# SwiftRL: Towards Efficient Reinforcement Learning on Real Processing-In-Memory Systems

Kailash Gogineni[1] Sai Santosh Dayapule[1] Juan Gómez-Luna[2] Karthikeya Gogineni[3]
Peng Wei[1] Tian Lan[1] Mohammad Sadrosadati[2] Onur Mutlu[2] Guru Venkataramani[1]

[1]George Washington University, USA [2]ETH Zürich, Switzerland [3]Independent



## ABSTRACT

Reinforcement Learning (RL) is the process by which an agent learns optimal behavior through interactions with experience datasets, all of which aim to maximize the reward signal. RL algorithms often face performance challenges in real-world applications, especially when training with extensive and diverse datasets. For instance, applications like autonomous vehicles include sensory data, dynamic traffic information (including movements of other vehicles and pedestrians), critical risk assessments, and varied agent actions. Consequently, RL training is significantly memory-bound due to sampling large experience datasets that may not fit entirely into the hardware caches and frequent data transfers needed between memory and the computation units (e.g., CPU, GPU), especially during batch updates. This bottleneck results in significant execution latencies and impacts the overall training time. To alleviate such issues, recently proposed memory-centric computing paradigms, like Processing-In-Memory (PIM), can address memory latency-related bottlenecks by performing the computations inside the memory devices.

In this paper, we present SwiftRL, which explores the potential of real-world PIM architectures to accelerate popular RL workloads and their training phases. We adapt RL algorithms, namely Tabular Q-learning and SARSA, on UPMEM PIM systems and first observe their performance using two different environments and three sampling strategies. We then implement performance optimization strategies during RL adaptation to PIM by approximating the Q-value update function (which avoids high performance costs due to runtime instruction emulation used by runtime libraries) and incorporating certain PIM-specific routines specifically needed by the underlying algorithms. Moreover, we develop and assess a multi-agent version of Q-learning optimized for hardware and illustrate how PIM can be leveraged for algorithmic scaling with multiple agents. We experimentally evaluate RL workloads on OpenAI GYM environments using UPMEM hardware. Our results demonstrate a near-linear scaling of 15× in performance when the number of PIM cores increases by 16× (125 to 2000). We also compare our PIM implementation against Intel(R) Xeon(R) Silver 4110 CPU and NVIDIA RTX 3090 GPU and observe superior performance on the UPMEM PIM System for different implementations.

## KEYWORDS

Reinforcement learning, Processing-in-memory, Multi-agent systems, Memory bottleneck, Performance analysis

## 1 INTRODUCTION

In recent years, Reinforcement Learning (RL) has seen important breakthroughs in various domains such as robotics, games, and healthcare [1–5]. All of these applications involve active interactions with the environment, from which observations are made in order to train the RL agent. Extending RL to real-world applications presents challenges, particularly in scenarios such as self-driving cars, where exploration and training in the field can be impractical and may even raise safety concerns while piloting a car due to delayed decisions stemming from the performance bottlenecks of underlying RL-based decision-making modules [6, 7].

Learning effective RL policies using *pre-collected* experience datasets reduces safety risks and the need for real-time interactions with the environment during training [6–8]. In this setting, a behavior policy interacts with the environment to collect a set of experiences and learns the optimal policy from pre-generated datasets during the training phase. Such offline RL has achieved considerable success in a diverse set of safety-critical applications, including healthcare decision-making, robotic manipulation skills, and certain recommendation systems [7, 9, 10]. Nevertheless, training from logs to learn a behavior policy and making *data-driven decisions* is a performance-intensive process, and there may be a vast amount of data points during the training phase [7]. Furthermore, frequent (re)training will be necessary on newly acquired data on such safety-critical applications, where modern processor-centric systems face the challenge of having to perform costly data movement between memory and processor units before performing RL computations, negatively impacting both the total execution time and the resulting energy consumption [11–14].

The Processing-In-Memory (PIM) [11, 12, 15–17] computing paradigm, which places the processing elements inside or close to the memory chips, is well positioned to address the performance bottlenecks of memory-intensive workloads. Despite being researched for decades, real-world PIM chips have only recently entered the commercial market. The UPMEM PIM computing platform [18] is the first commercially available architecture designed to accelerate memory-bound workloads [19–25]. Recent studies leverage PIM architectures to provide high performance and energy benefits on bioinformatics, neural networks, machine learning, database kernels, homomorphic operations and more [21–39]. However, no prior work has explored the adaptation of RL workloads on this real-world PIM architecture and evaluated its potential to accelerate the RL training phase, which is critical in efficiently learning effective policies.

In this paper, we present SwiftRL, where we accelerate RL algorithms, namely Tabular Q-learning [1, 40, 41] and SARSA [1], on UPMEM PIM systems and measure their performance using two different environments and three sampling strategies. We implement performance optimization strategies during RL adaptation to the PIM system via approximating the Q-value update function (which avoids high performance costs due to emulation used by

runtime libraries) and by adding certain custom PIM-specific routines needed by the underlying algorithms. Further, we evaluate the multi-agent version of Q-learning, showing how a real PIM system may be used for algorithmic scaling with multiple agents. Our experimental analysis demonstrates the performance and scalability of RL workloads across thousands of PIM cores on real-world OpenAI environments [42].

In summary, our paper makes the following contributions:

- We present a roofline model that highlights the memory-bounded behavior of RL workloads during their training phases. This motivates our SwiftRL design that accelerates the RL algorithms with PIM cores attached to the memory banks responsible for storing training datasets.
- We study the benefit of real in-memory computing systems on two RL algorithms learning under two distinct environments and various sampling strategies for experience data: sequential, stride-based, and random.
- We conduct scalability (strong scaling) tests by evaluating our RL workloads on thousands of PIM cores. Across all of our workloads, we observe a near-linear scaling of 15× in performance when the number of cores increases by 16× (125 to 2000 PIM cores).
- Our experimental results demonstrate superior performance of the real PIM system over implementations on Intel(R) Xeon(R) Silver 4110 CPU and NVIDIA RTX 3090 GPU, where the measured performance speedups of PIM adaptations are *at least* 1.62× and 4.84× respectively.
- We open-source our PIM implementations of RL training workloads at https://github.com/kailashg26/SwiftRL.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Reinforcement Learning

Reinforcement learning is a process where an agent learns to make decisions by mapping specific situations to actions in order to maximize a cumulative reward. The agent is not given explicit instructions on which actions to take; instead, it must experiment with different actions to discover which ones generate the greatest rewards [1]. In many real-world domains like guided navigation and autonomous mission controls, the RL agent learns entirely from a dataset of past interactions rather than interacting in real-time with the environment [7]. The dataset can be collected from agents following suboptimal or exploratory policies [7, 43]. The data logs include tasks such as *Frozen Lake and Taxi* [42]. To illustrate, frozen lake environment involves crossing a frozen lake on foot from start to goal without falling into any holes. The player may not always move in the intended direction due to the slippery nature of the frozen lake [42]. The taxi environment involves navigating to passengers in a grid world, picking them up, and dropping them off at one of four locations [42]. The end goal in this setting is still to optimize a reward function.

To elaborate on the operational workflow of offline reinforcement learning, we illustrate the process in Figure 1. We collect a large set of experiences using an unknown behavior policy[1] $\pi_\beta$, and the obtained dataset is labeled as $\mathcal{D}$ (Figure 1①). We perform this step once prior to the training phase of the reinforcement learning algorithm. In the offline training phase, the learning algorithm processes data tuples known as experiences[2] in the dataset, which includes states, actions, rewards, and next states $(s_i, a_i, r_i, s_i')$ [6]. The learning algorithm uses this data to repeatedly update a policy, specifically refining the quality values associated with state-action pairs until the expected rewards are reached. This involves reading (memory reads) from the dataset, learning, and then writing (memory writes) these updated quality values associated to the state-action pairs to the Q-table (②). After thoroughly training the policy (i.e., constructing the final Q-table) for a number of episodes[3], the policy is then ready for testing and deployment.

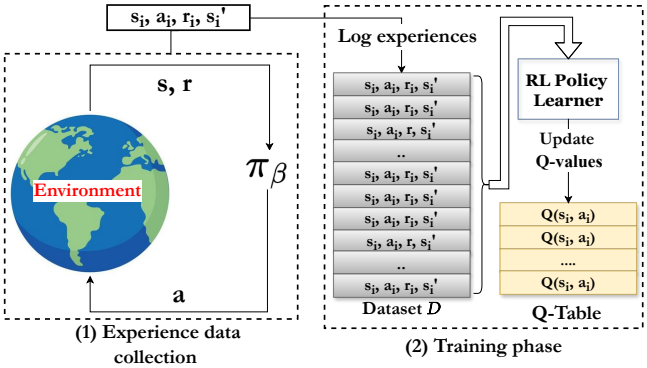


**Figure 1: The Operational Workflow of an Offline RL System. An unknown behavior policy $\pi_\beta$ is used to gather the dataset, denoted as $\mathcal{D}$. Training occurs without any interaction with the environment, and the policy (i.e., final Q-table) is deployed only after it is fully trained. The architecture is adapted from [7].**

RL algorithms typically exhibit memory-bounded behavior due to repeated memory accesses during the training phase. This involves iterating over the dataset multiple times to refine the policy and construct the final Q-table. To quantify the memory-boundedness of the CPU versions of our RL workloads (Q-learning [1, 40, 41], and SARSA learner [1]), we employ a roofline model [44] to visualize the extent to which our workloads are constrained by memory bandwidth and computational limits. Figure 2 shows the roofline model on an Intel(R) Core(TM) i7-9700K CPU (Coffee lake) with Intel Advisor [45].

The shaded area at the intersection of DRAM bandwidth and the peak compute performance roof is defined as the memory-bound area in the roofline plot. We make a key observation from Figure 2 that both the Q-learner and SARSA-learner CPU versions are in the memory-bound region. This is because their performance is primarily constrained by low DRAM bandwidth, which prevents the RL algorithms from achieving the maximum possible hardware performance. As a result, these RL workloads are potentially suitable for PIM.

[1] A *behavior policy* in reinforcement learning is the strategy an agent uses to explore and gather data from an environment [7]

[2] We refer to experiences as transitions or samples interchangeably [7].

