Characterizing and Optimizing the End-to-End Performance of Multi-Agent Reinforcement Learning Systems

Kailash Gogineni*, Yongsheng Mei*, Karthikeya Gogineni†, Peng Wei*, Tian Lan*, Guru Venkataramani*
*George Washington University, USA †Independent

***Abstract*— Multi-Agent Reinforcement Learning Systems (MARL) can unlock the potential to model and control multiple autonomous decision-making agents simultaneously. During online training, MARL algorithms involve performance-intensive computations, such as exploration and exploitation phases originating from a large observation-action space and a huge number of training steps. Understanding and mitigating the MARL performance limiters is key to their practical adoption.**

In this paper, we first present a detailed workload characterization of MARL workloads under different multi-agent settings. Our experimental analysis identifies a critical performance bottleneck that affects scaling within the mini-batch sampling on transition data. To mitigate this issue, we explore a series of optimization strategies. First, we investigate cache locality-aware sampling that prioritizes intra-agent neighbor transitions over other randomly picked transition data samples within the baseline MARL algorithms. Next, we explore importance sampling techniques that preserve the learning performance/distribution and capture the neighbors of important transitions. Finally, we design an additional algorithmic optimization that reorganizes the transition data layout to improve the cache locality between different agents during the mini-batch sampling process.

We evaluate our optimizations using popular MARL workloads on multi-agent particle games. Our work highlights several opportunities for enhancing the performance of multi-agent systems, with end-to-end training time improvements ranging from 8.2% (3 agents) to 20.5% (24 agents) compared to the baseline MADDPG, affirming the usefulness of deeply understanding MARL performance bottlenecks and mitigating them effectively.

***Index Terms*—Multi-Agent Systems, Performance Analysis, Reinforcement Learning, Performance Optimization**

I. Introduction

Multi-Agent Reinforcement Learning (MARL) algorithms are a new and evolving class of algorithms that can tackle real-world problems involving complex inter-agent interactions and decision-making in a shared environment. These settings are vastly different from single-agent RL scenarios and involve joint state-action spaces with multiple agents either cooperatively or competitively engaging with other agents in a shared environment. Consequently, as the number of agents grows, the state-action space expands exponentially, presenting unique computational complexities [1]. Applications of MARL span various domains, such as coordinating robot fleets [2], [3], transportation management [4], [5], game AI [6], [7], autonomous driving, recommendation systems, large language models, aviation, and search-and-rescue drone missions, to name a few [8]–[14].

In MARL [15], [16], individual agents optimize their actions and interactions with the environment. They decide their actions based on the current observations and evaluate state-action pairs using reward functions. The function determining the action is known as a policy, and the agents seek to find optimal policies that maximize their total accumulative (discounted) rewards. The function representing the reward estimates is known as the value function.

In the context of large-scale MARL involving approximately 50 agents working on a competitive task, our experiments show that it takes up to *seven days* to train to reach 1 million time steps and achieve optimal policy, even on the latest high-performance Nvidia Ampere architecture [17]. This severely limits the practical use of MARL in real-world systems and poses significant *computational challenges*, especially as the number of agents scales higher.

Multi-agent learning workloads typically follow the Centralized Training Decentralized Execution (CTDE) model and consist of two main stages: *action selection*, which efficiently runs on the GPU due to its parallel processing demands, and *update all trainers*, where the sampling phase, characterized by irregular accesses, is CPU-bound, while the actor-critic network updates are GPU-bound. In the action selection stage, as depicted in Figure 1, each agent in the environment has its own actor network (AN), which generates actions based on its current observation. Agents simultaneously execute these actions in a shared environment and receive rewards and the next state as the output. The experiences of each agent are stored in a replay buffer, which is essential for back-propagation. During the *mini-batch sampling* phase, each agent collects historical transition data of all other agents stored in the experience replay buffer. This sampling approach allows the algorithm to reuse transition data for updating the current policy. During training, each agent has a centralized critic that calculates Q-values (based on global information) to update the decentralized actors using the joint observation-action space of all agents. In the update all trainers phase, both the actor and critic networks are updated following the target Q calculation and the mini-batch sampling phase.

As the number of agents grows linearly, our performance analysis of key MARL workloads (MADDPG [18] and MATD3 [19]) on the recent Nvidia RTX 3090, CPU-GPU architecture [17] demonstrated a prominent trend: the update all trainers phase dominates the overall MARL performance (up

to ≈80%), surpassing all other phases significantly (Figure 2). This key finding prompted us to conduct a more granular analysis, breaking down the update all trainers phase into its constituent modules: mini-batch sampling, target Q calculation, and Q loss - P loss. In our analysis, the sampling phase was found to be the dominating component of the overall MARL training times. (Figure 3).

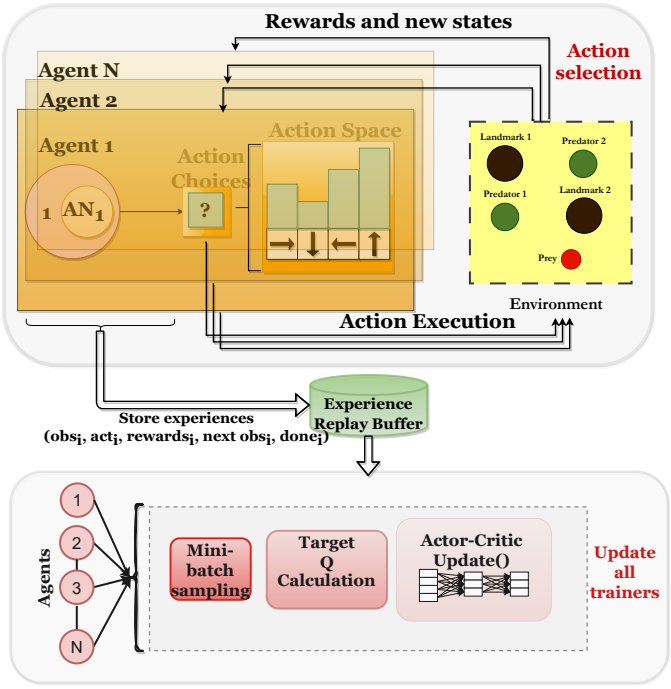


Fig. 1: MARL architecture and its typical implementation using a decentralized actor and a centralized critic.

To address the MARL performance bottlenecks, we delve into various avenues for performance optimization. First, a cache locality-aware sampling accelerates transition data access latencies by including the neighboring intra-agent transitions for the training phase. Second, we explore an information-prioritized locality-aware sampling that emphasizes certain information-rich transition data indices to maintain algorithmic learning performance. Third, we also explore data layout reorganization in the experience replay buffer to accelerate the sampling process between the agents. This technique involves rearranging the transition data of all agents in a locality-aware order to achieve better performance.

