TD3lite: FPGA Acceleration of Reinforcement Learning with Structural and Representation Optimizations

Chan-Wei Hu[1], Jiang Hu[1,2], Sunil P. Khatri[2]
Department of Computer Science and Engineering, Texas A&M University[1]
Department of Electrical and Computer Engineering, Texas A&M University[2]
{huchanwei123, jianghu, sunilkhatri}@tamu.edu

***Abstract*—Reinforcement learning (RL) is an effective and increasingly popular machine learning approach for optimization and decision-making. However, modern reinforcement learning techniques, such as deep Q-learning, often require neural network inference and training, and therefore are computationally expensive. For example, Twin-Delay Deep Deterministic Policy Gradient (TD3), a state-of-the-art RL technique, uses as many as 6 neural networks. In this work, we study the FPGA-based acceleration of TD3. To address the resource and computational overhead due to inference and training of the multiple neural networks of TD3, we propose TD3lite, an integrated approach consisting of a network sharing technique combined with bitwidth-optimized block floating-point arithmetic. TD3lite is evaluated on several robotic benchmarks with continuous state and action spaces. With only 5.7% learning performance degradation, TD3lite achieves $21\times$ and $8\times$ speedup compared to CPU and GPU implementations, respectively. Its energy efficiency is $26\times$ of the GPU implementation. Moreover, it utilizes $\sim 25-40\%$ fewer FPGA resources compared to a conventional single-precision floating-point representation of TD3.**

***Index Terms*—Reinforcement learning, hardware acceleration, FPGA, neural network, block floating-point**

I. Introduction

Reinforcement Learning (RL) is a branch of machine learning that focuses on sequential decision-making in an environment with uncertainty [1]. It has a wide range of applications, such as robotics [2], [3], healthcare [4], and finance [5]. Unlike supervised learning, where training and inference can be largely decoupled, the application (decision-making) of RL is tightly integrated with training. Thus, the traditional training-on-server and application-on-edge paradigm for deep learning cannot be applied for RL. This makes hardware approaches for RL challenging to realize, because the combined computational load of inference and training demands high computing efficiency as well as carefully engineered hardware acceleration techniques.

The recent flurry of interest in RL research was triggered by deep Q-learning (DQN) [6], which integrates deep neural networks (DNNs) with RL, and its well known success playing Google AlphaGo [7]. A DQN employs two DNNs. Generally, implementing the training and inference of the two neural networks is more challenging than many existing DNN acceleration works that perform either training or inference of a single DNN. Furthermore, state-of-the-art RL techniques increase the number of neural networks. DDPG [8], which is more advanced than DQN, uses four networks. Twin-Delayed Deep Deterministic Policy Gradient (TD3) [9], which is an improvement over DDPG, employs six networks. With this increase in the number of neural networks, there is a growing need for improving computational and resource efficiency in RL hardware acceleration.

In this work, we perform FPGA acceleration for TD3 [9]. We trade off learning performance and computing resource-efficiency through two integrated techniques: network sharing and block floating-point arithmetic with bit-width optimization. Existing RL FPGA acceleration approaches [10]–[17] pay little attention to the tradeoff except [18], which uses fixed-point arithmetic. In a large portion of existing works [11]–[13], [15], [16], the RL agent is only partially realized on an FPGA. Some other works [10], [17] are restricted to a discrete state/action space. By contrast, our approach, called TD3lite, realizes the entire RL agent on an FPGA, with a continuous state/action space. TD3lite is able to deliver a high computational and resource efficiency with minimal learning performance loss. Our contributions are summarized below.

- To the best of our knowledge, this work is the first FPGA acceleration technique for TD3, a state-of-the-art RL technique.
- To the best of our knowledge, this is the first study on neural network sharing in hardware acceleration.
- Further, this is the first investigation of block floating point arithmetic for RL FPGA acceleration.
- In TD3lite, the entire computation of the TD3 agent, including training of 6 networks, is realized on an FPGA.
- We test TD3lite on four widely used benchmarks from MuJoCo with a continuous state/action space. TD3lite achieves $21\times$ and $8\times$ speedup against CPU-based and GPU-based implementations (baselines). The energy efficiency of TD3lite is $47\times$ and $26\times$ better respectively, than the baselines. Moreover, the resource utilization is $\sim 25-40\%$ lower than a conventional single-precision floating-point representation of TD3.

II. Background

A. Reinforcement Learning

In reinforcement learning (RL), an agent interacts with an environment in discrete time steps, as shown in Figure 1. At each time step t, the agent takes an action a_t according to its policy π based on state s_t observed from the environment. Then, the environment returns a scalar reward r_t and moves to the next state s_{t+1}. The objective of an agent is to optimize

its policy π that maximizes the discounted cumulative reward $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$, where γ is the discount factor and T is the episode length. The state-action *value function* $Q(s_t, a_t)$ estimates the cumulative reward by taking action a_t at state s_t.

