Wiley
IET Computers & Digital Techniques
Volume 2024, Article ID 7855250, 13 pages
https://doi.org/10.1049/2024/7855250

Research Article

A FPGA Accelerator of Distributed A3C Algorithm with Optimal Resource Deployment

Fen Ge,[1,2] **Guohui Zhang**,[1,2] **Ziyu Li**,[1,2] **and Fang Zhou**[1,2]

[1]*College of Integrated Circuits, Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[2]*Key Laboratory of Aerospace Integrated Circuits and Microsystem, Ministry of Industry and Information Technology, Nanjing, China*

Correspondence should be addressed to Fen Ge; gefen@nuaa.edu.cn

Received 25 November 2023; Revised 7 April 2024; Accepted 3 May 2024; Published 27 May 2024

Academic Editor: Roger Woods

The asynchronous advantage actor-critic (A3C) algorithm is widely regarded as one of the most effective and powerful algorithms among various deep reinforcement learning algorithms. However, the distributed and asynchronous nature of the A3C algorithm brings increased algorithm complexity and computational requirements, which not only leads to an increased training cost but also amplifies the difficulty of deploying the algorithm on resource-limited field programmable gate array (FPGA) platforms. In addition, the resource wastage problem caused by the distributed training characteristics of A3C algorithms and the resource allocation problem affected by the imbalance between the computational amount of inference and training need to be carefully considered when designing accelerators. In this paper, we introduce a deployment strategy designed for distributed algorithms aimed at enhancing the resource utilization of hardware devices. Subsequently, a FPGA architecture is constructed specifically for accelerating the inference and training processes of the A3C algorithm. The experimental results show that our proposed deployment strategy reduces resource consumption by 62.5% and decreases the number of agents waiting for training by 32.2%, and the proposed A3C accelerator achieves 1.83× and 2.39× improvements in speedup compared to CPU (Intel i9-13900K) and GPU (NVIDIA RTX 4090) with less power consumption respectively. Furthermore, our design shows superior resource efficiency compared to existing works.

1. Introduction

Reinforcement learning (RL) is a machine learning algorithm that maximizes long-term rewards by interacting with the environment so that the agent learns optimal behavioral strategies. By introducing the deep neural network (DNN), deep reinforcement learning (DRL) greatly improves the learning ability of algorithms for high-dimensional inputs. The asynchronous advantage actor-critic (A3C) algorithm [1] is widely regarded as one of the most effective and powerful algorithms among various DRL algorithms, which is commonly applied in intelligent control systems in various fields, including autonomous driving [2, 3], unmanned aerial vehicle [4, 5, 6], robotics [7, 8, 9], and gaming [10, 11].

However, the introduction of DNN has led to a significant increase in the time and cost associated with model training. The substantial time and computational costs severely hinder the widespread adoption of DRL. Furthermore, the distributed asynchronous learning framework significantly increases the algorithmic complexity and computational demands of the A3C algorithm. Typically, it is deployed on multicore GPUs for computation, leading to relatively high energy consumption. Additionally, when using a GPU, the mix of small DNN architectures, small training batch sizes, and contention for the GPU for both inference and training can lead to a severe underutilization of the computational resources [12]. Field programmable gate array (FPGA) has proven to be excellent hardware acceleration platforms for computationally intensive algorithms (DNN, DRL, etc.) due to its programmability, high parallelism, and low power consumption [13]. Consequently, there has been a growing focus on using FPGA platforms to accelerate DRL algorithms [14, 15, 16, 17].

This paper aims to accelerate A3C algorithm for both inference and training on FPGA. Compared to previous research, we focus on how to improve the resource utilization of the distributed DRL accelerator. The main contributions of this paper can be summarized as follows:

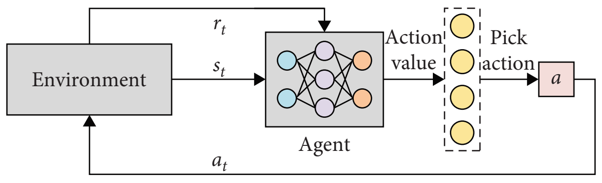

Figure 1: The principle of RL.

(1) A hardware deployment strategy for distributed DRL algorithm is proposed. This strategy focuses on optimizing three key aspects: the deployment of agents, the deployment of computing units, and the allocation of hardware resources. Its goal is to achieve a balanced interaction, inference, and training time, minimize idle waiting delays of computing units, and enhance the overall utilization of hardware device.

(2) A FPGA-based A3C algorithm accelerator architecture is proposed. We design dedicated acceleration units for different computational stages of the A3C algorithm and allocate mutually independent storage space for these units to improve data access efficiency. In addition, we optimize the hardware structure and control for different computational processes to further increase the speed and parallelism.

2. Background and Related Work

2.1. Deep Reinforcement Learning. In RL, an agent interacts with the environment over a discrete number of time steps. As shown in Figure 1, at each time step t, the agent obtains the current state s_t from the environment and selects an action a_t from the action space A according to its policy π. After executing the selected action a_t in the environment, the agent receives the reward value r_t and the next state s_{t+1}.

The aim of RL is to maximize cumulative rewards by learning a certain strategy through the interaction between the agent and the environment. RL can either learn a strategy directly or optimize the strategy indirectly by obtaining a value function that evaluates the quality of each state-action pair. DRL can obtain efficient decision-making strategies in complex, high-dimensional environments by integrating deep learning technique, and the RL framework. Typically, DRL employs DNN to approximate the value functions in RL and selects actions based on the outputs of DNN.

The computation of DRL consists of two main tasks: inference and training. In the inference task, training data are generated through interaction with the environment. The training task uses the data generated from the inference task to update the parameters of the DNN, thus improving the performance of the agent. The inference and training tasks are interrelated and together facilitate the learning and optimization of the agent in complex environments.

2.2. A3C Algorithm. The A3C algorithm, which builds upon the actor critic algorithm [18], employs a distributed training approach to simultaneously deploy multiple agents for interacting with the environment. This method enhances the speed and diversity of experience acquisition, as illustrated in Figure 2. The A3C algorithm creates a central agent as a template for the distributed agents and duplicates it N times, resulting in N local agents.

Under the distributed training framework, the computation of each agent is independent of each other and can be performed simultaneously. Additionally, the A3C algorithm uses an asynchronous parameter updating mechanism to prevent access conflicts that may arise when multiple local agents concurrently copy or update the parameters of the central agent.

2.3. The Computational Flow of A3C Algorithm. The complete computational flow of A3C algorithm can be divided into four parts: parameter synchronization (PS), inference (Inf), environment interaction (Env), and training (Train), as shown in Figure 3. The training process is further divided into four stages: loss gradient calculation (Loss GC), backpropagation (BP), parameter gradient calculation (Parameter GC), and parameter update.

The computational flow of the A3C algorithm is as follows: First, the central agent copies (the process of copying parameters is called PS) its network parameters (i.e., central θ) to a local agent (i.e., local θ). Subsequently, the computation of inference begins (i.e., Inf), and the process generates the output feature map based on the input feature map and the DNN parameters of each layer (i.e., θ). The agent then applies the action selected by Inf to the environment and receives the corresponding reward value (i.e., Env). When $T_{\max}$ inferences are performed, the training computation starts. The first step of training is Loss GC, which computes how much the input feature mapping values should be changed to optimize the objective function (i.e., gradients of the input feature map). BP propagates the gradient of feature map computed by Loss GC from the last layer of the network to the first. The goal of Parameter GC is to compute the gradient of the DNN parameters (i.e., $d\theta$). After obtaining the gradients of the DNN parameters, the central agent updates the parameters using an optimization algorithm (e.g., RMSProp).