[3] Note that each *episode* involves computing the quality values and updating the *quality* values associated with state-action pairs in the Q-table.

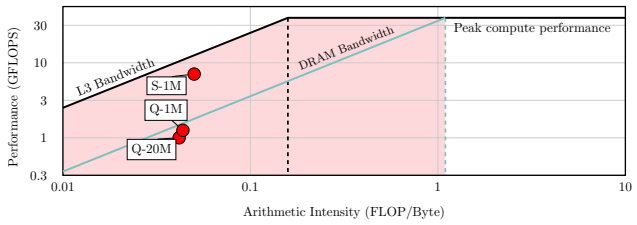


**Figure 2: The Roofline model depicts the performance characteristics of CPU versions in RL workloads, where "Q" refers to Q-learner [1, 40, 41], "S" signifies the SARSA learner [1], and "1M" and "20M" indicate the data size in millions of transitions. The workloads are tested on an Intel i7-9700K CPU.**

## 2.2 Processing-In-Memory

Recently, real-world Processing-In-Memory (PIM) [12, 21–34, 36, 38, 46–61] systems have emerged and are now part of the market landscape, with UPMEM [12, 21–34, 36, 38, 39, 46–59, 62–66] pioneering the first-ever commercialization of a PIM architecture. Additionally, there have been announcements regarding Samsung HBM-PIM [67, 68], Samsung AxDIMM [69], SK Hynix AiM [70], and Alibaba HB-PNM [71]. All these architectures have been prototyped and evaluated on real systems, sharing key and significant characteristics, as illustrated in Figure 3 for the UPMEM PIM system. First, these PIM systems feature a host CPU processor integrated with standard main memory, a deep cache hierarchy, and PIM-enabled memory modules. Second, the PIM-enabled memory contains multiple PIM chips connected to the host CPU through a memory channel. Third, the PIM processing elements operate at relatively low frequencies, typically a few hundred MegaHertz. Each PIM core (i.e., processing element; PIM PE) may include a small private instruction memory and a small data storage (scratchpad cache) memory. Fourth, PIM PEs can access data in their local memory bank, and there is typically no direct communication channel among the PIM cores. However, communication between multiple PIM cores occurs typically through the host CPU processor in architectures like UPMEM [19, 20, 22], HBM-PIM [67, 68], and AiM [70].

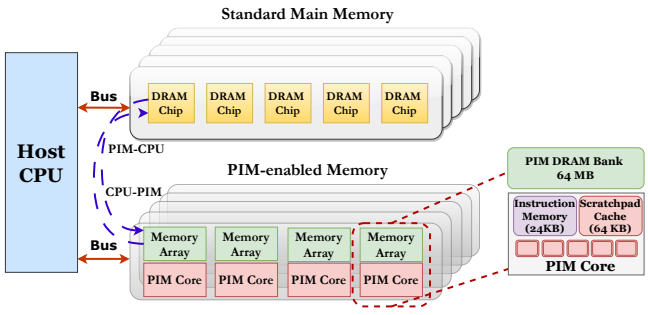


**Figure 3: Organization of a State-of-the-Art Processing-In-Memory (PIM) Architecture. Adapted from [19, 21, 24, 26, 28].**

In this study, we use UPMEM PIM, which uses conventional 2D DRAM arrays and tightly integrates them with general-purpose PIM cores, namely DRAM Processing Units (DPUs), on the same chip. UPMEM-based PIM systems use the Single Program Multiple Data (SPMD) [72] programming model. The DPUs are programmed using the C language with additional library calls [19, 20, 73, 74], while the host library offers C++, Python, and Java API support. The UPMEM SDK [74] supports common C data types and interfaces seamlessly with the LLVM compilation framework [75]. For a comprehensive listing of supported instructions, we refer the reader to the UPMEM PIM user manual [73].

The state-of-the-art UPMEM architecture has 20 PIM-enabled DIMMs, each with two ranks of 8 PIM chips. In total, 2,560 DPUs (PIM cores) are deeply pipelined and implemented with fine-grained multi-threading [76, 77] providing a peak throughput of 1 TOPS (*Tera operations/second*). Each PIM chip has eight 64-MB DRAM banks, with a programmable PIM core, 24-KB instruction memory (IRAM), and a 64-KB scratchpad memory (WRAM) coupled to each bank [26]. The DPUs have in-order 32-bit RISC-style instruction set architecture operating at 450 MHz [19, 20]. The DPU has 24 hardware threads, each with 24 32-bit general-purpose registers. They include native support for 32-bit integer addition/-subtraction and 8-bit multiplication. The complex operations, such as the multiplications on 64-bit integers, 32-bit floating point operations, and 64-bit floating point operations, are emulated by the run-time library and take tens to thousands of cycles [21, 22, 24, 26].

The conventional main memory and PIM-enabled memory modules exhibit distinct data layouts. The host processor can access MRAM banks for tasks such as copying input data (from main memory to MRAM, i.e., CPU to DPU) and retrieving results (from MRAM to main memory, i.e., DPU to CPU). Since there are no direct communication channels between DPUs, inter-DPU communication occurs exclusively through the host CPU, utilizing parallel CPU-DPU and DPU-CPU data transfers [24, 26, 78].

Even though we demonstrate adaptation of RL workloads to UPMEM architecture, our proposed optimization strategies are versatile and can be deployed on other real PIM hardware, resembling the architecture illustrated in Figure 3. Thus, we use the terms PIM core, PIM thread, DRAM bank, scratchpad, and CPU-PIM/PIM-CPU data transfer, which correspond to the DPU, tasklet, MRAM bank, WRAM, and CPU-DPU/DPU-CPU transfer in the PIMs implemented by UPMEM. For more detailed analysis of the UPMEM architecture, we refer the reader to [21, 22, 26, 50].

# 3 SWIFTRL DESIGN AND IMPLEMENTATION

In this section, we first study the memory behavior of RL workloads and their performance bottlenecks. We then demonstrate our workload adaptation strategies for PIM.

## 3.1 Memory Behavior of RL workloads

The training phase of offline reinforcement learning is heavily influenced by the need to access experiences stored in a dataset, which were gathered using a specific behavior policy. This phase often encounters memory bottlenecks due to two primary reasons: ① RL algorithms sequentially process large volumes of historical experience data for optimal policy learning, and ② Different sampling strategies (impacting data locality) are used during the learning

process. For instance, complex environments (e.g., Atari [79], StarCraft [80, 81]) typically require the agent to explore a broad range of the state-action space in early time-steps, so the agent performs random sampling. This random sampling can result in irregular memory access patterns and poor data locality as the state-action space expands [82–84]. To mitigate these bottlenecks, we first implement our workloads with different sampling strategies and test the efficiency of PIM. We distribute data chunks across various PIM cores in memory to accelerate the training phase and execute batch updates for each iteration in near-bank PIM cores [21, 22, 24].

## 3.2 Implementation of RL algorithms on PIM Architecture

Tabular Q-learning [1, 40, 41] and SARSA [1] are popular RL algorithms widely used in various applications [85–93] and as part of machine learning for hardware/software systems [94–98]. Both the algorithms learn from Q-tables. The Q-tables store the *quality* values associated with state-action pairs.

*3.2.1* ***Q-learning.*** Tabular Q-learning [1, 40, 41] is a widely-used model-free and off-policy RL workload [1, 40, 41] that learns through a trial-and-error approach [1, 40, 41, 85–91]. Agents interact with the environment based on some off-policy approach [1, 7] like *random selection* or *epsilon greedy*. In order to train the workload offline, we employ a behavior policy (i.e., random action selection) [1, 7] to collect the dataset $\mathcal{D}$ once. While we use the random action selection, other policies such as *epsilon greedy* and *boltzmann* can also be used to execute actions on the environment and log the experiences [1, 99, 100]. The objective in this offline setting is still to collect enough experiences and learn a policy (i.e., constructing the final Q-table) that maximizes the expected return [1].

Each experience tuple in the dataset is represented as $\mathcal{D}_i = (s_i, a_i, r_i, s'_i)$, where $i$ denotes the index of the transition within the dataset. $s'$ represents the next state resulting from taking action $a$ in state $s$, and the reward $r$ is determined by the state-action pair $(s, a)$. $\gamma$ is a discount factor that determines the balance between the immediate and future rewards [1]. $\alpha$ is the learning rate parameter that determines the rate at which the quality values associated with the state-action pairs in the Q-table are updated [1].

To illustrate the offline training phase of the tabular Q-learning algorithm, we outline the steps in Algorithm 1. The Q-learning algorithm initializes a Q-table with arbitrary values. For each episode (line 5), multiple batches (line 7) are selected to iteratively update the Q-values (line 12) based on the total number of experiences in the dataset [1]. The term $\max_{a'} Q(s', a')$ is the maximum Q-value for the next state $s'$, across all possible actions $a'$ (line 10). In other words, it calculates the highest Q-value for the next state $s'$, by trying out all the available actions $a'$, ensuring the selection of the most rewarding action for the next state.

We note that multi-agent reinforcement learning has recently been widely adopted in several popular domains from gaming to autonomous driving [101–110]. In this decentralized Q-learning approach, each agent maintains its own experience dataset and updates its Q-table to make decisions [111]. In our modeling, the independent learners do not observe the actions of other agents in the system.

**Algorithm 1:** Tabular Q-learning Algorithm

1: **Inputs:**
2: Experience data collected offline.
3: Initialize a Q-table with arbitrary/zero values.
4: Hyper-parameters: Learning Rate ($\alpha$), Discount Factor ($\gamma$), *num_episodes*.
5: **for** each *episode* in *num_episodes* **do**
6: **Batched updates:**
7: **for** each experience $(s_i, a_i, r_i, s'_i)$ in selected batch **do**
8: **Q-learning Update:**
9: Calculate Q-value target for the experience:
10: $q_{\text{value target}} \leftarrow \mathrm{r} + \gamma \cdot \max_{a'} Q(s', a')$
11: Update Q-values for the current state-action:
12: $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (q_{\text{value target}} - Q(s, a))$
13: **end for**
14: **end for**
15: **Output:**
16: Final Q-table with the learned Q-values.

