2024 34th International Conference on Field-Programmable Logic and Applications (FPL) | 979-8-3315-3007-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/FPL64840.2024.00040

A Heterogeneous Acceleration System for Attention-Based Multi-Agent Reinforcement Learning

Samuel Wiggins[1], Yuan Meng[1], Mahesh A. Iyer[2], Viktor Prasanna[1]
[1]*Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California*
[2]*Intel Corporation*
Contact: {wigginss, ymeng643, prasanna}@usc.edu, mahesh.iyer@intel.com

***Abstract*—Multi-Agent Reinforcement Learning (MARL) is an emerging technology that has seen success in many AI applications. Multi-Actor-Attention-Critic (MAAC) is a state-of-the-art MARL algorithm that uses a Multi-Head Attention (MHA) mechanism to learn messages communicated among agents during the training process. Current implementations of MAAC using CPU and CPU-GPU platforms lack fine-grained parallelism among agents, sequentially executing each stage of the training loop, and their performance suffers from costly data movement involved in MHA communication learning. In this work, we develop the first high-throughput accelerator for MARL with attention-based communication on a CPU-FPGA heterogeneous system. We alleviate the limitations of existing implementations through a combination of data- and pipeline-parallel modules in our accelerator design and enable fine-grained system scheduling for exploiting concurrency among heterogeneous resources. Our design increases the overall system throughput by 4.6× and 4.1× compared to CPU and CPU-GPU implementations, respectively.**

***Index Terms*—Multi-Agent Reinforcement Learning, Hardware Accelerator, Heterogeneous Computing**

I. Introduction

Multi-agent reinforcement learning has seen success in a variety of applications, including swarm systems [1], federated control [2], NoC design [3], etc. Compared to single-agent learning, multi-agent settings introduce partial observability and non-stationarity, which can hinder agents' collective learning performance [4]. As a result, one key optimization focus for MARL algorithm developers is effective inter-agent communication. Multi-Actor-Attention-Critic (MAAC) [5] is a state-of-the-art MARL algorithm. It addresses the communication problem in MARL by adopting attention-based training. The training of MAAC agent policies involves sending embedded messages to a centralized Multi-Head Attention (MHA) mechanism, which allows agents to dynamically select which agents to attend to, thus improving reward convergence in cooperative and competitive settings [6], [7]. This is at the expense of adding complexity to the training function, leading to higher computation cost [8]. In real-world applications of MARL algorithms (e.g., recommendation systems [9] and traffic networks [10]), the agent policies are trained in a centralized manner on a data-center device, leveraging actors collecting data using simulation software on processors.

However, MARL training is a highly time-consuming process. The necessity of training a centralized MHA mechanism alongside several Multi-Layer Perceptrons (MLPs) for actor and critic networks, each with varying compute and memory characteristics, poses unique challenges when optimizing for system throughput: *1. System Resource Utilization:* on emerging heterogeneous platforms, efficient mapping, scheduling and load balancing of tasks to saturate the compute power of the heterogeneous devices in the system is a critical challenge. Current CPU-GPU implementations of MARL [11], [12] simply partition the entire simulation and training phases onto different devices, where the load imbalance between CPU and GPU leads to underutilization of the heterogeneous compute power. *2. Intensive Data Movement:* the attention-based communication learning mechanism involves intricate data aggregation paths and significant data movement. Even if agents are parallelized using data-parallel resources on CPU or GPU, the communication overhead from these operations cannot be trivially hidden. These challenges are not efficiently addressed in current CPU and CPU-GPU implementations of MAAC [11], leading to suboptimal system throughput and poor scalability with increasing number of agents.

CPU-FPGA heterogeneous systems have emerged as popular platforms for accelerating AI workloads [13]–[17]. In this work, we propose a novel acceleration system based on a CPU-FPGA heterogeneous platform to address the challenges discussed above and achieve high-throughput MAAC training. Such a system is naturally suitable for MARL tasks because the data-movement-intensive computations in attention-based training can be improved using a spatial architecture, while the environment simulations placed on CPU processors allow plug-and-play of application-specific software. Our main contributions are:

- We parallelize environment sampling on the CPU with the training pipeline on the FPGA, and further exploit concurrency by partitioning and scheduling the training process onto both the CPU and the FPGA. This improves the system throughput by minimizing device idle times.
- We develop dedicated acceleration modules for the specialized multi-head attention and MLP layers in MAAC

This work was supported by Intel Corporation and in part by the U.S. National Science Foundation (NSF) under grants CNS-2009057 and SPX-2333009.