2.4. Related Work. With the development of DRL algorithms, the training method of DRL is transformed from single agent to multiple agents. Based on the number of agents working in the accelerated DRL algorithm, we classify DRL accelerators into single-agent DRL accelerators represented by the DQN algorithm [19] and multi-agent DRL accelerators represented by the distributed DRL algorithm.

Traditional single-agent acceleration work focuses on tabular RL algorithms acceleration [20, 21]. However, the key component of tabular RL, the Q table, is prone to becoming excessively large which prevents tabular RL from handling complicated tasks. To overcome this challenge, works [22, 23, 24, 25] designed dedicated acceleration architectures for different DRL algorithms, respectively. Nevertheless, all these works only support the inference computation in DRL algorithms and cannot be applied to the on-chip training process of the algorithms.

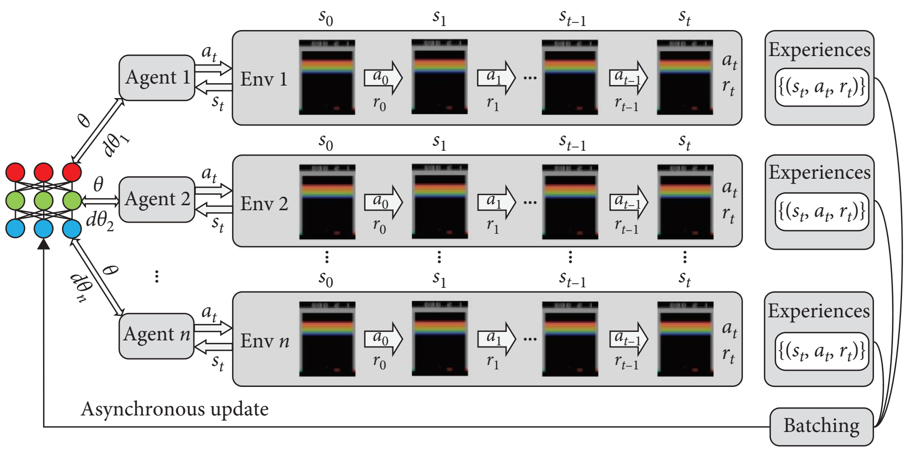

FIGURE 2: The principle of A3C algorithm.

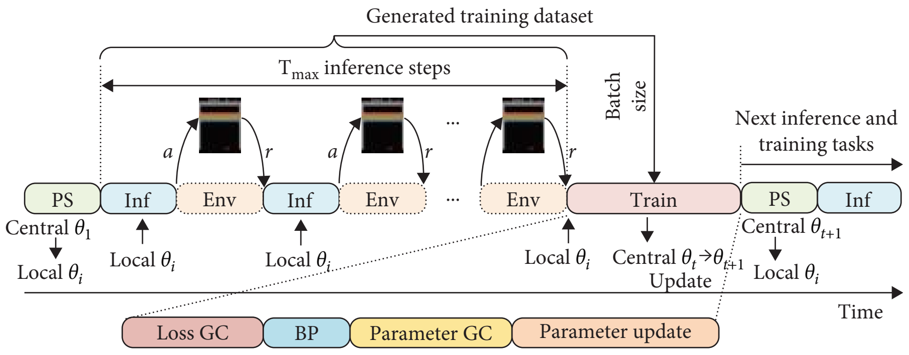
Generated training dataset
T_{max} inference steps
Batch size
Next inference and training tasks
PS
Inf
Env
Train
Central θ_1
Local θ_i
Central $\theta_t \to \theta_{t+1}$ Update
Central θ_{t+1}
Time
Loss GC
BP
Parameter GC
Parameter update

FIGURE 3: The computational flow of A3C algorithm.

Work [26] implemented the DRL inference and training process on-chip by configuring the computational units into different modes. Work [27] implemented a high-throughput PPO accelerator based on a systolic-array architecture coupled with a novel memory-blocked data layout, targeting both phases of the algorithm. Further, work [28] proposed a quantization-aware training algorithm in fixed point. It is the first DRL hardware accelerator that supports both inference and training with dual bit-precision in fixed-point for DRL. Although single-agent DRL accelerators, represented by [26, 27, 28], efficiently achieve computational acceleration of algorithms for inference and training. However, these architectures cannot be migrated to more advanced multi-agent DRL algorithms.

With the improvements to single-chip computing capability, on-chip distributed RL acceleration has been proposed recently [29, 30]. However, work [29] did not conduct multiple agents on-chip synchronization, restricting the learning speed of RL model. Work [30] instead only focuses on the interconnection part of the different agents but focuses little on the acceleration architecture of single agent. Work [31] proposed a hardware accelerator (DARL) for supporting single-agent and multi-agent DRL algorithms. However, DARL ignores the resource wastage problem caused by the distributed training characteristics of multi-agent DRL algorithms and the resource allocation problem caused by the imbalance between the computational amount of DRL algorithms' inference and training, which makes the main resource consumption of accelerator show an almost linear increase with the growth in the number of agents. Work [32] presented iSwitch, an in-switch acceleration solution that moves the gradient aggregation from server nodes into the network switches, thus they can reduce the number of network hops for gradient aggregation. However, work [32] still did not focus on the resource optimization that should be considered for hardware acceleration of distributed DRL algorithms.

To overcome these challenges, we propose a hardware deployment strategy designed for distributed DRL algorithm. The strategy focuses on how to optimize the resource wastage caused by the growth in the number of agents in distributed DRL algorithms and balance the resource consumption of training and inference. Then, we design an architecture for the A3C algorithm based on the proposed strategy to accelerate the inference and training of the algorithm.

3. The Hardware Deployment Strategy for Distributed DRL Algorithm

3.1. The Deployment Strategy for the Number of Distributed Agents. The A3C accelerator can deploy more agents to reduce the computational latency caused by delays in agent interaction and improve the efficiency of computing resource

Table 1: Abbreviation list.

Notation	Meaning
N_e	The number of experience saturated agents
N_r	The number of reward saturated agents
N_l	The number of agents limited by hardware resources
N_o	The optimal number of deployed agents
t_e	The iteration time of the state in the environment
t_i	The interaction time between host computer and accelerator
t_f	The computation time for forward inference
t_b	The training time for backward propagation
t_w	The waiting time due to all resources being occupied
t_s	The total time consumed to complete all exploration
L	The maximum number of explorations
$T_{\max}$	The upper limit of inference
$t_{\min}$	The minimum time to complete all exploration tasks, related to the number of agents
t_r	The real time to complete all exploration
M, M^*	The computational parallelism of accelerators, the optimal computational parallelism of accelerators
η, η^*	The ratio of training resources to inference resources, the optimal ratio of training resources to inference resources

utilization. However, an increase in the number of agents results in higher resource overhead, which may exceed the limitations of most embedded platforms. In this paper, we propose a methodology for determining the optimal number of distributed agents. It quantifies the advantages contributed by each newly added agent to assess its impact on reducing the training cycle of the A3C agent.

We identify three criteria to determine the optimal number of distributed agents: the number of experience saturated agents N_e (the notations used in our proposed deployment strategy here and below are explained in Table 1), the number of reward saturated agents N_r, and the maximum number of agents N_l that can be deployed on hardware devices. A unified deployment strategy is proposed to determine the optimal number of agents N_o based on the numerical relationship among N_e, N_r, and N_l, as presented in Equation (1):

$$N_o = \begin{cases} N_e & N_l > N_e > N_r \\ N_r & N_l > N_r > N_e \\ N_l & N_e \geq N_r > N_l \end{cases}. \tag{1}$$

(1) $N_l > N_e > N_r$: The optimal number of agents in this situation is N_e. When the number of agents reaches N_r, the experience contributed by the newly added agents effectively utilizes the computing resources of the accelerator, thereby reducing agent calculation waiting time and improving the exploration speed of A3C agents per second.

