PEARL: FPGA-Based Reinforcement Learning Acceleration with Pipelined Parallel Environments

Jiayi Li[1,2], Hongxiao Zhao[1,2], Wenshuo Yue[1,2], Yihan Fu[1,2], Daijing Shi[1,2], Anjunyi Fan[1,2], Yuchao Yang[1,2,3,4*] and Bonan Yan[1,2*]

[1]Institute for Artificial Intelligence, Peking University, Beijing, China [2]Beijing Advanced Innovation Center for Integrated Circuits, School of Integrated Circuits, Peking University, Beijing, China [3]School of Electronic and Computer Engineering, Peking University, Shenzhen, China [4]Chinese Institute for Brain Research, Beijing, China
{lijiayi, hongxiaozhao}@stu.pku.edu.cn, {yws2017, yihanfu, daijingshi, anjunyif, yuchaoyang, bonanyan}@pku.edu.cn

***Abstract*—Reinforcement learning (RL) is an effective machine learning approach that enables artificial intelligence agents to perform complex tasks and make decisions in dynamic situations. Training an RL agent demands its repetitive interaction with the environment to learn optimal policies. To efficiently collect training data, parallelizing environments is a widely used technique by enabling simultaneous interactions between multiple agents and environments. However, existing CPU-based RL software frameworks face a key challenge of slow multi-environmental update computation. To solve this problem, we present a novel FPGA-based RL accelerating framework–PEARL. PEARL instantiates multiple parallel environments and accelerates them with a carefully designed pipeline scheme to hide data transfer latency within the computation time. We evaluate PEARL on respective RL environments and achieve 4.36× to 972.6× speedup over the existing fastest software-based framework for parallel environment execution. When scaling the number of environments from 1024 to 43008 (42×) in *CliffWalking* benchmark, the power consumption increases marginally by 3%, while LUT and flip-flops utilization rise by 2.24× and 3.08×, respectively. This demonstrates efficient resource usage and power management in PEARL. Further, PEARL allows users to define and add their environments within the framework flexibly. We have established an open-source repository for users to utilize and expand. We also implement PEARL with the existing RL algorithm and save 7%-15% training time. All the source code is available online https://github.com/Selinaee/FPGA_Gym.**

***Index Terms*—Reinforcement learning, domain-specific accelerator, FPGA, parallel environments computing, multi-agent reinforcement learning**

I. Introduction

Reinforcement Learning (RL) is notably efficacious in solving sequential decision-making tasks. It provides artificial intelligence (AI) agents the capability of making decisions within an environment with a goal to optimize cumulative rewards over time. RL paradigms play a crucial role in training systems across fields such as games and strategies, robotics control, and natural language processing, as exemplified by DeepMind's AlphaGo and OpenAI's ChatGPT [1]. One significant challenge inherent to RL pertains to managing the trade-off between exploration (acquiring knowledge about the environment through experimentation) and exploitation (leveraging existing information for reward maximization) [2], [3].

In RL, agents interact with the environment to gain experience. Through this repetitive interaction, agents learn optimal action policies to achieve their goals (Fig. 1). A single interaction process is often described as a *step*. The specific process is illustrated in Fig. 1(a): an agent selects its *action* based on its *observation* of the environment and its *action policy*. Upon receiving the *action*, the environment *state* transitions to a new state called *next state*. At the same time, the agent receives *next observation* about this state, a *reward* signal corresponding to the *action*, and an indication of whether the current episode has terminated called *termination*. After one *step*, the agent will add the tuple $\langle observation, action, next\ observation, reward, termination \rangle$ into a dynamic dataset specific for this task for training to update a new *action policy* (often in the form of neural networks). With tens of millions of *steps*, the agent gradually learns the optimal *action policy*. This process requires vast computing resources. Simultaneously parallel environments as Fig. 1(b) is a popular way to generate more knowledge that can be added to dynamic datasets, improving data utilization.

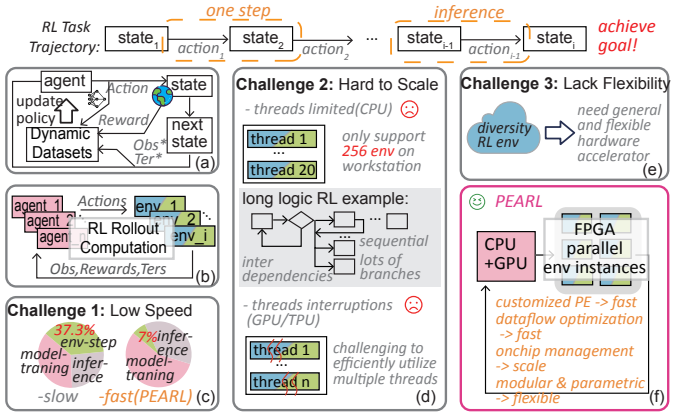

Fig. 1. Introduction. (a) One RL environment step rollout, (b) Parallel RL environments, (c) Challenge 1: Low Speed, (d) Challenge 2: Hard to Scale, (e) Challenge 3: Lack of Flexibility, (f) PEARL solution and features. *Obs and Ter represent *observation* and *termination* respectively.

