FPGA-Gym: An FPGA-Accelerated Reinforcement Learning Environment Simulation Framework

Jiayi Li[◇], Hongxiao Zhao[◇], Wenshuo Yue[◇], Yihan Fu[◇], Daijing Shi[◇], Anjunyi Fan[◇], Qinghao Wang[◇], Yaodong Yang[◇†], Bonan Yan[◇*]
[◇]Peking University [†]Beijing Institute for General AI
[*]Corresponding author: bonanyan@pku.edu.cn

Abstract

Reinforcement learning (RL) faces the key challenge of balancing exploration (gathering information about the environment through trials) and exploitation (exploiting current knowledge to maximize rewards), especially in the open world. To boost exploration efficiency, parallel environment execution is a widely used technique that instantiates multiple environments and executes them in parallel. However, this is computationally challenging using CPUs limited by the total thread numbers. In this work, we present FPGA-Gym, an FPGA-CPU joint acceleration framework for accelerating RL environment parallel executions. By offloading environment steps to FPGA hardware, FPGA-Gym achieves a 4.36 to 972.6× speedup over the existing fastest software-based framework for parallel environment execution. Moreover, FPGA-Gym is a general RL acceleration framework compatible with existing RL algorithms and frameworks. Its modular and parameterized features allow users to conveniently customize new environments without extensive FPGA knowledge. We demonstrate multiple representative RL benchmarks (e.g. Cartpole, CliffWalking, Seaquest etc.) with Deep Q-Network and proximal policy optimization algorithms. Additionally, we provide a standard environment library similar to Gymnasium based on FPGA-Gym framework. The framework and the library are available at https://github.com/Selinaee/FPGA_Gym.

1 Introduction

Reinforcement learning (RL) is highly effective in addressing sequential decision-making problems. It empowers artificial agents to make decisions within an environment to maximize cumulative rewards over time. RL faces the key challenge of balancing exploration (gathering information about the environment through trials) and exploitation (exploiting current knowledge to maximize rewards) [1, 2], especially in the open world. To boost exploration efficiency, parallel environment execution is a widely used technique that instantiates multiple environments and executes them in parallel, allowing multiple agents to interact with the environment simultaneously, thereby rapidly accumulating experience (Fig. 1(a)). However, updating a large number of environments in parallel places a substantial demand on computational resources [3–5]. Fig. 1(b) reveals that parallel environment step computation inflicts long latency, highlighting the significant computational demands required for environment evolution.

The key challenge in parallel RL environment execution lies in rapidly updating numerous parallel environments [3, 4, 6, 7]. On conventional hardware platforms, multithreading is the major tackle for parallelism. Central processing units (CPUs) can only manage a certain level of parallelism with limited thread numbers. Graphics processing units (GPUs) [8] and tensor processing units (TPUs) are also under intense investigation to parallelize environment execution for RL. However, RL environment execution presents unique challenges in sequential Markov decision process (MDP)

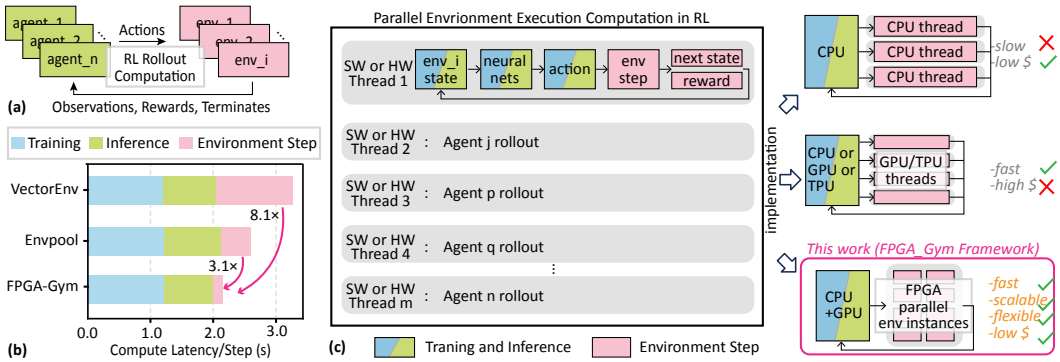

Figure 1: (a) Rollout computation of RL; (b) Compute latency breakdown in different RL acceleration frameworks; (c) Illustration of typical realizations of parallel RL environment execution.

computation [9–12]. The agent-environment interactions are modeled as Markov chains, characterized by iterative dynamic state transitions, such as environment resets and reward functions [13–15]. These elements are difficult to parallelize and scale efficiently on GPUs or TPUs (Fig. 1c).