In summary, our paper makes the following contributions:

- We systematically study the performance profiles of training phases within two *state-of-the-art* multi-agent reinforcement learning systems (MADDPG and MATD3) using two multi-agent particle environments with 3 to 24 agents on Nvidia Ampere Architecture (RTX 3090) [17]. For the *first* time, our performance analysis presents key insights into the computational bottlenecks confronting several MARL algorithms from a systems perspective.
- We present performance enhancement opportunities for hardware-level and algorithmic optimizations to improve the runtime of a key performance limiter within MARL, namely the mini-batch sampling phase. Our proposed optimizations include: ① Cache locality-aware neighbor data sampling to improve intra-agent memory accesses, ② Information prioritized cache locality-aware sampling to improve the sampling distribution, and ③ Transition data layout reorganization to improve the inter-agent cache locality.
- Our experimental results demonstrate end-to-end training time acceleration (1.2× for 24 agents), with performance improvements ranging from 8.2% (3 agents) to 20.5% (24 agents) compared to the baseline MARL workload - MADDPG, and attains an average 2× faster sampling compared to existing prioritization approaches (PER-MADDPG) while preserving the mean scores in both cooperative and competitive tasks. Finally, our transition data layout reorganization strategy shows promise of a steadily rising trend (from a slowdown of 37% for 3 agents to a speedup of 25.84% for 24 agents) in the sampling phase for a predator-prey environment for the MADDPG algorithm.

II. Background

In this section, we introduce our MARL workloads and multi-agent particle environments [18].

A. Multi-Agent Reinforcement Learning

Typically, MARL settings with N agents is defined by a set of states, $S = S_1 \times ... \times S_N$, a set of actions $A = A_1 \times ... \times A_N$. Each agent selects its action by using a policy $\pi_{\theta_i} : O_i \times A_i \rightarrow [0, 1]$. The state transition ($T : S \times A_1 \times A_2 \times ... \times A_N$) function produces the next state $S^{'}$, given the current state and actions for each agent. The reward, $R_i : S \times A_i \rightarrow \mathbb{R}$ for each agent is a function of global state and action of *all other agents*, with the aim of maximizing its own expected return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where γ denotes the discount factor and T is the time horizon.

MADDPG. In MADDPG [18], each agent learns an individual policy that maps the observation to its action to maximize the expected return, which is approximated by the critic. MADDPG lets the critic of agent i to be trained by minimizing the loss with the target Q-value and y_i using $\mathcal{L}(\theta_i) = \mathbb{E}_D[(Q_i(S, A_1, ...A_n) - y_i^2]$, and $y_i = r_i + \gamma \overline{Q}_i(S^{'}, A_1^{'}, ...A_n^{'})_{a_j^{'}=\overline{\pi}(o_j^{'})}$, where S and $A_1, ...A_n$ represent the joint observations and actions respectively. D is the experience replay buffer that stores the samples of *observations, actions, rewards, and new observations* for all the agents obtained after the training episodes. The MARL framework has four networks- actor, critic, target actor, and target critic. $\overline{Q}_i$ and $\overline{\pi}(o_j^{'})$ are the target networks for the stable learning of critic (Q_i) and actor networks. The target actor estimates the next action from the policy using the state output by the actor network. The target critic aggregates the output from the target actor to compute the target Q-values, which update the critic network and assess the quality of the actions taken by agents. The target networks help achieve training stability.

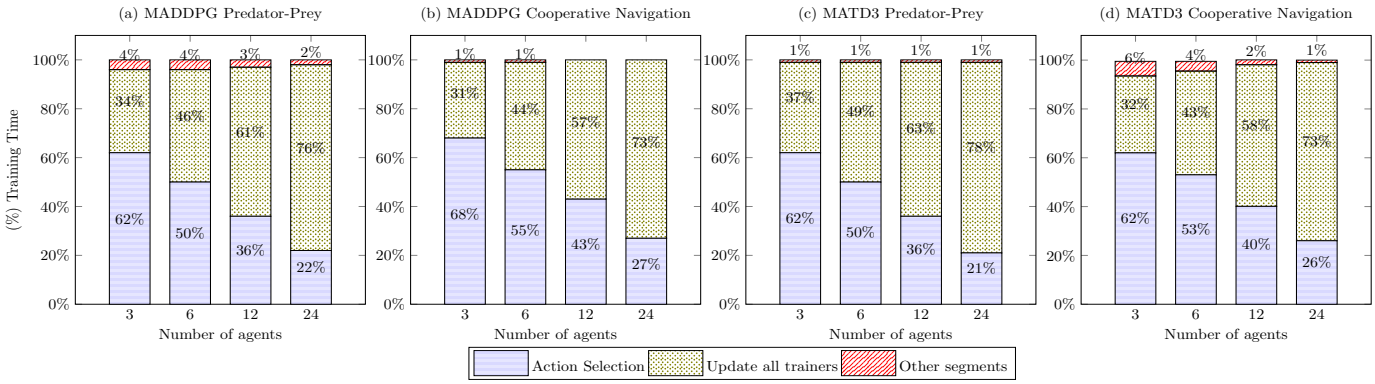


Fig. 2: End-to-end training time percentage breakdown for two MARL workloads, both involving 3 to 24 agents for Predator-Prey and Cooperative Navigation multi-agent particle games. The total training times are detailed in Table I.

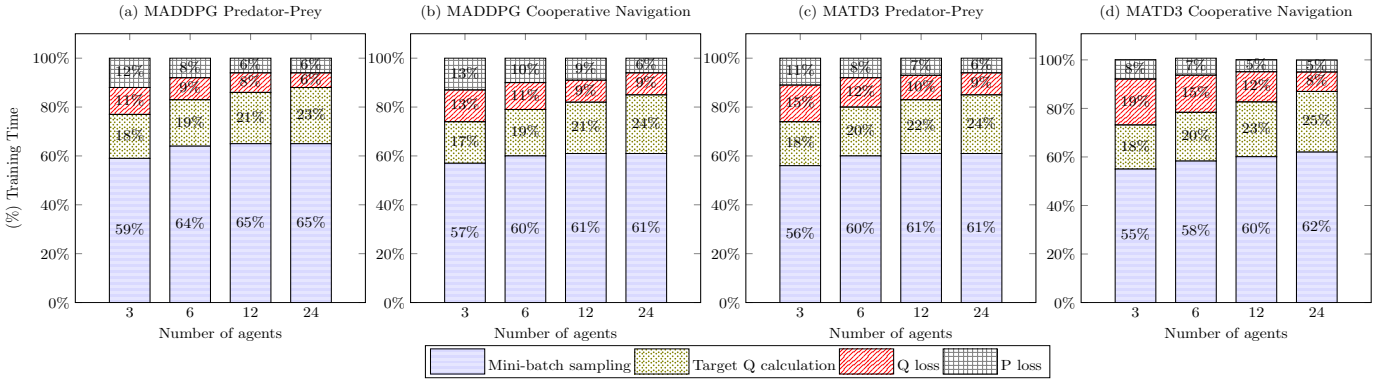


Fig. 3: Training time breakdown within update all trainers on two different MARL workloads with 3 to 24 agents for Predator-Prey and Cooperative Navigation multi-agent particle games.

MATD3. MATD3 [19] uses the twin delayed critics to tackle the over-estimation bias problem [19] and incorporates small amounts of noise to the actions sampled from the buffer. As the change of critic values needs to be reflected in the policies of other agents, MATD3 employs delayed policy updates for target networks and the policies to obtain an accurate critic before using it to update the actor network.

MADDPG and MATD3 find utility in a variety of domains, such as UAV systems, distributed control, robotic teams, and automated trading, as highlighted by several studies [16], [20]–[24].