**PIM Implementation:** Initially, the training dataset ($\mathcal{D}$) resides in the main memory of the host CPU. The first step involves transferring individual chunks of the training dataset to the local memories (DRAM banks) of PIM cores. To maximize parallelism in the Q-value update kernel, we partition the training dataset ($\mathcal{D}$) so that each PIM core ($\mathcal{P}_i$) handles a distinct chunk of data ($C_D$), enabling faster memory accesses. Secondly, within each PIM core ($\mathcal{P}_i$), Q-learning updates (line 12 from Algorithm 1) are computed for each transition ($\mathcal{T}_i$) in ($C_D$) using a single hardware thread (this work focuses solely on PIM-core parallelism). Each PIM core ($\mathcal{P}_i$) is allocated a Q-table with arbitrary values, and all PIM cores train in parallel asynchronously updating their local Q-tables using the *state-action-reward-next state* trajectories. The third step involves transferring partial results obtained from processing the individual data chunk in a specific PIM core back to the host processor to aggregate the final Q-table. The operational workflow of SwiftRL execution on a real PIM system is described in Figure 4.

We implement six versions of Q-learning with different input data types, and three sampling strategies (*SEQ* - sequentially sampling experiences, *RAN* - randomly selecting experiences to update for thorough exploration, and *STR* - selecting experiences at regular intervals). We note that experimenting with different sampling strategies enables us to assess their impact on the computational workload during the sampling phase.

- `Q-learner-FP32` trains with 32-bit real values for $\max_{a'} Q(s', a')$, and Q-table initialization and no scaling optimization. We use 32-bit data types as they provide a more accurate representation of transition data across episodes than 16-bit data.
- `Q-learner-INT32` trains with 32-bit fixed point representations of $\max_{a'} Q(s', a')$, Q-table initialization, and we scale up the reward $r$ for each experience $\mathcal{T}_i$, learning rate $\alpha = 0.1$, and the discount factor $\gamma = 0.95$ by using a constant scale factor=10,000 (scale factor is chosen to prevent overflow and underflow errors while ensuring sufficient precision for floating-point multiplications) to mitigate the cost of

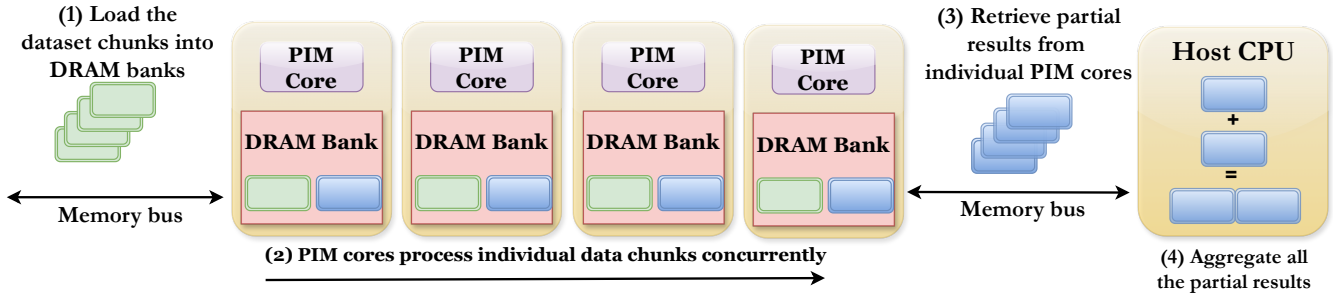


**Figure 4: SwiftRL execution on a real PIM system. The execution phase comprises four main steps: (1) loading the input dataset chunks into individual DRAM banks of PIM-enabled memory, (2) executing the RL workload (kernel) on PIM cores in parallel operating on different chunks of data, (3) retrieving partial results from DRAM banks to the host CPU, and (4) aggregating partial results on the host processor.**

floating-point multiplications for the update equation (line 12) in Algorithm 1. We scale down after the Q-value update and finally store the descaled value in the Q-table. This hybrid implementation is motivated by the fact that real-world PIM cores only support arithmetic operations of limited precision. For instance, UPMEM DPUs [73] execute naive 8-bit integer multiplication and emulate the 32-bit integer multiplication using shift-and-add instructions [21, 22]. Apart from UPMEM, other accelerators like HBM-PIM [67] and AiM [70] feature only 16-bit floating point operations. Additionally, replacing the compiler-generated 16-bit and 32-bit multiplications with *custom 8-bit built-in multiplications* [24] may be adopted to boost the training time further and reduce the number of instructions, but this optimization, which is specific to UPMEM, might only apply to some environments (e.g., frozen lake) which have limited value range that fits in 8 bits. Additionally, we implement custom routines such as *linear congruential generator [112]* to replicate the functionality of the `rand()` function within PIM cores (some standard library functions are not supported by UPMEM PIM architecture).

- We evaluate the performance of the aforementioned two versions using different sampling strategies with the data laid out in diverse memory access patterns. The six implemented versions include: `Q-learner-SEQ-FP32`, `Q-learner-RAN-FP32`, `Q-learner-STR-FP32`, `Q-learner-SEQ-INT32`, `Q-learner-RAN-INT32`, `Q-learner-STR-INT32`.

***Multi-agent Q-learning.*** For multi-agent Q-learning, we employ a random policy to explore the environment and log individualized experiences. Subsequently, we load the agent-specific datasets into the PIM cores' local memories (DRAM banks). The only difference in this workload compared to the Q-learning is that each PIM core will have agent-specific experiences to learn from, enabling multiple independent learners concurrently. We pin each agent to a PIM core in this design and iteratively train on its unique dataset. The host processor retrieves the final Q-tables for multiple agents upon completion of the training process. Notably, the aggregation step would be unnecessary in this setting as the learners operate independently throughout the training.

*3.2.2* ***SARSA Learning.*** SARSA (State-Action-Reward-State-Action) is an on-policy algorithm that learns the optimal policy by continuously updating the policy toward achieving the maximum reward [1]. The only difference for SARSA compared to Q-learning is that, SARSA employs an epsilon-greedy approach [1] to select next action $a'$ (Equation 1) and this function uses a custom routine (`rand()` function) to generate random action [1].
The SARSA update equation is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(\mathrm{r} + \gamma Q(s',a') - Q(s,a)\right) \tag{1}$$

The terms are similar to Q-learning, and the SARSA algorithm has the same training pattern, but the key difference is $Q(s',a')$ term, where the action $a'$ is the actual next action taken, following the policy being learned. Instead, in Q-learning maximum Q-value across all possible actions is used [1].

**PIM Implementation:** To extract parallelism in the SARSA learning update kernel, the first step involves partitioning the training dataset ($\mathcal{D}$) into subsets of equal size so that each PIM core ($\mathcal{P}_i$) receives a unique chunk of data ($C_D$). Secondly, a hardware thread is dedicated to computing SARSA updates (Equation 1) within each PIM core ($\mathcal{P}_i$). We note that, similar to the Tabular Q-learning implementation, we currently focus on the core-level parallelism in this work. In the third step, the results obtained from processing individual chunks of data within each PIM core ($\mathcal{P}_i$) are then transferred back to the host processor. The host processor finally aggregates all the partial results from multiple PIM cores, facilitating the final Q-table learned using SARSA update rule [1]. SARSA learner follows the same arithmetic intensity as Q-learning since only one floating-point multiplication is needed, and we substitute it with the fixed-point representation and scaling optimization.

We implement different variations of SARSA learning, featuring different data types (FP32 and INT32) and diverse sampling strategies. `SARSA-learner-INT32` learning uses 32-bit fixed point representations of $Q(s',a')$, $Q$-table initialization, and we scale up the reward $r$ for each experience $\mathcal{T}_i$, learning rate $\alpha = 0.1$, and the discount factor $\gamma = 0.95$ by using a constant scale factor=10,000 and scale down after the experience update and

finally transfer Q-values back to the host CPU in original precision. We implement the `SARSA-learner-INT32` as a substitute for the naive version (`SARSA-learner-FP32`) that trains with 32-bit real values (for variables mentioned above), which takes relatively longer execution cycles as the UPMEM PIM only supports native integer multiplications [19, 21, 22, 24] (specifically 8-bit). We evaluate the performance of the two variations mentioned above using diverse sampling strategies (*SEQ* - sequentially sampling experiences, *RAN* - randomly selecting experiences to update for thorough exploration, and *STR* - selecting experiences at regular intervals). The six implemented versions for SARSA are: `SARSA-SEQ-FP32, SARSA-RAN-FP32, SARSA-STR-FP32, SARSA-SEQ-INT32, SARSA-RAN-INT32, SARSA-STR-INT32`.

# 4 EXPERIMENTAL EVALUATION

In this section, we first describe our experimental setup. Second, we evaluate SwiftRL in terms of training quality (Section 4.2), and performance scaling characteristics, specifically strong scaling results (Section 4.3). Third, we compare our PIM implementations to state-of-the-art CPU and GPU implementations (Section 4.4).

## 4.1 Experimental Setup

Table 1 summarizes the specifications of the UPMEM PIM, CPU, and GPU systems. We perform our experiments on a real-world PIM server [156] with 2,524 PIM cores running at 425 MHz and 158 GB of DRAM memory. The table also outlines the characteristics of the baseline CPU and GPU systems used for comparative analysis.

**Table 1: Evaluated UPMEM PIM system [21, 22, 24] and baseline CPU [113] and GPU [114] specifications.**

| Metric | UPMEM PIM System | Intel Xeon Silver 4110 CPU [113] | NVIDIA Ampere RTX 3090 GPU [114] |
|---|---|---|---|
| Processor Node | 2x nm | 14 nm | 8 nm |
| Total Cores | 2,524 | 8 (16 threads) | 82 cores (10496 SIMD lanes) |
| Frequency | 425 MHz | 2.4 GHz (3 GHz Turboboost) | 1.70 GHz |
| Peak Performance | 1,088 GOPS | 38 GFLOPS | 35,580 GFLOPS |
| Main Memory | 158 GB | 132 GB | 24 GB |
| Memory Bandwidth | 2145 GB/s | 28.8 GB/s | 936.2 GB/s |
| Component TDP | 280 W | 85 W | 350 W |

The RL workloads are evaluated using popular environments, namely frozen lake and taxi, developed by Gym [42]. The taxi environment has a state space of $Discrete(500)$ since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), 4 destination locations, and an action space of $Discrete(6)$, while the frozen lake environment has a state space of $Discrete(16)$ since the map size is $4 \times 4$ and an action space of $Discrete(4)$. In our experiments, we use a learning rate of 0.1, $\gamma$-the discount factor, set to 0.95, and train the workloads for 2,000 episodes. To obtain a partially trained policy, we train a random behavior policy online and log the experiences until the policy performance achieves a performance threshold (Average reward) for frozen lake and taxi, respectively. We collected 1 million transitions for frozen lake and 5 million for the taxi paradigm. We collected more data for the taxi environment [1] because it encompasses 31.25× more states compared to the frozen lake envrionment [1, 42].