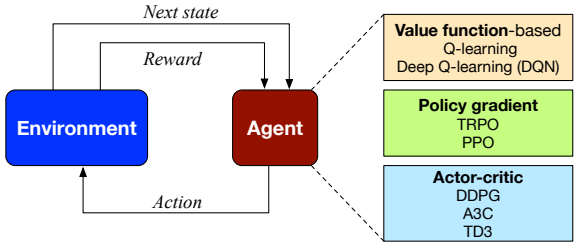

Fig. 1. Overview of RL system and algorithms.

A well-known early RL algorithm is Q-learning [1], which tabulates $Q(s_t, a_t)$ for limited discrete state and action spaces. A more advanced approach is function approximation, where $Q(s_t, a_t)$ is approximated by a continuous function, to handle much bigger spaces. A recent approach is deep Q-learning (DQN) [6], where the value function is approximated by deep neural networks. To reduce the correlation among the network training data, DQN requires to store some historical state transitions, which are randomly sampled and used as training data. This technique is called *experience replay* and needs significant storage space. Besides the *main network*, DQN employs a *target network*, which provides a reference for the main Q-network training with stability. Alternatively, one can approximate the policy π with a function instead of the Q-value. This approach is called *policy gradient*, and is usually faster than value-function based techniques. Neural network-based policy approximation techniques include Trust Region Policy Optimization (TRPO) [19] and Proximal Policy Optimization (PPO) [20]. Sometimes, policy gradient methods face instability problems and this motivates the *actor-critic* approach, where the actor plays the role of policy and the critic evaluates the value function. One recent influential actor-critic approach is Deep Deterministic Policy Gradient (DDPG) [8], which uses 4 networks. A3C (Asynchronous Advantage Actor-Critic) [21] is a lightweight multi-agent approach friendly to parallel processing. We use the TD3 approach, which is described next.

B. Twin-Delayed Deep Deterministic Policy Gradient (TD3)

Twin-Delayed Deep Deterministic Policy Gradient (TD3) [9] is built upon DDPG [8]. DDPG includes four DNNs, which are the main actor, the main critic, the target actor and the target critic. The main actor network $\pi_\phi(a_t|s_t)$ computes an action a_t based on the current state s_t, where ϕ indicates network weights. Meanwhile, the main critic network takes the state and action as input, and outputs value function $Q_\theta(s_t, a_t)$, where θ denotes network weights. Target networks have the same structure as their corresponding main networks, and their weights are periodically synchronized with the main networks. The less frequently updated target networks regulate the main network training. The weights in the target actor network and target critic network are denoted as $\phi^{'}$ and $\theta^{'}$, correspondingly. In TD3, two sets of critic networks, including both the main and target, are used with an identical structure but different initial conditions. The two main critic networks compute value functions Q_{θ_1} and Q_{θ_2}, respectively, where θ_1 and θ_2 are the network parameters of the first and second main critic networks. The Q-value being actually used is $\min(Q_{\theta_1}, Q_{\theta_2})$. $\theta_1^{'}$ and $\theta_2^{'}$ represents the network parameters for two target critic networks, $Q_{\theta_1^{'}}$ and $Q_{\theta_2^{'}}$. A brief TD3 pseudo code is provided in Algorithm 1. At each time step t, only the main actor network interacts with the environment and stores transitions to the experience replay buffer (lines 2-4). This is the *decision-making stage*. In the *training stage* (lines 5-10), gradients are computed and parameters are updated according to training data sampled from the replay buffer. Unlike the main critic network, which is trained at every time step, all target networks and the main actor network are updated every d time steps (line 7), where d is a parameter. Overall, TD3 uses 6 networks, $\pi_\phi, \pi_{\phi'}$, $Q_{\theta_1}, Q_{\theta_1^{'}}, Q_{\theta_2}$, and $Q_{\theta_2^{'}}$.

Algorithm 1 TD3 algorithm

Require: Randomize parameters θ_1 and θ_2 for main critic networks Q_{θ_1} and Q_{θ_2}
Require: Randomize ϕ for main actor network π_ϕ
Require: Initialize parameters for target networks $\theta_1^{'} \longleftarrow \theta_1$, $\theta_2^{'} \longleftarrow \theta_2$ and $\phi^{'} \longleftarrow \phi$
Require: Initialize experience replay buffer R
1: **while** $t \leq$ termination time **do**
2: Main actor network takes action $a_t \sim \pi_\phi(s_t) + \epsilon$, ϵ is a random noise for exploration.
3: Receives reward r_t and next state s_{t+1}.
4: Push tuple (s_t, a_t, r_t, s_{t+1}) to replay buffer R.
5: Get a batch B of transition data from R.
6: Compute gradients and update θ_1 and θ_2 based on B.
7: **if** $(t\%d) == 0$ **then**
8: Compute gradients and update ϕ based on B.
9: Synchronize ϕ', θ_1', and θ_2' with ϕ, θ_1, and θ_2.
10: **end if**
11: **end while**

