Accelerating Multi-Agent DDPG on CPU-FPGA Heterogeneous Platform

Samuel Wiggins[1], Yuan Meng[1], Rajgopal Kannan[2], Viktor Prasanna[1]
[1]*Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California*
[2]*DEVCOM Army Research Lab*
Contact: {*wigginss, ymeng643, prasanna*}*@usc.edu, rajgopal.kannan.civ@army.mil*

***Abstract*—Multi-Agent Reinforcement Learning (MARL) is a key technology in artificial intelligence applications such as robotics, surveillance, energy systems, etc. Multi-Agent Deep Deterministic Policy Gradient (MADDPG) is a state-of-the-art MARL algorithm that has been widely adopted and considered a popular baseline for novel MARL algorithms. However, existing implementations of MADDPG on CPU and CPU-GPU platforms do not exploit fine-grained parallelism between cooperative agents and handle inter-agent communication sequentially, leading to sub-optimal throughput performance in MADDPG training. In this work, we develop the first high-throughput MADDPG accelerator on a CPU-FPGA heterogeneous platform. Specifically, we develop dedicated hardware modules that enable parallel training of each agent's internal Deep Neural Networks (DNNs) and support low-latency inter-agent communication using an on-chip agent interconnection network. Our experimental results show that the speed performance of agent neural network training improves by a factor of 3.6× - 24.3× and 1.5× - 29.5× compared with state-of-the-art CPU and CPU-GPU implementations. Our design achieves up to a 1.99× and 1.93× improvement in overall system throughput compared with CPU and CPU-GPU implementations, respectively.**

***Index Terms*—Multi-Agent Reinforcement Learning, FPGA Acceleration, MADDPG**

I. Introduction

Multi-Agent Reinforcement Learning (MARL) is an area of Machine Learning that has seen popularity in various domains, including energy systems, self-driving cars, competitive games, network routing, etc [1]. The goal of MARL is to develop adaptive agents that can effectively coordinate or compete with other agents to achieve collective or individual goals. MARL extends traditional reinforcement learning by introducing the challenge of learning in a dynamic environment that is also affected by other agents. Therefore, MARL algorithms rely on effective inter-agent communication to improve the agents' learning performance [2].

Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [3] is a state-of-the-art Centralized-Training Decentralized-Execution (CTDE) MARL algorithm. Training occurs in an actor-critic manner, where each agent has a value network that facilitates the training of its policy network. MADDPG requires inter-agent communication, i.e., the sharing of action activities and state-transition information among agents during the neural network training phase. MADDPG has been widely adopted in literature and is often used as a baseline to compare against novel MARL algorithms. Specifically, MADDPG has been applied to several applications, including task partitioning and resource allocation in mobile edge computing [4] robotics [5], optimizing energy for hybrid electric vehicles [6], etc.

Existing implementations of MADDPG are highly sequential, with all agents mapped to a single CPU thread sharing an instance of the training environment. Some implementations use the massive parallel compute cores of GPUs to accelerate the training of agent Deep Neural Networks (DNNs) [3], [7], [8]. The efficiency of developing high-speed MADDPG systems is dependent upon several factors, including the platform's ability to resolve dependency requirements within MARL execution loops, and the suitability of the system memory hierarchy for parameter aggregation and communication. However, existing CPU-only and CPU-GPU platforms cannot balance the above factors, leading to various time overheads that impede the MADDPG system from achieving optimal throughput.

FPGAs are emerging as a popular platform of choice for accelerating computation- and memory-intensive deep learning applications. [9]–[11]. We propose accelerating the training process of MADDPG by leveraging the rich power compute and memory resources of FPGAs. We justify the use of a CPU-FPGA heterogeneous platform for MADDPG acceleration based on the following considerations: (1) Policy learning: FPGAs have large distributed on-chip SRAM, which can provide low-latency parameter accesses and aggregations during policy weight learning and updates; FPGAs also have rich logic resources that can enable fine-grained, fast inter-agent communication between agents; (2) Environment simulation: CPUs are general-purpose, so they can be used for executing application-dependent software simulations and allow plug-and-play in the MARL data collection process of different applications.

In this work, our main contributions are:

- We break down and identify key computational kernels of MADDPG and map them onto a CPU-FPGA platform.
- We offload compute-intensive training of agent DNNs using parallel pipelines on FPGA.