B. Multi-Agent Player Games

In many practical multi-agent scenarios, several agents simultaneously explore a common environment and perform competitive (e.g., predator-prey), cooperative (e.g., cooperative navigation), and mixed tasks [16], [18]. In cooperative setups, all agents share observations, and training is conducted centrally. In contrast, each agent aims to outperform its adversaries in competitive settings.

We explore a range of 2D tasks involving agents in cooperative and competitive scenarios [18]. Agents interact with landmarks and other agents in a shared environment to achieve various goals. The observation space of the agents is a high-level feature vector containing relative distances to other agents, along with additional information like communication and velocity [18]. For decision-making, agents have discrete action space and typically include five actions corresponding

TABLE I: End-to-end training times for MADDPG and MATD3 with varying numbers of agents trained for 60,000 episodes in Predator-Prey and Cooperative Navigation tasks.

Environment	Algorithm	Training Time (sec)
Predator-Prey	**MADDPG**	
	3 Agents	3365.99
	6 Agents	8504.99
	12 Agents	23406.16
	24 Agents	82768.15
	MATD3	
	3 Agents	3838.97
	6 Agents	9039.11
	12 Agents	24678.43
	24 Agents	80123.24
Cooperative Navigation	**MADDPG**	
	3 Agents	2403.64
	6 Agents	5888.64
	12 Agents	15722.43
	24 Agents	52421.81
	MATD3	
	3 Agents	2785.53
	6 Agents	6369.42
	12 Agents	17081.71
	24 Agents	55371.91

to *static, move right, move left, move up or down*.

We chose predator-prey tasks (competitive) and cooperative navigation tasks (cooperative) to conduct a comprehensive workload characterization and validate the effectiveness of our optimization techniques [18] and emphasize variations in training time complexity (predator-prey tasks take approximately $1.5\times$ longer training time compared to cooperative tasks). In predator-prey tasks, *N* predators work cooperatively to block the way of *M* fast-paced prey agents. The prey agents are environment-controlled and try to avoid collisions with predators. On the other hand, in cooperative navigation tasks, *N* agents work together to reach *L* landmarks, and the rewards encourage the agents to get closer to the landmarks in these settings.

For the predator-prey environment, when there are 3 agents, the observation spaces are as follows: Agent 1 (Predator) has an observation space of `Box(16,)`, agent 2 (Predator) has `Box(16,)`, agent 3 (Predator) has `Box(16,)`, and agent 4 (Prey) has `Box(14,)`. For example, `Box(16,)` would represent a 16-dimensional continuous space containing floating-point values. This flexibility allows agents to have a fine-grained perception of the environment or to make precise and nuanced decisions. In a larger-scale scenario with 24 agents, each agent (Predator) has a `Box(98,)` observation space, except for agents 25 to 32 (Preys), which have `Box(96,)`.

Moving on to the cooperative navigation environment, for 3 agents, each agent's observation space is `Box(18,)`, and the number of available actions is 5. With 6 agents, each agent has `Box(36,)` as their observation space. In a scenario with 12 agents, the observation space for each agent is `Box(72,)`, and in a 24-agent setup, each agent has an observation space of `Box(144,)`.

III. MARL Performance Profile

Our performance analysis[1] shows super-linear trend in various performance metrics (besides total training time): total instructions increase by $3-4\times$, cache misses by $2.5-4.5\times$, and dTLB load misses by $3-4\times$ on average for both cooperative and competitive games (Figure 4). As discussed in Section II-B, the predator-prey tasks exert a much stronger influence than cooperative tasks due to collaboration between the predator agents. They collaborate to maximize their shared return and capture the prey.

Specifically, within mini-batch sampling (the largest time consumer), every agent samples a set of mini-batch samples uniformly from the replay buffers of other agents and then updates its critic network. Each agent performs *lookup-read-write* operations, and this process scales with the number of agents, denoted as N, and is repeated for all N agents. As a result, the time complexity to collect the transition set is $O(N^2B)$, where B represents the batch size. From Figure 4, we note that a similar computational bottleneck is observed in cooperative scenarios where all the agents are trained

[1]We omit the *environment interactions* phase for the characterization study since it primarily depends on task complexity.

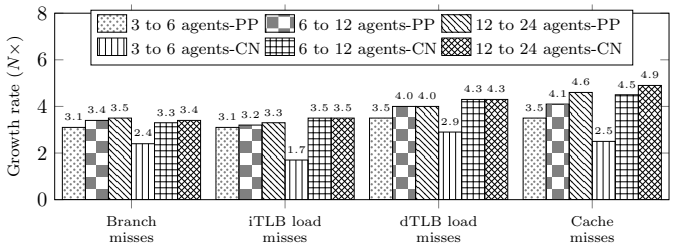


Fig. 4: Hardware Performance analysis of update all trainers averaged across two MARL workloads (MADDPG & MATD3) as the number of agents increases linearly. These workloads are trained using the Predator-Prey (PP) and Cooperative Navigation (CN) environments.

collectively to reach landmarks while avoiding collisions with each other.

Figure 5 illustrates the simplified view of the mini-batch sampling phase where each agent selects a batch of transitions (size=1024) from all other agent's replay buffers. Each agent trainer iterates through different agent IDs. The indices array maps the random reference points for each agent ID to retrieve transitions from the replay buffer. The model parameters in the multi-agent setting determine the number of samples and agents.

To analyze the memory access behavior, we profiled the training time growth of the sampling phase. As expected, we found that as the number of agents increases linearly, the run-time grows by approximately $3\times$. A careful examination led us to the conclusion that the sampling phase involves only one level of indirection (as shown in Figure 5), achieved by mapping the indices array reference points to the replay buffer storage locations. But, it is also worth noting that the actor and critic networks are periodically updated, causing the sampling stage to be called millions of times (iterations), which puts pressure on the cache bandwidth and capacity. This pressure arises because, for each iteration, the indices array dynamically changes to explore a broad state-action space, which can result in highly irregular memory accesses.

The target Q calculation phase is *second* largest time-consuming phase within update all trainers. Note that, in Figure 3, the computation time as a percentage within update all trainers increases with the number of agents for target Q, whereas the run-time proportion of Q loss - P loss decreases slightly.

Each agent performs the next action calculation, target Q next, and target Q values as a function of all other agents' joint observation-action space. To calculate the next action, the agent i uses its policy network to determine the next action-a' from the next state-S'. In this phase, each agent's policy network involves multiplication/addition operations with input-weight matrix resulting in performance impact. The obtained a' and S' data are aggregated and concatenated into a single vector in order to compute the target Q next amongst the cooperating agents. The input space (dimension) for the Q-

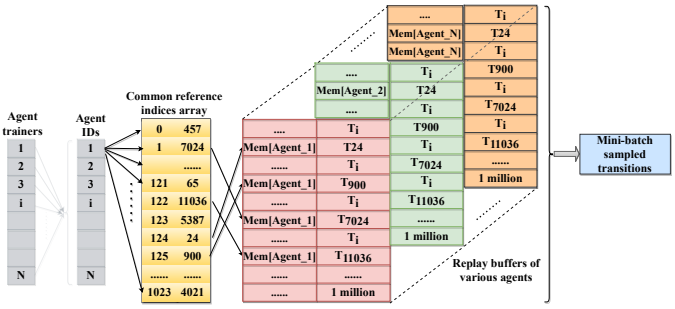


Fig. 5: Illustration of mini-batch sampling phase, where 1024 (batch size) transitions are captured for each agent as a function of all other agents. This process involves random memory accesses, where the reference points from the common indices array are used to retrieve transitions from specific memory locations in the replay buffer storage, with a maximum size of 1 million.