C. Block Floating-Point Arithmetic

The main idea of Block Floating-Point (BFP) representation is to let multiple floating point numbers, which form a block, share a single exponent. It is more compact than conventional floating-point representation since all numbers in the block share the same exponent, and provides better precision/range than a fixed-point representation. BFP is especially suitable for DNN hardware designs [22], whose major computing component is the MAC (multiply-accumulate). The floating-point multiplication has similar complexity as fixed-point arithmetic. After the "multiply", all operands naturally have the same value for their exponents and then the expensive floating-point addition is reduced to simpler fixed-point addition.

III. Previous Related Works

A survey for FPGA acceleration for reinforcement learning is provided in [23]. Mainstream reinforcement learning algorithms can be broadly categorized to (I) the classical tabular techniques, such as Q-learning [1], and (II) neural network-based function approximation approaches, which are

the latest trend. Here we summarize the previous works of the latter category, to which our work belongs. Several previous works [10]–[13] target FPGA acceleration of deep Q-learning (DQN) [6], [7], which employs two deep neural networks. The work of [10] implements only MLP layers without convolution, and is restricted to a discrete action space. In [11], a binary neural network is developed on the FPGA for the inference part, while the network training is conducted through servers. Another work [12] is able to perform partial training on an FPGA, wherein some layers are trained on the FPGA while the other layers are fixed. By doing so, a low precision can be used the fixed layers, thereby limiting the resource overhead. The acceleration of DQN in [13] is partly on an FPGA while the other parts are computed on an ARM core of the SoC on which the FPGA resides. It simplifies the neural network training by adopting online sequential extreme learning machine, which is effective for only certain types of neural networks. A couple of previous works [14], [15] investigate acceleration of the policy gradient method, which has a single neural network for approximating the policy and does not have a value function network. The work of [14] implements the TRPO (Trust Region Policy Optimization) training algorithm. It is targeted for stochastic policies while our work is focused on deterministic policies. Later, another work [15] realizes a lightweight training algorithm PPO (Proximal Policy Optimization) on a heterogeneous CPU-FPGA platform. In other words, only a part of the computing is carried out on the FPGA. There are works that implement actor-critic learning, which contain both value-function (critic) networks and policy (actor) networks. One such approach is DDPG (Deep Deterministic Policy Gradient), which is partially implemented on FPGA in [16]. In [18], DDPG is implemented using fixed-point operation on an FPGA. We have observed that fixed-point approach causes easily discernible degradation in the learning curves, though the numerical error is not reported. One recent actor-critic technique is A3C (Asynchronous Advantage Actor-Critic), which has multiple agents and avoids experience replay. An FPGA acceleration technique for A3C is proposed in [17]. Although A3C is able to handle continuous action space, the experiment in [17] is performed for only discrete action space. We focus on both continuous state/action space, and apply block floating-point to TD3 with a sharing technique.

IV. TD3lite Design

A. Computing Flow and Platform

The major computing tasks of TD3 (Algorithm 1) are depicted in the block diagram of Figure 2. The black arrows indicate the forward propagation (FW) dataflow in the decision-making stage, which generates actions to interact with the environment. The training stage contains three components: forward propagation (FW, shown in blue arrows), backward propagation (BP, shown in red arrows), and weight update (WU, shown in green arrows). The computing flow for one full iteration of TD3lite is depicted in Figure 3.

We describe the computing flow and tasks with a structural view of Figure 2 and a temporal view of Figure 3. The host CPU emulates the environment, the TD3lite agent is completely implemented on the FPGA and the communication

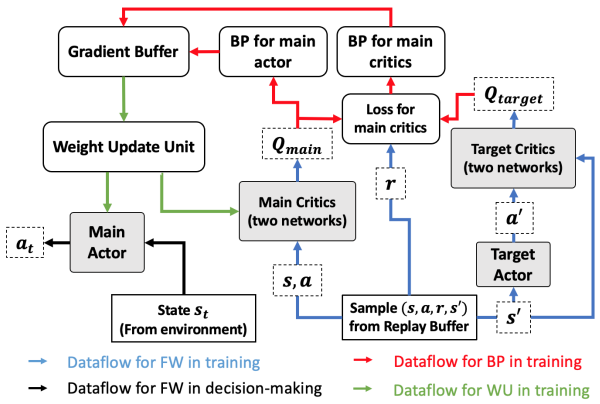

Fig. 2. Computing task overview.