This work is supported by the U.S. National Science Foundation (NSF) under grant OAC-2209563 and the Army Research Lab (ARL) under grant W911NF2220159.

Distribution Statement A: Approved for public release. Distribution is unlimited.

- We enable efficient all-to-all inter-agent communication between agents during the training stage using an on-chip ring network.
- We implement our design on a CPU-FPGA platform and demonstrate up to a 1.99× and 1.93× higher throughput compared to CPU-only and CPU-GPU implementations, respectively.

II. Background

A. Multi-Agent Deep Deterministic Policy Gradient

We consider an $N-$agent partially observable Markov Game composed of a set of agents, a state space, an action space for each agent i, a reward function ,and transition probability from each state-action pair to another state. For agent i, we denote its policy as a probability distribution over its action space: $\pi_i\left(a_t \mid s_t\right)$ is the probability of taking action a_t upon state s_t at a certain time step t. Each agent i aims to find an optimal policy that maximizes its own expected cumulative reward until terminal time step T: $R_i=\sum_{t=0}^{T} \gamma^i r_i^t$. To achieve this aim, Multi-Agent Deep Deterministic Policy Gradient (MADDPG) follows the actor-critic paradigm and utilizes two Deep Neural Network (DNN) models for each agent - one for approximating the action-value (i.e., value network) and one for approximating the policy (i.e., policy network) [3].

Each agent has its own policy network denoted $\mu_{\theta_i}(s)$. An agent's value network is denoted $Q_i^{\pi}(s, a_1, ..., a_N)$. The value networks are centralized as they take all the agents' action information as its input. Both DNNs use the Stochastic Gradient Descent (SGD) algorithm for optimization [12]. They are trained collaboratively. The gradient for optimizing the policy network is:

$$\begin{aligned} \nabla_{\theta_i} J(\mu_i) = \mathbb{E}_{s,a\sim\mathbb{D}} [\nabla_{\theta_i}\mu_i(a_i|o_i) * \\ \nabla_{a_i} Q_i^{\mu}(s, a_1, ..., a_N)|_{a_i=\mu_i(o_i)}] \end{aligned} \tag{1}$$

The centralized value network is updated as:

$$\begin{aligned} \mathcal{L}(\theta_i) = \mathbb{E}_{x,a,r,x'}\left[(Q_i^{\mu}(s, a_1, ..., a_N) - y)^2\right], \\ y = r_i + \gamma Q_i^{\mu'}(s', a_1', ...a_N')|_{a_j'=\mu_j'(o_j)}, \end{aligned} \tag{2}$$

where $\mu' = \{\mu_{\theta_1'}, ..., \mu_{\theta_N'}\}$ is the set of target policies used for training stability [13], [14]. τ is a constant scaling factor when updating target networks.

The complete MADDPG algorithm is shown in Algorithm 1. Figure 1 shows the breakdown of Training-in-Simulation for MADDPG using two distinct phases. **Sample Generation** (Algorithm 1 line 6-8) involves agents taking an action derived from their policy network in the environment, where transition information is stored in a shared replay buffer. **Model Update** (Algorithm 1 line 9-13) involves optimizing objective functions across each agent's internal DNN networks. Each MADDPG agent includes a total of four DNN networks: (1) policy, (2) critic, (3) target policy, and (4) target critic networks. Multi-Layer Perceptrons (MLPs) are commonly used to represent these networks.

Algorithm 1 MADDPG Algorithm

```
1:  Input: Initial NN parameters $\theta_i$
2:  Output: Learnt policies $\mu_{\theta_i}$
3:  Initialize Environment simulator $\leftarrow$ ENV
4:  for $episode$ = 1,2,...$M$ do
5:      for $t$ = 1 to max-episode-length do
6:          for each agent $i$, select action $a_i \leftarrow \mu_{\theta_i}$(state $s$)
7:          reward $r$, next state $s'$ $\leftarrow$ENV($\{a_1$,...,$a_N\}$)
8:          Store transition $\{s,\ a,\ r,\ s'\}$ in replay buffer $\mathcal{D}$
9:          for $Agent$ = 1,2,...$N$ do
10:             Sample a random mini-batch of samples from $\mathcal{D}$
11:             Update value network by minimizing loss function
                $\mathcal{L}(\theta_i)$ from equation 2
12:             Update policy network using the sampled policy
                gradient $\nabla_{\theta_i} J$ from equation 1
13:         end for
14:         Update target network parameters for each agent $i$:
            $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
15:     end for
16: end for=0
```