There are three significant challenges for the aforementioned parallel RL environments. (a) The computing latency of parallel RL environment steps is high [4]. As Fig. 1(c) shows, in the process of parallel 64 *CartPole* environments for training, the time of environments step accounts for 37.3% of the entire time. (b) The deployment of RL at scale remains a significant challenge. Despite the potential of GPUs and Tensor Processing

978-3-9826741-0-0/DATE25/© 2025 EDAA

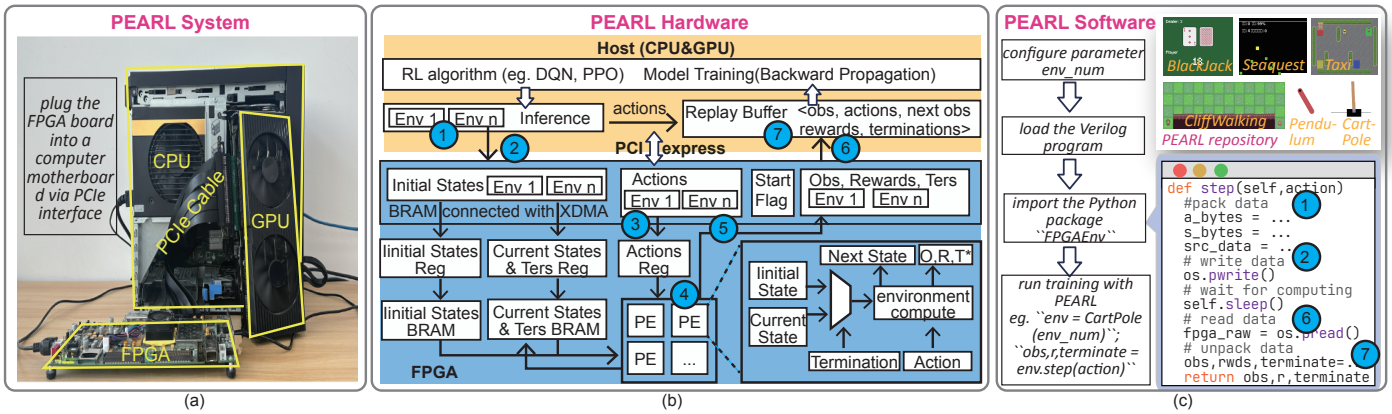

Fig. 2. (a) PEARL hardware implementation. (b) PEARL hardware architecture. (c) PEARL software interface and workflow. *O, R, T represents *observation*, *reward* and *termination*.

Units (TPUs) for parallel processing, their effectiveness is limited in RL caused of the fact that the logic complex RL tasks involve long sequential Markov decision processes. This complexity might lead to thread divergence and undermine the efficiency of parallel processing in RL environments. (c) There is a lack of a general and flexible hardware parallelization solution to address the high diversity of RL environments.

To tackle these challenges, we propose PEARL, a field-programmable gate array (FPGA) based hardware-software codesign acceleration framework, for RL environments parallel computing [5]–[7]. The main contributions of this work are:

- To improve *computing throughput*, we propose a pipeline scheduling scheme that achieves high throughput by integrating efficient data compression and local-store techniques, hiding the data transfer latency of caching intermediate computation results behind ongoing environment calculations.
- For *scalability*, we propose a series of on-FPGA intermediate data management methodologies for RL, focusing on synchronization between multiple instances, self-termination control and efficiently utilizing on-chip resources. As the number of parallel environments increases, the demand for on-chip resources does not increase significantly.
- For *flexibility*, we make PEARL framework modular and parametric. We publish a programming template for customizing users' environments (https://github.com/Selinaee/FPGA_Gym/tree/main/Template). PEARL is compatible with multiple algorithms and environments.

Through respective RL benchmarks evaluation, PEARL achieves a throughput improvement ranging from 4.36× to 972.6× and power efficiency improvements ranging from 6× to 2651× compared to EnvPool [4], the fastest CPU-based environment parallelism library. Additionally, by integrating the off-policy Deep Q-Network (DQN) [8] and the on-policy Proximal Policy Optimization (PPO) [9] algorithms, PEARL reduces training time by 7%-15% compared to EnvPool. In *CliffWalking* benchmark, scaling the number of environments from 1024 to 43008 (42×) results in only a 3% increase in power consumption, while look-up tables (LUTs) and flip-flops (FFs) utilization increases by 2.24× and 3.08×, respectively.

II. Preliminaries

A. Reinforcement Learning Algorithm

There is extensive research on RL algorithms. They can be broadly categorized into offline and online RL algorithms. Offline policy algorithms store the data in an experience replay buffer and sample it as needed for training, while online policy algorithms directly use feedback data as it is received.

DQN is a foundational off-policy algorithm in the field of RL, while PPO is one of the most widely used on-policy algorithms, dominating policy optimization due to its stability. Stable-Baselines3 is a popular and easy-to-use library that provides reliable implementations of state-of-the-art RL algorithms for training and evaluating agents. PEARL offloads the computational load of the environment to FPGA and can be highly compatible with these algorithms. To advance RL algorithms research, multiple RL environment repositories have been developed to evaluate algorithms' effectiveness. Among these, the Gymnasium library stands out for its standardized interfaces and widespread use [10]. The module interfaces of PEARL are consistent with those of Gymnasium.