(2) $N_l > N_r > N_e$: The optimal number of agents in this situation is N_r. When the number of agents reaches N_e, the experience introduced by the newly added agents enhances the diversity of training samples. This enables A3C agents to achieve higher cumulative rewards within the same exploration steps, ultimately accelerating the convergence speed of A3C agents.

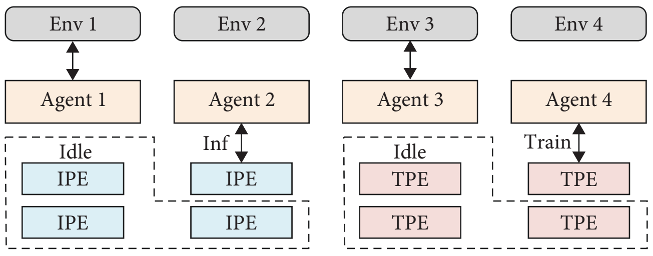

Figure 4: The idle issue.

(3) $N_e \geq N_r > N_l$: The optimal number of agents in this situation is N_l. Due to the limited number of agents that a hardware device can deploy, it is necessary to maximize the number of agents deployed to improve the utilization of computing resources and achieve the best acceleration performance.

3.2. The Deployment Strategy for the Number of PEs. The distributed nature of the A3C algorithm means that some agents may be involved in the inference while others are engaged in the training or interaction process. If the number of deployed inference processing engines (IPE) and training processing engines (TPE) in the computational unit matches the number of agents, it can result in the idle issue of computational resources, as illustrated in Figure 4. In the figure, IPE and TPE represent the hardware resources used by a local agent to complete the inference or training, respectively.

We model the computational process of the A3C algorithm to determine an appropriate level of PE parallelism. The number of agents is denoted as N, and the number of deployed IPEs and TPEs is presented as M. We define the iteration time of the state in the environment as t_e, the data transfer time between the accelerator and the host computer environment as t_i, the computation time for forward inference as t_f, and the training time for backward propagation as

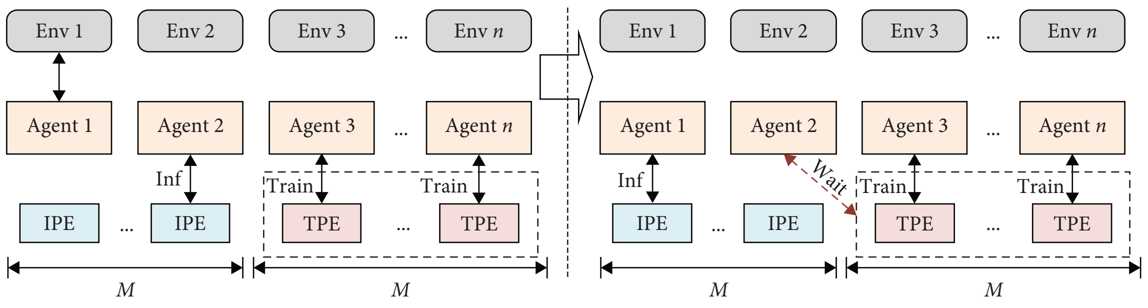

Figure 5: The issue of congestion.

t_b. The accelerator can support M agents performing inference and training computation simultaneously. If all PEs are occupied, other agents must wait for their turn, resulting in a waiting time defined as t_w.

When the upper limit of inference steps is set at $T_{\max}$, and the maximum number of exploration tasks is L, the time required for the agent to complete all exploration tasks can be expressed as $t_s = L \times ((t_e + 2t_i + t_f)T_{\max} + t_b)$. In the ideal case, the shortest time for the A3C algorithm to complete L rounds of tasks is $t_{\min} = t_s/N$. Nevertheless, the presence of waiting time and the uneven distribution of computational tasks among threads result in a real execution time t_r that exceeds $t_{\min}$.

Based on the above modeling, we propose a deployment strategy for the number of PEs as follows: We adjust the number of PEs to make the t_r change while ensuring that the total resource cost of the accelerator remains unchanged. By recording the relationship between M and t_r, the number of PEs corresponding to the shortest real execution time t_r can be found, denoted as M^*.

3.3. The Strategy for Reallocation of Resources. The training process of A3C algorithm has more computational tasks compared to the inference process. Therefore, at the same computational parallelism, the training must take more time than the inference. This will also bring a new congestion problem to the A3C accelerator, as shown in Figure 5. In order to solve this problem, we propose a reallocation strategy for the accelerator computing resources, which reallocates the resources of the inference module and the training module without changing the total computing resources of the accelerator, so as to reduce the training latency of the agents.

The deployment strategy for the number of PEs aims to determine the optimal number of PE denoted as M^*. We obtain the real execution time t_r for different numbers of PEs by varying the computational parallelism of inference and training. Our proposed reallocation strategy of computational resources is to derive the ratio η of training resources to inference resources based on a parallelism of M^*. Then, we gradually reduce the inference computational resources and use that part of the resources in the training computation. Therefore, we can get the relationship between t_r and η under different resource allocation ratios, so as to find the resource allocation ratio η^* corresponding to the shortest execution time t_r.

4. The Hardware Architecture of A3C Accelerator

4.1. System Architecture. We first divide the computational task into five parts: forward inference computation, loss gradient computation, backpropagation computation, parameter gradient computation, and parameter update computation. Then, an FPGA-based A3C algorithm accelerator architecture is proposed, as shown in Figure 6. It is provided with an environment by a host computer (PC) and computational acceleration for inference and training of agents is completed by FPGA.

We utilize N independent virtual environments in the host computer for supporting the learning process of N agents. On the FPGA accelerator, three computational units are designed for computing forward inference, backpropagation, and parameter gradient. In addition, the loss gradient computation unit (LGCU) is deployed for computing the gradient of the output layer and the RMSProp Unit is deployed for parameter updating.

The execution process of proposed accelerator is as follows: The host computer program extracts the environment state value s_t at time t and transfers it to the memory unit of the FPGA accelerator. Next, the forward computation unit (FCU) maps the input s_t into the policy output π (s_t; θ) and the value output v (s_t; θ). Finally, the policy output is fed to the environment corresponding to the thread to complete the interaction. When the time step t reaches $T_{\max}$, the LGCU retrieves the forward computation results from the memory unit and computes the loss gradient of the actor network and the critic network. Afterward, the loss gradient of the output layer is propagated layer by layer to the input layer by the backward computation unit (BCU), while the gradient value $d\theta$ of the parameter is obtained by the parameter gradient computation unit (PGCU). When the gradient calculation of the parameters is completed, the parameter update of the central agent is done by the RMSProp Unit.