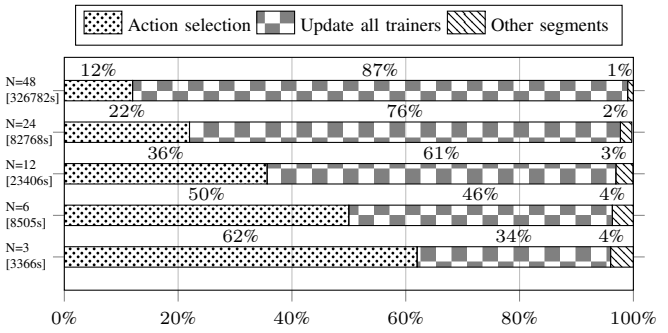


Fig. 6: The breakdown of training time for the MADDPG workload is presented, spanning from 3 to 48 agents in a Competitive environment, specifically Predator-Prey. The total training time of MARL algorithms, depicted in seconds, is displayed on the y-axis within square brackets.

function increases quadratically with the number of agents [1]. The target critic values for each agent i are computed using target Q next values from the target actor network. We note that each agent has to read the other agent's policy values; as such, for N agents, there are $N \times (N-1)$ memory lookup operations corresponding to the next action-a'.

Backpropagation is the *third* largest phase of update all trainers. This phase is dominated by the back-propagation of the critic network that computes the mean-squared error loss between the target critic and critic networks, and the actor network is updated by minimizing the Q values (critic network). As the number of agents increases, the main challenge is the trainable parameters increase, and *N* policy and *N* critic networks are built for all *N* agents, which incurs extra time to update the weights for each agent.

A. Scalability Tests

In our study, we aimed to observe the update all trainers trend in MADDPG when using the predator-prey environment. We conducted tests on the Nvidia Ampere Architecture [17] to assess scalability and profiled the training time. Scalability is crucial as it allows systems to handle increased workloads efficiently, ensuring optimal performance and adaptability in various scenarios. In applications like robotics, scalability enhances operational efficiency and reliability, enabling simultaneous task execution and obstacle management. In Figure 6, we notice an exponential increase in the total training time to *3.5 days for 48 agents; 7 days including environment interactions* (shown on the y-axis). Also, the overall contribution from *update all trainers* modules ranges from 34% to 87% as we increase the number of agents. This growth rate is especially noteworthy in MARL with a larger number of agents, primarily due to the expanding size of the observation-action space [25], [26]. For instance, for a single experience tuple from a 3-agent setup with an observation space of `[Box(16,), Box(16,), Box(16,), Box(14,)]` in the predator-prey environment, upgrading to a 24-agent configuration results in an approximate sixfold expansion ($6\times$) of the observation space. Conversely, in a cooperative setting, where agents cooperate instead of competing, the observation space expands by a factor of eight ($8\times$). Contrary to cooperative tasks, in the predator-prey task, agents must balance cooperation and competition to optimize their collective performance with predator-prey agents, emphasizing scalability issues with the increase in the number of agents.

B. Our Key Findings

In summary, we make the following key observations from MARL performance characterization:

1) As the number of agents increases, the overall MARL training time for updating all trainers grows in a super-linear fashion primarily due to the increasing observation space and amount of interactions with all the neighboring agents in a shared environment.
2) The transition data sampling phase dominates the overall training time of the MARL training, which is largely influenced by irregularity in memory access patterns on account of the index values within the mini-batch.
3) For the sampling phase, the number of cache misses on average grows by more than $3\times$ (approx.) for both the cooperative and competitive scenarios, and that could vary significantly based on lookup patterns as each agent has to gather all other agents' transition data. Cache misses are indicative of the working set sizes, and they become particularly relevant in large-scale multi-agent models involving 12 or more agents. In such cases, the sampling phase entails significant data movement and gathering operations.
4) As the number of agents increases from 3 to 24, the dimension of Q function and target Q also grows exponentially due to the significant increase in the size of observation space (number of float values).

IV. Opportunities for MARL Performance Optimization

This section explores several opportunities for optimizing the key performance bottleneck stemming from the mini-batch sampling phase identified in our performance profiling studies (Section III). First, we customize the sampling process to streamline the address fetch patterns and guide the hardware prefetcher to improve its efficiency. Second, we present a new optimization strategy called information-prioritized locality-aware sampling. In contrast to randomly selecting transitions, this method chooses neighbors from the replay buffer associated with high-priority transitions to improve the learning efficiency. Third, we reconsider how we store transitions from all agents since it can directly affect the computation time spent on the sampling phase. To tackle this issue, we explore an algorithmic optimization - data layout reorganization.

A. Intra-Agent Cache-aware Mini-batch Sampling

The default random mini-batch sampling used in MARL algorithms incurs high training time because each agent must gather random transitions from other agents in order to update their critic and actor networks.

The sampling stage has the index buffer, which stores the lookup indices for each agent's replay buffer. Due to the difficulty in predicting these random memory addresses ahead of time, the memory requests in the mini-batch sampling phase encounter load misses for every reference point in the index array, with cache prefetchers having little ability to predict these indices ahead of time. Therefore, there is a need to design cache-aware sampling strategies that could assist the hardware prefetcher in reducing expensive trips up and down the memory hierarchy.

Algorithm 1 demonstrates the intra-agent cache locality-aware sampling approach. We modify the sampling phase implementation of MARL workloads. Initially, every agent trainer gathers a list of random reference points, MB_idx (1024 random reference points are uniformly sampled based on the replay buffer storage size). For each agent, our intra-agent cache-aware approach selects a reference point (idx) and then accesses all transitions from idx to $idx + neighbors$. This process involves retrieving the transitions within that range from the replay buffer storage and obtaining the corresponding output data.

Algorithm 1: Cache locality-aware sampling

Input: Mini-batch indices MB_idx; replay buffer D; neighbors n; num_agents
Output: *Final list* (mini-batch transitions)
for $agent\_id$ ***in*** $agent\_trainers$ **do**
  $MB\_idx \leftarrow$ *random.sample(len(D), batch size)*
  for $agent \leftarrow 1, 2, \ldots, num\_agents$ **do**
    for idx ***in*** MB_idx **do**
      Output data $\leftarrow D[idx : idx + neighbors]$
      Final list.append(Output data)

Figure 7 depicts the new workflow from our intra-agent cache locality-aware sampling. We effectively steer the hardware prefetcher towards fetching transition data (memory accesses that follow a sequential pattern from the chosen reference point) from contiguous memory locations into the cache. Our experimental analysis (Section VI-A) demonstrates that this simple optimization can significantly improve performance while preserving the MARL reward values observed with random mini-batch sampling.

B. Algorithmic Optimizations

We explore additional opportunities for performance optimization to tune MARL algorithms in the sampling phase, which dominates their overall training time.

1) Information Prioritized Locality-aware Sampling: When estimating the expected value using stochastic updates, it is important that the updates correspond to the same distribution as expected. However, cache locality-aware sampling can introduce bias by changing this distribution in an uncontrolled way, ultimately altering the solution to which the estimates will converge. To tackle this issue and obtain the performance guarantee, we apply the importance-sampling weights on top of the baseline MARL workload provided in the following lemma 1.