979-8-3315-3007-5/24/$31.00 ©2024 IEEE
DOI 10.1109/FPL64840.2024.00040

training. We further apply optimizations to maximize system throughput and reduce resource consumption.
- We parameterize our design and propose an efficient design space exploration (DSE) method that returns optimal combinations of key design parameters to generate high-throughput accelerator implementations on any target FPGA. Our DSE runs in linear time and avoids exponential-time exhaustive search over the design parameter choices.
- We implement our design on a CPU-FPGA platform and demonstrate a 4.6× and 4.1× higher system throughput compared to CPU and CPU-GPU implementations, respectively.

II. Background

A. Multi-Actor-Attention-Critic

We consider a partially observable variant of N-agent Markov Games [18], where each agent i receives an observation (o_i) that contains only partial information from a state space $\mathcal{S}$. Each agent i aims to find an optimal policy π_i, denoted as a probability distribution over its action space $\pi_i : \mathcal{S} \rightarrow P(\mathcal{A}_i)$ that maximizes its own total expected reward over time T: $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where γ is a discount factor. MAAC uses a policy-critic approach in training [19], which leverages training of two separate Deep Neural Networks (DNNs) collaboratively for each agent i - one to estimate a value based on input observations and actions (i.e., critic network $Q_i(o, a)$) and another to approximate the agent's policy function (i.e., policy network $\pi_i(o)$).

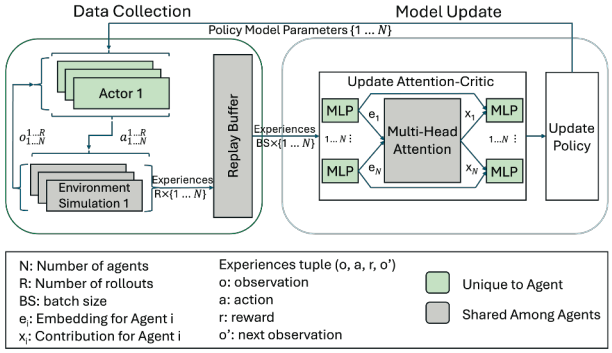

Fig. 1. Training Process of Multi-Actor-Attention-Critic (MAAC)

MAAC involves attention-based communication learning during its critic and policy networks training process. As shown in Figure 1, the training process of MAAC can be viewed as two phases: **(1) Data Collection**: Multiple actors perform parallel inferences on agents' policy networks to interact with the environment, generating and storing experiences (i.e., data points for training) into a global replay buffer $\mathbb{D}$. **(2) Model Update**: Batches of experience are sampled from the global replay buffer to perform stochastic gradient descent (SGD) [20] on all agents' policy and critic networks. The communication learning refers to inference and weight updates of a Multi-Head Attention (MHA), which occurs during critic network training. In the MHA, each agent queries other agents for their observation and action embedding information in order to collaboratively estimate their own value function.

Calculating each agent's Q-value ($i \in 1...N$) involves MLPs unique to each agent, as well as the centralized shared MHA.

$$Q_i^{\psi}(o, a) = f_i(g_i(o_i, a_i), x_i) \tag{1}$$

g_i is a 1-layer MLP embedding function used to calculate embeddings and f_i is a 2-layer MLP. Embeddings from all agents are sent to the MHA to compute a unique contribution message x_i for each agent per head as follows:

$$x_i = \sum_{j \neq i} \left[\text{softmax} \left(\frac{e_j^T W_k^T W_q e_i}{\sqrt{D_k}} \right) h(W_v g_j(o_j, a_j)) \right] \tag{2}$$

h is an activation function. W_k, W_q and W_v are projection matrices that transform embeddings (e_i and e_j) into Keys, Queries and Values [21]. D_k is the dimension of the Keys. Messages for each agent are then concatenated across each head, then sent to each agent's unique 2-layer MLP f_i as shown in Equation 1 to calculate final Q-values. DNNs are trained using SGD for optimization. All critic networks are updated to minimize a joint regression loss function [5].

The computations in MHA used in MAAC are different compared to MHA in image classification or representation models [21] in the following aspects: First, every agent i computes Queries and Values from a different set of embeddings from all agents other than itself (shown as $j \neq i$ in Equation 2), requiring specialized index handling for different agents; Second, instead of sharing the same tensor input for Keys, Queries and Values in [21], the MHA in MAAC takes a one-dimensional observation embedding for the Keys and takes $(N-1)$-dimensional observation-action embeddings as the inputs to the Queries and Values computations. These unique characteristics make the computation of MHA in MAAC memory-bound, where stacking data-parallel resources proves to be inadequate, while leveraging large distributed on-chip SRAM of spatial architectures becomes advantageous for alleviating memory-bound problems.