To address the challenge, this work proposes to use field-programmable gate arrays (FPGAs) as the hardware acceleration foundation [16–21]. FPGAs are programmable semiconductor chips consisting of reconfigurable digital logic circuit blocks and interconnects. **To the best of our knowledge, this work presents the first holistic FPGA-CPU joint acceleration framework, termed as `FPGA-Gym`, for massively instantiating and parallelizing RL environment computation.** It offloads the execution of environment steps onto FPGAs via a high-bandwidth peripheral component interconnect express (PCIe) interface to the CPU. This work aims to overcome the limitations of CPUs and GPUs in handling logically complex and data-intensive RL tasks.

`FPGA-Gym` is fast and scalable. We propose pipeline and time division multiplexing techniques for acceleration [22]. Compared to `EnvPool`, the fastest CPU-based environment parallelism library at present, `FPGA-Gym` achieves a 4.36 to 972.6× speedup. In a grid-world task like *CliffWalking*, `FPGA-Gym` can instantiate more than 8,000 parallel environments in one FPGA chip.

`FPGA-Gym` is easy to customize. We implement `FPGA-Gym` in parameterized and modular manners. It provides a plug-in usage for users without FPGA knowledge. Customizing new environments on top of `FPGA-Gym` is convenient, only requiring revising the computing module of a single environment inside our provided template. Other parameterized parallel operations and interactions between FPGA and CPU are already defined in the framework backbone code.

`FPGA-Gym` is general and compatible with the existing algorithms and libraries. We release an open-source FPGA-Gym library, including various types of RL environments. Moreover, `FPGA-Gym` focuses on the environment execution and is compatible to the existing learning algorithms, e.g. deep Q-network (DQN) [23, 24] and proximal policy optimization (PPO) [25, 26], without modifying the learning framework source code.

2 Related Works

CPU-based RL frameworks: Gymnasium [27], `VectorEnv` and `Sample Factory` [28] offer multithreading or multiprocessing to run each environment in an independent thread or process. However, running programs with tens or hundreds of threads is troublesome in that the users have to take care of thread synchronization, context-switching overhead, and memory bandwidth limitations. `EnvPool` [22] leverages C++ backend for optimized parallel computing and memory management, achieving a 2.8× speedup compared to its Python counterpart. Nonetheless, it requires the developer to manually translate the Python-written environment into C++. As the number of instantiated environments increases, the parallelization becomes increasingly constrained by the fixed number of CPU threads, thereby limiting its scalability.

GPU-/TPU- based RL frameworks: GPUs/TPUs demonstrate significant performance boosts in computing large-scale linear algebra. Naturally, researchers would like to apply this advantage to RL acceleration. Researchers have adapted the source code of several pure-compute RL environments, e.g. *MuJoCo*, etc. [8, 29–42], to be compatible with GPU/TPU using CUDA or JAX toolkits for parallelism. Complex real-world or intricate logical tasks can be challenging to simulate using GPUs or TPUs due to their inefficiency in parallelizing sequential logic operators [43]. Moreover, the process of deploying environments to these hardware accelerators involves significant software engineering efforts, including the need to work with different programming models and low-level libraries. These challenges and limitations can make it difficult to fully leverage the computational power of GPUs and TPUs for such purposes.

3 `FPGA-Gym` Framework

3.1 User workflow

`FPGA-Gym` accommodates the needs of two user groups:

Standard environment users, who would like to accelerate the RL rollouts and training, can simply ① plug the FPGA board into a computer motherboard via PCIe interface→② configure the parallel environment number parameter ("env_num")→③ load the Verilog program onto the FPGA→④ import the Python environments→⑤ run the program. `FPGA-Gym` provides a set of reliable implementations of state-of-the-art RL algorithms. The FPGA_Gym library has included off-the-shelf RL environment examples. CPUs only handle data packing and transmission since the parallel environment computation runs entirely on the FPGA. This setup requires minimal dependencies, only using the "os" and "NumPy" [44] packages, making it highly compatible with various parallel RL algorithm libraries. Fig. 2(a) shows the typical source code to work with Stable-Baseline3 [45], a popular training framework for RL in Python, with the help of `FPGA-Gym`.

Custom environment users, who would like to customize new environments, need to ① implement the new environment step function in Verilog HDL [46] with the provided template (detailed in Section 3.3 and the supplementary materials) → ② update the basic environment computation module in the template and modify the parameters in the template (see the supplementary materials for step-by-step guidance).

