GPU-Podracer: Scalable and Elastic Library for Cloud-Native Deep Reinforcement Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

Deep reinforcement learning (DRL) has revolutionized learning and actuation in applications such as game playing and robotic control. The cost of data collection, i.e., generating transitions from agent-environment interactions, remains a major challenge for wider DRL adoption in complex real-world problems. Following a cloud-native paradigm to train DRL agents on a GPU cloud platform is a promising solution. In this paper, we present a scalable and elastic library *GPU-podracer* for cloud-native deep reinforcement learning, which efficiently utilizes millions of GPU cores to carry out massively parallel agent-environment interactions. At a high-level, GPU-podracer employs a tournament-based ensemble scheme to orchestrate the training process on hundreds or even thousands of GPUs, scheduling the interactions between a leaderboard and a training pool with hundreds of pods. At a low-level, each pod simulates agent-environment interactions in parallel by fully utilizing nearly $7,000$ GPU CUDA cores in a single GPU. Our GPU-podracer library features high scalability, elasticity and accessibility by following the development principles of containerization, microservices and MLOps. Using an NVIDIA DGX SuperPOD cloud, we conduct extensive experiments on various tasks in locomotion and stock trading and show that GPU-podracer outperforms Stable Baseline3 and RLlib, e.g., GPU-podracer achieves nearly linear scaling.

1 Introduction

Deep reinforcement learning (DRL), which balances the exploration (of uncharted territory) and exploitation (of current information), has revolutionized learning and actuation in applications such as game playing [23] and robotic control [29]. DRL employs a trial-and-error manner to generate transitions from agent-environment interactions, along with the learning procedure. However, the cost of data collection remains a major challenge for wider DRL adoption in real-world problems with complex and dynamic environments. Therefore, a compelling solution is massively parallel training on hundreds or even thousands of GPUs, say millions of GPU cores.

Existing DRL frameworks are not satisfactory with respect to scalability and accessibility. As shown in Fig. 1, OpenAI Baselines [7], Stable Baselines 3 [19] and OpenAI Spinning Up [1] utilize a single GPU, while RLlib [13] and rlpyt [24] can support multiple GPUs. However, there is no existing DRL framework for a cloud with hundreds or even thousands of GPUs. We aim to fully utilize two core techonlogies: 1). GPUDirect technology [25] that provides a path for data to bypass CPUs and travel on "open highways" offered by GPUs, storage, and networking devices; and 2). massively parallel simulations using thousands of GPU cores on a single GPU. On the other hand, the above frameworks, except OpenAI Spinning Up serving an educational purpose, involve a steep learning curve or a lack of customization flexibility, which results in low accessibility.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

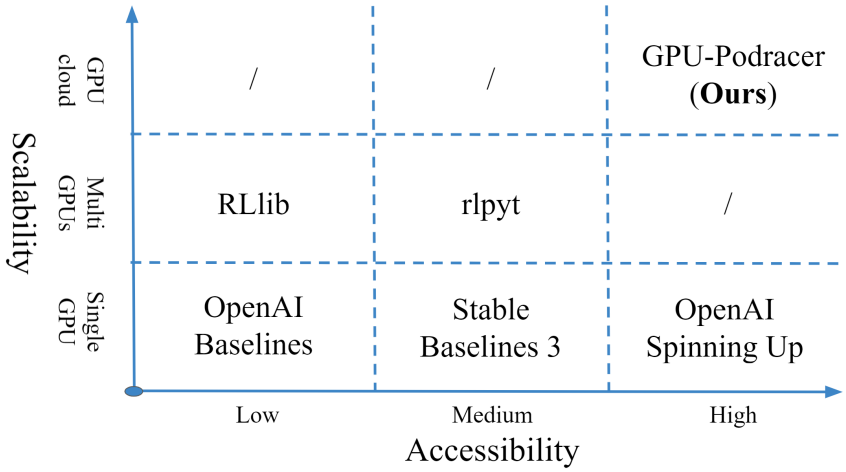

Figure 1: Comparison of different frameworks/libraries.

Scaling out the training process of DRL agents to hundreds or even thousands of GPUs is challenging for researchers and practitioners. The *cloud-native* paradigm aims to scalably and elastically utilize the cloud computing resources. Therefore, we believe it is practically promising to schedule the training of DRL agents by following the cloud-native paradigm, such as employing standardized software stack, e.g., Kubernetes (K8s) [3], and adopting core technologies including containers, microservices, continuous integration (CI) and continuous delivery (CD) [2, 9].