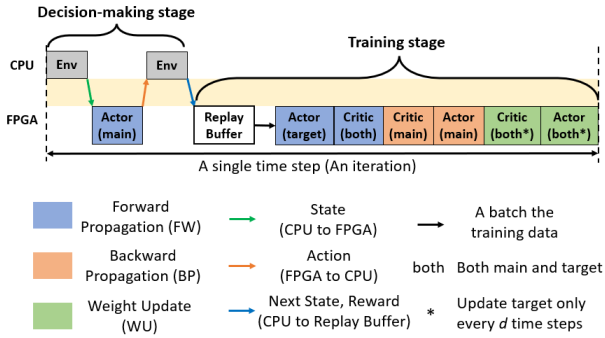

Fig. 3. Computing flow of one full iteration of TD3lite.

between them is through the PCIe 4.0 bus. In the decision-making stage of step t, the host CPU sends the current state s_t to the FPGA agent, receives an action a_t generated by the FPGA agent, calculates reward r_t, and transitions to the next state s_{t+1}. A number of state transitions are stored in the experience replay buffer in the form of (s_t, a_t, r_t, s_{t+1}). In the training stage of step t, a batch B of transitions are sampled from the replay buffer to serve as training data. For each sampled transition (s, a, r, s'), the target actor takes s' as input and generates action a'. Next, the main and target critic networks perform FW according to (s, a) and (s', a'), respectively, and produce Q_{main} and Q_{target}. Please note this done for two sets of main/target critic networks and the Q-values are the minimum between them. Then, the two main critic networks are trained according to a loss defined by Q_{target} and r. Meanwhile, the main actor network is trained according to loss of $-Q_{main}$. More details of the loss functions can be found in [8], [9]. The gradients obtained from BP are applied for WU of the main networks according to the RMSprop method [24].

B. The Accelerator Design

The hardware architecture of TD3lite is depicted in Figure 4. The off-chip memory contains the experience replay buffer. The weight buffer, the intermediate buffer storing all the intermediate results, and the gradient buffer are implemented using BRAMs. The FP-to-BFP converter helps convert a block of floating-point numbers, such as a matrix, into block floating-point representation. It first extracts the largest exponent in a block, which becomes the shared exponent. Then, the mantissa

of each value is aligned based on the shared exponent. The BFP-to-FP converter performs block floating-point to conventional floating-point number conversion. A 16-bit Linear-Feedback Shift Register (LFSR) [25] adds random noise from a Gaussian distribution with zero mean and unit variance to the main actor output in each decision-making stage for exploration (line 2 in Algorithm 1).

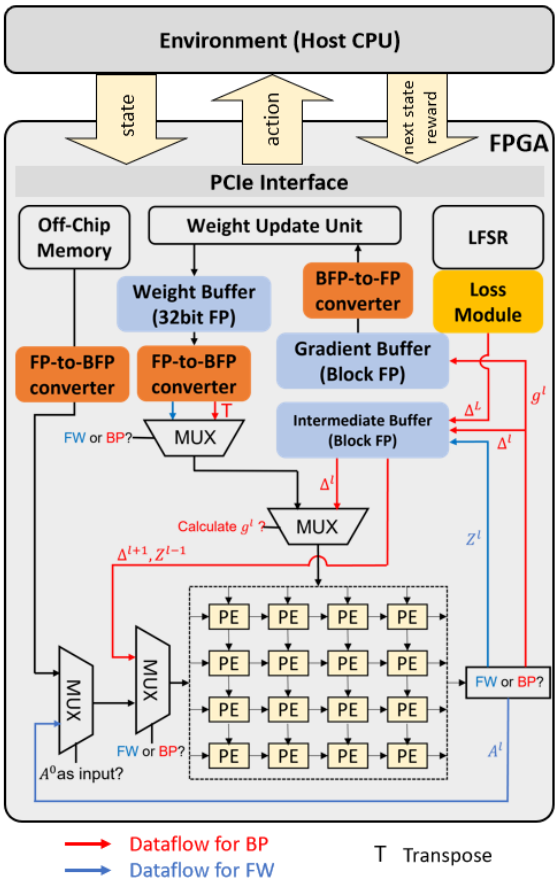

Fig. 4. Hardware architecture of TD3lite.

The FW and BP computations of all the networks are performed on a systolic array with 16×16 processing elements (PEs). Each PE is a fixed-point MAC unit with a weight register so that the dataflow through the systolic array is weight-stationary. The FW computations are

$$\underset{M\times n_l}{Z^l} = \underset{M\times n_{l-1}}{A^{l-1}} \times \underset{n_{l-1}\times n_l}{W^l}, \quad A^l = ReLU(Z^l) \tag{1}$$

where l is the network layer index, n_l is width of layer l, W^l is the matrix of weights from layer $l-1$ to l, A^l is the output of layer l and Z^l is an intermediate result of layer l.