B. RL Acceleration Framework

RL acceleration framework is a programming framework to accelerate the execution of RL environments. There exist many related works based on CPUs with just-in-time compilation and GPUs. Gymnasium's VectorEnv [10] implements parallelization with Python. In contrast, EnvPool [4] leverages a C++ backend to optimize parallel computing and memory management, resulting in a 2.8× speedup over its Python-based counterpart. However, as the number of instantiated environments increases, the parallelization is constrained by the fixed number of available CPU threads, thereby limiting scalability. Sample Factory [11] introduces a fully asynchronous approach termed "double-buffered sampling", where environment execution and neural network forward passes are processed concurrently on separate sets of environments, significantly enhancing throughput. Despite this performance gain, the method fails to support RL algorithms that require synchronous execution, thus limiting its flexibility. PodRacer exploits the massively

parallel processing capabilities of TPUs to optimize the execution of multiple RL environments in parallel. However, its implementation details are not accessible for customized environments [12]. WarpDrive achieves high throughput by translating the environment dynamics and the agent's decision-making processes into GPU-compatible CUDA code [13]. The drawback of this approach is that the environments must be purely compute-based, i.e., matrix operations so that they efficiently and easily accelerated on GPUs and TPUs [14].

III. PEARL Hardware Architecture & Software Interface

A. Architecture

Fig. 2 presents the PEARL methodology, including hardware architecture and software interface. There are three major components in PEARL implementation: the host (CPU/GPU), the transmission module (PCI express interface), and the FPGA implementation of parallel environments computing accelerator [15]–[17]. The host utilizes the tuples $\langle \text{obs}, \text{action}, \text{next obs}, \text{reward}, \text{termination} \rangle$ from the replay buffer to optimize model weights, update the *action policy* using efficient RL algorithms, and infer the agent's *action* based on the *current observation* and *action policy*. The entire workflow is as follows:

①: The host packs the *actions* into bytes.
②: Thess bytes information are sent to FPGA via PCI express.
③-⑤: FPGA performs the environments *step* compute.
⑥: The host reads the results named "fpga_raw".
⑦: The host unpacks the "fpga_raw" from bytes form into *next observations*, *rewards*, and *terminations*.

The tuples are added to the replay buffer to train a better *action policy* until the agent achieves the task goal. The transmission module acts as a critical bridge in the architecture, managing data transfer and synchronization between the host and the FPGA [18].

We model the workflow and the latency of each step. T_1 to T_7 represent time costs of step ① to ⑦ depicted in Fig. 2(b) and (c), respectively:

$$
\begin{aligned}
T_1 &\approx \alpha \cdot k_1 + \beta & T_7 &\approx \alpha \cdot k_2 + \beta \\
T_2 &\approx \frac{k_1 \times env\_num}{v_t} & T_6 &\approx \frac{k_2 \times env\_num}{v_t} \\
T_3 &= \frac{k_1 \times env\_num}{v_b} & T_5 &= \frac{k_2 \times env\_num}{v_b} \\
T_4^{\text{without\_pipeline}} &= c_c \times env\_num & T_4^{\text{with\_pipeline}} &= c_c
\end{aligned}
\tag{1}
$$

where α (unit: seconds/GByte) is the per-byte processing time for data packing/unpacking, β (unit: seconds/GByte) is the fixed overhead time for initiating the packing/unpacking process, env_num is the number of environments to update, k_1 (unit: GByte) is the data size of *action* per environment, k_2 (unit: GByte) is the data size of *observation*, *reward*, and *termination* per environment, v_b (unit: GBytes/second) is the transfer speed between BRAM and register, v_t (unit: GBytes/second) is the transfer (PCI express) bandwidth between CPU and FPGA, c_c (unit: second) is the time to compute one environment *step*. We illustrate the time in Fig. 3 and optimize each *step*.

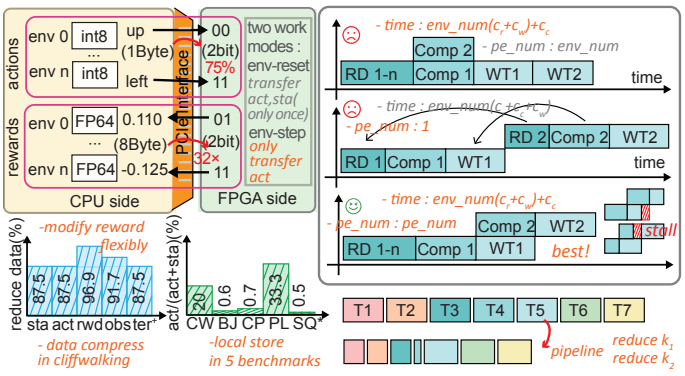

Fig. 3. Dataflow optimization: data compressing, local storing, and pipelining of data transferring and computing. $^+$Sta, act, rwd, obs, and ter represent STA_WL, ACT_WL, RWD_WL, OBS_WL, and Ter_WL, where STA_WL, ACT_WL, RWD_WL, OBS_WL, and Ter_WL (unit: bits) represent the bit width of the *state*, *action*, *reward*, *observation*, and *termination* information for a single environment, respectively. *CW, BJ, CP, PL, and SQ represent the benchmarks of *CliffWalking*, *BlackJack*, *CartPole*, *Pendulum*, and *Seaquest*, respectively.

B. DataFlow Optimization

FPGAs enable highly precise customization of data flow and control flow, with each logic unit tailored to specific task requirements [19]–[22]. This allows for rapid execution of bitwise operations, custom mathematical computations, and specialized LUTs, resulting in significantly higher computational speeds. Each environment *step* is completed within tens of clock cycles, as shown in Table I. However, there are data transfer challenges between heterogeneous platforms. We optimize dataflow for RL by reducing the transformation data (reduce k_1, k_2) and pipeline technique as shown in Fig. 3.