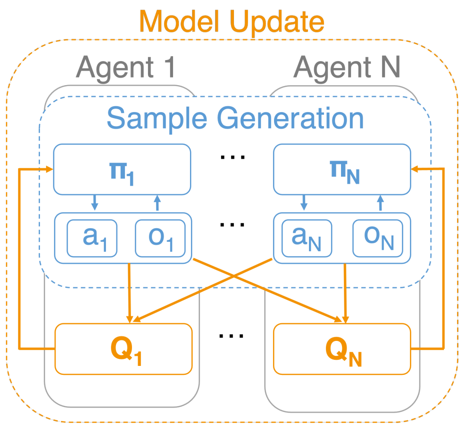


Fig. 1. Centralized Critics and Decentralized Actors approach of MADDPG.

B. Challenges in MADDPG Acceleration

Each training iteration of MADDPG involves several primitives with high variations in their arithmetic intensities and memory requirements (i.e. forward/backward propagations of SGD, replay operations, inter-agent communication, etc). In existing CPU implementations, these variations result in non-uniform data access latencies through multi-level cache hierarchy [15]. Mapping agents to separate CPU threads can potentially incur additional latency penalties when performing all-to-all inter-agent communication compared to a single-threaded approach. Dedicated data layout and custom memory system can be developed using FPGA to hide these data access overheads efficiently.

Some MADDPG systems utilize the high-bandwidth global memory and data-parallel compute cores of GPUs to accelerate the training of agent DNNs, where high-throughput batched independent forward and backward propagations can be mapped. However, the centralized training of each agent's critic and policy networks is dependent upon an all-to-all exchange of action activations between agents [3]. This communication causes data dependencies unsuitable for the independent data-parallel architecture of GPUs [16], especially if inter-agent communication time dominates the time compared to forward and backward propagations.

Figure 2 summarizes the execution time breakdown for Sample Generation and Model Update in one iteration of MADDPG on two typical simulation baselines (Predator-Prey, Cooperative-Communication) [3] on CPU and CPU-GPU coupled platforms, with Sample Generation times normalized to 1. We observe in all cases that the Model Update phase is the bottleneck in current MADDPG systems with either platform in both of the simulation environments.

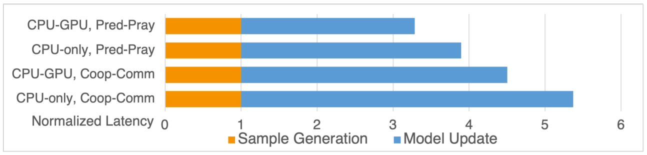


Fig. 2. Execution time breakdown

C. Related Work

There is a plethora of work in acceleration and parallelization of single-agent Reinforcement Learning (RL). [17] deploy multiple actors and workers to facilitate parallel sample generation and DNN training on CPU and GPU platforms. In [18], a CPU-FPGA implementation of single-agent DDPG is proposed in a robotic arm control scenario. However, MARL has unique challenges and additional computations that cannot be directly addressed by these single-agent RL works.

Most MARL work in literature focuses on optimizing reward rather than speed performance. There is limited work focused on MARL Training-in-Simulation acceleration. In [19], a multi-agent variation of Q-learning, Q-Learning Real-Time-Swarm (Q-RTS), is used in control low-powered microcontroller-based agents using a centralized FPGA controller. In [20], a CPU-FPGA implementation of IC3Net [21] exists with an additional pruning system for increased acceleration. To our knowledge, our work is the first CPU-FPGA implementation targeted at MADDPG Training-in-Simulation acceleration.

III. Accelerator Implementation

A. Overview

We map our agents to collect data points from the environment on the CPU and pipeline learner modules on the FPGA to perform neural network training. Figure 3 shows the overall system architecture. Similar to the CPU-only and CPU-GPU baselines, all agents are mapped to a single CPU thread and perform the Sample Generation phase sequentially. Transition information (i.e., state, action, etc.) is collected by agents iteratively and stored in a shared replay buffer contained in CPU global memory. The communication between the Host CPU and device FPGA is achieved through the PCIe interface. Batches of transition information are sampled from the replay buffer and streamed to the FPGA, where gradients and neural network weights are computed. Each agent learner contains a value and policy network training pipeline responsible for updating neural network weights and biases. The updated policy network weights are then sent back to the CPU, where Sample Generation resumes with the new weights.