The BP computations are specified by

$$\begin{aligned} \underset{M\times n_l}{\Delta^l} &= (\underset{M\times n_{l+1}}{\Delta^{l+1}} \times \underset{n_{l+1}\times n_l}{(W^{l+1})^\top}) \cdot \underset{1\times n_l}{ReLU'(Z^l)} \\ \underset{n^{l-1}\times n_l}{g^l} &= \underset{n^{l-1}\times M}{(Z^{l-1})^\top} \times \underset{M\times n_l}{\Delta^l} \end{aligned} \tag{2}$$

The error of the last layer is obtained by $\Delta^L = Loss \cdot ReLU'(Z^L)$. For the other layers, Δ^l is the error of layer l propagated from layer $l+1$ and g^l is the gradient for W^l.

In the decision-making stage, the main actor network performs FW (blue line) to compute the action. All the intermediate results and middle layer outputs are stored in the intermediate buffer. The action is the output from the last layer.

The training stage includes both FW and BP specified in Eq. (1) and (2). Weights are loaded to the systolic array after the FP-to-BFP conversion. The BP computations follow the red lines in Figure 4. The loss module computes the loss functions using a single-precision floating-point multiplier and adder. The gradients obtained through BP are stored in the gradient buffer. The weight update unit (WUU) has two multipliers and an adder. It uses single-precision floating-point representation because it tracks the moving average of the squared gradient, which requires high precision. The overhead of float-point computing for a few multipliers and adders is limited.

C. Block Floating-Point (BFP) Computation

In TD3lite, a matrix or a vector is treated as a block for BFP representation. For example, a BFP vector can be expressed as $A_i = m_i^A \times 2^{e_A}$, where A_i is the i-th element in vector A, with the shared exponent e_A and its mantissa m_i^A, and 2^{e_A} is shared by all elements of the block. To illustrate BFP arithmetic, a dot product of two vectors A and B is given by

$$\begin{aligned} A \cdot B &= \sum_{i=1}^{N} ((m_i^A \times 2^{e_A}) \times (m_i^B \times 2^{e_B})) \\ &= (m^A \cdot m^B) \times (2^{e_A + e_B}) \end{aligned} \tag{3}$$

where $m^A \cdot m^B$ is computed via fixed-point arithmetic. Thus, each PE in the systolic array performs fixed-point MAC computation for the mantissa part. This reduces resource utilization and improves speed compared with a floating-point based MAC. The shared exponents are added only once for the entire block. Further, we study different bit-widths for the mantissa of BFP in order to explore the tradeoff between learning performance and computing-resource efficiency. BFP with short bitwidth can accelerate computing speed and reduce FPGA resource, at the expense of learning performance degradation.

D. Network Sharing with Expansion

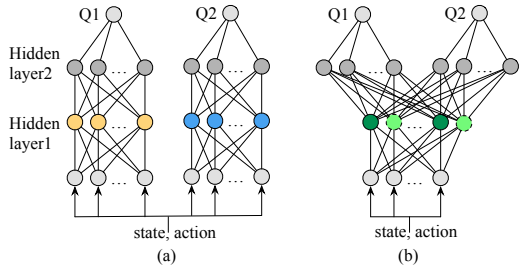

Fig. 5. (a) Two separated (main or target) critic networks; (b) Two critic networks with shared hidden layer1.

We propose a sharing technique, called **topology sharing**, for the critic networks to compensate the learning performance loss due to bitwidth reduction in BFP while the training computing overhead is still limited. Please note this purpose is different from the typical goal of resource utilization reduction through sharing. Figure 5(a) shows two main (or target) critic

TABLE I
LEARNING PERFORMANCE COMPARISON AMONG DIFFERENT NUMBER REPRESENTATIONS.

	Single-precision FP			**Mantissa bit-width in BFP**					
	GPU	CPU	FPGA	24	20	16	**12**	11	10
Hopper	3416.23	3376.46	3479.66	3211.42	3187.25	2983.64	**2818.35 (-19%)**	2511.78	1433.81
HalfCheetah	8451.76	8411.87	8409.1	8172.82	8066.3	6817.67	**6604.87 (-21.4%)**	3194.12	2434.88
Walker	4478.12	4510.03	4452.94	4312.44	4093.21	3771.18	**3686.01 (-17.2%)**	2732.36	2331.72
Ant	5110.32	5033.1	5117.83	4998.14	4452.67	4112.13	**3920.22 (-23.4%)**	3192.75	2432.18

networks in TD3. Since the two networks perform a similar task, their hidden layers can be shared as shown in Figure 5(b). There are two special designs in this sharing.