## 4.2 RL Training Quality

The trained policy in Q-learning and SARSA is evaluated with the frozen lake and taxi environments. The hyper-parameters for the evaluation include 1,000 episodes with synchronization period ($\tau$) set to 50 [115, 116]. $\tau$ refers to the communication rounds for the Inter-PIM core communication, where the total number of episodes, denoted by $\mathcal{E}$, is assumed to be divisible by $\tau$. The algorithm outputs the final aggregated Q-estimate as the average of all local Q-tables. In this context, $Comm_{rounds}$ is defined as $\mathcal{E}/\tau$, representing the rounds of communication required in the training phase. Our performance results of PIM implementations takes into account the estimated time, which includes the impact of $Comm_{rounds}$, specifically in the context of Inter-PIM core communication.

For the frozen lake environment, for `Q-learner-SEQ`, we estimate the average mean reward for 1000 episodes with synchronization period ($\tau$) set at 10, 25, and 50 is observed to be 0.74, 0.7295, and 0.70, respectively. These are relatively same or slightly better than CPU implementation. For the `SARSA-learner-SEQ` with a $\tau$ of 50, a mean reward of 0.71 is registered against the 0.723 of the CPU version. We note that the `Q/SARSA-learner-RAN/STR` also perform on par with `Q/SARSA-learner-SEQ`.

For the taxi environment, we evaluate an approximated model `Q-learner-SEQ` algorithm with synchronization period ($\tau$) set at 50 is observed to be -7.9 against the -8.6 of the CPU version. The `SARSA-learner-SEQ` exhibits similar behavior with $\tau$ of 50, a mean reward of $-8.8$ against the $-8.2$ of the CPU version. Even with INT32, we convert the values back from INT32 to FP32 using scaling optimization before the PIM cores transfer the partial results to the host CPU.

## 4.3 Performance Analysis of PIM Kernels: Scaling PIM Cores

In this section, we evaluate the performance scaling characteristics of our RL workloads using *strong scaling* experiments. Figure 5 illustrates the performance scaling results across 125-2000 PIM cores for various versions of our RL workloads. We present the total execution time, which is further broken down into (1) the execution time of the PIM kernel, (2) communication time between the host processor and PIM cores for initial dataset transfer (CPU-PIM), (3) communication time between PIM cores and the host CPU for final result transfer (PIM-CPU), and (4) communication time between PIM cores for Q-values transfer (Inter PIM core). The Inter PIM core synchronization is estimated using the synchronization period ($\tau$), where the total number of episodes, denoted by $\mathcal{E}$, is

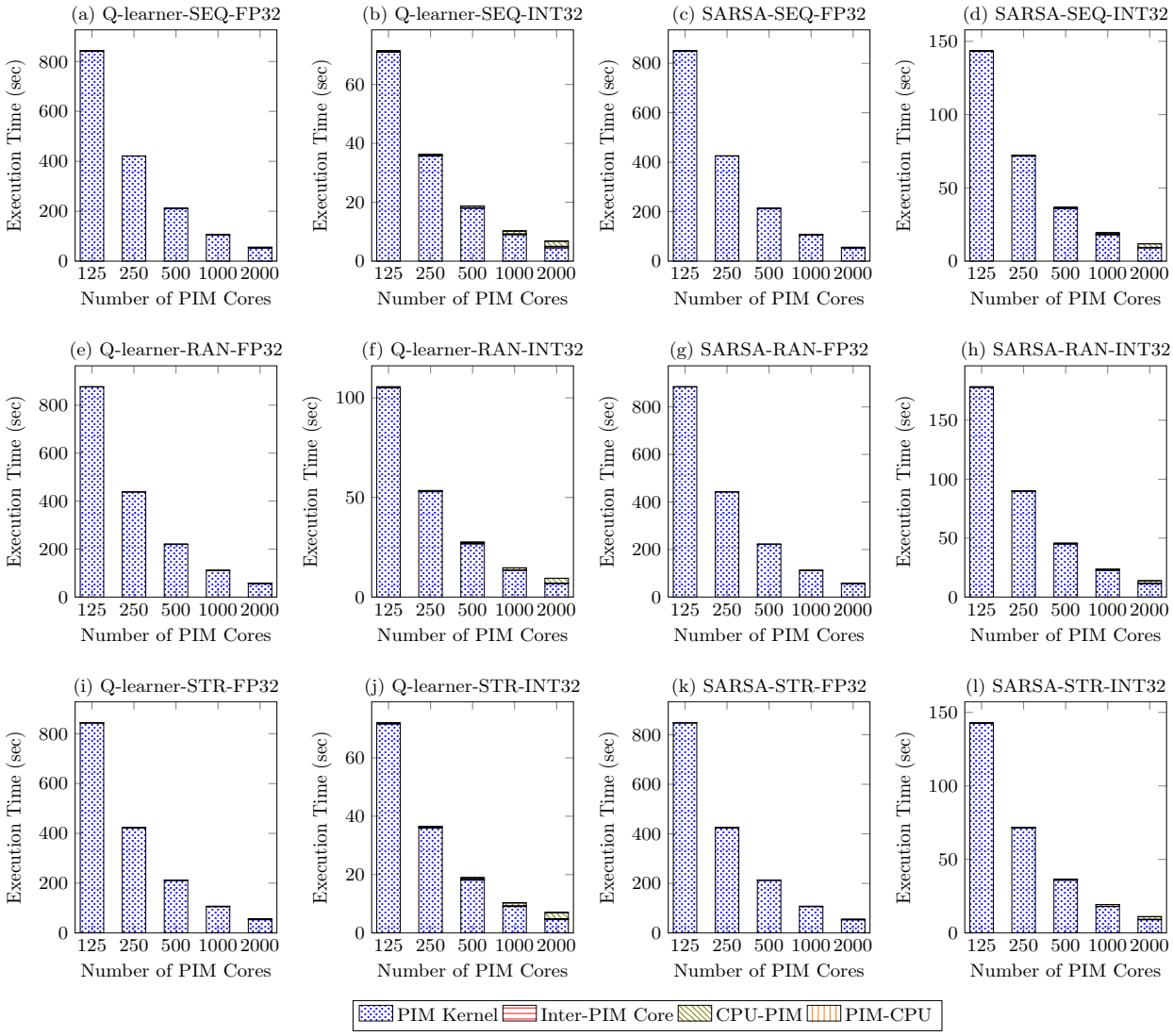


**Figure 5: Execution time (measured in seconds) of RL workloads on 125, 250, 500, 1,000 and 2,000 PIM cores (x-axis) with each PIM core running with single thread for a frozen lake environment. In this illustration, SEQ-RAN-STR represent sequential, random and stride-based sampling techniques and both the Q-learning and SARSA learners are evaluated with FP32 and INT32 data types. The synchronization period $\tau$ and the stride value in this experiment is set to 50 and 4 respectively.**

assumed to be divisible by $\tau$. Given that we train for 2,000 episodes with $\tau$ set to 50, the value of $Comm_{rounds}$ is 40. During this process, the local results are aggregated before advancing to next episode.

To demonstrate how the performance of RL workloads scales with an increasing number of PIM cores, we maintain a fixed dataset size for the frozen lake and taxi environments [42]. We make four observations from Figure 5 and Figure 6. First, we observe that the PIM kernel time scales linearly with the PIM cores. On average, across all RL workloads for FP32 and INT32, the speedup from 125 PIM cores to 2,000 PIM cores exceeds 15×. The scaling for the taxi environment also follows a similar trend in performance scaling (Figure 6). This speedup can be attributed to the large memory bandwidth and increased concurrency offered by scaling the number of PIM cores.

Second, the communication cost between PIM cores is relatively small for RL workloads with the frozen lake scenario due to minimal data transfer between PIM cores. The largest fraction of Inter PIM core synchronization over the total execution is 21.19% for `Q-learner-STR-INT32` with 2000 PIM cores for the taxi environment. Even so, 2000 PIM core configuration significantly reduces the overall execution time for `Q-learner`. This trend is similar in `Q-learner-SEQ-INT32` with 20.79% over the total execution