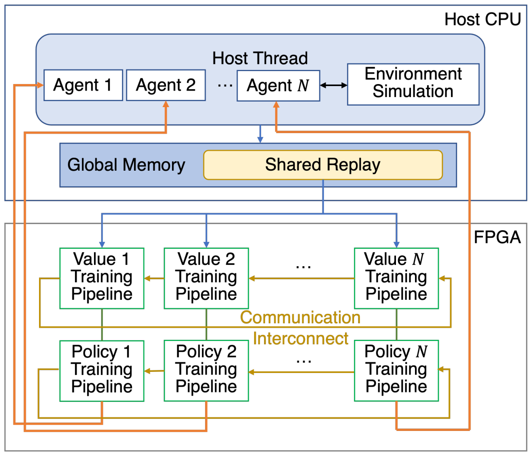
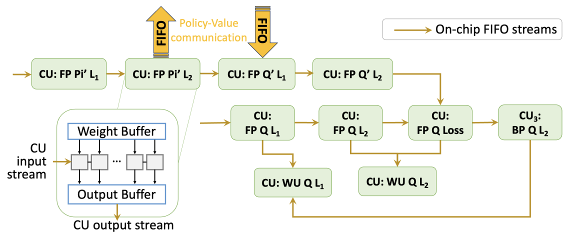

Host CPU
Host Thread
Agent 1
Agent 2
...
Agent N
Environment Simulation
Global Memory
Shared Replay
FPGA
Value 1 Training Pipeline
Value 2 Training Pipeline
...
Value N Training Pipeline
Communication Interconnect
Policy 1 Training Pipeline
Policy 2 Training Pipeline
...
Policy N Training Pipeline

Fig. 3. System Overview

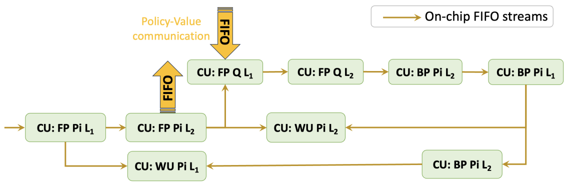
MADDPG agents need to communicate action activations (i.e., output of the policy network) between each other during training of value and policy networks. Using FPGA, we can enable low-latency inter-agent communication through the use of a custom interconnect. Our primary design principle is to balance between latency and area consumption. A naïve interconnect approach is to use a crossbar which has $O(N^2)$ area complexity. Although each pair of agents have a wire for direct communication, this interconnect would be hard to scale with a large number of agents given the limited on-chip resources of an FPGA device. To mitigate the large area complexity of the interconnect between agent training pipelines, our design uses a ring interconnect with an area complexity of $O(N)$. This comes with the trade-off that $O(N)$-cycle latency is needed for all the agents (training pipelines) to receive messages from all the other agents (training pipelines). However, this does not put significant overhead in training because the time spent communicating action activations through wires can be hidden by the computations (i.e., layer propagations) within each DNN model.

B. Learner Module

On the FPGA, each agent is mapped to a learner module with corresponding value and policy training pipeline. Each pipeline contains different connections between neural networks, as shown in Figure 4 using a 2-layer MLP example. The value training pipeline for an $L-$layer value network consists of $n = 5 \times L$ stages: Forward Propagation (FP) through L layers of the target value, target value and value networks (plus an additional stage for Loss computation), Backward Propagation (BP) through $L-1$ layers of the value network and



(a) Value training pipeline



(b) Policy training pipeline

Fig. 4. Multiple pipeline dataflow of both pipelines in the learner module for L=2-Layer MLP networks. Pi, Q, Pi', Q' denotes the policy, value, target policy, and target values networks, respectively. L_i denotes the i^{th} layer.

Weight Update (WU) for each of the L value layers. The policy training pipeline contains $m = 5 \times L - 1$ stages: FP through L layers of the policy and value networks, BP through L layers of the value network and $L-1$ layers of the policy network, and WU for L number of value layers. Each stage corresponds to a unique Compute Unit (CU) containing a systolic array of F parallel Multiply-Accumulate elements. Each propagation step for a CU may take several passes through the systolic array, as the size of the input data is usually larger than the size of the systolic arrays. We use F (unroll factor) to control the degree of concurrent processing of different output neurons in a data-parallel manner. In practice, a higher unroll factor corresponds to higher throughput during the FP, BP, and WU stages since there would be fewer passes through larger systolic arrays. However, given the limited FPGA resources, fine-tuning the unroll factors F for each stage of the training pipelines is needed to optimize performance.