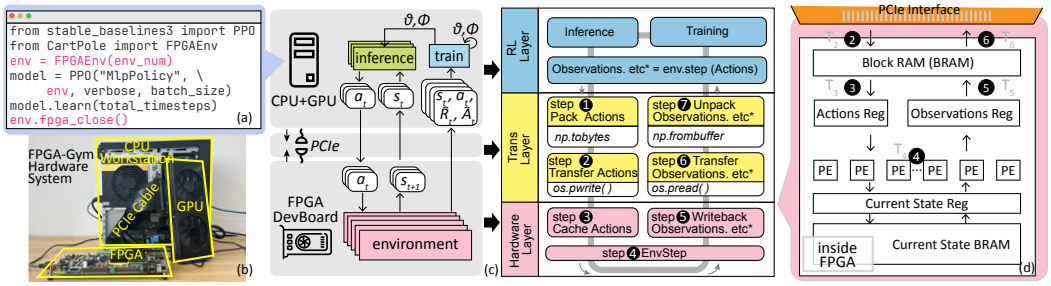

Figure 2: (a) Typical source code to run `FPGA-Gym`; (b) Photo of the proposed `FPGA-Gym` hardware system; (c) System architecture diagram of `FPGA-Gym` in a hierarchical manner. "Observations Reg" stores the observations, rewards, and terminate signals in FPGA computing rollouts. "PE" is the processing element synthesized in FPGA, which is a basic computing circuit module for a single environment instance.

3.2 Implementing `FPGA-Gym` framework

`FPGA-Gym` offloads the dominating environment step computation onto FPGA. The environment step in RL refers to the iterative process of obtaining the corresponding observation, reward, and termination based on the action generated by the agent. Each environment update is done in 7 steps in `FPGA-Gym`, as illustrated in Fig. 2(c): ❶Pack data: CPU packs environment initial states and agent actions into bytes. ❷Transfer actions: CPU transfers the packed data to FPGA block random

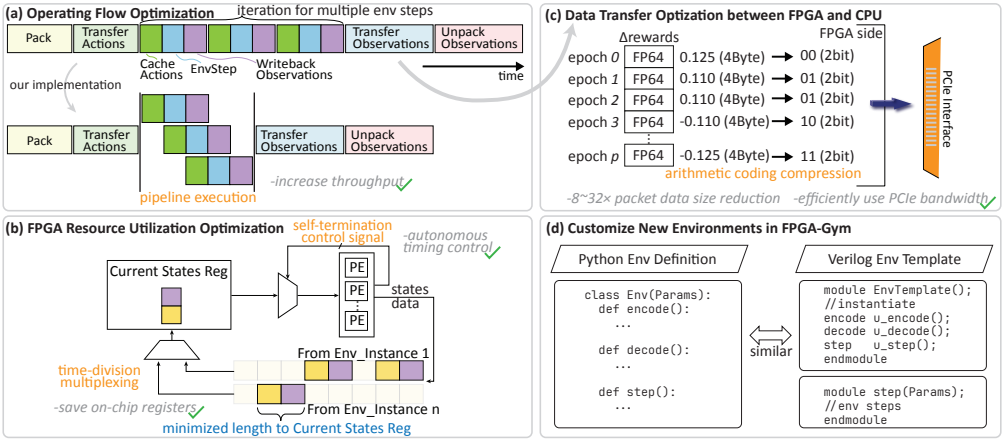

Figure 3: (a) The operating flow optimization; (b) FPGA resource utilization optimization; (c) data transfer optimization; (d) a template to customize new environments in `FPGA-Gym`. The orange text labels the proposed techniques.

access memory (BRAM) via PCIe. ❸Cache actions: reading initial states, current states, actions, and terminations from BRAM into registers for parallel computation. ❹Environment step: when the last step execution ends, the compute units use the initial states and actions as inputs. Otherwise, they use the current states and actions as inputs. The compute units then output the next states, observations, rewards, and termination flags. Then the current states are replaced by the next states. ❺Writeback observations: storing observations, rewards, and terminates into BRAM. ❻Transfer observations: transferring observations, rewards, and terminates from BRAM to CPU via PCIe. ❼Unpack: CPU unpacks observations, rewards, and terminates bytes into Python object/data types. ❶, ❷, ❻, and ❼ is realized on the CPU side in Python. It seamlessly integrates with reinforcement learning algorithms within Python code. ❸, ❹, ❺ runs independently on FPGA and is implemented with Verilog HDL [46]. To boost the computational throughput, we propose the following techniques to achieve highly pipelined and compressed computation.

3.2.1 Pipeline design

To achieve acceleration, we first model and profile the aforementioned workflow latency step by step. T_2, T_3, T_4, T_5 and T_6 represent time costs of step ❷ to ❻ depicted in Fig. 2(c), respectively:

$$
\begin{aligned}
T_2 &= \frac{k_1 \times env\_num}{v_t} & T_3 &= \frac{k_1 \times env\_num}{v_b} \\
T_4^{\text{without\_pipeline}} &= c \times env\_num & T_4^{\text{with\_pipeline}} &= c \\
T_5 &= \frac{k_2 \times env\_num}{v_b} & T_6 &= \frac{k_2 \times env\_num}{v_t}
\end{aligned}
\tag{1}
$$

where env_num is the number of environments to update, c (unit: second) is the time to compute one environment step, 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: GByte/second) is the transfer speed between BRAM and register, v_t (unit: GByte/second) is transfer (PCIe) bandwidth between CPU and FPGA. For steps ❶ and ❼, we adopted efficient data packing and unpacking strategies that consider both software and hardware aspects. Thanks to the robust functionality of NumPy, these processes execute very quickly and occupy only a small fraction of the total runtime. The measured and estimated time $T_{2\sim6}$ will be given in Section 4 and the supplementary materials.