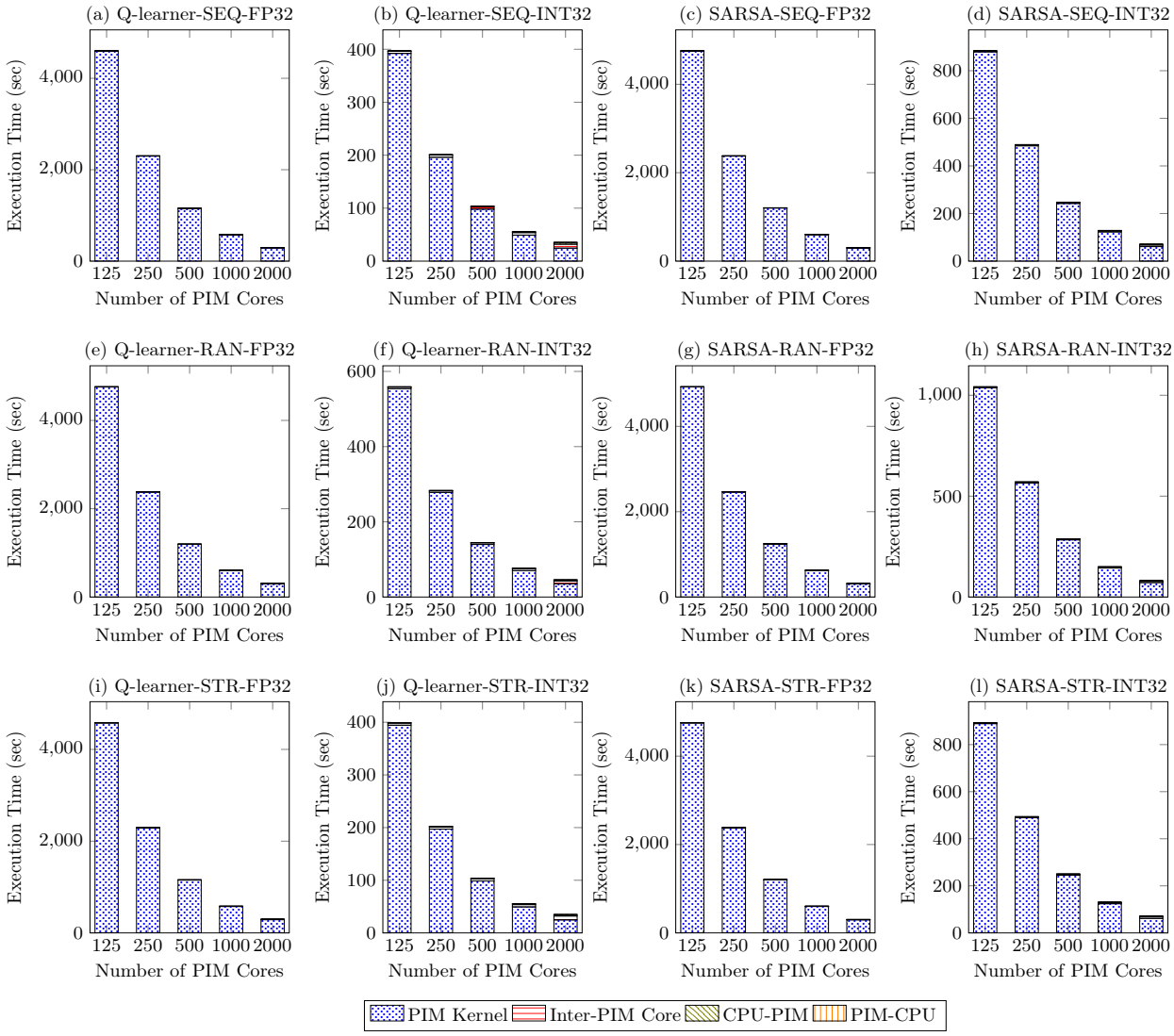


**Figure 6: Execution time (sec) (y-axis) of RL workloads on 125, 250, 500, 1,000 and 2,000 PIM cores (x-axis) with each PIM core running with single thread for a taxi environment. In this illustration, SEQ-RAN-STR represent sequential, random and stride-based sampling techniques and both the Q-learning and SARSA learners are evaluated with FP32 and INT32 data types. The synchronization period $\tau$ and the stride value in this experiment is set to 50 and 4 respectively.**

time spent on inter-PIM core communication, followed by 17% for `Q-learner-RAN-INT32` in the taxi environment (Figure 6). This is because the taxi environment requires approximately 47× more data (Q-values compared to frozen lake [42]) that needs to be transferred between the PIM cores. Overall, the prominence of inter-PIM core communication correlates with the amount of data exchanged, with taxi environment exhibiting dominance due to its higher data exchange demands compared to frozen lake.

Third, in terms of initial setup costs, `Q-learner-STR-INT32` incurs the highest cost at 29.61% in the frozen lake environment due to the initial dataset transfer (CPU-PIM) amount to each PIM core is comparatively more substantial than the small amount of data (Q-values) being transferred between the PIM cores.

Fourth, the communication cost associated with initial setup (CPU-PIM) and final PIM-CPU transfers exhibit negligible overhead on the total execution time for the taxi environment.

## 4.4 Comparison with CPU/GPU platforms

We implemented the CPU and GPU versions of all of our RL algorithms that are widely considered as state-of-the-art baselines [1, 40, 41, 117]: (1) CPU-V1: Multiple threads update a shared Q-table through the update function. Each thread operates on a portion of the dataset independently, and the same Q-table is used

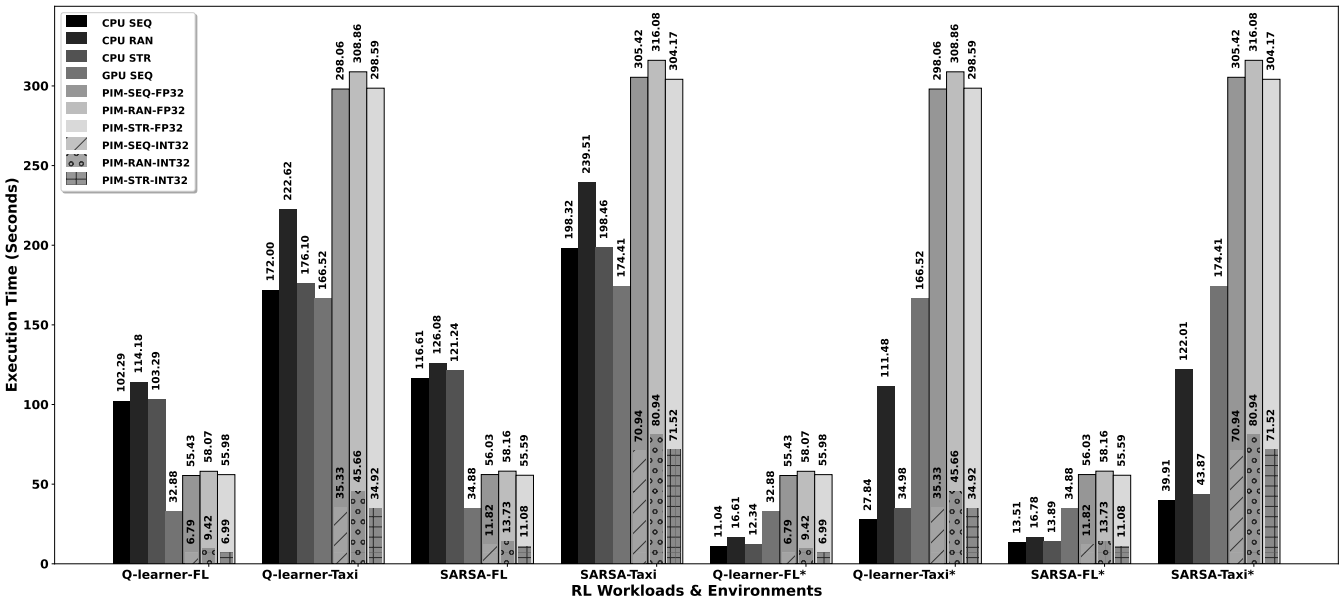


**Figure 7: Execution time (in seconds) of CPU, GPU, and PIM for the training phase of RL workloads. The solid bars represent PIM execution time without scaling optimization, utilizing FP32. Meanwhile, the patterned bars depict execution time with scaling optimization, employing INT32. For PIM evaluation, we utilize 2,000 PIM cores (best-performing number) for workload execution. The results for performance comparison with CPU-V2 version, Q-learning, and SARSA with FL and taxi are denoted in (*). The RL algorithms, `Q-learner-FL`, `Q-learner-Taxi`, `SARSA-FL`, and `SARSA-Taxi`, refer to Tabular Q-learning and SARSA algorithms trained with frozen lake and taxi environments.**

for updates, and (2) CPU-V2: Distributed version, where multiple threads update the portions of data independently by using local Q-tables.

Figure 7 illustrates the execution times of Q-learning and SARSA learning on PIM, CPU, and GPU with frozen lake and taxi environments [42]. We make four observations. First, from our analysis, we observe that both `Q-learner` and `SARSA` show higher execution times across SEQ/STR/RAN due to floating-point operations, which are not natively supported by our PIM architecture [19, 20]. Despite this limitation, our `Q-learner-SEQ-FP32` and `SARSA-SEQ-FP32` workloads are 1.84× and 2.08× faster than their counterpart CPU versions (CPU-V1) for frozen lake task.

Second, for frozen lake, when using random sampling to prioritize exploration (`Q-learner-RAN-FP32`), we observed a speedup of 1.96× compared to the CPU-V1 version. However, in the taxi environment, compared to the CPU-V1, our PIM implementation (`Q-learner-SEQ-RAN-STR-FP32`) is almost 0.64× slower on average due to the large number of floating point operations performed corresponding to huge state-action size. Compared to the CPU-V2 in the taxi environment, we observe a slowdown in execution time for sequential and stride-based sampling techniques against the CPU-V2. This is due to CPU hardware prefetcher's ability to enhance cache locality for sequential and stride memory access patterns, where CPU-cache latencies are lower than that of PIM-DRAM.

Third, using fixed-point representation (INT32) offers higher performance than the floating-point (FP32) format. For example, `Q-learner-SEQ-INT32` is 8.16× faster than `Q-learner-SEQ-FP32`, and the trend is similar across various sampling strategies. This is the result of using natively supported instructions (even though 32-bit integer multiplications are emulated by the run-time library [21, 22]).

Fourth, the GPU version of Q-learning with sequential sampling outperforms `Q-learner-SEQ-FP32-FL` by 1.68×, benefiting from the Ampere architectures' large set of SIMD lanes and enhanced memory bandwidth. Notably, our `Q-learner-SEQ-INT32-FL` achieves a substantial speedup of 4.84× over the GPU version due to INT32 instructions. The `SARSA-SEQ-INT32-FL` achieves a speedup over `SARSA-SEQ-FP32-FL` by 4.73×.

Finally, our findings highlight the UPMEM architectures potential for accelerating the training of multiple independent Q-learners, where each agent trains an offline dataset of size 10,000 (frozen lake) transitions and learns individual optimal policies. To illustrate, when training 1,000 agents, each with 10,000 transitions, for 2,000 episodes, the overall execution time is approximately 996.52 seconds. Scaling up to 2,000 agents increases the execution time to about 1,943.78 seconds on an Xeon CPU.

Our PIM implementation with fixed-point representation introduces agent-level parallelism, demonstrating algorithmic scalability. By training individual agents on PIM cores, we achieve significant speedup compared to their baseline CPU version, which utilizes multiple independent Tabular Q-learners. Specifically, SwiftRL achieves a speedup of approximately 11.23× for 1,000 agents and 21.92× for

2,000 agents when executing on 1,000 and 2,000 PIM cores, respectively.