B. Limitations of Existing MAAC Implementations

CPU-GPU implementations of MAAC [11] offload the training of critic and policy networks to the GPU. The GPU performs full batch data-parallel layer propagations sequentially, moving aggregated results back and forth from GPU global memory. However, the attention-based training process of MAAC consists of various kernels (i.e., linear layers, batch normalization, softmax, scaled-dot product, etc.) with different memory and compute characteristics with intricate data indexing and aggregation paths. Even if agents are parallelized using data-parallel resources on CPU or GPU, the communication overheads from these data movements cannot be trivially hidden, and they increase as the number of agents is scaled up. Moreover, existing CPU and CPU-GPU implementations of MAAC execute the data collection and model update process sequentially, with no exploitation of overlapping these two distinct phases. Parallelizing and balancing these two phases

on heterogeneous systems is crucial for achieving high system resource utilization and throughput.

Figure 2 summarizes the single-training-iteration execution time breakdown across two different environment benchmarks from the widely-used multi-agent particle environment (MPE) [22] on CPU and CPU-GPU platforms, with Data Collection times normalized to 1. We observe that the Model Update phase is a major bottleneck of existing MAAC implementations across both platforms and environment benchmarks. Even if the two phases are fully parallelized, the system utilization would still be low due to the severe load-imbalance causing significant idling of the CPU.

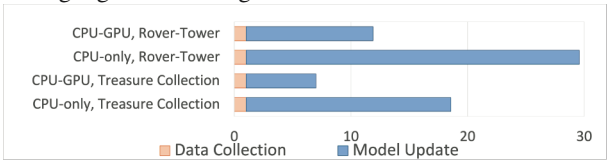

Fig. 2. Normalized Execution Time Breakdown

C. Related Work

MARL is an emerging area where there are limited works for parallelization and acceleration. Most works in Multi-Agent RL focus on optimizing reward convergence through algorithmic methods on novel inter-agent communication mechanisms. This leads to unique challenges in acceleration which are unaddressed in existing literature [23]. In [24], a CPU-FPGA heterogeneous design for Multi-Agent Deep Deterministic Policy Gradient (MADDPG) accelerates the training of agent critic and policy networks, utilizing a ring interconnect for its all-to-all communication of pre-defined static messages. In [25], a centralized controller on FPGA for table-based Q-learning is used to coordinate microcontroller-based agents. [26] introduces a real-time sparse training accelerator for MARL algorithm IC3Net [27] focused on a network pruning system. To our knowledge, our work is the first to accelerate MARL with attention-based inter-agent communication.

III. Heterogeneous Acceleration System Design

A. System Overview

Figure 3 shows the system overview of MAAC on our CPU-FPGA heterogeneous platform. Our acceleration system is composed of a pool of parallel CPU Simulation Threads, a CPU Host Thread that coordinates the necessary data transfers between the CPU and FPGA, an FPGA, and a CPU Training Thread. We perform the Data Collection phase on the CPU Simulation Threads, which deploys general-purpose software [22] that can simulate a wide range of application environments. We deploy R parallel environment simulations for Data Collection. Each one of the N actors holds a unique agent's policy network. The inference processes of all agents sharing an environment are computed sequentially on a CPU Simulation Thread. Experiences tuples are sent to replay buffer $\mathbb{D}$ that resides in CPU DDR memory. The Model Update phase performs SGD (involving Forward Propagations (FP) and Backward Propagations (BP) through all the agent DNNs and centralized Attention modules) using a batch of experiences from $\mathbb{D}$. The FPGA and the CPU Training Thread collaboratively execute the Model Update phase.

To address the challenge of *System Resource Utilization*, we partition the tasks in the Model Update phase and deploy them on both CPU and FPGA. By allocating part of the training process to the CPU, we effectively utilize the resources on both devices. Specifically, as shown in Figure 3, we assign the BP of the policy update on the CPU Training Thread, where new policy weights can be directly used in the Data Collection phase by the CPU Simulation Threads without causing additional PCIe weight transfer overheads. The complete critic update and the FP of the policy update are accelerated on the FPGA. Q-values and policy activations are sent from the FPGA via PCIe to the CPU. Overall, our partitioning technique only introduces a size of $\{\text{batch size} \times N \times (\text{policy activations} + \text{Q-values})\} \approx$ in the magnitude of hundreds of kilobytes additional data traffic compared to the alternative mapping choice of offloading the entire Model Update on the accelerator. The latency overheads from this additional data traffic are trivial compared to the performance gain of load-balancing the computations on CPU and FPGA.