4.2. The Proposed FCU. Each forward processing engine (FPE) in the FCU is responsible for performing the forward propagation computation of the neural network. It includes the multiply-accumulate operation between the input feature maps and the weights, the computation of ReLU, the computation of Softmax for actor network and the computation of linearity for critic network. In the above calculations, the multiply-accumulate calculation and the calculation of the

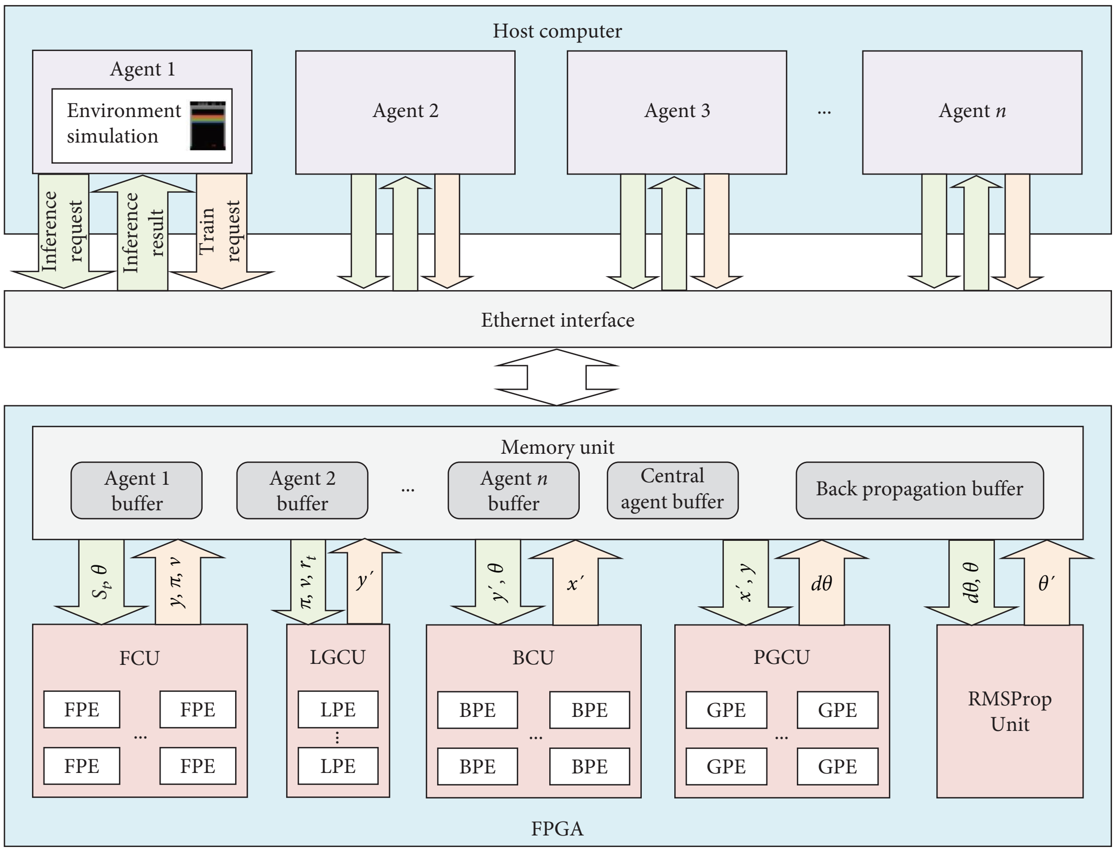

Figure 6: System architecture.

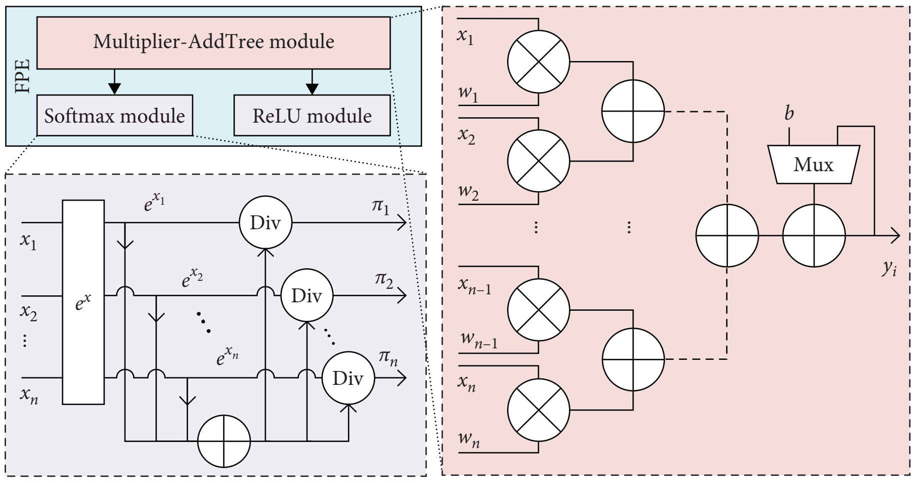

Figure 7: The structure of FPE.

linearity can be considered as matrix multiplication operations, while the calculation of the ReLU and the calculation of Softmax are different activation methods. Therefore, each FPE is divided into three submodules, which correspond to the above computational process, and its structure is shown in Figure 7.

In the designed FPE, the Multiplier-AddTree module performs the parallel computation of n weights and input nodes. In the A3C algorithm, the actor network and the critic network share all neural network parameters except for the output layer, and each shared layer uses ReLU as the activation function. Therefore, only the Multiplier-AddTree module and the ReLU module are needed to compute these shared layers.

When calculating the output layer, it is necessary to use the Softmax module and the Multiplier-AddTree module to complete the calculation. The Multiplier-AddTree module completes the computation of the output of the critic network, while the Softmax module is used to complete the

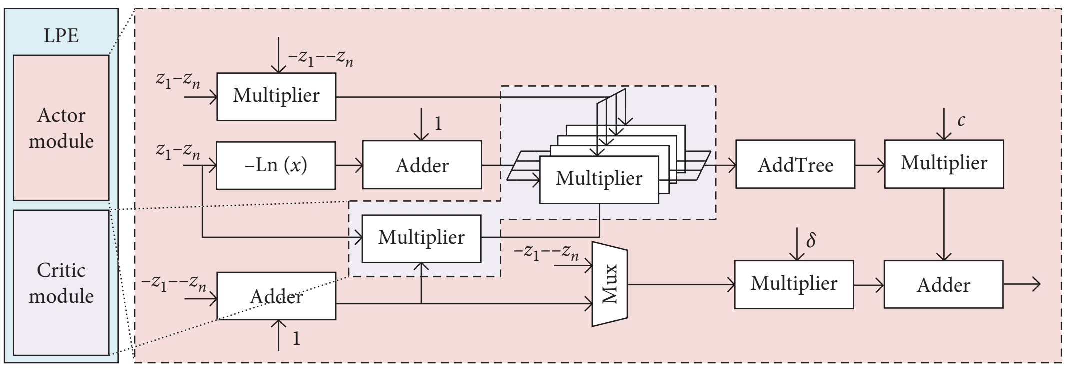

FIGURE 8: The structure of LPE.

computation of the output of the actor network, which will be fed back to the host computer as the basis for selecting the action.

4.3. The Proposed LGCU.

4.3. The Proposed LGCU. We define the last shared layer as the input layer during backpropagation. The calculation of the loss gradient is to compute the gradient value of the loss function with respect to the input layer during backpropagation. We simplify the derivative of the loss function to facilitate hardware implementation of the process.