We instantiate multiple environment compute units on FPGA so that they can execute the step of multiple environments in a few clock cycles (often around 3∼10 ns/cycle) at the same time. Instead of realizing the aforementioned 7 steps sequentially, we implement a pipeline design specifically between step ❹ and step ❺ so that the computing (❹) and access to BRAM (❺) inside FPGA occur simultaneously to improve the computing throughput. Because the computation overlaps with the data transfer in time, c (the time required to compute a single environment) and $(pe\_num \cdot k_2/v_b)$

(the time required to transfer the computed data of a single environment to BRAM) should be equal to each other to avoid the case that one step waiting for the other to complete.

3.2.2 Time-division multiplexing & self-termination reset

Storing a large number of current states would consume FPGA on-chip registers as the parallelism is high. We carry out time-division reuse of this part of resources (Fig. 3(b)). Only when this part of data is used, will it write in current states register from BRAM. This technique leads to an efficient usage of FPGA on-chip computing and memory resources,i.e. the FPGA on-chip hardware resources do not grow dramatically as the number of environments ("env_num") increases. Additionally, in a vectorized parallelism approach, managing individual environment resets upon the reception of a "done" signal can be challenging. To solve this difficulty, we employ the self-termination reset in the control flow. Each computation unit receives the action associated with a specific environment, accompanied by the current state and termination signal from the preceding step, as well as the initial state. If the termination signal is asserted, the environment automatically reverts to the initial state as the input for the subsequent step calculation. This method voids significant wait times or processing overhead.

3.2.3 Local access & data compression

The quantity of data transmitted has a direct impact on the values of the coefficients k_1 and k_2 in Equation 1. To optimize data transmission, we employ the following techniques in `FPGA-Gym`. Firstly, the state of the environment is preserved within the FPGA and updated automatically following each calculation, except the initial state, which is received only once at the beginning. In subsequent iterations, only the action sent from the CPU is accepted as an input. Secondly, for tasks where the reward value encompasses several types but varies significantly, our framework uses arithmetic coding to compress the reward values (illustrated in Fig. 3(c)). The reward values are replaced by a categorical code, followed by batch numerical processing upon returning to the CPU. Lastly, for tasks that allow for the concatenation of multiple values into a 32-bit number, we use bit-packing and unpacking techniques to compress data, thereby enhancing data utilization efficiency. Details can be found in the supplementary materials.

3.3 Customizing new environments

`FPGA-Gym` is a general RL acceleration framework that supports customized environments. We have capsulated the aforementioned workflow, pipeline, and compression into the backbone code of `FPGA-Gym`. We provide a template for the customized environment step function (an epitome is given in Fig. 3(d)). This source code template leaves all of the environment step functions empty and the user can directly use Verilog HDL language (very similar to Python when it comes to the flow control) to implement their environment or translate the environment Python code into Verilog HDL code. Once done, the next step is to set up the key parameters in the template and synthesize digital logic circuits on FPGA with the filled template.

4 Experiments

This section investigates the efficacy and compatibility of the `FPGA-Gym` framework for RL. In particular, we would like to quantitatively answer the following questions:① *Q1*: How effective is `FPGA-Gym` in accelerating different types of RL environments? ② *Q2*: How significant are the proposed design techniques (detailed in Section 3.2) on the parallel environment execution speedup? ③ *Q3*: How to work with `FPGA-Gym` together the existing RL training algorithms and frameworks?

Benchmarks & Baselines: We select the typical environments *CartPole*, *Pendulum*, *CliffWalking*, and *BlackJack* in `Gymnasium` [27] as the representatives for benchmarking. These 4 environments cover discrete/continuous variable space, partially/globally observable simulation, and various categories including classic control, grid world, and strategy games (Table 1). The acceleration framework baselines are `EnvPool` [22] and `VectorEnv` [27]. `EnvPool` is the latest and fastest (to the best of our

Table 1: Typical Environments For Benchmarking

Environment Name	Environment Type	Action Space Type	Observation Space Type	Rewards Type	Observability
CartPole	physical control	discrete	continuous	discrete	full
Pendulum	physical control	continuous	continuous	continuous	full
MountainCar	physical control	discrete	continuous	discrete	full
CliffWalking	gird world	discrete	discrete	discrete	full
FrozenLake	gird world	discrete	discrete	discrete	full
Taxi	gird world	discrete	discrete	discrete	full
BlackJack	strategy game	discrete	discrete	discrete	partial
Seaquest	atari game	discrete	discrete	discrete	full

knowledge) before this work to implement the above 4 environments. `VectorEnv` is a widely-used vectorized environment native to Gymnasium libraries using multithreading.