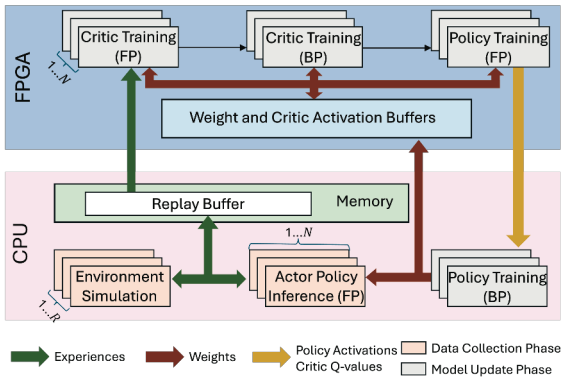

Fig. 3. System Overview

B. Accelerator Design

Figure 4 shows the overall accelerator design of MAAC training on FPGA. It is a pipelined design composed of forward and backward propagations of two types of pipeline stages: (1) Linear Layer stages and (2) Multi-Head Attention (MHA) stages. Each pipeline stage computes a single experience (i.e., a tuple of $\{\text{observation}_i$, action_i, next observation_i, $\text{reward}_i\}$ for all N agents, $0 \leq i < N$) from the batch of BS experiences at a time, where different experiences in a batch are processed in a pipelined concurrent manner. To address the challenge of training with *Intensive Data-Movement*, we allocate on-chip FIFO pipes to directly stream intermediate outputs among agents' embedding layers and the MHA. This ensures 1-cycle amortized latency in embedding collection from each agent before computing attention score. Each linear layer stage is composed of an array of multiplier-accumulator units used to compute matrix-vector multiplication for a given input. Note that a BP linear layer stage also performs gradient aggregation of the corresponding layer (vector outer product).

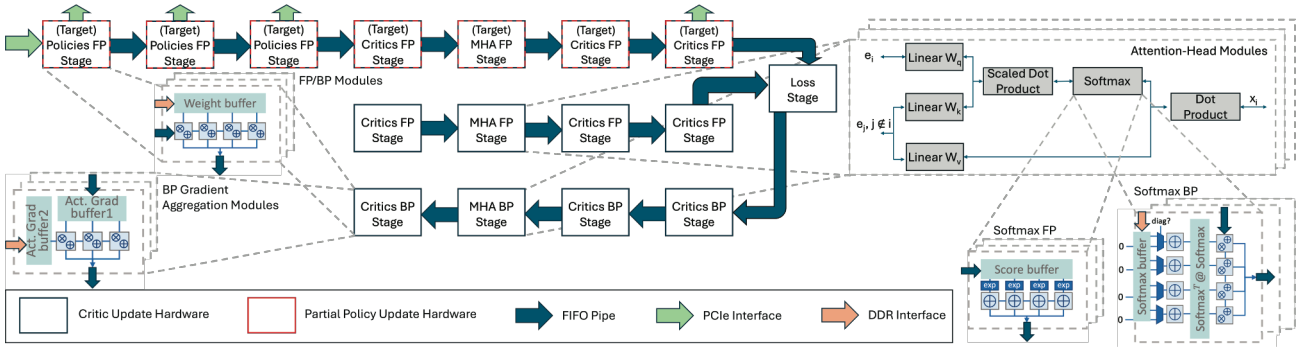

Fig. 4. FPGA Training Architecture

We use a feature-parallel factor PF_F to control the degree of concurrent processing of different output features (i.e., neurons) in a data-parallel manner for each individual stage in the pipeline. A higher feature-parallel factor corresponds to higher throughput for each stage at the cost of higher resource consumption. For linear layer stages, we also exploit agent-level parallelism by dedicating parallel modules for each agent or subset of agents depending on an agent-parallel factor PF_A.

Multi-Head Attention stages are composed of multiple sub-stages, including linear layers for Key, Query, Value generation, (scaled) dot-product layers, and a softmax layer as seen in Figure 4. Each head calculates message x_i for one agent, thus needing N rounds through these sub-stages to calculate messages for all agents in a pipelined manner. We exploit parallelism among attention heads using MHA head-parallel factor PF_H, corresponding to the number of dedicated attention-head modules that execute in parallel. Linear layers for Key and Value generation use systolic arrays that fully exploit parallelism (i.e., spatially unroll the loop) across its input dimension of size $N-1$ (e_j embeddings include embeddings for all agents except for agent i, while e_i denotes the embeddings for only agent i). Furthermore, to mitigate performance degradation resulting from high-latency data movement during the computation of attention scores and softmax functions, we allocate an SRAM buffer to store the complete attention score. This strategy restricts external memory accesses to only retrieving weights, which do not scale up with increasing number of agents.