# 5 KEY TAKEAWAYS

Our design and evaluation of SwiftRL, the first-known implementation that accelerates RL workloads on processing-in-memory systems, gave us several insights:

- RL workloads demonstrate reduced performance potential on UPMEM hardware PIM accelerator due to instruction emulation by the runtime library as the floating-point operations are not supported by the UPMEM platform. To tackle this, we adopt 32-bit fixed-point representations (Section 4.3).
- The most suited reinforcement learning (RL) algorithms for the UPMEM PIM architecture are those that have memory-intensive tasks and require minimal communication between the inter-PIM cores. For instance, our study shows multi-agent Q-learning demonstrates better hardware adaptability (Section 4.4).
- Scaling the PIM cores linearly leads to a nearly proportional reduction in the execution time for a given working set size (Section 4.3).
- We demonstrate that PIM is beneficial for random memory accesses. However, when it comes to accessing data sequentially or in stride-based patterns, the CPU hardware prefetcher's strong capability in managing data locality results in improved performance (Section 4.4).

# 6 RELATED WORK

To our knowledge, our work represents the first instance of adapting RL on real PIM architectures. We have already extensively compared SwiftRL to state-of-the-art CPU-based and GPU-based systems and presented the strong scaling experiments in Sections 4.4 and 4.3, respectively. In this section, we briefly summarize other related works in two categories: ① Outlining recent advancements in leveraging PIM systems to accelerate workloads, including deep learning and machine learning. ② Reviewing prior efforts to accelerate RL and highlighting how our work distinguishes itself.

## 6.1 Processing-in-Memory Systems

*6.1.1 DL Training and Inference.* Prior efforts leverage PIM systems to accelerate deep learning (DL) inference and training phases [36, 47, 118–122] . For instance, various proposals have been studied to accelerate DL inference phases, including CNNs [13, 123, 124], RNNs [14, 70], and recommendation systems [69, 71]. Another avenue of exploration in academia and industry capitalizes on the analog computation capabilities of non-volatile memories (NVMs), particularly for tasks like matrix-vector multiplication, thereby facilitating the training of deep neural networks [120, 121, 125]. Samsung's AxDIMM is an illustrative prototype, embedding an FPGA fabric in the DIMM's buffer chip, specifically designed to accelerate recommendation inference in Deep Learning Recommendation Models (DLRMs) [69]. Additionally, SK Hynix has introduced the Accelerator-in-Memory, a PIM architecture based on GDDR6, featuring specialized multiply-and-accumulate units and lookup-table-based activation functions to expedite deep learning workloads [126].

*6.1.2 PIM for ML algorithms.* Few related prior works propose solutions for ML algorithms and evaluate the performance benefits of PIM technologies [127–129]. For instance, UPMEM PIM is the first real-world processing-in-memory architecture used to accelerate ML workloads encompassing tasks like linear regression, logistic regression, and K-nearest neighbors [24]. Another line of work leverages different memory technologies (e.g., 3D-stacked DRAM [127, 129], SRAM [128]) to accelerate memory-bound machine learning applications [128–131]. None of these works present a comprehensive implementation and evaluation of RL algorithms utilizing a real processing-in-memory architecture.

*6.1.3 UPMEM PIM system.* Several studies have focused on characterizing and outlining the architecture of UPMEM's PIM system [18, 21, 22, 46, 132]. There are several works that explore accelerating variety of applications and algorithms UPMEM's PIM system, such as ML training/inference [21–24, 32, 36, 39, 133, 134], bioinformatics [30, 33, 38, 64, 135], analytics & databases [37, 51, 62, 136], security [34, 137], distributed optimization algorithms [31] and more [25–30, 32, 35, 50]. However, none of the prior works have explored reinforcement learning (RL) algorithms on UPMEM's PIM system, a gap that we fill by implementing and conducting a comprehensive evaluation in this paper.

## 6.2 Accelerating Reinforcement Learning Workloads

*6.2.1 Distributed Training.* Prior works on distributed training have been proposed to accelerate the training phase of RL workloads [138–143]. Another strategy for multi-agent RL acceleration is to restrict the agent interactions to one-hop neighborhoods and adopt a distributed training strategy to accelerate the training phase [144]. However, training on VM-based approaches still requires extensive management of the cluster and deploying the training jobs. Prior studies, like FA3C [139], have focused on accelerating multiple parallel worker scenarios, where each agent is controlled independently within their own environments using single-agent RL algorithms. Contrary to that, SwiftRL designs a distributed learning architecture, with PIM cores executed concurrently without extra cluster management.

*6.2.2 Quantization.* Low-precision (Quantization) training for neural networks reduces the neural network weights, enables faster compute operations, and minimizes the memory transfer computation time. Quantization aware training [145, 146], post-quantization training [147, 148], and mixed precision [149] demonstrated that neural networks may be quantized to a lower precision without significant degradation in the accuracy or rewards. Furthermore, to speed up the training, prior works have shown that half-precision quantization can yield significant performance benefits and improve hardware efficiency by reducing precision from FP32 to FP16 or even lower while achieving adequate convergence [150]. Other relevant approaches include QuaRL [151], where the authors demonstrated that applying quantization on RL algorithms and quantizing the policies down to $\leq 8$ bits led to substantial speedups compared to full precision training. In contrast, we accelerate the training phase of offline RL workloads with large datasets on a real-world PIM architecture that exhibits a memory-bounded behavior.

## 7 CONCLUSION

In this paper, by adapting and implementing popular RL algorithms on a real Processing-in-Memory (PIM) architecture, we explore the potential of memory-centric systems in Reinforcement Learning (RL) training. We evaluate our PIM-based Q-learning and SARSA algorithm implementations on the UPMEM PIM system with up to 2000 PIM cores. We explore several optimization strategies that will enhance the performance of these RL workloads under different input data types and sampling strategies. We evaluate the quality, performance, and scalability of RL workloads on PIM architectures compared to state-of-the-art CPU and GPU baselines. Our findings indicate that PIM systems offer superior performance compared to CPUs and GPUs when handling memory-intensive RL workloads. Our studies demonstrate a near-linear scaling of 15× in performance when the number of PIM cores increases by a factor of 16× (125 to 2000). Our research results demonstrate that PIM systems have the potential to serve as effective accelerators for a diverse range of RL algorithms in the future.

## ACKNOWLEDGMENTS

This research is based on work supported by the National Science Foundation under grant CCF-2114415. We thank UPMEM for providing hardware resources to perform this research. We acknowledge the generous gifts from our industrial partners, including Google, Huawei, Intel, and Microsoft. This work is supported in part by the Semiconductor Research Corporation (SRC), the ETH Future Computing Laboratory (EFCL), and the AI Chip Center for Emerging Smart Systems (ACCESS).

## REFERENCES

[1] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction. Second Ed." *A Bradford Book*, 2018.
[2] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster Level in StarCraft II using Multi-Agent Reinforcement Learning," *Nature*, vol. 575, 2019.
[3] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving Rubik's Cube with a Robot Hand," *arXiv preprint arXiv:1910.07113*, 2019.
[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, 2016.
[5] C. Yu, J. Liu, S. Nemati, and G. Yin, "Reinforcement Learning in Healthcare: A Survey," *ACM CSUR*, vol. 55, 2021.
[6] Prudencio, Rafael Figueiredo and Maximo, Marcos R.O.A. and Colombini, Esther Luna, "A survey on offline reinforcement learning: Taxonomy, review, and open problems," *IEEE TNNLS*, 2023.
[7] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems," *arXiv preprint arXiv:2005.01643*, 2020.
[8] R. Agarwal, D. Schuurmans, and M. Norouzi, "An Optimistic Perspective on Offline Reinforcement Learning," in *ICML*, 2020.
[9] S. Tang, M. Makar, M. Sjoding, F. Doshi-Velez, and J. Wiens, "Leveraging Factored Action Spaces for Efficient Offline Reinforcement Learning in Healthcare," *NeurIPS*, vol. 35, 2022.
[10] X. Zhan, H. Xu, Y. Zhang, X. Zhu, H. Yin, and Y. Zheng, "DeepThermal: Combustion Optimization for Thermal Power Generating Units Using Offline Reinforcement Learning," in *AAAI*, vol. 36, 2022.
[11] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," *Microprocessors and Microsystems*, vol. 67, 2019.
[12] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A Modern Primer on Processing in Memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, 2022.
[13] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *APSLOS*, 2018.
[14] A. Boroumand, S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu, "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks," in *PACT*, 2021.
[15] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-In-Memory: A Workload-driven Perspective," *IBM Journal of Research and Development*, vol. 63, 2019.
[16] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," *arXiv preprint arXiv:1905.09822*, 2019.
[17] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Enabling Practical Processing in and near Memory for Data-Intensive Computing," in *DAC*, 2019.
[18] F. Devaux, "The True Processing-In-Memory Accelerator," in *Hot Chips*, 2019.
[19] UPMEM Website, https://www.upmem.com, 2023.
[20] Introduction to UPMEM PIM. Processing-In-Memory (PIM) on DRAM Accelerator (White Paper), 2018.
[21] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware," in *IGSC*, 2021.
[22] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-In-Memory System," *IEEE Access*, vol. 10, 2022.
[23] J. Gómez-Luna, Y. Guo, S. Brocard, J. Legriel, R. Cimadomo, G. F. Oliveira, G. Singh, and O. Mutlu, "Machine Learning Training on a Real Processing-in-Memory System," in *ISVLSI*, 2022.
[24] J. Gómez-Luna, Y. Guo, S. Brocard, J. Legriel, R. Cimadomo, G. F. Oliveira, G. Singh, and O. Mutlu, "Evaluating Machine Learning Workloads on Memory-Centric Computing Systems," in *ISPASS*, 2023.
[25] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures," *ACM POMACS*, vol. 6, 2022.
[26] C. Giannoula, I. Fernandez, J. Gómez-Luna, N. Koziris, G. Goumas, and O. Mutlu, "SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures," *ACM SIGMETRICS PER*, vol. 50, 2022.
[27] Item, Maurus and Gómez-Luna, Juan and Guo, Yuxin and Oliveira, Geraldo F and Sadrosadati, Mohammad and Guo, Yuxin and Mutlu, Onur, "TransPimLib: Efficient Transcendental Functions for Processing-In-Memory Systems," in *ISPASS*, 2023.
[28] J. Chen, J. Gómez-Luna, I. El Hajj, Y. Guo, and O. Mutlu, "SimplePIM: A Software Framework for Productive and Efficient Processing-In-Memory," in *PACT*, 2023.
[29] N. Abecassis, J. Gomez Luna, O. Mutlu, R. Ginosar, A. Moisson-Franckhauser, and L. Yavits, "GAPiM: Discovering Genetic Variations on a Real Processing-in-Memory System," *bioRxiv*, 2023.
[30] S. Diab, A. Nassereldine, M. Alser, J. G. Luna, O. Mutlu, and I. E. Hajj, "High-throughput Pairwise Alignment with the Wavefront Algorithm using Processing-in-Memory," *arXiv preprint arXiv:2204.02085*, 2022.
[31] S. Rhyner, H. Luo, J. Gómez-Luna, M. Sadrosadati, J. Jiang, A. Olgun, H. Gupta, C. Zhang, and O. Mutlu, "Analysis of Distributed Optimization Algorithms on a Real Processing-In-Memory System," *arXiv preprint arXiv:2404.07164*, 2024.
[32] C. Giannoula, P. Yang, I. F. Vega, J. Yang, Y. X. Li, J. G. Luna, M. Sadrosadati, O. Mutlu, and G. Pekhimenko, "Accelerating Graph Neural Networks on Real Processing-In-Memory Systems," *arXiv preprint arXiv:2402.16731*, 2024.
[33] S. Diab, A. Nassereldine, M. Alser, J. Gómez Luna, O. Mutlu, and I. El Hajj, "A framework for high-throughput sequence alignment using real processing-in-memory systems," *Bioinformatics*, vol. 39, 2023.
[34] H. Gupta, M. Kabra, J. Gómez-Luna, K. Kanellopoulos, and O. Mutlu, "Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System," in *IISWC*, 2023.
[35] G. F. Oliveira, J. Gómez-Luna, S. Ghose, A. Boroumand, and O. Mutlu, "Accelerating Neural Network Inference with Processing-in-DRAM: From the Edge to the Cloud," *IEEE Micro*, vol. 42, 2022.
[36] P. Das, P. R. Sutradhar, M. Indovina, S. M. P. Dinakarrao, and A. Ganguly, "Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware," in *SOCC*, 2022.
[37] C. Lim, S. Lee, J. Choi, J. Lee, S. Park, H. Kim, J. Lee, and Y. Kim, "Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs," *ACM PACMMOD*, vol. 1, 2023.
[38] L.-C. Chen, C.-C. Ho, and Y.-H. Chang, "UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification," in *DAC*, 2023.
[39] Y. Wu, Z. Wang, and W. D. Lu, "PIM-GPT: A Hybrid Process-in-Memory Accelerator for Autoregressive Transformers," *arXiv preprint arXiv:2310.09385*, 2023.
[40] C. J. Watkins and P. Dayan, "Q-Learning," *Springer Machine Learning*, vol. 8, 1992.
[41] G. Li, Y. Wei, Y. Chi, Y. Gu, and Y. Chen, "Sample Complexity of Asynchronous Q-Learning: Sharper Analysis and Variance Reduction," *NeurIPS*, vol. 33, 2020.
[42] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI GYM," *arXiv preprint arXiv:1606.01540*, 2016.