First, we compress the data transfer volume. The CPU side initially represents *actions* in a more extensive data format, e.g. INT8 for *actions*. These are then compressed to a smaller bit representation (e.g. 2 bits) on the FPGA side for transmission. In addition, the computation results *rewards* are represented on the FPGA in category coding. After the CPU retrieves the data, it can efficiently process and convert it into actual values using tools like NumPy. As the example in Fig. 3, this approach reduces the data transfer volume by 32×. Moreover, the ability to flexibly modify rewards, which is unique to RL, allows users to fine-tune parameters effectively without rebuilding and reloading the Verilog program. Through data compression, the key coefficients k_1 and k_2 in Eq. 1 are significantly reduced. For example, in *CliffWalking*, the technique reduces data length of

TABLE I
Typical Environments For Benchmarking PEARL

Environment Name	Environment Type	Obs/Action space type	Observability	Compute Time(ns)
CartPole [23]	physical control	discrete	full	800
Pendulum [10]	physical control	continuous	full	960
MountainCar [24]	physical control	discrete	full	840
CliffWalking [25]	gird world	discrete	full	20
FrozenLake [25]	gird world	discrete	full	20
Taxi [26]	gird world	discrete	full	10
BlackJack [10]	strategy game	discrete	partial	70
Seaquest [27]	atari game	discrete	full	80

state, *action*, *reward*, *observation*, and *termination* by 87.5%, 87.5%, 96.9%, 91.7%, and 87.5%, respectively.

Second, the *initial environment states* are loaded onto the FPGA and cached to reduce transfer data [28], [29]. PEARL has two work modes: in env-reset mode, the system initializes the environments. The host sends *initial environment states* and *actions* to FPGA; in env-step mode, the host only sends *actions* to FPGA. After each *step*, the computation results *next environment states* are stored inside FPGA as the next step environment *current states*. Only the necessary information for training, i.e. *next observations*, *rewards*, and *terminations*, shall communicate with the CPU. In most RL environments, compared with environment *states*, the agent *action* data length is minimal. This design significantly reduces the data volume k_1 sent from the CPU to the FPGA. In *CliffWalking*, *BlackJack*, *CartPole*, *Pendulum*, and *Seaquest* benchmark, the ratio of bidwidth act/(act+sta) is 20%, 0.6%, 0.7%, 33.3% and 0.5%, respectively (Fig. 3).

Third, we design an efficient pipeline for high throughput. As shown in Fig. 3, if we only use one compute unit to sequentially execute ④environment compute, from ③ to ⑤, the total computation time, as given by Eq. 2, is:

$$Loop\_time^{\text{serial}} = env\_num \cdot (c_r + c_c + c_w) \quad (2)$$

$$Loop\_time^{\text{parallel}} = env\_num \cdot (c_r + c_w) + c_c \quad (3)$$

where c_r (unit: second) is the time computing unit takes to read one environment's corresponding *action*, c_w (unit: second) is the time computing unit takes to write one environment's *observation*, *reward*, and *termination*. In contrast, PEARL significantly reduces the total computation time by leveraging FPGA to generate multiple compute units for parallel execution [30]. When we employ env_num processing elements to parallelize the execution of ④ environment compute, the required time becomes Eq. 3. The single BRAM one port has connected with the PCIe-XDMA (DMA Subsystem for PCIe) IP core [31], [32], only one port can used to write or read data at the same time, so ③ and ⑤ can not overlap. In addition, ⑤ usually needs more time than ③, as compared to k_2, k_1 is much smaller. Therefore, we design a pipeline scheme between ④ and ⑤. As Fig. 2, there are pe_num computing units, one computing unit can compute one environment in c_c. env_num environments are divided into (env_num/pe_num) groups for computation. When the first batch of pe_num environments completes their calculations, the results are cached and written into BRAM, and the second batch of environments begins computation simultaneously. This process continues until all the env_num environments have been processed. When $pe\_num = c_c/c_w$, the pipeline achieves minimal idle time and the loop time is the same with env_num computing units. By time-division multiplexing, PEARL achieves optimal resource utilization.

C. On-FPGA Intermediate Data Management

In PEARL, how to manage intermediate data is central to efficiently handling the data flow and ensuring the high-throughput execution of RL environments. The following method outlines the on-chip management strategy implemented

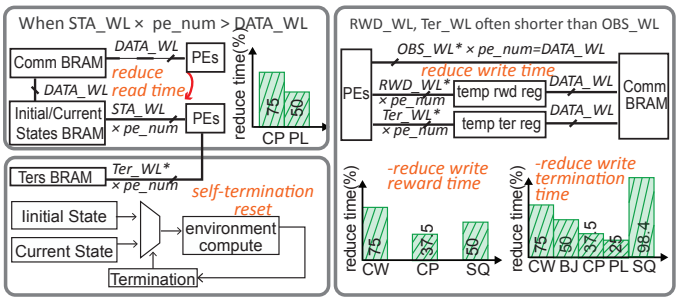

Fig. 4. Intermediate data management.

in our design, as illustrated in Fig. 2 and Fig. 4. There are three BRAMs in PEARL: *Comm-BRAM*, *Initial-States-BRAM*, and *Current States-&-Ters-BRAM*. Comm-BRAM stores *initial states*, *actions*, *start_flag*, *observations*, *rewards*, and *terminations*. Comm-BRAM allows efficient data transfer between the CPU and the FPGA via the XDMA protocol, but the bandwidth is limited in DATA_WL.