Given a set of MAAC algorithm hyper-parameters, the parallel factors described above need to be tuned in order to deliver the optimal performance for a target FPGA device. The exploration for optimal parallel factors (PF_A, PF_H, and PF_F for all layers) is later discussed in Section IV. We also leverage hardware and algorithm optimizations for delivering high system throughput with minimal hardware resources:

Partial Hardware Re-use for Policy and Critic: The policy update has identical feature dimensions and compute dataflow as the critic update, but with different DNN weights. Laying out the (FP) training pipeline of these two DNNs spatially would require doubling the number of modules in the update stages, thus severely limiting the parallel factors adoptable within each stage. To alleviate this issue and reduce resource consumption, we opt to reuse the hardware modules used for the target policy and critic FPs in the attention-critic update for the policy update phase. Reusing hardware reduces the cost of allocating additional resources, with a very small additional latency overhead since switching to different weight matrices can be executed within the pipeline.

Intra-Iteration Dependency Relaxation with Inter-Iteration Dependency Preservation: In MAAC, policy and critic networks are trained interactively (the training process of policy networks needs a forward propagation through updated critic networks). We use a "lagged critic" mechanism to facilitate concurrent training of policy and critic networks in the same iteration, similar to parallelizing single-agent DDPG [28]. The implementation of a lagged critic mechanism decouples the training iterations of the policy network from the most recent critic updates. Specifically, we let policy training utilize a slightly outdated critic network, i.e., the critic updated one training iteration behind, to guide and inform the policy network updates. This method effectively mitigates the sequential dependency inherent in the same iteration, promoting a concurrent and synchronized training process for both networks, but still preserves their dependency and weight synchronization between adjacent iterations. Although policy network updates lag the critic updates by one training iteration, its effect is negligible compared to the millions of training iterations that take place in the end-to-end training process [29].

IV. Design Space Exploration

As detailed in Section III-B, the design parameters need to be tuned for delivering optimal performance on a given FPGA device. To perform the design space exploration (DSE), we develop an accurate performance model to estimate the impact of these design parameters (agent-parallel factor PF_A, MHA head-parallel factor PF_H, and feature-parallel factors $PF_F^i, i \in \mathbb{S}$, where $\mathbb{S}$ represents the set of all pipeline stages performing FP and BP through linear and MHA layers), and constraints such as the available SRAM, DSPs, etc. on the performance of the design.

The objective of DSE is to maximize the system throughput (defined in Equation 6), which is inversely proportional to the total latency of performing a batched gradient update of the networks for all agents on the FPGA. Therefore, we aim to

minimize the total latency of the pipeline accelerator design for processing a batch of BS experiences:

$$T_{\text{total}} = \max_{i \in \mathbb{S}}(T_{\text{stage } i}) \times (\text{num\_stages} + BS) \quad (3)$$

Specifically, based on our accelerator design, assuming there are N agents in the MARL application, the pipeline stage latency T_{stage} for a linear layer with input feature size F_{in} and output feature size F_{out} in FP and BP are:

$$T_{\text{stage } i=\text{linear}} = \max\{\frac{(N/PF_A) \times F_{in} \times (F_{out}/PF_F^i)}{freq}, T_{LoadW}\} \quad (4)$$

where T_{LoadW} denotes the latency for loading weights from external memory computed as $\frac{F_{in}F_{out}}{\text{bandwidth}}$.

The pipeline stage latency T_{stage} for a MHA layer with input feature size F_{in}, output feature size F_{out}, and H attention heads can be derived by multiplying the number of agents processed and pipeline 4-sub-stage fill/drain with the longest latency in the sub pipeline stage of a MHA module (i.e., the key and value encoding stage):

$$T_{\text{stage MHA}} = (N+4) \times \max\{\frac{F_{in} \times (F_{out}/PF_H)}{freq}, T_{LoadW}\} \quad (5)$$

For all (FP and BP) linear layer and MHA stages, we monitor the DSP usage and on-chip SRAM buffer requirements. These metrics, parameterized by the parallel factors, are assessed to ensure they remain within resource limits.

To accomplish the objective of minimizing T_{total} (Equation 3) is essentially to minimize the bottleneck stage, i.e., the stage with the longest latency to complete. This optimization problem is thus equivalent to finding the combinations of PF_A, PF_H, and $PF_F^i \forall i \in \mathbb{S}$ that ideally load balances all the pipeline stages. An exhaustive search over the entire design space would lead to an $O(N \times H \times F^{|\mathbb{S}|})$ complexity. In this work, we propose an efficient heuristic to identify optimal design parameter choices in $O(|\mathbb{S}|+H+N)$ time complexity, as shown in Algorithm 1. Our algorithm first determines the computation requirement ratios among all linear layers, and fixes the relative ratio among assigned parallel resources along the feature dimensions PF_F; Then, it proceeds to balance the pipeline stage latencies between the bottleneck linear stage and the MHA stage by tuning PF_H (which controls the MHA stage latency) and PF_A (which controls the linear layer latencies) in an interleaving manner until reaching resource limit. Finally, it fine-tunes the PF_F based on available resources.