Hardware Environment Setup: The hardware platform we use has 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. For the rollout length, 10^5 iterations are used to average the speed measurement experiments. All the training experiments set fixed 5 random seeds for reproducibility. For training, we employ DQN and PPO algorithms. PPO is implemented by directly calling the RL training programming framework Stable-Baseline3 [45, 47].

4.1 Experiments on acceleration

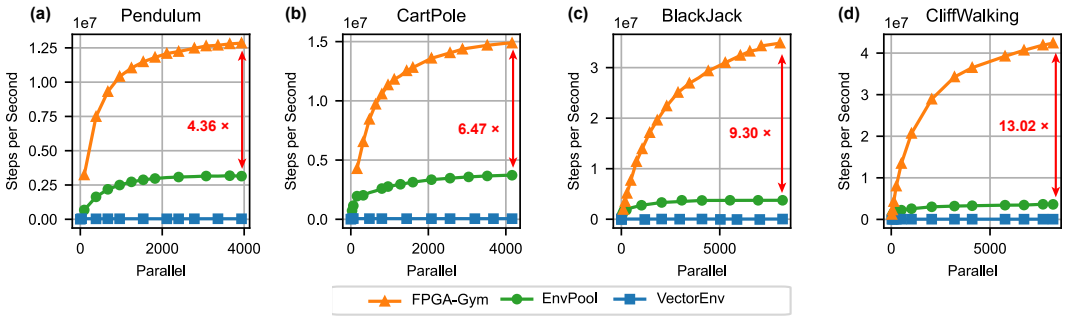

Figure 4: Measured throughputs of parallel environment steps (rollouts).

To answer *Q1*, we measure the throughputs at different parallelism of *Pendulum*, *CartPole*, *BlackJack*, and *CliffWalking* environments compared with `EnvPool` and `VectorEnv` frameworks, shown in Fig. 4. The x-axis is the number of instantiated parallel environments to compute. The y-axis is the total number of steps updated per second, which represents the computational throughput of executing environments in parallel (higher is better). Compared to `EnvPool`, `FPGA-Gym` improves the throughput by 4.36× for *Pendulum*, 6.47× for *CartPole* when executing 4000 instantiated environments in parallel, and 9.30× for *BlackJack*, 13.02× for *CliffWalking* with 8000 parallel environments. When running 2048 parallel *Seaquest* environments, `FPGA-Gym` achieves a 972.6× speedup compared to `EnvPool`. This validates the significant improvement of our approach to accelerate parallel RL environment execution. As the number of parallel environments increases, `EnvPool` reaches its maximum throughput earlier than `FPGA-Gym`, indicating a better scalability of `FPGA-Gym`. This confirms the original motivation of our proposed solution. The `EnvPool`'s curves in Fig. 4 reach saturation because of the limited number of computing threads; `FPGA-Gym`'s curves get lower slopes due to the BRAM bandwidth and PCIe transmission bandwidth.

4.2 Experiments on pipeline-driven efficiency and hardware resource utilization

To answer *Q2*, we test the execution latency breakdown before and after implementing pipeline optimization as well as the hardware resource utilization improvements. Fig. 5 shows the measured

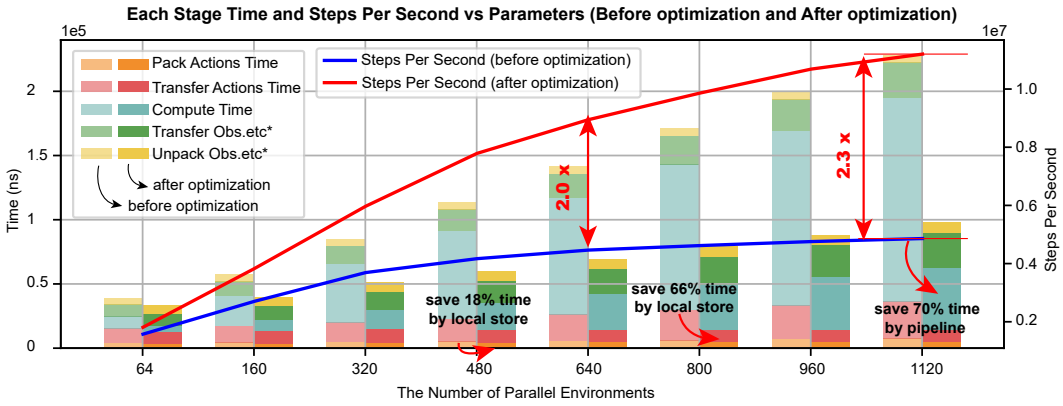

Figure 5: Each Stage Time and Steps Per Second vs The Number of Parallel Environments (Before and After Optimization) $*$: "Obs. etc" represents for the observations, rewards, and terminates.

computing latency of each stage in the workflow and steps per second before and after pipeline optimization. Here we take the *CartPole* task [27] as an example.