1) The shared hidden layer is larger than each of the original ones. In Figure 5, if there are k orange nodes and k blue nodes, there are $K > k$ green nodes. This expansion captures more information than the original hidden layers and thus can compensate the learning performance loss caused by the reduced bitwidth in BFP.
2) BP is performed only on a subset of shared nodes, e.g., dark green nodes in Figure 5(b). Different subsets are randomly selected at different time steps of TD3. The randomness is realized via the LFSR [25]. Since BP is the largest component of TD3 computing, this expanded sharing hardly increases computing overhead if only k nodes are selected. Please note all nodes are used in FW.

There are other variants for the sharing, e.g., sharing both hidden layers, and they are analyzed in Section V.

V. EVALUATION AND ANALYSIS

A. Benchmarks and Experiment Platforms

To evaluate the proposed TD3lite, we use four MuJoCo benchmarks, *Hopper*, *HalfCheetah*, *Walker* and *Ant*, from OpenAI Gym [26]. MuJoCo benchmarks are widely used in RL literature [27], [28] for learning performance evaluation since they provide environments with continuous state/action spaces. For example, *Hopper* is a single-legged jumper with 11 physical states and 6 actions. The objective is to control the hopper to hop forward as fast as possible without falling to the ground. *HalfCheetah* and *Walker* have a 17-dimensional state and 6-dimensional action space, and *Ant* has 111-dimensional state and 8-dimensional action space.

In our implementation, both main and target actor networks have two hidden layers of size 256. For main and target critic networks without topology sharing, the size of the input layer is the concatenation of state and action, and the output size is 1 as each network calculates an estimated Q-value. With topology sharing, as shown in Figure 5(b), the size of shared hidden layer is 300, and in each training iteration, 256 neurons are selected randomly to perform BP and WU. All TD3lite networks together have about 300K parameters. All the weights in the networks are updated by the RMSprop method [24], which is implemented using single-precision floating-point arithmetic and is fully stored in BRAM. The learning rate is 3×10^{-4}, and the batch size $|B|$ is 256.

We implement TD3lite on the Xilinx Alveo U50 acceleration card operating at 160MHz using Xilinx Vitis 2020.1 and Vitis HLS. The resources available on the Alveo U50 is presented in the last row of Table IV. Two baselines are compared with TD3lite – a CPU-based and a GPU-based implementation. For all three designs, an AMD Ryzen 7 3800x operating at 3.9 GHz with 32G RAM is used as the host CPU. For the GPU-based platform, we use RTX 2070 Super at 1605 MHz, with 2560 cores and 8GB GDDR6 memory. The replay buffer is implemented in 8GB HBM (High Bandwidth memory) on the Alveo U50 board. We use an efficient and high performance deep learning library, PyTorch [29], to implement TD3 on the CPU-based and GPU-based baselines. PyTorch automatically provides optimizations, such as data parallelism, custom caching tensor allocation and multiprocessing, for general-purpose parallel hardware, such as GPU.

B. Precision and Learning Performance

The **learning performance** is measured by the average maximum cumulative reward over 10 runs of 5000 episodes each, which is the metric used in the TD3 paper [9]. Besides single-precision floating-point arithmetic, we investigate different bitwidth options for the proposed BFP arithmetic and show the results in Table I. In our BFP implementation, the shared exponent is set to 8 bits, which is the same as the IEEE 754 standard. From Table I, BFP alone without bitwidth reduction (24-bit mantissa) causes very small learning performance loss. It is no surprise that the learning performance degrades along with bitwidth reduction. Although TD3lite adopts BFP with a 12-bit mantissa, which has large performance loss, we will show that this loss can be largely compensated by our proposed network topology sharing. Moreover, a 12-bit mantissa for our BFP implementation offers a reasonable performance drop and a good reduction in resource utilization.

We study the topology sharing (TS) with several variants, as described in Section IV-D. One variant is hard sharing (HS), which updates all nodes during the training stage, while TS only updates a subset of nodes, and different subsets are updated in different time steps. In HS1, we share the first hidden layer, and in HS2, both hidden layers are shared. These variants are implemented on the FPGA and summarized below.

- **TD3-ori**: TD3 implementation on FPGA with neither BFP nor network sharing.
- **BFP12**: BFP with 12-bit mantissa only.
- **TS1**: Topology sharing on the first hidden layer as shown in Figure 5(b) without BFP.
- **BFP12-TS1 (TD3lite)**: BFP with 12-bit mantissa & TS1.
- **BFP12-TS2**: BFP12 and TS2, which is topology sharing for both hidden layers.
- **BFP12-HS1-256**: BFP12 with hard sharing of the first hidden layer with 256 nodes.
- **BFP12-HS1-300**: BFP12 and hard sharing with 300 nodes for the first hidden layer.
- **BFP12-HS2-256**: BFP12 with hard sharing of both hidden layers with 256 nodes.