IV. Design Space Exploration

In this section, we discuss our method for placing the learner modules of N agents on the target FPGA device and the resource allocation to the compute units in each learner training pipeline. Specifically, the objective is to determine the optimal unroll factors F for each training pipeline stage (i.e., CU), in order to maximize the training pipeline throughput given the capacity of an arbitrary FPGA device.

A. Training Pipeline Performance Tuning

Assume a large batch of samples is streaming through each training pipeline (batch size $B \gg$ total number of stages ns in the Agent and Critic training pipelines); each pipeline stage processes the DNN layer propagation of 1 sample at a time. The training pipeline throughput can be modeled as

$$TP = \frac{1}{\max_{j\in[1,ns]} \left(T^{j}_{\text{FP or BP or WU}}\right)} \tag{3}$$

where $T_{\text{FP or BP or WU}}$ stands for the pipeline stage with the longest latency, which can be either a FP, BP, or WU Compute Unit (CU). Given a fixed amount of compute resources, load unbalance between pipeline stages can lead to sub-optimal throughput due to faster-processing CUs idling and waiting for slower-processing CUs to complete. Therefore, maximizing TP is equivalent to minimizing the longest pipeline stage, i.e., tuning the unroll factors in all the CUs such that their latencies are balanced.

Based on the data parallel processing of a CU explained in Section III-B, the latency (i.e., number of compute cycles) spent by each CU processing 1 sample can be modeled using the layer's input dimension ID, output dimension OD, and its data-parallel unroll factor F:

$$T_{\text{FP or BP or WU}} = \frac{ID \times OD}{F} \tag{4}$$

Note that the compute resources (number of DSPs) and memory resources (number of SRAM banks) consumed by the corresponding CU in the above equation are: $C_{\text{DSP}} = 5 \times F$ (each float multiplier-adder consumes 5 DSPs using Vitis HLS); $C_{\text{SRAM}} = F$ (the layer weight matrix needs to be partitioned into different banks for supporting concurrent accesses by parallel multiplier-adders).

The goal is to let

$$T^{i}_{\text{FP or BP or WU}} = T^{j}_{\text{FP or BP or WU}}, \forall i, j \in [1, ns]. \tag{5}$$

We associate a base unroll factor f with the CU responsible for FP of the 1st layer. The optimal unroll factors of all the CUs can be calculated using f based on Equations 4 and 5 (e.g., $F_{\text{FP, layer 2}} = f \times \frac{ID2 \times OD2}{ID1 \times OD1}$, etc.).

B. Learner Resource Allocation

A modern multi-die FPGA is composed of several Super-Logic Regions (SLR) [22], each having a different amount of on-chip resources, and the communication wires between different SLRs are limited. To avoid long-latency cross-SLR communication between CUs in the same training pipeline, we place all the CUs serving the same agent in the same SLR. Assume there are N agents and S SLRs; we assign agents to SLRs such that the number of agents assigned to an SLR is proportional to the available compute resources (i.e., number of DSPs) in that SLR: $N_1 : DSP_{\text{SLR}}1 = N_2 : DSP_{\text{SLR}}2 = \ldots N_S : DSP_{\text{SLR}}S, N_1 + N_2 + \ldots + N_S = N$.

Then, we can determine the base unroll factor f for each agent's training pipeline by finding the maximum value of f such that

$$\begin{gathered}\sum_{i\in[1\ldots ns]} \sum_{j\in\text{Agent}[1\ldots N_x]} C^{\text{Agent } j, \text{ stage } i}_{\text{DSP}} < \text{DSP Bound of SLR } x \\ \sum_{i\in[1\ldots ns]} \sum_{j\in\text{Agent}[1\ldots N_x]} C^{\text{Agent } j, \text{ stage } i}_{\text{SRAM}} < \text{SRAM Bound of SLR } x\end{gathered} \tag{6}$$

As a result, all the unroll factors F for each CU in each agent's training pipeline can be derived from their base unroll factor f using Equations 4 and 5.