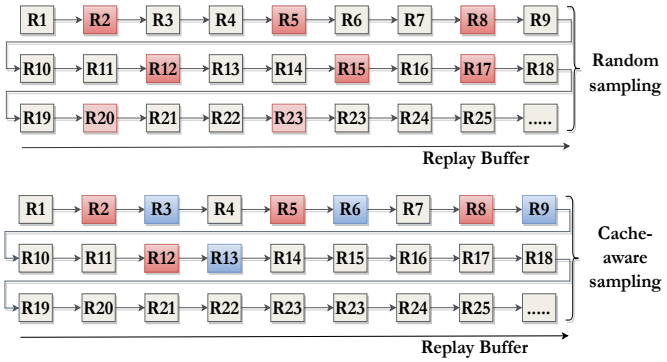


Fig. 7: Illustration of random sampling and cache locality-aware sampling. The reference points (R2, R5, R8, R12) are highlighted in red. Transitions associated with each reference point are shown in blue, with one neighbor highlighted for each reference point.

Prior studies have considered prioritized experience replay to improve training efficiency and rewards [27], [28]. These strategies involve assigning weights to transitions that enhance the sampling performance and algorithm efficiency. In our approach, we also study how to incorporate a prioritization scheme into our framework, which we attempt to combine with the cache locality-aware sampling technique mentioned earlier. To elaborate, we first generate the common indices array set using proportional sampling [27], with the likelihood of selecting an index influenced by the priority associated with that index. Subsequently, we employ a predictor to determine the optimal neighbors for the selected priority reference based on the normalized weight (0 to 1). This process continues

until the batch size (1024) is reached to update the actor-critic networks.

The predictor takes a normalized weight as input and returns a list of optimal neighbors based on the value of the weight (specific to each reference point). This determination is made based on set threshold levels of granularity.

Lemma 1: The importance-sampling weights at step i for eliminating the bias of changing the sampling strategy are given by:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}, \quad (1)$$

where N is the buffer size and $P(i)$ is the cache locality-aware sampling probability. β represents the compensation parameter. When $\beta = 1$, it indicates full compensation, akin to importance sampling [27].

Lemma 1 provides the weights that can be incorporated into the learning by updating the weighted temporal-difference and adjusts the sampling probabilities of experiences to prioritize and sample those experiences more frequently during training. This process involves selecting reference points from a list of experiences based on a randomly generated value within the range of cumulative priorities. We then calculate the weights for those indices using importance sampling and use a predictor to determine the optimal neighbors in the training phase [27].

2) Transition Data Layout Reorganization: Here, we reconsider how the transitions (experiences) are stored in the experience replay buffer. We redesign the replay buffer as key-value stores, i.e., instead of storing transitions separately for each agent in distant memory locations, we transform the replay buffer into a hash map with key-value pairs. The key represents the index, and the corresponding values include transition data histories of all agents sequentially. This modification significantly reduces the sampling phase overheads from being proportional to the number of agents, N, times the length of mini-batch indices, m, to just a single loop iterating m times. The baseline sampling approach had a time complexity of $O(N.m)$. The new approach using the key-value pairs has a time complexity of $O(m)$, resulting in a significant performance improvement, especially as the number of agents increases. By leveraging this locality-aware key-value table, we can prefetch data for multiple agents simultaneously rather than sampling each agent's data individually.

V. Implementation

We implement our proposed optimizations discussed in Section IV on the baseline MARL workloads: MADDPG[2] [18], and MATD3[3] [19]. We use the popular multi-agent particle environment developed by OpenAI[4] as our learning framework and show the performance improvements in cooperative and competitive scenarios. The MARL stages, such as the action selection (neural network computations), are executed on the GPU due to its parallel processing capabilities. On the other hand, the mini-batch sampling phase is CPU-bound, while the actor-critic updates usually run on the GPU.

Evaluation: ① To evaluate our cache-aware mini-batch sampling optimization (Section IV-A), we compare our performance with baseline multi-agent learning methods, namely MADDPG and MATD3, which are foundational algorithms in various applications. ② To evaluate our importance-based optimization (Section IV-B1), we integrate our cache locality-aware sampling into the existing prioritization approaches [27] for MARL workloads. Subsequently, we compare our information prioritized locality-aware sampling (Section IV-B1) with the state-of-the-art prioritization schemes (PER-MADDPG and PER-MATD3 [29]) that have low overhead in terms of weight computation. ③ We compare the data layout reorganization approach (Section IV-B2) to the baseline MARL workload, MADDPG.

Hardware: We evaluate our approach on an NVIDIA RTX 3090. The full platform description can be found in Table II. The server runs Ubuntu Linux 20.04.5 LTS and is equipped with CUDA 9.0, cuDNN 7.6.5, PCIe Express v4.0, and the NCCL v2.8.4 communication library. The Python version used is 3.7.15, and the machine supports TensorFlow (v2.11.0), TensorFlow-GPU (v2.1.0), and OpenAI GYM (v0.10.5). Performance profiles were observed using the Perf [30] tool and NVProf [31] to profile the hardware behavior in a multi-core (using all cores) configuration. The hardware prefetcher is enabled by default, and all CPU cores are operated at their maximum frequency.

TABLE II: Evaluation Platform.

Device	**NVIDIA Geforce RTX 3090**
Architecture	Ampere
Power	350 W
CUDA Cores	10496
Base Clock	1.40 GHz
Device Memory	24 GB, 384-bit bus, GDDR6X
Host	**AMD Ryzen 3975WX**
L1 Cache size	2 MiB (split between L1d and L1i)
L2 Cache size	16 MiB
L3 Cache size	128 MiB (shared)
TLB size	3072 4K pages
Cores / Threads	32 Cores / 64 Threads
Main Memory	512 GB, 2200 MHz, 64-bit bus, DDR4

Software Settings: The actor and critic networks are parameterized by a two-layer ReLU MLP with 64 units per layer, and the mini-batch size is 1024 for sampling the transitions. In all of our experiments, we use Adam optimizer [32] with a learning rate of 0.01, maximum episode length as 25 (max episodes to reach the terminal state), and τ = 0.01 for updating the target networks. γ is the discount factor, which is set to 0.95. The size of the replay buffer is 1 million, and the network parameters are updated after every 100 samples added to the replay buffer. The workloads are trained for 60K episodes without using explicit vector instructions for parallelization of the action selection phase. We use the default hyper-parameters recommended by the state-of-the-art baselines.

[2]https://github.com/openai/maddpg

[3]https://github.com/JohannesAck/MATD3implementation

[4]https://github.com/openai/multiagent-particle-envs

VI. Performance Evaluation

In this section, we evaluate the performance benefits of our optimization techniques.

A. Performance Improvements: Mini-batch Sampling Phase & End-to-end Training Time

We begin by studying our cache-aware sampling optimization, where we pick different numbers of neighbor transition data samples in the experience replay buffer to understand the relative performance gains. Figure 8 shows the performance improvements in two scenarios: one with 64 neighbors and 16 reference points to optimize spatial locality and another with 16 neighbors and 64 reference points to sufficiently preserve the randomness property of sampling in the transitions.