First, we reduce the read data time by using higher bandwidth BRAMs within the FPGA chip. For example, if we only use Comm-BRAM, when (STA_WL $\times$ pe_num $>$ DATA_WL), the read data time of one group with pe_num environments needs (STA_WL $\times$ pe_num)/DATA_WL clocks. Using higher bandwidth Initial-States-BRAM and Current-States-BRAM reduces read data time to one clock cycle. DATA_WL (unit: bits) is the Comm-BRAM bandwidth in bits per clock cycle. In *CartPole* and *Pendulum* benchmark, the technique reduces 75%, 50% read *initial states* and *current states* time from BRAM, respectively.

Second, we utilize self-termination reset to manage environment resets in a vectorized parallel setting, ensuring efficient and uninterrupted execution. After each *step*, the *termination* message of every environment, along with the *current states*, will be cached in BRAM. The *initial state*, *current state*, and *termination* signal are synchronously fed into the PE, where the PE will decide whether to use the *initial state* or the *current state* based on the *termination* signal (Fig. 2(b), Fig. 4).

Third, we reduce the time of writing computation results by using temporary registers. Since *reward* and *termination* usually occupy smaller bits than *observation* in one RL environment. These pieces of information are stored at different addresses in Comm-BRAM. Writing all of these data simultaneously would result in unnecessary time consumption. We use a counter for the written state judgment. There are three write states: (a) write only *observations*, (b) write *observations* and *rewards*, and (c) write *observations*, *rewards*, and *terminations*. *Reward* and *termination* information with a data length less than DATA_WL will be temporarily stored separately in registers until the combined data length reaches DATA_WL, at which point it will be written to Comm-BRAM. In *CliffWalking*, *CartPole*, and *Seaquest* benchmark, the technique reduces 75%, 37.5%, and 50% write *reward* time to Comm-BRAM, respectively. In *CliffWalking*, *BlackJack*, *CartPole*, *Pendulum*, and *Seaquest* benchmark, the technique reduces 75%, 50%, 37.5%, 25% and 98.4% write *termination* time to Comm-BRAM, respectively.

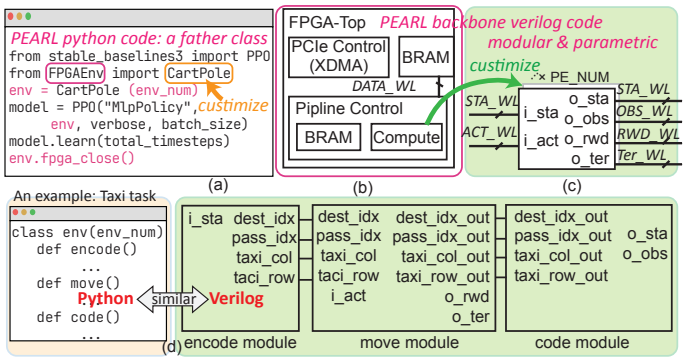

Fig. 5. Modular & Parametric Design. (a) RL training with PEARL, (b) PEARL backbone Verilog code, (c) customized instance, (d) an example of a customized environment implementation in PEARL

D. Modular & Parametric Implementation

PEARL is designed to be modular and parametric, allowing for flexible customization for new RL environments and good compatibility with RL training algorithms, such as DQN and PPO. Fig. 5(a) shows the example of using PEARL run *CartPole* with PPO algorithm from RL training framework Stable-Baselines3. Users can modify the env_num with given environments or fill their design logic according to the standard interfaces and modify some parameters. On the software part, we define an abstract class to help users to expand new environments. To add a new environment, import the FPGAEnv class, modify parameters such as STA_WL in the software, and adjust data packaging according to the specific characteristics of the environment. On the hardware part, the backbone of the data flow and on-FPGA intermediate data management Verilog code have already been established, as shown in Fig. 5(b). Users just modify parameters and customize the single compute instance according to the standard interfaces, as shown in Fig. 5(c). In Fig. 5(d), we provide an example of implementing *Taxi* from Python code to Verilog code. More detailed guidance is available in here.

IV. Evaluation

A. Benchmarks & Experimental Setup

We evaluate PEARL using widely-used representative RL benchmarks with different types (see Table I). The basic parameters of these environments are listed in Table II. For hardware configuration, we adopt an off-the-shelf Xilinx VC707 development board with 8-lane PCIe2.0 interfacing to a workstation with an Intel Core i9-10900K CPU and 128GB DDR4 memory. The acceleration framework baselines for this work are EnvPool [4] and VectorEnv [10]. Prior to this work, EnvPool was the latest and fastest framework for the aforementioned 5 environments. VectorEnv, a vectorized environment in the Gymnasium libraries, employs multithreading for parallelization.