The x-axis is the number of parallel environments. The left y-axis is the execution time (stacked bar chart, lower values are better). The right y-axis is the steps per second (curves, higher is better). In each group of the stacked bars, the left (right) stacked bars are the execution time before (after) optimization. As the number of parallel environments increases, the computing module dominates the entire execution time. For the task with 1,120 instantiated *CartPole* environments in parallel, the pipeline technique saves up to 70% of compute time. Moreover, storing environment states locally saves CPU to FPGA's transmission time by 66.0% and data packing time by 33.1%. With everything considered, `FPGA-Gym` achieves a 2.3$\times$ speedup than before optimization.

It is convenient in `FPGA-Gym` to scale out the number of parallel environments. By tuning the "env_num" parameter, the `FPGA-Gym` framework can automatically complete the instantiation with more FPGA resources. Table 2 shows FPGA resource utilization at different parallel environment numbers. "pe_num" is the number of compute modules that compute the step of one environment. "c" is the time that a single RL environment computes. Look-up tables (LUTs), flip-flops (FFs) are the components to implement combinational logic, and registers to store intermediate results, respectively. The numbers in the last 3 rows are the FPGA implementation results for how many of the corresponding components are used. As shown, the LUTs and FFs of parallel *BlackJack* increase only 8.49% and 19.07% as env_num increases from 96 to 8000 (84$\times$), thanks to the proposed time-division multiplexing technique. Computing circuits (LUTs) and registers (FFs) can be shared among the environment instances.

Table 2: FPGA resource utilization

Environment	*Cliffwalking*		*Cartpole*		*Pendulum*		*BlackJack*	
env_num	64	8000	480	4000	192	4032	96	8064
pe_num / c (ns)	2/20		20/800		24/960		16/160	
power(W)	4.118	4.258	8.388	8.898	9.87	10.38	4.471	4.644
LUTs	27364	43481	175725	196896	228021	260307	46832	50811
FFs	26829	42821	254842	258832	267556	390253	41937	49938

4.3 Experiments on end-to-end agent training

To answer *Q3*, we evaluate the effects of `FPGA-Gym` on the existing RL algorithms and frameworks. We deploy the PPO algorithm from Stable-Baselines3 in 64 parallel *CartPole* environments with three frameworks–`VectorEnv`, `EnvPool`, and `FPGA-Gym`. The results are shown in Fig 6 (a) and (b). In Fig. 6, the x-axis of (a), (c), and (e) is the time of the entire RL training process, including

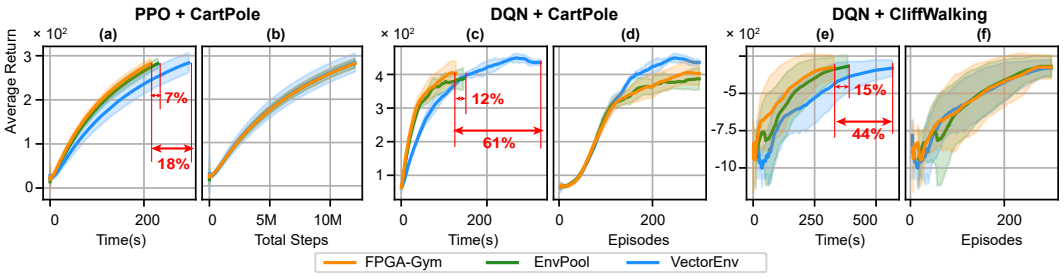

Figure 6: End_to_End_Train.

training, inference, and environment steps, etc. The x-axis of (b) is the total steps executed in the RL training process. The x-axis of (d), and (f) is the number of episodes executed during RL training. The y-axis is the measured average rewards during training. Fig. 6(a)(b) indicates that the training performance of the three methods is similar for the same number of steps. However, `FPGA-Gym` consistently achieves this performance faster than `EnvPool` and `VectorEnv`. When the total steps reach 12,000,000, the training time of `FPGA-Gym` is reduced by 18% and 7% compared to `VectorEnv` and `EnvPool`, respectively. Fig. 6(c)(d) presents the training performance of the *CartPole* using the DQN algorithm with 64 parallel environments. When episode number reaches 300, the training time of `FPGA-Gym` is reduced by 61% and 12% compared to `VectorEnv` and `EnvPool`, respectively. Fig. 6(e)(f) illustrates the training performance of the *Cliffwalking* environment using the DQN algorithm with 160 parallel environments. `FPGA-Gym`, `EnvPool` and `VectorEnv` achieve similar average rewards at the same episodes. When the number of episodes reaches 300, the training time of `FPGA-Gym` is reduced by 44% and 15% versus `VectorEnv` and `EnvPool`, respectively. The above results validate the compatibility of `FPGA-Gym` with various RL algorithmic libraries and frameworks, as well as the acceleration of training.