Compared to the baseline MADDPG, our proposed optimization significantly reduces the sampling phase training time by 37.2% for the predator-prey environment with 24 agents (64 neighbors and 16 reference points). This improvement is consistent across predator-prey and cooperative navigation scenarios, each involving 3-6-12-24 agent configurations[5]. Figure 9 shows the end-to-end training time reduction compared to the baseline MARL workloads. We observe that as the number of agents grows from 3 to 24 for a competitive task, the training time reduces from 8.2% to 20.5%, which shows that our optimization improves the end-to-end training time by about $1.2\times$ for a 24-agent setting. Furthermore, as the frequency of sampling function calls increases, the performance gains in end-to-end training time become more pronounced with our cache-aware optimization.

Similar to MADDPG, in the case of MATD3, for 3 and 6 agents in a predator-prey environment, we observe a sampling phase time reduction of 36% with 16 neighbors and 64 reference points. Another key finding is that cache-aware MATD3 also exhibits superior performance compared to the baseline MATD3 within the sampling phase, consequently translating into overall performance improvements. The end-to-end training time reduction for MATD3 with 16 neighbors and 64 reference points in predator-prey environment ranges from 6.8% to 10.25% for 3 to 6 agents, respectively.

Using the Perf tool, we profile the mini-batch sampling phase to obtain low-level CPU metrics. By implementing our cache locality-aware optimization within MADDPG, we significantly reduced cache misses. Specifically, we observed a decrease of approximately 16.1%, 21.8%, 25%, and 29% in cache misses when dealing with 3, 6, 12, and 24 agents, respectively. These improvements were particularly prominent when we had 16 neighbors and 64 reference points (more randomness) in a predator-prey scenario.

Figure 10 illustrates the game scores achieved during the training iterations, depicting the average reward for all agents in multi-agent settings. It is worth noting that in the cooperative navigation environment with 12 agents, our cache-aware optimization shows slight degradation (the point

[5]Given the limitations in available space, we present charts related to MADDPG as a representative MARL workload. We observed similar trends in MATD3 as well.

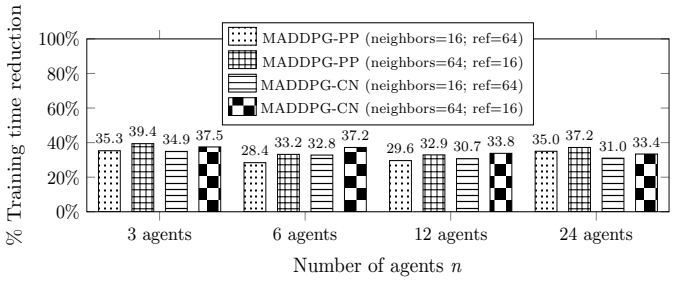


Fig. 8: Comparing the mini-batch sampling phase for MADDPG across various environments: PP (Predator-Prey) and CN (Cooperative Navigation). We utilize 16 neighbors with 64 reference points and 64 neighbors with 16 reference points.

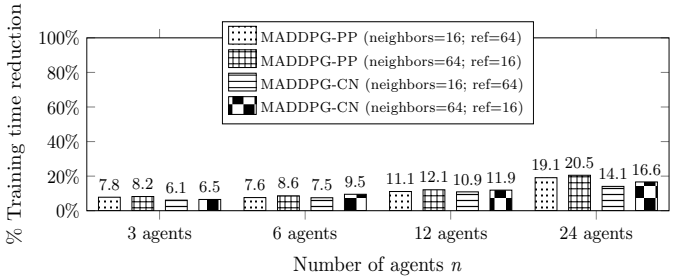
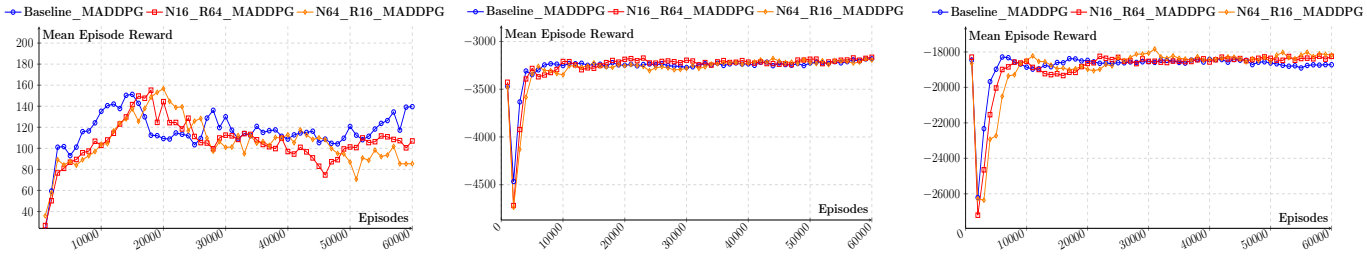


Fig. 9: Comparing the total training time for MADDPG across various environments: PP (Predator-Prey) and CN (Cooperative Navigation). We utilize 16 neighbors with 64 reference points and 64 neighbors with 16 reference points.

where the curve starts to converge) in rewards when altering the uniform distribution. To address this bias and maintain the distribution while improving performance, we introduce information-prioritized sampling. Figure 11 demonstrates that this approach enhances the learning performance by giving priority to experiences with a significant impact on overall rewards while improving the training time through selecting the neighbor samples that preserve spatial locality. The trend in competitive tasks is relatively uneven, but this instability is inherited from the baseline. We note that MARL training can be unstable in some environments due to complex inter-agent interactions and may need hyper-parameter tuning to successfully train for optimal policies.

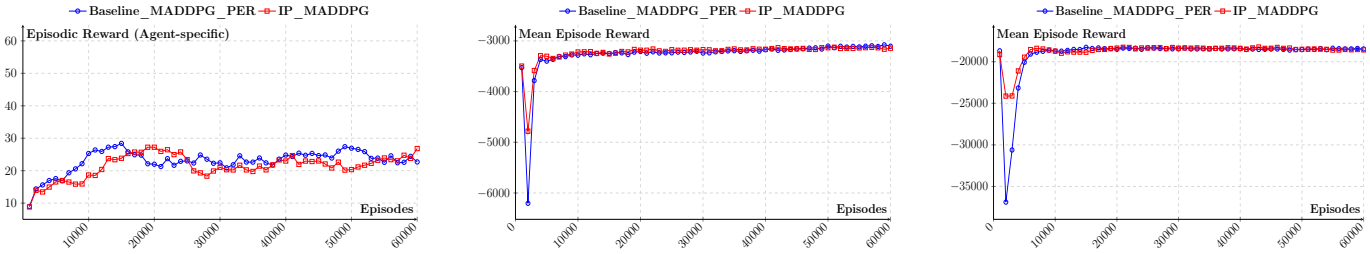
B. Cross-validation

To understand how our optimizations perform across different CPU-GPU architectures, we consider an Intel i7-9700K CPU with 8 cores and a Nvidia Pascal micro-architecture (GTX 1070 - GPU enabled). To evaluate the compute-heavy multi-agent setting, we test MADDPG with the predator-prey environment. Both Figure 12 and Figure 13 illustrate the performance benefits (training time reduction) achieved through the application of our proposed optimization across different computing platforms.