B. Performance Boost

Fig. 6 shows the high throughput of PEARL. The x-axis represents five respective benchmarks. Each benchmark includes performance comparisons of four different baselines: VectorEnv, EnvPool, PEARL w/o pipeline, and our optimized

TABLE II
Experiment Setup: Environment Parameter

Env Name	*Cliff-Walking*	*Black-Jack*	*Cart-Pole*	*Pendulum*	*Seaquest*
STA_WL	8	152	128	64	736
ACT_WL	2	1	1	32	3
OBS_WL	8	16	128	96	512
RWD_WL	1	2	1	32	32
DATA_WL	1024	512	32	32	1024
pe_num	256	256	20	24	16
c_c	20	80	800	960	80

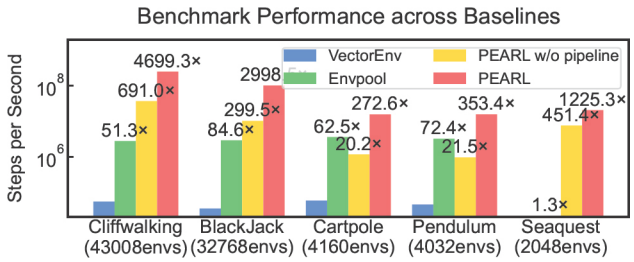

Fig. 6. Computing throughput comparison between VectorEnv [10], EnvPool [4], and PEARL (this work).

approach PEARL. PEARL w/o pipeline refers to the case where pipelining and parallelism optimizations are not used in computing execution, and environment *states* are not stored within the FPGA. The y-axis, on a log scale, measures the throughput (steps per second) of each baseline. For stability, we iterate 10,000 *steps* and calculate the average performance. PEARL has a performance boost in all benchmarks, achieving 4699.3×, 2998.5×, 272.6×, 353.4×, and 1225.2× speedup versus the most popular VectorEnv in *CliffWalking*, *BlackJack*, *CartPole*, *Pendulum*, and *Seaquest*, respectively. Compared to EnvPool, the existing fastest software-based framework for parallel environment execution, PEARL achieves 4.36× to 972.6× throughput. Compared to "w/o pipeline", PEARL achieves 2.7× to 13.5× throughput of these representative tasks, demonstrating the effectiveness of our dataflow optimizations. Due to the limitations of CPU threads, VectorEnv can only support a maximum of 256 environments (env_num) for Seaquest on the workstation with 20 threads. Therefore, the data for VectorEnv is measured at 256 environments.

Fig. 7 illustrates the training acceleration performance and compatibility of PEARL with various RL algorithms and frameworks. The x-axis represents training time (in seconds), while the y-axis reflects the average return, corresponding to the cumulative rewards accumulated by the agent over multiple steps. A higher average return indicates more effective learning and improved decision-making by the agent throughout the training process. PEARL reduces training time by 7%, 12%, and 15% compared to EnvPool, and by 18%, 61%, and 44% compared to VectorEnv when training with PPO on *CartPole*, DQN on *CartPole*, and DQN on *CliffWalking*, respectively.

C. Power Efficiency

Fig. 8 shows the high power efficiency of PEARL. The x-axis represents the env_num. The y-axis represents the power efficiency (steps/seconds/W). The power of PEARL consists of two parts: the power of data packing and transfer on the

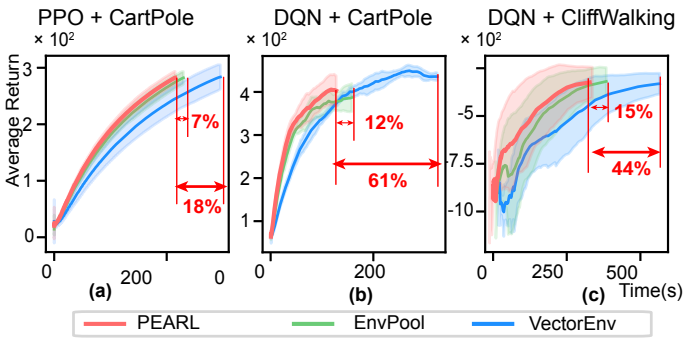

Fig. 7. RL end-to-end training performance of VectorEnv, EnvPool, and PEARL (this work).

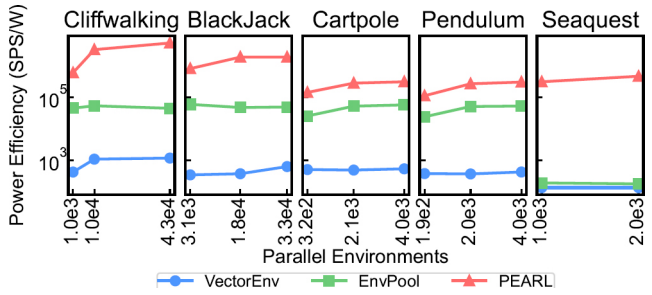

Fig. 8. Power Efficiency of VectorEnv, EnvPool, and PEARL (this work).

CPU and the FPGA post-implementation power. Compared with VectorEnv and EnvPool, PEARL achieve 4583×, 5134×, 583×, 738×, 3483× and 120×, 40×, 6×, 6×, 2651× power efficiency respectively, in *CliffWalking*, *BlackJack*, *CartPole*, *Pendulum*, and *Seaquest* benchmark.

D. FPGA Utilization

Fig. 9 shows the efficient data management of PEARL. The x-axis represents the *env_num*. The y-axis represents the resource utilization of LUTs, FFs, and BRAMs on FPGA. As the number of parallel environments increases, the FPGA resource usage remains relatively stable, with no significant growth in resource consumption. For example, when scaling the number of environments from 1024 to 43008 (42×) in the CliffWalking benchmark, the power consumption increases marginally by 3%, and LUT and FF utilization rise by 2.24× and 3.08×, respectively.