V. Experiments and Evaluation

A. Metrics

The primary metric for measuring MADDPG Training-in-Simulation speed is throughput in terms of iterations executed per second (IPS):

$$IPS = \frac{1}{T_{\text{iteration}}} = \frac{1}{T_{SG} + T_{\text{MU}}}, \tag{7}$$

where T_{SG} and T_{MU} are the execution times of the Sample Generation and Model Update phases, respectively. Note that the FPGA kernel focus on the Model Update phase, where $1/T_{\text{MU}}$ for each agent is equivalent to the TP that we optimize in Section IV-A.

B. Simulation Environments and Platform Overview

We use two benchmarks (Cooperative-Communication and Predator-Prey) with a varying number of agents from Multi-Agent Particle Environment [3] to evaluate the performance of our accelerated system. Cooperative-Communication involves two MADDPG agents where a "speaker" agent must guide a "listener" agent to a desired landmark destination. Predator-Prey involves three slower "Predator" MADDPG agents working together to capture a single faster "Prey" DDPG agent.

We compare our CPU-FPGA heterogeneous design against two different setups: CPU-only homogeneous platform and CPU-GPU heterogeneous platform. For the CPU-only and CPU-GPU implementation, all agents are mapped to a single CPU thread, where the training of internal neural networks happens sequentially agent-by-agent on the CPU or GPU, respectively. The CPU-only baseline can distribute training among multiple training threads. The CPU-GPU baseline offloads training to an NVIDIA RTX A5000 GPU. The CPU-only and CPU-GPU baselines use Python v3.11.3 and PyTorch v2.0.1, where the CPU-GPU version additionally uses CUDA v11.8.0. FPGA kernels are developed using Xilinx Vitis HLS v2022.2. OpenCL is used to implement transfers from the host CPU to the FPGA via PCIe. Table I provides a detailed list of the device specifications used to conduct our experiments.

TABLE I
PLATFORM SPECIFICATIONS

Platform	CPU AMD EPYC 7763	GPU NVIDIA RTX A5000	FPGA Xilinx Alevo U200
Frequency	2.45 GHz	2.0 GHz	300 MHz
Memory Bandwidth	205 GB/s	768 GB/s	77 GB/s
On-Chip Memory	256 MB L3 Cache	6 MB L2 Cache	35 MB
Peak Performance	3.58 TFLOPS	27.8 TFLOPS	18.6 TOPS

C. Resource Utilization

We perform the design space exploration described in section IV to generate the design parameters used in each experiment. We summarize the post-synthesized hardware resource utilization obtained from both experiments in Table II. We observe increased utilization for Predator-Prey than Cooperative-Communication due to the higher number of agents.vWe implement Cooperative Communication with a more aggressive unroll factor F across its CUs since more resources are available for the training pipeline of each agent.

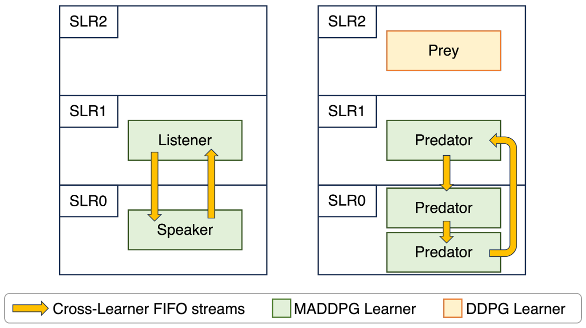


Fig. 5. Device map for **(left)** Cooperative-Communication, **(right)** Predator-Prey on Xilinx Alevo U200 FPGA.

TABLE II
ACCELERATOR RESOURCE UTILIZATION

Experiment	BRAM	FlipFlop	LUT	DSP
Cooperative-Communication	1060 (49%)	1182K (50%)	774K (65%)	3320 (49%)
Predator-Prey	1519 (73%)	1438K (61%)	829K (70%)	2447 (36%)

Figure 5 displays the device map for both experiments, showing the three SLR regions contained within the U200 FPGA. We let Cooperative-Communication occupy a total of two SLRs (one for each agent's learner module), while Predator-Prey occupies all three (two learner modules in SLR1, one each for SLR0 and SLR2). The resource constraints for design space exploration are set accordingly.