5 Limitation & Future Work

The primary motivation behind `FPGA-Gym` is to create a comprehensive hardware-enabled RL acceleration framework that maximizes the potential of heterogeneous computing platforms, including CPUs, GPUs, and FPGAs. In its current form, `FPGA-Gym` supports a basic combination of GPU for neural networks, CPU for flow control, and FPGA for rollout acceleration. Our plans include exploring efficient task distribution and delicate scheduling among CPUs, GPUs, and FPGAs to enhance performance in dynamic, unpredictable open-world scenarios. Furthermore, we aim to enhance the scalability of `FPGA-Gym` by integrating a network of multiple FPGAs in upcoming versions. In the future, we plan to improve the BRAM bandwidth for better speedup ratios using multiple PCIe controllers and faster PCIe interfaces. With these enhancements, `FPGA-Gym` is set to deliver increased speed and scalability.

6 Conclusion

This study presents `FPGA-Gym`, a novel acceleration framework that combines FPGA and CPU components to enhance the performance of RL environments' parallel executions. By offloading environment steps to FPGA reconfigurable hardware, `FPGA-Gym` achieves a significant speedup of 4.36 to 972.6$\times$ over the fastest existing software-based framework. We provide a modular design and parametric approach to simplify the deployment of environment updates for users without extensive FPGA expertise. A specialized FPGA-based parallel template is introduced for step operations in RL environments, similar to JAX's functionality but focusing on individual environments as the smallest units of computation. This approach offers flexibility and customization. The `FPGA-Gym` framework is compatible with existing RL algorithms and frameworks, including DQN and PPO algorithms. A standard environment library, similar to Gym library, is provided based on the `FPGA-Gym` framework. The library is publicly available `https://github.com/Selinaee/FPGA_Gym`.

Acknowledgments and Disclosure of Funding

This work was supported by National Natural Science Foundation of China (no. T2350006, 92264201, 92364102); and the 111 Project under Grant B18001. This work is sponsored by Beijing Nova Program. The authors declare no competing interests.

References

[1] Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Zhenrui Li, Guojun Peng, Yue-Jiao Gong, Yining Ma, and Zhiguang Cao. MetaBox: A Benchmark Platform for Meta-Black-Box Optimization with Reinforcement Learning. *Conference on Neural Information Processing Systems (NeruIPS)*, 2024.

[2] Zhecheng Yuan, Sizhe Yang, Pu Hua, Can Chang, Kaizhe Hu, and Huazhe Xu. RL-ViGen: A Reinforcement Learning Benchmark for Visual Generalization. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[3] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *International Conference on Learning Representations (ICLR)*, 2019.

[4] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[5] Jiaming Ji, Borong Zhang, Jiayi Zhou, Xuehai Pan, Weidong Huang, Ruiyang Sun, Yiran Geng, Yifan Zhong, Josef Dai, and Yaodong Yang. Safety gymnasium: A unified safe reinforcement learning benchmark. *Conference on Neural Information Processing Systems (NeruIPS)*, 2023.

[6] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning (ICML)*, 2018.