E. Comparison With Related Works

Table III compares PEARL with the existing software-based RL acceleration frameworks. It showcases the introduction of FPGA to build a complete RL acceleration, which brings performance and efficiency gains. Table IV lists and compares the latest related FPGA-based RL domain-specific accelerators. Because our focus is on increasing the throughput of environment steps, while other works concentrate on the action generation process, the comparison is not directly aligned. Therefore, we only compare the types of environments supported. PEARL focuses on the parallel environment execution instead of specific whole learning algorithms to find the sweet point of flexibility, versatility, and efficiency.

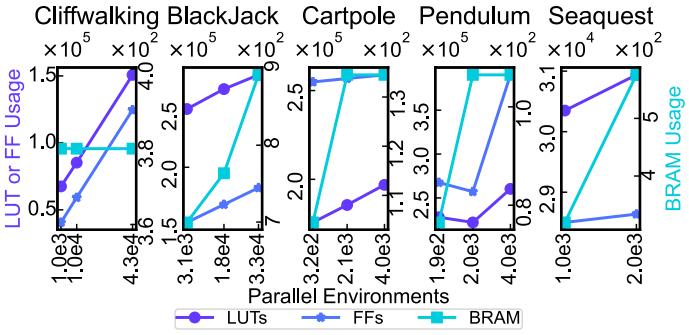

Fig. 9. FPGA resource utilization of 5 environments across different parallel environment numbers.

TABLE III
COMPARISON OF PARALLEL RL ENVIRONMENT FRAMEWORK

Environment Name	VectorEnv [10]	Envpool [4]	**PEARL (This work)**
Hardware Platform	CPU	CPU	**CPU+FPGA**
Steps/Second (SPS)	54035 (1×)	2773265 (51×)	**253926929** (4699×)
Power Effficiency (SPS/W)	1172.89 (1×)	44845.81 (38×)	**5375707.70** (4583×)
Max Environment Number	<2000	<2000	>**40000**

V. CONCLUSION

In this work, we present PEARL, an FPGA-based acceleration framework (with hardware and ready-to-use software interface) for parallel RL environments. We leverage the inherent flexibility and parallelism of FPGAs, significantly improving the computational throughput and power efficiency compared to traditional CPU-based solutions. PEARL supports and is evaluated with various RL benchmarks and achieves satisfactory speedup and power efficiency improvements. PEARL is compatible with popular RL algorithms, e.g. DQN and PPO. PEARL showcases the potential of FPGA-based architectures in accelerating RL tasks. Future research shall explore incorporating GPUs and CPUs for further enhancement.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (T2350006, 92264201, 92364102), by the 111 Project under Grant B18001. This work is sponsored by Beijing Nova Program. Yuchao Yang acknowledges Fok Ying-Tong Education Foundation and Tencent Foundation through the Xplorer Prize.

TABLE IV
COMPARISON OF RL ACCELERATORS

Work	Hardware Platform	Acceleration	RL Env. Type
ASPLOS'19 [33]	Virtex VCU1525	A3C†	Discrete
FCCM'20 [34]	Alveo U200	PPO†	Continuous
ICCAD'20 [35]	ASIC	$A3C^{2}$†	Both
DAC'21 [36]	Alveo U50	DDPG†	Continuous
ICCAD'22 [37]	Alveo 280	HDQL†	Discrete
IPDPSW'21 [38]	PYNQ-Z1	DQN†	Discrete
FPL'22 [39]	Alveo U50	TD3†	Continuous
CF'22 [40]	Alveo U200	Replay Buffer	Discrete
PEARL	**Virtex VC707**	**Environment**	**Both**

† Support only one specific learning algorithm.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the Game of Go Without Human Knowledge," *Nature*, 2017.

[2] Z. Ma, H. Guo, J. Chen, Z. Li, G. Peng, Y.-J. Gong, Y. Ma, and Z. Cao, "MetaBox: A Benchmark Platform for Meta-Black-Box Optimization with Reinforcement Learning," *Conference on Neural Information Processing Systems (NeruIPS)*, 2024.

[3] Z. Yuan, S. Yang, P. Hua, C. Chang, K. Hu, and H. Xu, "RL-ViGen: A Reinforcement Learning Benchmark for Visual Generalization," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[4] J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. Yan, "Envpool: A Highly Parallel Reinforcement Learning Environment Execution Engine." in *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[5] A. Kouris, S. I. Venieris, and C.-S. Bouganis, "A Throughput-Latency Co-Optimised Cascade of Convolutional Neural Network Classifiers," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020.

[6] O. Sharif and C.-S. Bouganis, "A Framework for Designing Scalable Gaussian Belief Propagation Accelerators for use in SLAM," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

[7] M. Shafique, T. Theocharides, C.-S. Bouganis, M. A. Hanif, F. Khalid, R. Hafiz, and S. Rehman, "An Overview of Next-generation Architectures for Machine Learning: Roadmap, Opportunities and Challenges in the IoT Era," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[8] T. Hester, M. Vecerík, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, G. Dulac-Arnold, J. P. Agapiou, J. Z. Leibo, and A. Gruslys, "Deep Q-learning From Demonstrations," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint*, 2017.

[10] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," 2023.