D. Performance: IPS

Current MADDPG CPU-only implementations allow the learner to deploy several training threads to perform the Model Update phase across multiple CPU threads. Figure 6 shows the learners' training time with a varied number of training threads for the Predator-Prey simulation. We can see that the execution time increases when adding training threads. This is because cross-thread communication and aggregation of results between CPU threads can take significant time due to the non-uniform access latency from the multi-level cache hierarchy. Given this trend, we will utilize a single training thread for our experiments with the CPU-only baseline.

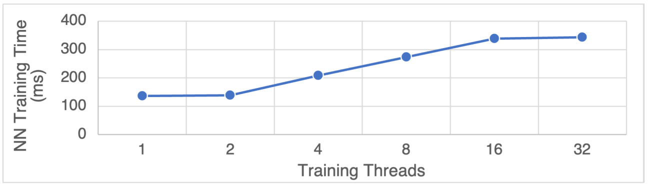


Fig. 6. Learners training time of CPU-only platform when varying the number of training threads.

Figure 7 shows a comparison between all three platforms - CPU-only, CPU-GPU, CPU-FPGA with respect to learner

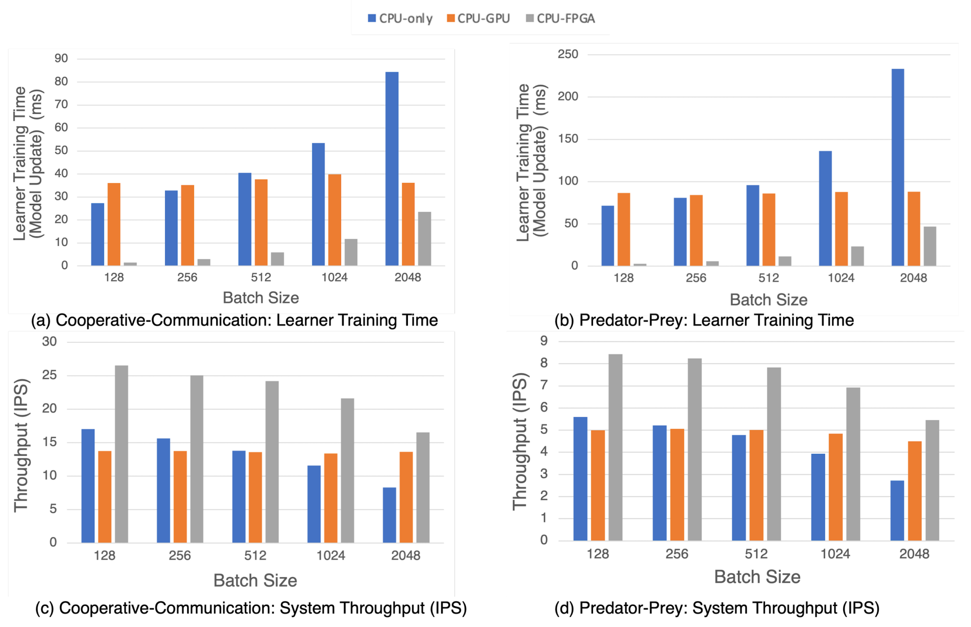


Fig. 7. Comparisons of learner training time and overall system performance measured in IPS between our design - CPU-FPGA and baselines - CPU-only, CPU-GPU varying the batch size.

training time and overall system throughput in the two simulation environments. We observe similar trends between both Cooperative-Communication and Predator-Prey as evident from Figures 7(a) vs. 7(b) and Figures 7(c) vs. 7(d). Cooperative-Communication has shorter learner training time and higher throughput values measured in IPS compared to Predator-Prey. This is attributed to the more complex environment of Predator-Prey with its additional agents, where agent inference and environment interaction time takes (4×) longer. We observe that our CPU-FPGA system's neural network training speedup and overall system throughput improvements are higher in the Predator-Prey simulation compared to Cooperative-Communication because of the following reasons: (1) an increased amount of agent-level parallelism using added learner modules to account for the additional two agents, (2) exploiting fine-grained inter-agent communication between an increased amount of agents.

In Figures 7(a) and 7(b), the learner training time is shown between the two environment scenarios with varying batch sizes. We first observe that it takes a certain batch size (in our case, size 512 or more) for the CPU-GPU platform to outperform the CPU-only platform. The added kernel and synchronization overheads coupled with the small neural network MLPs severely limit the high bandwidth data parallel architecture of modern GPUs. GPU threads can independently process samples from a batch in parallel during SGD, resulting in high-weight reusing as the batch size increases. Off-chip data access for weights is expensive, and its effects can be mitigated with high data reuse. However, with a relatively low arithmetic intensity of MADDPG, utilizing GPUs for small batches is dominated by the abovementioned overheads. Our implementation mitigates these problems, storing weights and biases in on-chip memory. This results in our CPU-FPGA system achieving a neural network training speedup of 3.6× - 24.3× and 1.5× - 29.5× compared with the CPU-only and CPU-GPU implementations.