(a) PP-6 agents (b) CN-6 agents (c) CN-12 agents

Fig. 10: The training outcomes for multi-agent games using the baseline MADDPG workload; adopting cache-aware sampling with two settings: one with $n = 16$ and $ref = 64$ (which enhances randomness), and another with $n = 64$ and $ref = 16$ (focusing on optimizing spatial locality). The environments include PP (Predator-Prey) and CN (Cooperative Navigation).



(a) PP-6 agents (b) CN-6 agents (c) CN-12 agents

Fig. 11: Results from Multi-Agent Reinforcement Learning (MARL) training using the PER-MADDPG algorithm with a prioritized sampling of information, conducted in both the Predator-Prey (PP) and Cooperative Navigation (CN) tasks, involving scenarios with 3 to 12 agents. In the Predator-Prey task, only the rewards for the predator agents are presented to illustrate the reward variations. On the other hand, in cooperative tasks, where agents work together, the emphasis lies on presenting the average scores achieved without detailing individual rewards.

An interesting observation is that using the CPU in isolation yields notable performance gains, surpassing the improvements attained from a system equipped with a GTX 1070. This phenomenon arises due to the intricacies of enabling CPU-GPU computation, which involves frequent data transfer and exerts pressure on memory and PCIe bandwidth. This effect becomes more pronounced when dealing with a smaller number of agents, attributed to insufficient data and computation to engage the GPU's processing capacity completely.

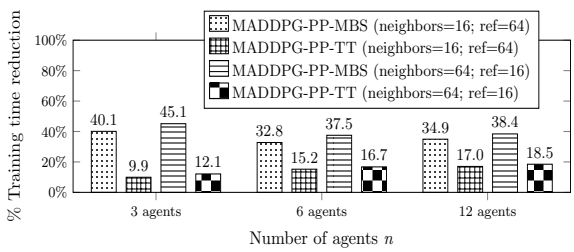


Fig. 12: Mini-batch sampling phase (MBS) and total training time (TT) savings on Intel i7-9700K CPU with 8 cores evaluated on MADDPG with predator-prey environment.

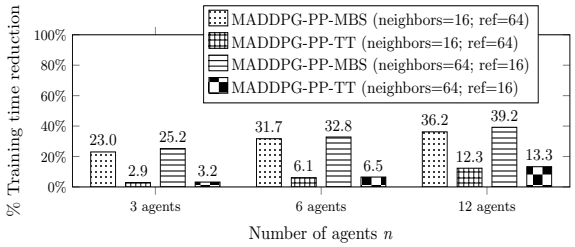


Fig. 13: Mini-batch sampling phase (MBS) and total training time (TT) savings on CPU-GPU (Nvidia Pascal-GTX 1070) evaluated on MADDPG with predator-prey environment.

C. Additional Opportunities for Performance Optimization

1) Information prioritized locality-aware sampling: We evaluate the end-to-end performance benefits of our information-prioritized locality-aware sampling and compare it to the state-of-the-art prioritized version of MADDPG, as algorithmic optimization involves calculating the weights and updating priorities in the training phase. In our evaluation setup, we adjust the selection of neighboring reference points based on their values. If a reference point's value is below 0.33 (T_1), we pick one neighbor (N_1). When the reference point value falls between 0.33 (T_1) and 0.66 (T_2), we opt for

two neighbors (N_2). If a reference point's normalized priority surpasses 0.66 (T_2), we choose four neighbors (N_3). These parameters collectively allow our algorithm to adaptively determine the number of neighbors selected, improving learning efficiency in multi-agent reinforcement learning. Averaging across 3, 6, and 12 agents, we observed 2× improvement in the efficiency of the mini-batch sampling phase for MADDPG in both competitive and cooperative. Figure 11 illustrates reward curves plotted over 60,000 episodes. This graph validates that our optimizations perform comparably to the state-of-the-art baseline, as indicated in the results. The curve in red is our optimization on top of PER [27], and the curve in blue is the baseline (PER-MADDPG [27]). This is achieved by strategically selecting the reference points that have high priority and combining them with cache-aware sampling to enable the hardware prefetcher to operate efficiently.

2) Data layout reorganization: Figure 14 shows the training time reduction for the mini-batch sampling phase. In cases involving 3 and 6 agents, the dominant factor in performance profile is the transition data layout reorganization phase, and we observe performance slowdown. This is because the time required for data reshaping must be combined with the layout reorganization time. However, in the case of 24 agents within the predator-prey environment, a substantial reduction of approximately 25.8% in the time taken for the sampling phase becomes noticeable (Figure 14). If we focus solely on inter-agent sampling and exclude data reshaping, we achieve a speedup of about 1.36×-2.26×-4.41×-9.55× for 3-6-12-24 agents respectively in competitive environments. Similarly, in cooperative scenarios, we can achieve speedups of 1.18×-1.71×-3.44×-7.03× for 3-6-12-24 agents, respectively.

VII. Related Work

Prior work have analyzed performance enhancing methods for training and inference times through software-hardware optimizations [33], [34]. To accelerate the single-agent RL algorithms using CPU-GPU platforms, several methods are proposed [35]–[42]. QuaRL [35] observed that quantizing the policies to $\leq$ 8 bits led to performance improvements and carbon emission reduction compared to full precision training only in single-agent settings. WarpDrive [42], provides high-throughput and scales almost linearly to many agents and run thousands of parallel environment simulations. However, our work emphasizes multi-agent scenarios where inter-agent communication within a common shared environment is pivotal, and these scenarios are directly applicable to real-world applications. AccMER [43], on the other hand, minimizes the transition data movement of cooperative MARL workloads by repeatedly reusing the transitions for a window of n steps. However, this approach specifically targets prioritized MARL workloads and cooperative tasks.

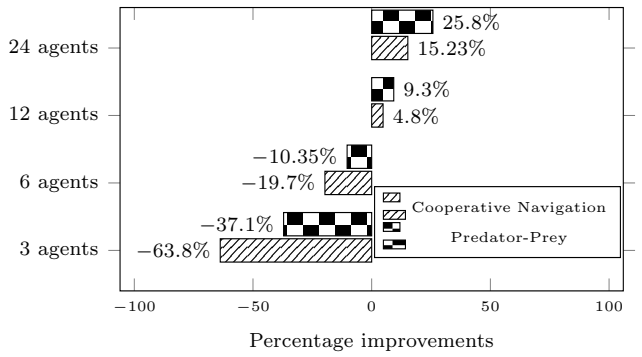


Fig. 14: Reduction in training time for the mini-batch sampling phase (MADDPG workload) after enabling transition data layout reorganization on predator-prey & cooperative tasks.

Prior studies, like FA3C [33], have focused on accelerating multiple parallel worker scenarios, where each agent is controlled independently within their own environments using single-agent RL algorithms. iSwitch [34] reduces the end-to-end network latency for synchronous training but also improves the convergence with faster weight updates for asynchronous training. However, MARL algorithms involve significant inter-agent interactions and related computations in a single shared environment. Agents in such MARL settings usually have a large observation-action space. To the best of our knowledge, this is the first work to present insights into comprehensive performance profiling that encompasses multiple agents from a systems perspective.