Our DSE generalizes our design to support arbitrary hyper-parameters of different MAAC applications, and is able to quickly generate optimal designs on different FPGA devices.

V. Evaluation

A. Experiment Setup

Metrics: The main metric optimized by an acceleration system for MARL is the system throughput in terms of number of Agent-gradient-updates Per Second (APS):

$$APS = \frac{\text{number of agents} \times \text{batch size}}{T_{iteration}}, \quad (6)$$

where $T_{iteration}$ is the single-training-iteration execution time. For our CPU-FPGA acceleration system, $T_{iteration} = \max(T_{DC}^{CPU} + T_{MU}^{CPU}, T_{MU}^{FPGA})$. T_{DC} and T_{MU} are the single-training-iteration execution times of the Data Collection and Model Update phases, respectively. T_{MU}^{FPGA} is obtained by implementing the design guided by our DSE.

Algorithm 1 Design Parameters Search

Inputs: Layer dimensions and number of operations (#ops) in the set of all layer propagations $\mathbb{S} = \mathbb{S}_{\text{linear}} \cup \mathbb{S}_{\text{MHA}}$
Initialize $PF_A \leftarrow 1, PF_H \leftarrow 1$
▷ Step 1: Balance FP/BP modules for linear layer propagations
Find the linear layer with the minimal #ops in $\mathbb{S}_{\text{linear}} \rightarrow$ min_linear, Set $PF_F^{\text{min\_linear}} \leftarrow 1$
for all other layer propagations $i \in \mathbb{S}_{\text{linear}}, i \neq$ min_linear **do**
    Set $PF_F^i \leftarrow \lceil \frac{\#\text{ops}(i)}{\#\text{ops(min\_linear)}} PF_F^{\text{min\_linear}} \rceil$
▷ Step 2: Balance MHA modules with linear layer propagations
Find the linear layer stage with the longest latency $T_{\text{stage linear}}$ based on all $PF_F \rightarrow$ max_linear
while $T_{\text{stage MHA}} \leq T_{\text{stage max\_linear}}$ and $PF_H < H$ **do**
    Increment PF_H and update $T_{\text{stage MHA}}$
while synthesized design is valid wrt all resource bounds **do**
    Increment PF_A; update $T_{\text{stage } i}$ and $DSP_{\text{stage i}} \forall i \in \mathbb{S}_{\text{linear}}$
    if $\max_{i \in \mathbb{S}_{\text{linear}}}\{T_{\text{stage } i}\} < T_{\text{stage MHA}}$ and $PF_H == H$ **then**
        break; ▷ Increasing PF_A no longer improves speed
    if $\max_{i \in \mathbb{S}_{\text{linear}}}\{T_{\text{stage } i}\} < T_{\text{stage MHA}}$ and $PF_H < H$ **then**
        Increment PF_H
    if $PF_A == N$ **then**
        Increment all $PF_F^i \forall i \in \mathbb{S}_{\text{linear}}$
Outputs: Design parameters $PF_A, PF_H, PF_F^i \forall i \in \mathbb{S}_{\text{linear}}$

Evaluation Environment: We evaluate our implementation using the Rover Tower simulation from the MPE. Different benchmarks share similar environment simulation times, observation and action dimensions, so the performance observations in our experiment is representative across different benchmarks. Our implementations use a batch size of 1024, four attention heads, and DNN hidden dimensions of 128 for training, consistent with the original MAAC hyper-parameter specifications [11].

We compare our CPU-FPGA implementation against two different setups: CPU-only homogeneous platform and CPU-GPU heterogeneous platform. The specifications for the devices used in each platform are detailed in Table I. CPU and CPU-GPU implementations use PyTorch to implement DNN training. We use the oneAPI development flow to implement our FPGA kernels [30].

TABLE I
Platform Specifications

Platform	CPU Intel Xeon Gold 6326	GPU NVIDIA RTX 3090	FPGA Intel DE10 Agilex 7
Technology	10 nm	8 nm	10 nm
Frequency	2.9 GHz	1.7 GHz	250 MHz
Memory Bandwidth	171 GB/s	936 GB/s	85 GB/s
On-Chip Memory	24 MB L3 Cache	6 MB L2 Cache	64 MB
Peak Performance	537 GFLOPS	35.6 TFLOPS	38 TFLOPS

FPGA Accelerator Setup: We perform DSE (Algorithm 1) to obtain the design parameters of our target FPGA device. We run the design parameter search algorithm on the Intel Xeon Gold 6326 CPU, which only takes under 2 seconds to generate the design parameters for each experiment on the target FPGA. The optimal design point to load-balance the system for our target hardware is determined at $PF_A = 1$, $PF_H = 1$, and

PF_F for each stage ranging from $1-16$ across all layer propagations. Table II describes our resource utilization for the Rover Tower simulation with varying numbers of agents.