TABLE II
LEARNING PERFORMANCE OF DIFFERENT COMBINATIONS OF NUMBER REPRESENTATIONS AND NETWORK SHARING VARIANTS.

	TD3-ori	**BFP12**	**TS1**	**BFP12-TS1 (TD3lite)**	**BFP12-TS2**	**BFP12-HS1-256**	**BFP12-HS1-300**	**BFP12-HS2-256**	**BFP12-HS2-300**
Hopper	3479.66	2818.35	3421.63 (-1.7%)	**3271.72 (-5.9%)**	2552.76	2387.1	2588.33	2132.44	2127.3
HalfCheetah	8409.1	6604.87	8201.82 (-2.5%)	**7996.51 (-4.9%)**	6098.8	6551.28	6327.11	5187.56	4968.5
Walker	4452.94	3686.01	4348.08 (-2.4%)	**4126.4 (-7.3%)**	3211.72	2635.5	2643.97	2011.22	2264.2
Ant	5117.83	3920.22	5003.4 (-2.2%)	**4874.53 (-4.8%)**	3287.13	3337.41	3476.54	2412.34	2529.37
Average loss	-	-20%	-2.2%	**-5.7%**	-29.4%	-32.3%	-30.8%	-46.2%	-44.9%

- **BFP12-HS2-300**: BFP12 and hard sharing with 300 nodes for both hidden layers.

Table II compares the learning performance of the above options. In BFP12-TS1, we can reduce the learning performance loss of BFP12 from 20% to merely 5.7% on average. It is reasonable since BFP12-TS1 has 300 neurons in the first hidden layer, which is more informative. One can also see that hard sharing and sharing two layers cause too much learning performance loss. Although the learning performance loss from TS1 alone is only 2.2%, we will show that it does not have the computing and resource efficiency provided by BFP12-TS1 (TD3lite). In Figure 6, we illustrate the cumulative reward over learning episode for *Hopper*. One can see that the final reward from TD3lite is close to the CPU and GPU implementations with single-precision floating point and TD3lite reaches high reward faster than the other two approaches.

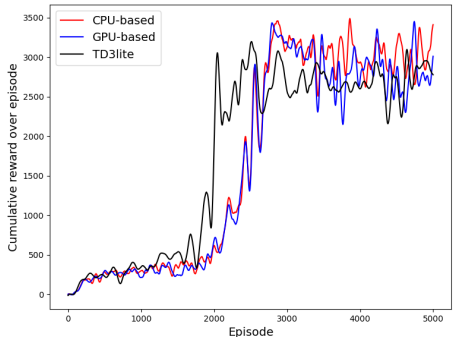

Fig. 6. Learning curves for the *Hopper* benchmark.

C. Computing Throughput

Computing throughput is measured by Full Iteration Processed per Second (**FIPS**), where a full iteration includes both the decision-making and training stages shown in Figure 3. The FIPS results are plotted in Figure 7. TD3lite achieves a 21× and 8× throughput compared to the CPU and GPU implementations, respectively. TS1 alone provides limited throughput improvement. One can observe that the main contributor of throughput improvement is BFP12. However, its significant learning performance loss (20%) needs to be compensated by TS1. This is why the combination of BFP12 and TS1, which is TD3lite, achieves the best tradeoff between learning performance and computing throughput.

D. Resource Utilization and Efficiency

Table III compares power consumption and energy efficiency in terms of FIPS per Watt. The energy efficiency of TD3lite is 47× and 26× of CPU and GPU implementations, respectively. Comparing BFP12 and TD3lite, one can tell that TD3lite slightly improves energy efficiency of BFP12 while compensating the learning performance loss of BFP12.

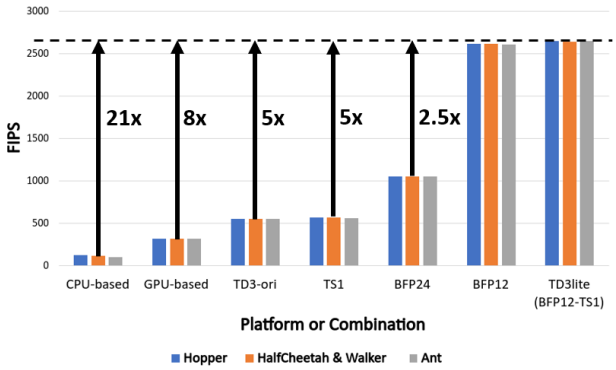

Fig. 7. Computing throughput in terms of FIPS.

TABLE III
POWER AND ENERGY EFFICIENCY COMPARISON.

	CPU	GPU	TD3-ori	TS1	BFP24	BFP12	TD3lite
Power (W)	52	74	26	25	25	24	24
FIPS	123	320	550	567	1053	2617	2647
FIPS/W	2.37	4.32	21.15	22.7	42.1	109	110.3

Table IV compares the FPGA resource utilization of TD3lite and others. TD3lite utilizes 35% less DSPs, 39% less flip-flops, 35% less LUTs, and 26% less BRAMs than TD3-ori, which is based on single-precision floating-point arithmetic.