Most of the existing literature has extensively discussed the challenges related to memory accesses in various applications, including recommendation systems, RL and large language models [44]–[48]. To tackle these issues, previous efforts have explored *Processing-In-Memory/Near-Memory* techniques to accelerate the inference and training phases. However, these approaches require significant changes to the hardware in terms of commercialization and adapting to rapidly evolving model designs. In contrast, we introduce two cost-effective methods for adapting the software to improve memory access prediction and optimize cache usage with hardware hints.

VIII. Conclusion

We present a detailed performance analysis of a new class of algorithms originating from the domain of Multi-Agent Reinforcement Learning (MARL). These workloads are computationally intensive and can run for several days, even for a relatively small number of agents (48) on the latest high-performance GPUs. We proposed and studied several optimizations to address the performance concerns that guide the hardware prefetchers to reduce the end-to-end training time. Our experimental results demonstrate end-to-end training time acceleration, with improvements ranging from 8.2% (3 agents) to 20.5% (24 agents) compared to the state-of-the-art MADDPG algorithm. Further, we achieve 2× speedup (sampling phase) for our information prioritized sampling compared to PER-MADDPG.

Acknowledgment

This research is based on work supported by the National Science Foundation under grant CCF-2114415.

REFERENCES

[1] H. U. Sheikh and L. Bölöni, "Multi-agent reinforcement learning for problems with combined individual and team reward," in *IJCNN*. IEEE, 2020, pp. 1–8.

[2] G. Swamy, S. Reddy, S. Levine, and A. D. Dragan, "Scaled autonomy: Enabling human operators to control robot fleets," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 5942–5948.

[3] Y. Yuan, J. Hao, F. Ni, Y. Mu, Y. Zheng, Y. Hu, J. Liu, Y. Chen, and C. Fan, "Euclid: Towards efficient unsupervised reinforcement learning with multi-choice dynamics model," *arXiv preprint arXiv:2210.00498*, 2022.

[4] A. L. Bazzan, "Opportunities for multiagent systems and multiagent reinforcement learning in traffic control," *Autonomous Agents and Multi-Agent Systems*, vol. 18, pp. 342–375, 2009.

[5] F. Ni, J. Hao, J. Lu, X. Tong, M. Yuan, J. Duan, Y. Ma, and K. He, "A multi-graph attributed reinforcement learning based optimization algorithm for large-scale hybrid flow shop scheduling problem," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 3441–3451.

[6] H. Jianye, X. Hao, H. Mao, W. Wang, Y. Yang, D. Li, Y. Zheng, and Z. Wang, "Boosting multiagent reinforcement learning via permutation invariant and permutation equivariant networks," in *The Eleventh International Conference on Learning Representations*, 2022.

[7] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[8] S. Gronauer and K. Diepold, "Multi-agent deep reinforcement learning: a survey," *Artificial Intelligence Review*, pp. 1–49, 2022.

[9] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *Handbook of Reinforcement Learning and Control*, pp. 321–384, 2021.

[10] M. Wen, J. Kuba, R. Lin, W. Zhang, Y. Wen, J. Wang, and Y. Yang, "Multi-agent reinforcement learning is a sequence modeling problem," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 509–16 521, 2022.

[11] M. W. Brittain, X. Yang, and P. Wei, "Autonomous separation assurance with deep multi-agent reinforcement learning," *Journal of Aerospace Information Systems*, vol. 18, no. 12, pp. 890–905, 2021.

[12] P. Razzaghi, A. Tabrizian, W. Guo, S. Chen, A. Taye, E. Thompson, A. Bregeon, A. Baheri, and P. Wei, "A survey on reinforcement learning in aviation applications," *arXiv preprint arXiv:2211.02147*, 2022.

[13] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," 2023.

[14] Y. Mei, H. Zhou, and T. Lan, "Projection-optimal monotonic value function factorization in multi-agent reinforcement learning," in *Proceedings of the 2024 International Conference on Autonomous Agents and Multiagent Systems*, 2024.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[16] L. Busoniu, R. Babuska, and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.

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

[18] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *NeurIPS*, vol. 30, 2017.

[19] J. Ackermann, V. Gabler, T. Osa, and M. Sugiyama, "Reducing overestimation bias in multi-agent domains using double centralized critics," *NeurIPS Deep RL Workshop*, 2019.

[20] T. Li, K. Zhu, N. C. Luong, D. Niyato, Q. Wu, Y. Zhang, and B. Chen, "Applications of multi-agent reinforcement learning in future internet: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 2, pp. 1240–1279, 2022.

[21] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, "Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications," *IEEE transactions on cybernetics*, vol. 50, no. 9, pp. 3826–3839, 2020.

[22] A. Oroojlooy and D. Hajinezhad, "A review of cooperative multi-agent deep reinforcement learning," *Applied Intelligence*, vol. 53, no. 11, pp. 13 677–13 722, 2023.

[23] A. Feriani and E. Hossain, "Single and multi-agent deep reinforcement learning for ai-enabled wireless networks: A tutorial," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1226–1252, 2021.

[24] 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.

[25] K. Gogineni, P. Wei, T. Lan, and G. Venkataramani, "Scalability Bottlenecks in Multi-Agent Reinforcement Learning Systems," *arXiv preprint arXiv:2302.05007*, 2023.

[26] ——, "Towards efficient multi-agent learning systems," *arXiv preprint arXiv:2305.13411*, 2023.

[27] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[28] Y. Mei, H. Zhou, T. Lan, G. Venkataramani, and P. Wei, "MAC-PO: Multi-agent experience replay via collective priority optimization," in *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, 2023, pp. 466–475.

[29] J. Ackermann, "Tensorflow-2 implementation of multi-agent reinforcement learning approaches," https://github.com/JohannesAck/tf2multiagentrl, 2020.

[30] V. Ramos, "Performance counters api for python," https://pypi.org/project/performance-features/, May 2019.

[31] Nvidia-Profiler-12.3, "Nvidia profiler user's guide," https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2023.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *ASPLOS*, 2019, pp. 499–513.

[34] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 279–291.

[35] 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," 2022.

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

[37] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "GA3C: GPU-based A3C for deep reinforcement learning," *CoRR abs/1611.06256*, 2016.

[38] M. Zhou, Z. Wan, H. Wang, M. Wen, R. Wu, Y. Wen, Y. Yang, Y. Yu, J. Wang, and W. Zhang, "Malib: A parallel framework for population-based multi-agent reinforcement learning." *J. Mach. Learn. Res.*, vol. 24, pp. 150–1, 2023.

[39] V. Egorov and A. Shpilman, "Scalable multi-agent model-based reinforcement learning," *arXiv preprint arXiv:2205.15023*, 2022.

[40] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1705.04862*, 2017.

[41] 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 *International Conference on Machine Learning*. PMLR, 2021, pp. 980–991.

[42] T. Lan, S. Srinivasa, H. Wang, and S. Zheng, "Warpdrive: fast end-to-end deep multi-agent reinforcement learning on a gpu," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 14 225–14 230, 2022.

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

[44] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[45] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 488–501.

[46] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 790–803.

[47] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "Recssd: near data processing for solid state drive based recommendation inference," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 717–729.

[48] K. Gogineni, S. S. Dayapule, J. Gómez-Luna, K. Gogineni, P. Wei, T. Lan, M. Sadrosadati, O. Mutlu, and G. Venkataramani, "Swiftrl: Towards efficient reinforcement learning on real processing-in-memory systems," *arXiv preprint arXiv:2405.03967*, 2024.