The loss function $L_\pi(\theta)$ of the actor network is defined by Equation (2). Assuming that the output of the last shared layer is a 1D vector $Y = (y_1, y_2, \ldots, y_l)$, and after the Softmax, it becomes $Z = (z_1, z_2, \ldots, z_i, \ldots, z_l)$, where z_i $(1 \le i \le l)$ represents the maximum value in the output vector Z. The inverse input map $X'_\pi = (x'_1, x'_2, \ldots, x'_l)_\pi$ of the actor network can be obtained by Equation (4):

$$L_\pi(\theta) = log\pi(s_t|a_t;\theta)(r_t - v_t) + c\sum - \pi_i(s_t;\theta)ln\frac{1}{\pi_i(s_t;\theta)}, \tag{2}$$

$$X'_\pi = c\begin{bmatrix} 1-lnz_1 \\ 1-lnz_2 \\ \vdots \\ 1-lnz_l \end{bmatrix}^T \begin{bmatrix} z_1(1-z_1) & -z_2z_1 & \cdots & -z_lz_1 \\ -z_1z_2 & z_2(1-z_2) & \cdots & \vdots \\ \vdots & \vdots & \ddots & z_{l-1}z_l \\ -z_1z_l & -z_2z_l & \cdots & z_l(1-z_l) \end{bmatrix} + \delta\begin{bmatrix} -z_1 \\ \vdots \\ 1-z_i \\ \vdots \\ -z_l \end{bmatrix}^T, \tag{3}$$

$$x' = \frac{\partial L_\pi(z)}{\partial y} = \frac{\partial L_\pi(z)}{\partial z}\frac{\partial z}{\partial y}. \tag{4}$$

By combining Equations (2) and (4), we can derive a simplified expression for the inverse input map of actor network X'_π by performing the derivative solver operation on x', as shown in Equation (3).

Equation (5) gives the loss function $L_v(\theta)$ of the critic network. Assuming that the output of the last shared layer is a 1D vector $Y = (y_1, y_2, \ldots, y_l)$ and the value output v is obtained by passing through the linear output layer, the inverse input map of the critic network $X'_v = (x'_1, x'_2, \ldots, x'_l)_v$ can be expressed as Equation (6):

$$L_v(\theta) = [r_t - v_t(s_t;\theta)]^2, \tag{5}$$

$$X'_v = -2\delta(w_1, w_2, \ldots, w_l). \tag{6}$$

Based on the above conclusion, the computation process for the inverse input map of the actor network and the critic network can be seen as a matrix operation. As a result, we design the LPE, as illustrated in Figure 8, to calculate the inverse input feature map of the actor network. Moreover, we calculate the inverse input feature map of the critic network by reusing the multipliers within the designated purple box to minimize resource overhead.

4.4. The Proposed BCU.

4.4. The Proposed BCU. The BCU consists of several backward processing engines (BPE), as shown in Figure 9. The control signal generation module (CSG module) and the input control module (IC module) are employed in the BPE to execute the derivative operation of the ReLU function. Additionally, the Multiplier-Acc module is utilized for the matrix multiplication between the feature map and weights.

In the designed BPE, considering the special derivative property of ReLU described in Equation (7), the CSG module is employed to capture the positive and negative information of the output feature map during the forward propagation calculation, instead of directly computing the derivative value of the ReLU function. During the calculation of forward inference, if the result $y_i > 0$, then a positive sign is stored in the buffer, whereas if $y_i = 0$, a zero value is stored instead, indicating both positive and negative information. Afterward, the captured information is utilized to regulate the input of Multiplier-Acc module by IC module, thereby

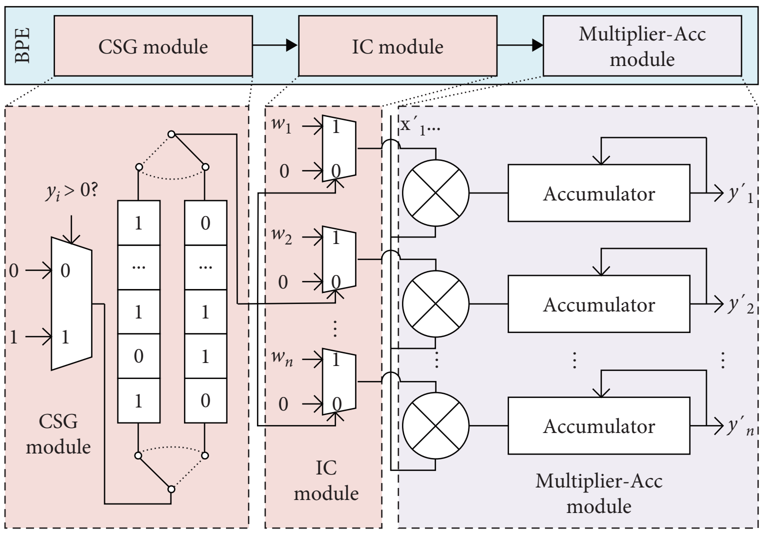

Figure 9: The structure of BPE.

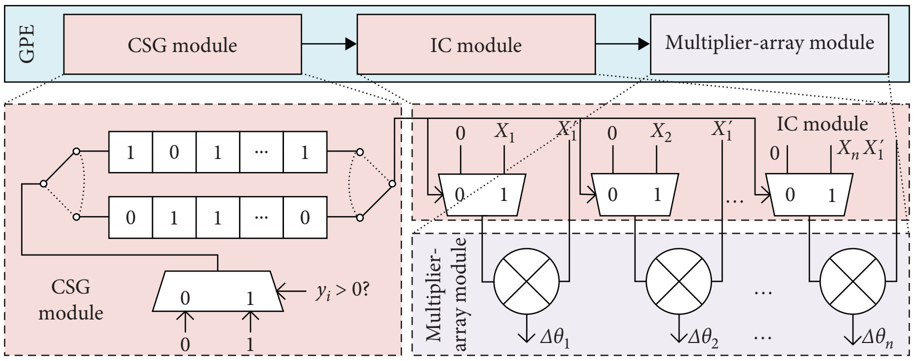

Figure 10: The structure of GPE.

indirectly accomplishing the derivative operation of the ReLU function:

$$\frac{\partial\,\mathrm{ReLU}(wx+b)}{\partial(wx+b)} = \frac{\partial y}{\partial(wx+b)} = \begin{cases} 0 & y=0 \\ 1 & y>0 \end{cases}. \quad (7)$$

In contrast to the Multiplier-AddTree module used in the FPE, we employ the Multiplier-Acc module for matrix multiplication operations in BPE. These two computational structures have an inherent transpose relationship in the storage and mapping of weights. Using this relationship, we can solve the problem of differences in weight mappings during forward and backward computation, effectively improving the access efficiency of parameters without increasing storage consumption.

4.5. The Proposed PGCU. The PGCU computes the gradient value $d\theta$ for each layer, as expressed by Equations (8) and (9). Here, x' refers to the reverse input feature map, A' stands for the derivative of the activation function, and x represents the input feature map computed during forward inference. The initial input x' for parameter gradient computation is calculated by the LGCU, while the subsequent layer inputs are acquired through the progressive propagation of the BCU:

$$d\theta_{L_1} = \frac{\partial L(\theta)}{\partial \theta_{L_1}} = \frac{\partial L(\theta)}{\partial y_L}\frac{\partial y_L}{\partial y_{L-1}}\frac{\partial y_{L-2}}{\partial y_{L-3}}\cdots\frac{\partial y_{L_1+1}}{\partial y_{L_1}}\frac{\partial y_{L_1}}{\partial \theta_{L_1}}, \quad (8)$$

$$dw = x'A'x, db = x'A'. \quad (9)$$

Like the BCU, the PGCU also requires computing the derivative of the activation function. We can apply the same approach used in the BCU to obtain these results. The structure of the gradient processing engine (GPE) is illustrated in Figure 10. The CSG module and the IC module are used to compute the derivatives of the ReLU function, while the n parallel multiplier array is responsible for calculating the parameter gradient values.