[43] Y. Meng, S. Kuppannagari, R. Rajat, A. Srivastava, R. Kannan, and V. Prasanna, "QTAccel: A Generic FPGA based Design for Q-Table based Reinforcement Learning Accelerators," in *IPDPSW*, 2020.

[44] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *ACM CACM*, vol. 52, 2009.

[45] Intel® Advisor User Guide, https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html#gs.150nvx, 2021.

[46] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi *et al.*, "A Case Study of {Processing-In-Memory} in {off-the-Shelf} Systems," in *USENIX ATC*, 2021.

[47] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-In-Memory for Energy-efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.

[48] I. Fernandez, C. Giannoula, A. Manglik, R. Quislant, N. M. Ghiasi, J. Gómez-Luna, E. Gutierrez, O. Plata, and O. Mutlu, "MATSA: An MRAM-based Energy-Efficient Accelerator for Time Series Analysis," *IEEE Access*, 2024.

[49] I. Fernandez, A. Manglik, C. Giannoula, R. Quislant, N. M. Ghiasi, J. Gómez-Luna, E. Gutiérrez, O. Plata, and O. Mutlu, "Accelerating Time Series Analysis via Processing using Non-Volatile Memories," *arXiv preprint arXiv:2211.04369*, 2022.

[50] B. Hyun, T. Kim, D. Lee, and M. Rhu, "Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology," in *HPCA*, 2024.

[51] A. Bernhardt, A. Koch, and I. Petrov, "pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories," in *DaMoN*, 2023.

[52] S. U. Noh, J. Hong, C. Lim, S. Park, J. Kim, H. Kim, Y. Kim, and J. Lee, "PID-Comm: A Fast and Flexible Collective Communication Framework for Commodity Processing-in-DIMM Devices," *arXiv preprint arXiv:2404.08871*, 2024.

[53] A. A. Khan, H. Farzaneh, K. F. Friebel, C. Fournier, L. Chelini, and J. Castrillon, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms," *arXiv preprint arXiv:2301.07486*, 2022.

[54] V. Zois, D. Gupta, V. J. Tsotras, W. A. Najjar, and J.-F. Roy, "Massively Parallel Skyline Computation For Processing-In-Memory Architectures," in *PACT*, 2018.

[55] H. Kang, Y. Zhao, G. E. Blelloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons, "PIM-trie: A Skew-resistant Trie for Processing-in-Memory," in *SPAA*, 2023.

[56] Y. Falevoz and J. Legriel, "Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture," in *Euro-Par*, 2023.

[57] A. Lopes, D. Castro, and P. Romano, "PIM-STM: Software Transactional Memory for Processing-In-Memory Systems," *arXiv preprint arXiv:2401.09281*, 2024.

[58] M. Mognol, D. Lavenier, and J. Legriel, "Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment."

[59] D. Kim, T. Kim, I. Hwang, T. Park, H. Kim, Y. Kim, and Y. Park, "Virtual PIM: Resource-Aware Dynamic DPU Allocation and Workload Scheduling Framework for Multi-DPU PIM Architecture," in *PACT*, 2023.

[60] J. Nider, J. Dagger, N. Gharavi, D. Ng, and A. Fedorova, "Bulk JPEG Decoding on In-Memory Processors," in *SYSTOR*, 2022.

[61] J. Nider, C. Mustard, A. Zoltan, and A. Fedorova, "Processing in Storage Class Memory," in *HotStorage*, 2020.

[62] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Adaptive Query Compilation with Processing-in-Memory," in *ICDEW*, 2023.

[63] O. Ferraz, Y. Falevoz, V. Silva, and G. Falcao, "Unlocking the Potential of LDPC Decoders with PiM Acceleration," in *ACSSC*, 2023.

[64] D. Lavenier, C. Deltel, D. Furodet, and J.-F. Roy, "BLAST on UPMEM," Ph.D. dissertation, INRIA Rennes-Bretagne Atlantique, 2016.

[65] R. Ma, S. Zheng, G. Wang, J. Pu, Y. Hua, W. Wang, and L. Huang, "Accelerating Regular Path Queries over Graph Database with Processing-in-Memory," *arXiv preprint arXiv:2403.10051*, 2024.

[66] A. A. Khan, J. P. C. De Lima, H. Farzaneh, and J. Castrillon, "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview," *arXiv preprint arXiv:2401.14428*, 2024.

[67] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim *et al.*, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *ISSCC*, 2021.

[68] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "Hardware Architecture and Software Stack for PIM based on Commercial DRAM technology: Industrial Product," in *ISCA*, 2021.

[69] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, vol. 42, 2021.

[70] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, "A 1ynm 1.25 V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *ISSCC*, 2022.

[71] D. Niu, S. Li, Y. Wang, W. Han, Z. Zhang, Y. Guan, T. Guan, F. Sun, F. Xue, L. Duan *et al.*, "184QPS/W 64Mb/mm 2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System," in *ISSCC*, 2022.

[72] O. Mutlu, "Lecture 20: Graphics Processing Units," video recording available at http://www.youtube.com/watch?v=dg0VN-XCGKQ, presentation available at: https://safari.ethz.ch/digitaltechnik/spring2020/lib/exe/fetch.php?media=onur-digitaldesign-2020-lecture20-gpu-beforelecture.pdf, Digital Design and Computer Architecture, ETH Zürich, 2020.

[73] "UPMEM User Manual. Version 2023.1.0,", 2023.

[74] UPMEM, "UPMEM Software Development Kit (SDK).", https://sdk.upmem.com, 2023.

[75] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[76] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *Advanced computer architecture*, 1986.

[77] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *Real-Time signal processing IV*, vol. 298, 1982.

[78] D. Lee, B. Hyun, T. Kim, and M. Rhu, "Analysis of Data Transfer Bottlenecks in Commercial PIM Systems: A Study with UPMEM-PIM," *IEEE CAL*, 2024.

[79] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013.

[80] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "StarCraft II: A New Challenge for Reinforcement Learning," *arXiv preprint arXiv:1708.04782*, 2017.

[81] Y. Mei, H. Zhou, T. Lan, G. Venkataramani, and P. Wei, "MAC-PO: Multi-Agent Experience Replay via Collective Priority Optimization," in *AAMAS*, 2023.

[82] K. Gogineni, Y. Mei, T. Lan, P. Wei, and G. Venkataramani, "AccMER: Accelerating Multi-Agent Experience Replay with Cache Locality-aware Prioritization," in *ASAP*, 2023.

[83] K. Gogineni, P. Wei, T. Lan, and G. Venkataramani, "Towards Efficient Multi-Agent Learning Systems," *MLArchSys, ISCA*, 2023.

[84] K. Gogineni, P. Wei, T. Lan, and G. Venkataramani, "Scalability Bottlenecks in Multi-Agent Reinforcement Learning Systems," *FastPath, ISPASS*, 2023.

[85] A. Perera and P. Kamalaruban, "Applications of Reinforcement Learning in Energy Systems," *Elsevier Renew. Sustain. Energy Rev.*, 2021.