In Figures 7(c) and 7(d), the overall system throughput measured in IPS is shown with varying batch sizes. The CPU-GPU platform is only able to surpass the CPU-only system's throughput with a batch size of 1024 and 512 for Cooperative-Communication and Predator-Prey, respectively. Again, synchronization and off-chip memory access overheads for small batch sizes are not hidden efficiently on GPUs. Our design achieves up to a 1.99× and 1.93× improvement compared to CPU and CPU-GPU implementations. While accelerating the value and policy training achieves significant speedup in system throughput, we are limited by the Sample Generation phase and the replay sampling time during the Model Update phase.

VI. Conclusion

We accelerated MADDPG on a CPU-FPGA heterogeneous platform by training agents on FPGA using parallel learner modules. We compared our system with baselines using implementations on CPU and CPU-GPU platforms and observe significant speedup in learners' training time. Future work includes looking into accelerating MARL algorithms onto heterogeneous platforms with more complicated communication strategies or a fully decentralized training scheme. Our current accelerator only looks into parallelizing the neural network training portion of MADDPG. Future works can look at acceleration during the Sample Generation phase (parallelize agent simulation interactions, mapping agents to separate CPU threads, etc.) or more complex replay management.

References

[1] 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.

[2] C. Zhu, M. Dastani, and S. Wang, "A survey of multi-agent reinforcement learning with communication," *arXiv preprint arXiv:2203.08975*, 2022.

[3] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems*, vol. 30, 2017.

[4] K. Lu, R.-D. Li, M.-C. Li, and G.-R. Xu, "Maddpg-based joint optimization of task partitioning and computation resource allocation in mobile edge computing," *Neural Computing and Applications*, pp. 1–18, 2023.

[5] R. Wu, J. Zhong, B. Wallace, X. Gao, H. Huang, and J. Si, "Human-robotic prosthesis as collaborating agents for symmetrical walking," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 306–27 320, 2022.

[6] J. Peng, W. Chen, Y. Fan, H. He, Z. Wei, and C. Ma, "Ecological driving framework of hybrid electric vehicle based on heterogeneous multi agent deep reinforcement learning," *IEEE Transactions on Transportation Electrification*, 2023.

[7] 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.

[8] C. Zhang, Y. Meng, and V. Prasanna, "A framework for mapping drl algorithms with prioritized replay buffer onto heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[9] Y. Meng, Y. Yang, S. Kuppannagari, R. Kannan, and V. Prasanna, "How to efficiently train your ai agent? characterizing and evaluating deep reinforcement learning on heterogeneous platforms," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[10] H. Zhou, B. Zhang, R. Kannan, V. Prasanna, and C. Busart, "Model-architecture co-design for high performance temporal gnn inference on fpga," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1108–1117.

[11] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 19–27.

[12] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[13] 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.

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

[15] I. Bevin Brett. (2016) Cpu memory performance. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html

[16] Intel. (2022) Compare benefits of cpus, gpus, and fpgas for different oneapi compute workloads. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.htmlgs.32zunb

[17] C. Zhang, S. R. Kuppannagari, and V. K. Prasanna, "Parallel actors and learners: A framework for generating scalable rl implementations," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021, pp. 1–10.

[18] C. Guo, W. Luk, S. Q. S. Loh, A. Warren, and J. Levine, "Customisable control policy learning for robotics," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 91–98.

[19] G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, M. Re, and S. Spanò, "Fpga implementation of q-rts for real-time swarm intelligence systems," in *2020 54th Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2020, pp. 116–120.

[20] J. Yang, J. Kim, and J.-Y. Kim, "Learninggroup: A real-time sparse training on fpga via learnable weight grouping for multi-agent reinforcement learning," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–9.

[21] A. Singh, T. Jain, and S. Sukhbaatar, "Learning when to communicate at scale in multiagent cooperative and competitive tasks," *arXiv preprint arXiv:1812.09755*, 2018.

[22] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 81–92.