[7] Xuehai Pan, Mickel Liu, Fangwei Zhong, Yaodong Yang, Song-Chun Zhu, and Yizhou Wang. Mate: Benchmarking multi-agent reinforcement learning in distributed target coverage control. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[8] Steven Dalton and Iuri Frosio. Accelerating Reinforcement Learning through GPU Atari Emulation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[9] Misha Laskin, Denis Yarats, Hao Liu, Kimin Lee, Albert Zhan, Kevin Lu, Catherine Cang, Lerrel Pinto, and Pieter Abbeel. URLB: Unsupervised Reinforcement Learning Benchmark. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[10] Christopher Yeh, Victor Li, Rajeev Datta, Julio Arroyo, Nicolas Christianson, Chi Zhang, Yize Chen, Mohammad Mehdi Hosseini, Azarang Golmohammadi, Yuanyuan Shi, et al. SustainGym: Reinforcement Learning Environments for Sustainable Energy Systems. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[11] Denis Tarasov, Alexander Nikulin, Dmitry Akimov, Vladislav Kurenkov, and Sergey Kolesnikov. Corl: Research-oriented deep offline reinforcement learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[12] Yun Qu, Boyuan Wang, Jianzhun Shao, Yuhang Jiang, Chen Chen, Zhenbin Ye, Liu Linc, Yang Feng, Lin Lai, Hongyang Qin, et al. Hokoff: Real Game Dataset from Honor of Kings and its Offline Reinforcement Learning Benchmarks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[13] Rong-Jun Qin, Xingyuan Zhang, Songyi Gao, Xiong-Hui Chen, Zewen Li, Weinan Zhang, and Yang Yu. Neorl: A near real-world benchmark for offline reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[14] Xiao-Yang Liu, Ziyi Xia, Jingyang Rui, Jiechao Gao, Hongyang Yang, Ming Zhu, Christina Wang, Zhaoran Wang, and Jian Guo. FinRL-Meta: Market environments and benchmarks for data-driven financial reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[15] Hua Wei, Jingxiao Chen, Xiyang Ji, Hongyang Qin, Minwen Deng, Siqin Li, Liang Wang, Weinan Zhang, Yong Yu, Liu Linc, et al. Honor of kings arena: an environment for generalization in competitive reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[16] Akhil Raj Baranwal, Salim Ullah, Siva Satyendra Sahoo, and Akash 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 (IEEE TCAD)*, 2021.

[17] Yuan Meng, Sanmukh Kuppannagari, and Viktor Prasanna. Accelerating Proximal Policy Optimization on CPU-FPGA Heterogeneous Platforms. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[18] Hyungmin Cho, Pyeongseok Oh, Jiyoung Park, Wookeun Jung, and Jaejin Lee. FA3C: FPGA-Accelerated Deep Reinforcement Learning. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[19] Chan-Wei Hu, Jiang Hu, and Sunil P. Khatri. TD3lite: FPGA acceleration of reinforcement learning with structural and representation optimizations. In *Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[20] Yuan Meng, Chi Zhang, and Viktor Prasanna. FPGA acceleration of deep reinforcement learning using on-chip replay management. In *Proceedings of the 19th ACM International Conference on Computing Frontiers (CF)*, 2022.

[21] Samuel Wiggins, Yuan Meng, Rajgopal Kannan, and Viktor Prasanna. Accelerating multi-agent DDPG on CPU-FPGA heterogeneous platform. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2023.

[22] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. Envpool: A Highly Parallel Reinforcement Learning Environment Execution Engine. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[23] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning From Demonstrations. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[26] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[27] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium. `https://zenodo.org/record/8127025`, 2023.

[28] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with asynchronous reinforcement learning. In *International conference on machine learning (ICML)*, 2020.

[29] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library. `http://github.com/RobertTLange/gymnax`, 2022.

[30] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[31] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[32] Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. Warpdrive: Fast end-to-end deep multi-agent reinforcement learning on a GPU. *Journal of Machine Learning Research (JMLR)*, 2022.

[33] Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. *arXiv preprint arXiv:2402.16801*, 2024.

[34] Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[35] Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Coyette, Paul Duckworth, Laurence I. Midgley, Tristan Kalloniatis, Sasha Abramowitz, Cemlyn N. Waters, Andries P. Smit, Nathan Grinsztajn, Ulrich A. Mbou Sob, Omayma Mahjoub, Elshadai Tegegn, Mohamed A. Mimouni, Raphaël Boige, Ruan de Kock, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in JAX. In *International Conference on Learning Representations (ICLR)*, 2023.

[36] Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Akbir Khan, Christian Schröder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob N. Foerster. Jaxmarl: Multi-agent RL environments and algorithms in JAX. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2024.

[37] Mathias Lechner, Lianhao Yin, Tim Seyde, Tsun-Hsuan Johnson Wang, Wei Xiao, Ramin M. Hasani, Joshua Rountree, and Daniela Rus. Gigastep - One Billion Steps per Second Multi-agent Reinforcement Learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[38] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac Gym: High Performance GPU Based Physics Simulation For Robot Learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[39] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A differentiable physics engine for large scale rigid body simulation. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[40] Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-minigrid: Scalable meta-reinforcement learning environments in JAX. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[41] Benjamin Ellis, Jonathan Cook, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob Foerster, and Shimon Whiteson. Smacv2: An improved benchmark for cooperative multi-agent reinforcement learning. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[42] Yiheng Zhu, Yang Zhan, Xuankun Huang, Yuwei Chen, Jiangwen Wei, Wei Feng, Yinzhi Zhou, Haoyuan Hu, Jieping Ye, et al. Ofcourse: A multi-agent reinforcement learning environment for order fulfillment. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[43] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[44] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 2020.

[45] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research (JMLR)*, 2021.

[46] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[47] Antonin Raffin. RL Baselines3 Zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.