[86] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," *IEEE Access*, vol. 7, 2019.

[87] J. Peng and R. J. Williams, "Incremental Multi-Step Q-Learning," in *Elsevier Machine Learning Proceedings*, 1994.

[88] J. Sharma, P.-A. Andersen, O.-C. Granmo, and M. Goodwin, "Deep Q-Learning With Q-Matrix Transfer Learning for Novel Fire Evacuation Environment," *IEEE SMC: Systems*, vol. 51, 2020.

[89] S. Spano, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, "An Efficient Hardware Implementation of Reinforcement Learning: The Q-Learning Algorithm," *IEEE Access*, vol. 7, 2019.

[90] A. Applebaum, C. Dennler, P. Dwyer, M. Moskowitz, H. Nguyen, N. Nichols, N. Park, P. Rachwalski, F. Rau, A. Webster *et al.*, "Bridging Automated to Autonomous Cyber Defense: Foundational Analysis of Tabular Q-Learning," in *AISec*, 2022.

[91] T. J. Perkins and M. D. Pendrith, "On the existence of fixed points for q-learning and sarsa in partially observable domains," in *ICML*, 2002.

[92] G. A. Rummery and M. Niranjan, *On-line Q-Learning Using Connectionist Systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[93] S. Shresthamali, M. Kondo, and H. Nakamura, "Adaptive Power Management in Solar Energy Harvesting Sensor Node Using Reinforcement Learning," *ACM TECS*, vol. 16, 2017.

[94] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning," in *MICRO*, 2021.

[95] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," *ACM SIGARCH Computer Architecture News*, 2008.

[96] S. M. PD, H. Yu, H. Huang, and D. Xu, "A Q-Learning Based Self-Adaptive I/O Communication for 2.5D Integrated Many-Core Microprocessor and Memory," *IEEE TC*, 2015.

[97] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, Energy-efficient, Fault-tolerant Network-on-Chip Design Using Reinforcement Learning," in *DATE*, 2019.

[98] G. Singh, R. Nadig, J. Park, R. Bera, N. Hajinazar, D. Novo, J. Gómez-Luna, S. Stuijk, H. Corporaal, and O. Mutlu, "Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning," in *ISCA*, 2022.

[99] J. Langford and T. Zhang, "The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits," *NeurIPS*, vol. 20, 2007.

[100] I. Caspi, G. Leibovich, G. Novik, and S. Endrawis, "Reinforcement Learning Coach," 2017.

[101] C. S. De Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson, "Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge?" *arXiv preprint arXiv:2011.09533*, 2020.

[102] M. Zhou, J. Luo, J. Villella, Y. Yang, D. Rusu, J. Miao, W. Zhang, M. Alban, I. Fadakar, Z. Chen *et al.*, "SMARTS: Scalable Multi-Agent Reinforcement Learning Training School for Autonomous Driving," in *CoRL*, 2021.

[103] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving," *arXiv preprint arXiv:1610.03295*, 2016.

[104] P. Palanisamy, "Multi-Agent Connected Autonomous Driving using Deep Reinforcement Learning," in *IJCNN*, 2020.

[105] S. Bhalla, S. G. Subramanian, and M. Crowley, "Training Cooperative Agents for Multi-Agent Reinforcement Learning," in *AAMAS*, 2019.

[106] A. Ozdaglar, M. O. Sayin, and K. Zhang, "Independent Learning in Stochastic Games," in *International Congress of Mathematicians*, 2021.

[107] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE T-ITS*, 2021.

[108] J. Yang, A. Nakhaei, D. Isele, K. Fujimura, and H. Zha, "CM3: Cooperative Multi-goal Multi-Stage Multi-Agent Reinforcement Learning," *arXiv preprint arXiv:1809.05188*, 2018.

[109] L. Matignon, G. J. Laurent, and N. Le Fort-Piat, "Independent reinforcement learners in cooperative markov games: A survey regarding coordination problems," *The Knowledge Engineering Review*, 2012.

[110] S. Gu, L. Yang, Y. Du, G. Chen, F. Walter, J. Wang, Y. Yang, and A. Knoll, "A Review of Safe Reinforcement Learning: Methods, Theory and Applications," *arXiv preprint arXiv:2205.10330*, 2022.

[111] M. Tan, "Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents," in *ICML*, 1993.

[112] P. L'Ecuer and F. Blouin, "Linear Congruential Generators of Order K>1," in *WSC*, 1988.

[113] I. S. P. (formerly Skylake), "Xeon® Silver 4110 Processor," Website: https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache-2-10-ghz.html.

[114] "NVIDIA Ampere Architecture," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[115] J. Woo, G. Joshi, and Y. Chi, "The Blessing of Heterogeneity in Federated Q-learning: Linear Speedup and Beyond," *arXiv preprint arXiv:2305.10697*, 2023.

[116] J. Qi, Q. Zhou, L. Lei, and K. Zheng, "Federated Reinforcement Learning: Techniques, Applications, and Open Challenges," *arXiv preprint arXiv:2108.11887*, 2021.

[117] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *ICML*, 2016.

[118] B. Li, Y. Wang, and Y. Chen, "HitM: High-Throughput ReRAM-based PIM for Multi-Modal Neural Networks," in *ICCAD*, 2020.

[119] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets," *IEEE TC*, vol. 68, 2018.

[120] Y. Luo and S. Yu, "Benchmark Non-volatile and Volatile Memory Based Hybrid Precision Synapses for In-situ Deep Neural Network Training," in *ASP-DAC*, 2020.

[121] H. Sun, Z. Zhu, Y. Cai, X. Chen, Y. Wang, and H. Yang, "An Energy-Efficient Quantized and Regularized Training Framework For Processing-In-Memory Accelerators," in *ASP-DAC*, 2020.

[122] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning," in *MICRO*, 2020.

[123] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: a DRAM based Accelerator for Accurate CNN Inference," in *DAC*, 2018.

[124] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *APSLOS*, 2017.

[125] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision," in *ISCA*, 2019.

[126] D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G.-M. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, V. Kornijcuk, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Lee, D. Ko, Y. Jun, I. Kim, C. Song, I. Kim, C. Park, S. Kim, C. Jeong, E. Lim, D. Kim, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep Learning Application," *IEEE JSSC*, vol. 58, 2023.

[127] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-memory Analytics Frameworks," in *PACT*, 2015.

[128] J. Vieira, N. Roma, P. Tomás, P. Ienne, and G. Falcao, "Exploiting Compute Caches for Memory Bound Vector Operations," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018.

[129] H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, "ORIGAMI: A Heterogeneous Split Architecture for In-Memory Acceleration of Learning," *arXiv preprint arXiv:1812.11473*, 2018.

[130] S. Bavikadi, P. R. Sutradhar, K. N. Khasawneh, A. Ganguly, and S. M. Pudukotai Dinakarrao, "A Review of In-Memory Computing Architectures for Machine Learning Applications," in *GLSVLSI*, 2020.

[131] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V. J. Reddi, P. R. Sutradhar, A. Joseph, and S. M. Pudukotai Dinakarrao, "A Survey on Machine Learning Accelerators and Evolutionary Hardware Platforms," *IEEE Des. Test Comput.*, vol. 39, 2022.

[132] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *Elsevier JSA*, 2022.

[133] N. Zarif, "Offloading embedding lookups to processing-in-memory for deep learning recommender models," Ph.D. dissertation, 2023. [Online]. Available: https://open.library.ubc.ca/collections/ubctheses/24/items/1.0435518

[134] S. Y. Kim, J. Lee, Y. Paik, C. H. Kim, W. J. Lee, and S. W. Kim, "Optimal Model Partitioning with Low-Overhead Profiling on the PIM-based Platform for Deep Learning Inference," *ACM TODAES*, 2024.

[135] D. Lavenier, R. Cimadomo, and R. Jodin, "Variant Calling Parallelization on Processor-in-Memory Architecture," in *BIBM*, 2020.

[136] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Accelerating Large Table Scan Using Processing-In-Memory Technology," *Springer Datenbank-Spektrum*, 2023.

[137] G. Jonatan, H. Cho, H. Son, X. Wu, N. Livesay, E. Mora, K. Shivdikar, J. L. Abellán, A. Joshi, D. Kaeli *et al.*, "Scalability Limitations of Processing-in-Memory using Real System Evaluations," *ACM POMACS*, 2024.

[138] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "GA3C: GPU-based A3C for Deep Reinforcement Learning," *ICLR*, 2017.

[139] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-Accelerated Deep Reinforcement Learning," in *ASPLOS*, 2019.

[140] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating Distributed Reinforcement Learning with In-Switch Computing," in *ISCA*, 2019.

[141] M. W. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, N. Momchev, D. Sinopalnikov, P. Stańczyk, S. Ramos, A. Raichuk, D. Vincent *et al.*, "ACME: A Research Framework for Distributed Reinforcement Learning," *arXiv preprint arXiv:2006.00979*, 2020.

[142] A. Stooke and P. Abbeel, "Accelerated Methods for Deep Reinforcement Learning," *arXiv preprint arXiv:1803.02811*, 2018.

[143] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient Parallel Methods for Deep Reinforcement Learning," *arXiv preprint arXiv:1705.04862*, 2017.

[144] B. Wang, J. Xie, and N. Atanasov, "DARL1N: Distributed Multi-Agent Reinforcement Learning with One-hop Neighbors," *CoRR abs/2202.09019*, 2022.

[145] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision," in *CVPR*, 2019.

[146] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *JMLR*, vol. 18, 2017.

[147] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference," in *DAC*, 2020.

[148] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving Neural Network Quantization without Retraining using Outlier Channel Splitting," in *ICML*, 2019.

[149] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed Precision Training," *arXiv preprint arXiv:1710.03740*, 2017.

[150] J. Björck, X. Chen, C. De Sa, C. P. Gomes, and K. Weinberger, "Low-Precision Reinforcement Learning: Running Soft Actor-Critic in Half Precision," in *ICML*, 2021.

[151] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-maron, A. Faust, and V. J. Reddi, "QuaRL: Quantization for Fast and Environmentally Sustainable Reinforcement Learning," *TMLR*, 2022.