4.6. The Proposed RMSProp Unit. The RMSProp Unit is responsible for asynchronously updating parameters of the central agent. The A3C algorithm requires gradient value aggregation prior to training, followed by parameter updates of the central network using the RMSProp gradient descent method [33]. Therefore, we design the structure of the RMSProp Unit, as shown in Figure 11.

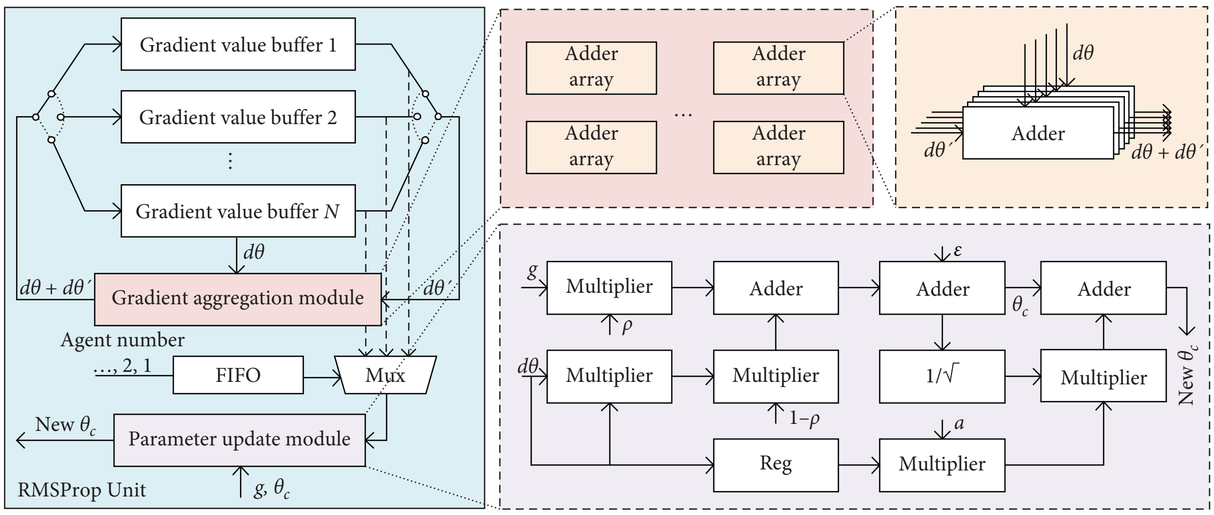

Figure 11: The structure of RMSProp Unit.

Table 2: DNN model.

No.	Layer type	Number of parameters	Number of output features
0	Input	—	1×128
1	Fully connected (FC1)	128×256	1×256
2		ReLU activation	
3	Fully connected (FC2)	256×128	1×128
4		ReLU activation	
5	Fully connected (FC3)	128×64	1×64
6		ReLU activation	
7	Fully connected (FC4)	64×4	1×4
8		Softmax (activation)/linear (value)	

We deploy multiple adder arrays with a parallelism degree of n in the Gradient aggregation module. These adders perform gradient aggregation calculations for each GPE in the PGCU. To implement the asynchronous parameter updating method of the A3C algorithm, we employ a synchronous first-in-first-out (FIFO) data buffer. This buffer retains the order in which the distributed agent perform aggregation computation.

When an agent finishes the gradient aggregation calculation, its corresponding agent number i is sent to the FIFO as a control signal. If the Parameter update module is idle, the selector reads the agent number i from the FIFO. Simultaneously, it accesses the central agent parameter memory and the gradient value buffer of the corresponding agent to perform the gradient update calculation. Additionally, the Parameter update module achieves continuous parameter value updates through multipipeline control, reducing the data waiting period.

5. Evaluation

5.1. Experimental Setup. We use the Atari game as a virtual training environment for validating the proposed deployment strategy and evaluating the performance of the accelerator. Additionally, the RAM version of the Atari game is used in our work, and the state information of the agents of any game under this version is described by a vector of size 1×128. Therefore, the input data for our A3C model have a uniform shape. The DNN architecture employed by the A3C algorithm is outlined in Table 2.

To evaluate the performance of the FPGA accelerator, we implemented the A3C algorithm on both CPU and GPU platforms for comparison. The CPU is Intel (R) Core (TM) i9-13900K working at 3,000 MHz. The GPU is NVIDIA GeForce RTX 4090 working at 2,235 MHz. The FPGA platform is Virtex-VC707 working at 200 MHz.

5.2. The Verification of Deployment Strategy for Distributed Algorithm. We investigate the impact of varying the number of agents and the inference upper limit $T_{\max}$ during model training. As a result, we found that $N_e = 8$ and $N_r = 6$. Additionally, after evaluating the size of hardware resources required for deploying each agent, we determined that $N_l = 16$. Based on Equation (1) in Section 3.1, we can conclude that the optimal number of agents N_o is 8. We measure the performance of the DRL platforms using the number of explorations processed per second (EPS) across all agents. If each A3C agent executes $T_{\max}$ inference tasks followed by a training task, with a $T_{\max}$ value of 5 and an achieved EPS of

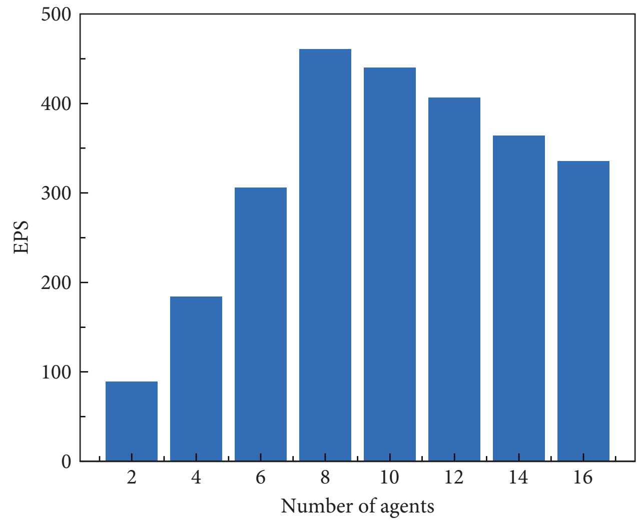

FIGURE 12: The EPS of different number of agents.

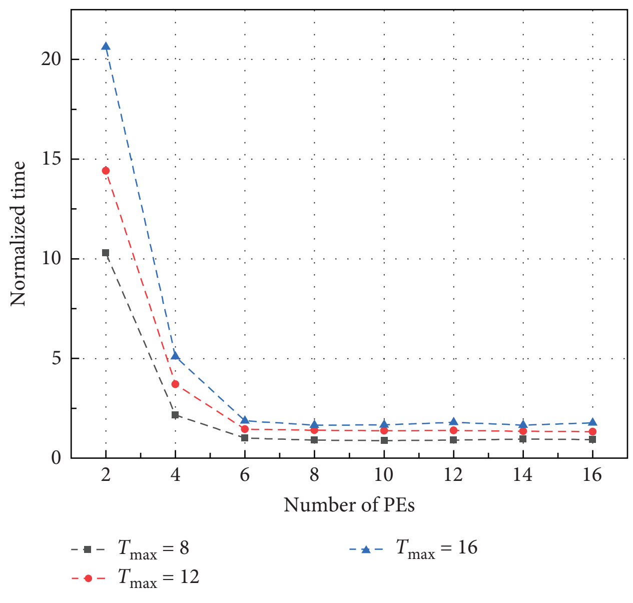

FIGURE 13: The normalized time of different number of PEs.