TABLE II
FPGA ACCELERATOR RESOURCE UTILIZATION

Parallel Factors (PF_A, PF_H, PF_F)	ALUTs	DSPs	RAMs	MLABs
(1, 1, 1-16)	76-83%	19-23%	71-84%	52-56%

B. MAAC Training Latency Breakdown

Figure 5 shows a timeline with latency breakdown of the various Data Collection and Model Update tasks assigned onto the CPU and FPGA for a $4-$agent scenario. The figure highlights our performance gain from two perspectives: (1) $3.8\times$ speedup in single-training-iteration latency due to our novel spatial architecture that exploits the compute and memory characteristics of each stage, processing each sample of the batch in a pipelined manner; (2) better exploitation of heterogeneous resource concurrency by mapping the backward propagation of policy updates onto the CPU. Note that even if we parallelize the CPU-GPU system by enabling concurrent data collection and training, we still observe $3.6\times$ (compared to $3.8\times$ with no overlap) higher performance in terms of single-training-iteration latency on our CPU-FPGA system, directly indicating higher APS. Although we incur additional PCIe latencies due to sending activations and Q-values compared to GPU training, the relatively small overheads can be completely overlapped with computation by our heterogeneous system as shown in Figure 5. The behavior in Figure 5 is generalizable to varying number of agents and hyper-parameters. With more agents, the ratio between PCIe transfer and compute times are still the same, so the overlaps shown in Figure 5 still apply. With varying hyper-parameters, such as increased batch size and hidden dimensions for policy and critic networks, both network updates will have higher latency, and the observations in Figure 5 remain the sam.e

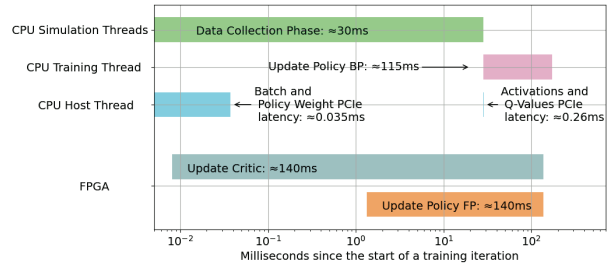

Fig. 5. Latency breakdown of a single training iteration

C. System Throughput & Scalability

The bar plots in Figure 6 show an APS comparison between all three platforms with a varying number of agents. Our CPU-FPGA accelerator outperforms both CPU and CPU-GPU systems across all agent scenarios, with up to $4.6\times$ and $4.1\times$ higher system throughput, respectively.

We demonstrate consistent speedup with scalability compared to the baseline platforms. This is evident as the APS shows minimal to no throughput degradation with increasing number of agents. On the CPU, as the number of agents increases, a larger amount of communication overheads lower the throughput. Both GPU and FPGA-based implementation demonstrate better scalability than CPU, while our FPGA design shows consistent speedup due to spatial architecture design that streams MHA and linear layer results in a near-memory fashion. A more powerful FPGA device would enable our design to further increase its performance gap over the other platforms, where higher parallel factors would be discovered from our DSE algorithm.

We use effective resource utilization (the line plots in Figure 6), defined as the achieved throughput divided by the theoretical peak throughput using the allocated compute resources, to demonstrate the speedup from the FPGA design compared to the GPU implementation. The effective resource utilization of the FPGA ranges from 56% to 68% depending on the number of agents compared to the GPU's 11% to 20% utilization. This low utilization is attributed to the full-batch layer propagation scheme of MAAC training on GPU, where it is unable to saturate the large number of available data-parallel CUDA cores. The intricate datapath of MAAC training is suited for FPGA, with its rich set of logic resources that can be tailored toward MAAC's various compute and memory-intensive operations.

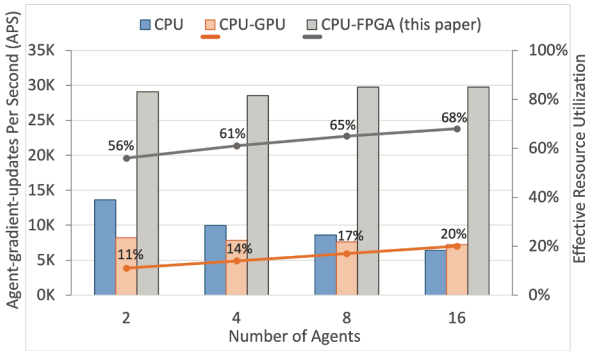

Fig. 6. APS comparison across CPU, CPU-GPU and CPU-FPGA systems

VI. CONCLUSION

We developed the first work to accelerate MARL with attention-based communication. We proposed a mapping on a CPU-FPGA heterogeneous system along with DSE for the FPGA accelerator design, which led to speedups of up to $4.6\times$ compared to CPU and CPU-GPU baselines. Our work showcases promising opportunities for adopting FPGA and spatial architectures in the field of multi-agent systems with sophisticated communication adaptation. There are multiple future research directions to explore. For instance, the development of a general-purpose scheduling algorithm that automatically assigns training tasks (MHA and MLP layers) onto heterogeneous devices based on the task dependency graph among training agents, as well as the development of scalable distributed FPGA systems tailored to support multi-agent systems.