[11] A. Petrenko, Z. Huang, T. Kumar, G. S. Sukhatme, and V. Koltun, "Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning," in *International conference on machine learning (ICML)*, 2020.

[12] M. Hessel, M. Kroiss, A. Clark, I. Kemaev, J. Quan, T. Keck, F. Viola, and H. van Hasselt, "Podracer Architectures for Scalable Reinforcement Learning," *arXiv preprint*, 2021.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A Performance Study of General-purpose Applications on Graphics Processors Using CUDA," *Journal of parallel and distributed computing (JPDC)*, 2008.

[14] T. Lan, S. Srinivasa, H. Wang, and S. Zheng, "WarpDrive: Fast End-to-End Deep Multi-Agent Reinforcement Learning on a GPU," *Journal of Machine Learning Research (JMLR)*, 2022.

[15] T. Martínek, J. Kořenek, and T. Čejka, "LGBM2VHDL: Mapping of LightGBM Models to FPGA," in *IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2024.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, 2011.

[17] A. Filgueras, M. Vidal, D. Jiménez-González, C. Álvarez, and X. Martorell, "FPGA Framework Improvements for HPC Applications," in *International Conference on Field Programmable Technology (ICFPT)*, 2023.

[18] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[19] D. Ludovici, F. Gilabert, S. Medardoni, C. Gomez, M. Gomez, P. Lopez, G. Gaydadjiev, and D. Bertozzi, "Assessing Fat-tree Topologies for Regular Network-on-chip Design under Nanoscale Technology Constraints," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009.

[20] B. Salami, K. Parasyris, A. Cristal, O. Unsal, X. Martorell, P. Carpenter, R. De La Cruz, L. Bautista, D. Jimenez, C. Alvarez, S. Nabavi, S. Madonar, M. Pericàs, P. Trancoso, M. Abduljabbar, J. Chen, P. N. Soomro, M. Manivannan, M. Berge, S. Krupop, F. Klawonn, A. Mekhlafi, S. May, T. Becker, G. Gaydadjiev, H. Salomonsson, D. Dubhashi, O. Port, Y. Etsion, L. Quoc Do, C. Fetzer, M. Kaiser, N. Kucza, J. Hagemeyer, R. Griessl, L. Tigges, K. Mika, A. Hüffmeier, M. Pasin, V. Schiavoni, I. Rocha, C. Göttel, and P. Felber, "LEGaTO: Low-Energy, Secure, and Resilient Toolset for Heterogeneous Computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020.

[21] M. Zahedi, G. Custers, T. Shahroodi, G. Gaydadjiev, S. Wong, and S. Hamdioui, "SparseMEM: Energy-efficient Design for In-memory Sparse-based Graph Processing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[22] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev, "The SARC Architecture," *IEEE Micro*, 2010.

[23] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, 1983.

[24] A. W. Moore, "Efficient Memory-based Learning for Robot Control," University of Cambridge, Computer Laboratory, Tech. Rep., 1990.

[25] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," *A Bradford Book*, 2018.

[26] T. G. Dietterich, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," *arXiv preprint*, 1999.

[27] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: an Evaluation Platform for General Agents," *arXiv preprint*, 2013.

[28] G. Zgheib and P. Ienne, "Evaluating FPGA Clusters under Wide Ranges of Design Parameters," in *27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[29] E. Pissadakis, N. Alachiotis, P. Skrimponis, D. Theodoropoulos, T. Korakis, and D. Pnevmatikatos, "ReFiRe: Efficient Deployment of Remote Fine-Grained Reconfigurable Accelerators," in *International Conference on Field-Programmable Technology (FPT)*, 2018.

[30] S. Perri, C. Zambelli, D. Ielmini, and C. Silvano, "Digital In-Memory Computing to Accelerate Deep Learning Inference on the Edge," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024.

[31] I. Mavroidis, I. Papaefstathiou, L. Lavagno, D. S. Nikolopoulos, D. Koch, J. Goodacre, I. Sourdis, V. Papaefstathiou, M. Coppola, and M. Palomino, "ECOSCALE: Reconfigurable computing and runtime system for future exascale systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[32] K. Koliogeorgi, N. Voss, S. Fytraki, S. Xydis, G. Gaydadjiev, and D. Soudris, "Dataflow Acceleration of Smith-Waterman with Traceback for High Throughput Next Generation Sequencing," in *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[33] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-Accelerated Deep Reinforcement Learning," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[34] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating Proximal Policy Optimization on CPU-FPGA Heterogeneous Platforms," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020.

[35] 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 (ICCAD)*, 2020.

[36] J. Yang, S. Hong, and J.-Y. Kim, "Fixar: A Fixed-Point Deep Reinforcement Learning Platform with Quantization-Aware Training and Adaptive Parallelism," in *Design Automation Conference (DAC)*, 2021.

[37] 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 (ICCAD)*, 2022.

[38] H. Watanabe, M. Tsukada, and H. Matsutani, "An FPGA-based On-Device Reinforcement Learning Approach using Online Sequential Learning," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021.

[39] C.-W. Hu, J. Hu, and S. P. Khatri, "TD3lite: FPGA Acceleration of Reinforcement Learning with Structural and Representation Optimizations," in *Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[40] Y. Meng, C. Zhang, and V. Prasanna, "FPGA Acceleration of Deep Reinforcement Learning using On-Chip Replay Management," in *ACM International Conference on Computing Frontiers (CF)*, 2022.