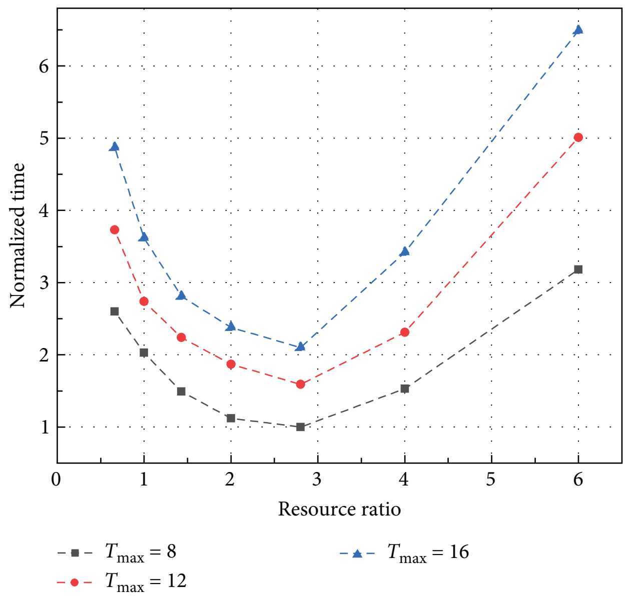

FIGURE 14: The normalized time of different resource ratio.

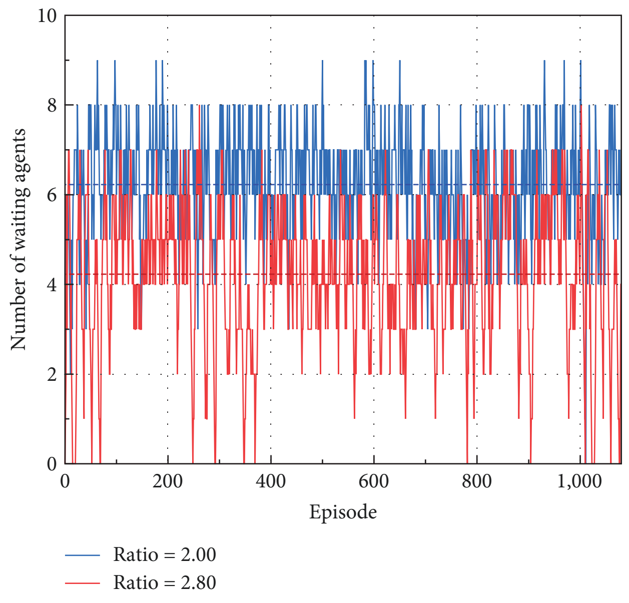

FIGURE 15: The number of waiting agents in different ratio.

500, the DRL platform handles 2,500 inference tasks, 500 extra inferences for value bootstrapping, and 500 training tasks per second. Then, we conducted experiments to assess the variation in EPS based on the deployment of different numbers of agents. The corresponding results are presented in Figure 12. The results show that the highest EPS can be achieved with the optimal number of agents.

We investigate the real execution time t_r, for varying number of PEs using a software simulation platform. The corresponding results are illustrated in Figure 13. When the number of deployed PEs exceeds 6, the decreasing trend of the normalized time remains almost constant. Therefore, we can infer that the most suitable number of PE deployments M^* is 6. Using the optimal number of PEs can reduce resource consumption by 62.5% compared to the worst configuration (the number of PEs is 16). This suggests that more PEs do not necessarily lead to an improvement in performance and confirms the effectiveness of our proposed strategy.

After obtaining the optimal number of PEs, we investigate the influence of the resources ratio of training to inference on the real execution time. Figure 14 displays the real execution time corresponding to various resource ratios. Based on these results, we determine that the optimal ratio of resources η^* is 2.8.

Furthermore, we compare the number of waiting agents under the initial and optimal resource ratios, as shown in Figure 15. When the optimal resource ratio of training to inference is used, the average number of agents waiting for training decreases from 6.2 to 4.2, achieving a 32.2% optimization.

Table 3: Resource overhead.

Component	LUT	FF	BRAM	DSP
Ethernet interface	8,066	2,207	—	—
Memory unit	16,519	12,901	242.5	3
FC unit	33,093	50,004	—	90
BC unit	89,371	84,598	—	224
PGC unit	8,799	16,934	—	32
LGC unit	11,135	19,512	—	36
RMSProp unit	61,492	98,157	500.5	304
Total	228,475 (303,600)	284,313 (607,200)	743 (1,030)	689 (2,800)

Table 4: The experiment results of each platform.

	CPU	GPU	FPGA
Platform	Intel i9-13900K	NVIDIA RTX4090	Xilinx VC707
Frequency (Hz)	3,000M	2,235M	200M
Power (W)	28.29	21.84	10.03
EPS	348.92	267.05	638.16
EPS/W	12.33	12.23	63.61

Table 5: The comparison results with previous works.

	Shiri et al.'s [22] study	Gankidi and Thangavelautham's [34] study	Meng and Kuppannagari's [27] study	Yang et al.'s [28] study	FA3C [29]	Ours work
Platform	Xilinx Artix7	Xilinx Virtex7	Xilinx U200	Xilinx U50	Xilinx VCU1525	Xilinx VC707
Algorithm	HDRL	DQN	PPO	DDPG	A3C	A3C
Frequency (Hz)	100M	150M	285M	164M	180M	200M
Precision	FIX	FP	FP32	FIX	FP32	FP32
LUT	3,345	—	501K	508.1K	677.3K	228.4K
DSP	39	—	3,744	2,302	2,348	689
BRAM	321	—	1,046	1,798	1,267	743
GOPS/GFLOPS	4.4GOPS	0.847	—	—	—	14.8
Normalized EPS to FA3C	256	—	1,354	7,696	2,550	3,669
Power (W)	0.873	9.7	—	20.4	18	10.03
LRE	0.0765	—	0.0027	0.0151	0.0038	0.0161
DRE	6.5641	—	0.3616	3.3432	1.086	5.3251
BRE	0.7975	—	1.2945	4.2803	2.0126	4.9381
EPS/W	293.2	—	—	377.3	141.7	365.8

5.3. The Evaluation of Accelerator Performance. The resource overhead of the designed A3C accelerator is presented in Table 3. The high LUT resource overhead is due to the fact that the accelerator needs a large number of control signals to control the distributed asynchronous learning method of the A3C algorithm. In addition, the distributed deployment approach of the agents and the large number of parameters and computation results to be stored cause a high consumption of BRAM.

We investigate the EPS and power consumption of the three platforms. The obtained results are presented in Table 4. The experiment indicates that the designed FPGA accelerator reaches 1.83× and 2.39× speedup compared to CPU and GPU. Additionally, the power consumption of the designed accelerator is merely 35.5% and 45.9% of the CPU and GPU, suggesting a significantly lower power consumption.

Table 5 shows the comparison results with previous works on several metrics. We focus on comparing the normalized

EPS to FA3C (since FA3C is the earlier work on distributed DRL accelerator, we normalize all the comparison data on EPS to FA3C), LUT resource efficiency (LRE), DSP resource efficiency (DRE), and BRAM resource efficiency (BRE). LRE, DRE, and BRE are obtained from the ratio of EPS to the consumed LUT, DSP, and BRAM, respectively.

As shown in Table 5, the proposed accelerator exhibits excellent computational performance in comparison to existing works. In terms of resource efficiency, our design presents significant advantages in LRE, DRE, and BRE compared to the latest works [27, 28, 29] on DRL accelerators. The results show that the work [22] outperforms ours in LRE and DRE due to the fact that it only accelerates inference for hierarchical deep reinforcement learning (HDRL). As a result, less LUT and DSP resources are consumed in their work. Specifically, our work achieves higher computational performance, resource efficiency, and energy efficiency compared to FA3C which is the first accelerator designed for A3C algorithms.