REFERENCES

[1] M. Hüttenrauch, A. Šošić, and G. Neumann, "Guided deep reinforcement learning for swarm systems," *arXiv preprint arXiv:1709.06011*, 2017.

[2] S. Kumar, P. Shah, D. Hakkani-Tur, and L. Heck, "Federated control with hierarchical multi-agent deep reinforcement learning," *arXiv preprint arXiv:1712.08266*, 2017.

[3] N. Anantharajaiah, Y. Xu, F. Lesniak, T. Harbaum, and J. Becker, "Dream: Distributed reinforcement learning enabled adaptive mixed-critical noc," in *2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2023, pp. 1–6.

[4] 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.

[5] S. Iqbal and F. Sha, "Actor-attention-critic for multi-agent reinforcement learning," in *International conference on machine learning*. PMLR, 2019, pp. 2961–2970.

[6] Z. Zhu, S. Wan, P. Fan, and K. B. Letaief, "An edge federated marl approach for timeliness maintenance in mec collaboration," in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2021, pp. 1–6.

[7] Z. Liang, J. Cao, S. Jiang, D. Saxena, J. Chen, and H. Xu, "From multi-agent to multi-robot: A scalable training and evaluation platform for multi-robot reinforcement learning," *arXiv preprint arXiv:2206.09590*, 2022.

[8] S. Wiggins., Y. Meng., R. Kannan., and V. Prasanna., "Characterizing speed performance of multi-agent reinforcement learning," in *Proceedings of the 12th International Conference on Data Science, Technology and Applications - DATA*, INSTICC. SciTePress, 2023, pp. 327–334.

[9] M. M. Afsar, T. Crump, and B. Far, "Reinforcement learning based recommender systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–38, 2022.

[10] J. Lee, J. Chung, and K. Sohn, "Reinforcement learning for joint control of traffic signals in a transportation network," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 2, pp. 1375–1387, 2019.

[11] "Maac implementation on cpu-gpu platform," 2023. [Online]. Available: https://github.com/shariqiqbal2810/MAAC

[12] "Graph convolutional marl implementation on cpu-gpu platform," 2023. [Online]. Available: https://github.com/PKU-RL/DGN

[13] F. Kreß, J. Hoefer, T. Hotfilter, I. Walter, E. M. El Annabi, T. Harbaum, and J. Becker, "Automated search for deep neural network inference partitioning on embedded fpga," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2022, pp. 557–568.

[14] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on fpga platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[15] T. C. Chau, X. Niu, A. Eele, J. Maciejowski, P. Y. Cheung, and W. Luk, "Mapping adaptive particle filters to heterogeneous reconfigurable systems," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 4, pp. 1–17, 2014.

[16] T. Santos, J. Bispo, and J. M. Cardoso, "A cpu-fpga holistic source-to-source compilation approach for partitioning and optimizing c/c++ applications," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 320–322.

[17] L. Stornaiuolo, M. Santambrogio, and D. Sciuto, "On how to efficiently implement deep learning algorithms on pynq platform," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2018, pp. 587–590.

[18] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine learning proceedings 1994*. Elsevier, 1994, pp. 157–163.

[19] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[20] S.-i. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] 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.

[23] S. Wiggins, Y. Meng, R. Kannan, and V. Prasanna, "Evaluating multi-agent reinforcement learning on heterogeneous platforms," in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications V*, vol. 12538. SPIE, 2023, pp. 493–497.

[24] ——, "Accelerating multi-agent ddpg on cpu-fpga heterogeneous platform," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–7.

[25] G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Re, A. Ricci, and S. Spano, "An fpga-based multi-agent reinforcement learning timing synchronizer," *Computers and Electrical Engineering*, vol. 99, p. 107749, 2022.

[26] 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.

[27] 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.

[28] 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.

[29] Y. Meng, C. Zhang, and V. Prasanna, "Fpga acceleration of deep reinforcement learning using on-chip replay management," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. CF '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 40–48. [Online]. Available: https://doi.org/10.1145/3528416.3530227

[30] "Intel oneapi." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/ overview.html