TABLE IV
RESOURCE UTILIZATION ON XILINX ALVEO U50. THE NUMBER IN PARENTHESES INDICATES THE PERCENTAGE CHANGE IN UTILIZATION COMPARED WITH OUR BASELINE FPGA IMPLEMENTATION, TD3-ORI.

Type	Flip-Flops	LUTs	BRAM	DSP
TD3-ori	611K	763K	717	3551
TS1	609K (-0.3%)	744K (-2.5%)	746 (-4%)	3461 (-2.5%)
BFP24	451K (-26%)	551K (-28%)	601 (-16%)	2899 (-18%)
BFP12	370K (-39%)	490K (-36%)	512 (-28.5%)	2237 (-37%)
TD3lite	373K (-39%)	498K (-35%)	527 (-26%)	2299 (-35%)
Available on FPGA	1743K	872K	1344	5952

VI. CONCLUSIONS

In this paper, we introduce TD3lite, an FPGA acceleration approach for TD3, a state-of-the-art RL technique. To address the computational and resource overhead resulting from the inference and training of multiple neural networks, TD3lite employs a network sharing technique in conjunction with the use of bitwidth-optimized block floating-point arithmetic. Experimental results from robotic benchmarks show that TD3lite is 21× and 8× faster than CPU and GPU implementations, respectively, at the cost of only 5.7% learning performance loss. TD3lite achieves 26× energy efficiency over a GPU implementation, and reduces FPGA resource utilization by $\sim 25-40\%$ compared to a single-precision floating-point realization of TD3.

ACKNOWLEDGMENTS

This work is partially supported by NSF CCF-1937396.

REFERENCES

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. A Bradford Book, 2 ed., 2018.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[3] L. Brunke, M. Greeff, A. W. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. P. Schoellig, "Safe learning in robotics: From learning-based control to safe reinforcement learning," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 5, 2021.

[4] C. Yu, J. Liu, S. Nemati, and G. Yin, "Reinforcement learning in healthcare: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–36, 2021.

[5] P. N. Kolm and G. Ritter, "Modern perspectives on reinforcement learning in finance," *Modern Perspectives on Reinforcement Learning in Finance (September 6, 2019). The Journal of Machine Learning in Finance*, vol. 1, no. 1, 2020.

[6] 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.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[8] 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.

[9] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning*, pp. 1587–1596, 2018.

[10] J. Su, J. Liu, D. Thomas, and P. Cheung, "Neural network based reinforcement learning acceleration on FPGA platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 68–73, Sept. 2016.

[11] Y. Kadokawa, Y. Tsurumine, and T. Matsubara, "Binarized P-network: deep reinforcement learning of robot control from raw images on FPGA," *IEEE Robotics and Automation Letters*, vol. 6, pp. 8545–8552, Oct. 2021.

[12] W. Jo, J. Lee, S. Park, and H.-J. Yoo, "An energy-efficient deep reinforcement learning FPGA accelerator for online fast adaptation with selective mixed-precision re-training," in *IEEE Asian Solid-State Circuits Conference*, 2021.

[13] H. Watanabe, M. Tsukada, and M. Hiroki, "An FPGA-based on-device reinforcement learning approach using online sequential learning," in *IEEE International Parallel and Distributed Processing Symposium Workshops*, pp. 96–103, 2021.

[14] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2018.

[15] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on CPU-FPGA heterogeneous platforms," in *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 19–27, 2020.

[16] C. Guo, W. Luk, S. Shui, A. Warren, and J. Levine, "Customisable control policy learning for robotics," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2019.

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

[18] 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*, pp. 259–264, 2021.

[19] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," *arXiv preprint arXiv:1502.05477*, 2015.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[21] V. Mnih, A. Badia, M. Mirza, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *arXiv preprint arXiv:1602.01783*, 2016.

[22] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High performance FPGA-based CNN accelerator with block-floating-point arithmetic," *IEEE Transactions on VLSI Systems*, vol. 27, pp. 1874–1885, Aug. 2019.

[23] M. Rothmann and M. Porrmann, "A survey of domain-specific architectures for reinforcement learning," *IEEE Access*, vol. 10, pp. 13753–13767, Jan. 2022.

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

[25] C. Condo and W. Gross, "Pseudo-random gaussian distribution through optimised lfsr permutations," *Electronics Letters*, vol. 51, no. 25, pp. 2098–2100, 2015.

[26] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," 2016.

[27] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, pp. 1329–1338, 2016.

[28] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," *arXiv preprint arXiv:1708.04133*, 2017.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.