6. Conclusion

This paper introduces a hardware deployment strategy for distributed algorithms. The proposed strategy optimizes the number of agents, PEs, and the allocation of resources. Its aim is to ensure a balanced interaction and training time among distributed agents in the accelerator while reducing idle waiting delays and improving the utilization of resources. Then, we propose an FPGA-based architecture for the A3C algorithm. The proposed architecture adopts modular computing unit design, independent storage resource allocation method, and parallel deployment of computing resources, which effectively improves the training efficiency of A3C algorithm. The experimental results show that our proposed deployment strategy reduces resource consumption by 62.5% and decreases the number of agents waiting for training by 32.2%, and the proposed A3C accelerator achieves 1.83× and 2.39× improvements in speedup compared to CPU and GPU, respectively. The power consumption of the accelerator is only 35.5% of the CPU and 45.9% of the GPU, which effectively reduces the execution overhead of the A3C algorithm. Further, the proposed A3C accelerator presents significant advantages in resource efficiency compared to existing works.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key R&D Program of China (No. 2022YFD2000700).

References

[1] V. Mnih, A. P. Badia, M. Mirza et al., "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, pp. 1928–1937, PLMR, 2016.

[2] A. Kendall, J. Hawke, D. Janz, P. Mazur, and D. Reda, "Learning to drive in a day," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254, IEEE, Montreal, QC, Canada, 2019.

[3] M. Kaushik, V. Prasad, K. M. Krishna, and B. Ravindran, "Overtaking maneuvers in simulated highway driving using deep reinforcement learning," *IEEE Intelligent Vehicles Symposium (IV)*, pp. 1885–1890, 2018.

[4] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3948–3955, IEEE, Vancouver, BC, Canada, 2017.

[5] M. A. Anwar and A. Raychowdhury, "NavREn-Rl: Learning to fly in real environment via end-to-end deep reinforcement learning using monocular images," in *IEEE International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pp. 1–6, IEEE, Stuttgart, Germany, 1995.

[6] A. Rodriguez-Ramos, C. Sampedro, H. Bavle, P. De La Puente, and P. Campoy, "A deep reinforcement learning strategy for UAV autonomous landing on a moving platform," *Journal of Intelligent & Robotic Systems*, vol. 93, no. 1-2, pp. 351–366, 2019.

[7] M. Liang, M. Chen, Z. Wang, and J. Sun, "A CGRA based neural network inference engine for deep reinforcement learning," *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 540–543, 2018.

[8] T. Zhang, Z. McCarthy, O. Jow et al., "Deep imitation learning for complex manipulation tasks from virtual reality teleoperation," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5628–5635, IEEE, Brisbane, QLD, Australia, 2018.

[9] E. Marchesini and A. Farinelli, "Enhancing deep reinforcement learning approaches for multi-robot navigation via single-robot evolutionary policy search," in *International Conference on Robotics and Automation (ICRA)*, pp. 5525–5531, IEEE, Philadelphia, PA, USA, 2022.

[10] D. Silver, A. Huang, C. J. Maddison et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[11] Y. Sun, B. Yuan, Y. Zhang et al., "Research on action strategies and simulations of DRL and MCTS-based intelligent round game," *International Journal of Control, Automation and Systems*, vol. 19, no. 9, pp. 2984–2998, 2021.

[12] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Reinforcement learning through asynchronous advantage actor-critic on a gpu," arXiv preprint, arXiv: 1611.06256, 2016.

[13] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: a survey," arXiv preprint, arXiv: 1806.01683, 2018.

[14] S. Spano, G. C. Cardarilli, L. Di Nunzio et al., "An efficient hardware implementation of reinforcement learning: the Q-learning algorithm," *IEEE Access*, vol. 7, pp. 186340–186351, 2019.

[15] N. K. Manjunath, A. Shiri, M. Hosseini, B. Prakash, N. R. Waytowich, and T. Mohsenin, "An energy efficient EdgeAI autoencoder accelerator for reinforcement learning," *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 182–195, 2021.

[16] K. Chen, L. Huang, M. Li, X. Zeng, and Y. Fan, "A compact and configurable long short-term memory neural network hardware architecture," *IEEE International Conference on Image Processing (ICIP)*, pp. 4168–4172, 2018.

[17] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, IEEE, Baltimore, MD, USA, 2017.

[18] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7–9, pp. 1180–1190, 2008.

[19] V. Mnih, K. Kavukcuoglu, D. Silver et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[20] Y. Meng, S. Kuppannagari, R. Rajat, A. Srivastava, R. Kannan, and V. Prasanna, "QTAccel: a generic FPGA based design for Q-table based reinforcement learning accelerators," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 107–114, IEEE, New Orleans, LA, USA, 2020.

[21] A. R. Baranwal, S. Ullah, S. S. Sahoo, and A. Kumar, "*ReLAccS*: a multilevel approach to accelerator design for reinforcement learning on FPGA-based systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1754–1767, 2021.

[22] A. Shiri, B. Prakash, A. N. Mazumder, N. R. Waytowich, T. Oates, and T. Mohsenin, "An energy-efficient hardware accelerator for hierarchical deep reinforcement learning," in *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 1–4, IEEE, Washington DC, DC, USA, 2021.

[23] D. P. Leal, M. Sugaya, H. Amano, and T. Ohkawa, "FPGA acceleration of ROS2-based reinforcement learning agents," in *Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 106–112, IEEE, Naha, Japan, 2020.

[24] M.-J. Li, A.-H. Li, Y.-J. Huang, and S.-I. Chu, "Implementation of deep reinforcement learning," in *International Conference on Information Science and Systems*, pp. 232–236, ACM, 2019.

[25] A. Shiri, A. N. Mazumder, B. Prakash, H. Homayoun, N. R. Waytowich, and T. Mohsenin, "A hardware accelerator for language-guided reinforcement learning," *IEEE Design & Test*, vol. 39, no. 3, pp. 37–44, 2022.

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

[27] Y. Meng and S. Kuppannagari, "Accelerating proximal policy optimization on CPU-FPGA heterogeneous platforms," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 19–27, IEEE, Fayetteville, AR, USA, 2020.

[28] J. Yang, S. Hong, and J.-Y. Kim, "FIXAR: a fixed-point deep reinforcement learning platform with quantization-aware training and adaptive parallelism," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 259–264, IEEE, San Francisco, CA, USA, 2021.

[29] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 499–513, ACM, 2019.

[30] Y. Wang, M. Wang, B. Li, H. Li, and X. Li, "A many-core accelerator design for on-chip deep reinforcement learning," in *International Conference on Computer-Aided Design*, pp. 1–7, IEEE, San Diego, CA, USA, 2020.

[31] H. Chen, M. Issa, Y. Ni, and M. Imani, "DARL: distributed reconfigurable accelerator for hyperdimensional reinforcement learning," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, IEEE, San Diego, CA, USA, 2022.

[32] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 279–291, IEEE, Phoenix, AZ, USA, 2019.

[33] T. Tieleman, "Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude," *COURSERA: Neural Networks for Machine Learning*, vol. 4, no. 2, pp. 26–31, 2012.

[34] P. R. Gankidi and J. Thangavelautham, "FPGA architecture for deep learning and its application to planetary robotics," in *IEEE Aerospace Conference*, pp. 1–9, IEEE, Big Sky, MT, USA, 2017.