In this paper, we present a scalable and elastic library *GPU-podracer* for cloud-native deep reinforcement learning, which efficiently utilizes millions of GPU cores to carry out massively parallel agent-environment interactions. At a high-level, GPU-podracer employs a tournament-based ensemble scheme to orchestrate the training process on hundreds or even thousands of GPUs, scheduling the interactions between a leaderboard and a training pool with hundreds of pods. At a low-level, each pod simulates agent-environment interactions in parallel by fully utilizing over $7,000$ GPU cores in a single GPU. Our GPU-podracer library features high scalability, elasticity and accessibility by following the development principles of containerization, microservices and MLOps.

Our main contributions are summarized as follows:

- We present a scalable and elastic open-source library for cloud-native deep reinforcement learning, *GPU-podracer*, that can utilize millions of GPU cores to train effective DRL agents for complex real-world problems.
- To accelerate data collection for efficient exploration, we propose a tournament-based ensemble training scheme and employ massive parallel simulations.
- GPU-podracer follows a *cloud-native* paradigm by realizing the development principles of containerization, microservices and MLOps (e.g., continuous integration and continuous delivery), and achieves high accessibility.
- Using an NVIDIA DGX SuperPOD cloud [25], we conduct extensive experiments on various tasks in locomotion and stock trading and show that GPU-podracer outperforms Stable Baseline3 [19] and RLlib [13], e.g., GPU-Podracer achieves nearly linear scaling.

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 presents our design principles. Section 4 describes the GPU-podracer library. In Section 5, we present experimental results. We conclude this paper in Section 6.

2 Related Works

We review open-source DRL frameworks/libraries and environment simulation packages.

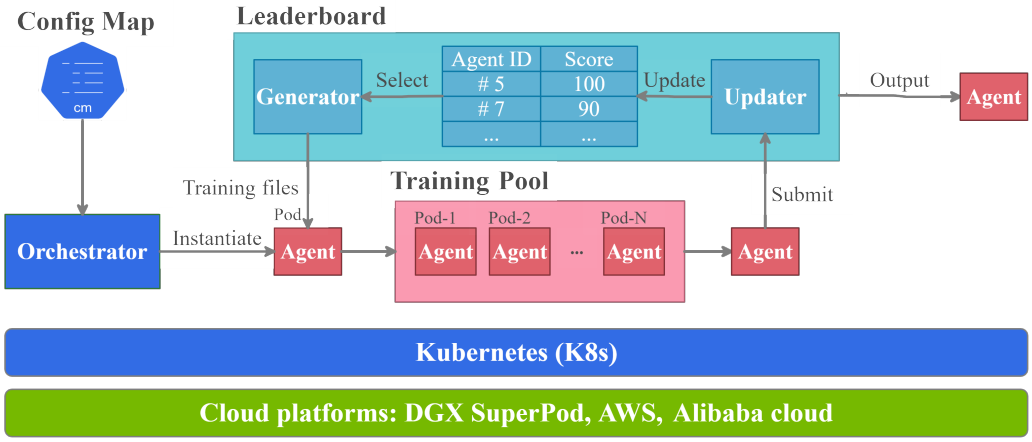

Figure 2: GPU-podracer employs a tournament-based ensemble training, where a leaderboard is updated by a training pool of pods.

2.1 Deep Reinforcement Learning Framework/Libraries

Many open-source DRL frameworks/libraries have been developed in recent years with varied capabilites. OpenAI Spinning Up [1] and Google Dopamine [5] are research frameworks for the fast prototyping of DRL algorithms. They both implement numerous DRL algorithms with simple and pedagogical codes. Stable Baseline3 [19] is a stable and efficient DRL library, introducing the parallelism of sampling through vectorized environment. RLlib [13] is a generic DRL library that derives its strength from Ray communication procotols that enable scalable, distributed training. Podracer [12] from DeepMind is closely related to our GPU-podracer, which also focuses on the efficient usage of large computing resources for training DRL agents. However, it is designed for Google's tensor processing units (TPUs) that are inaccessible to many researchers and practitioners.

2.2 Simulation Packages

Environment simulation is a critical component of DRL training, and lots of platforms that provide various task simulations are emerging to close the simulation-to-reality gap. OpenAI Gym [4] is a fundamental simulation toolkit for DRL research, which defines a standard interface for follow-up works. It includes a collection of benchmark problems, e.g., classic control, Atari games, and 2D and 3D robots. MuJoCo [26] and Isaac Gym [16] are two powerful platforms for robotic simulations. MuJoCo [26] is a popular physics simulator that efficiently simulates joint contact models. The recently released Isaac Gym [16] is a high-performance simulation environment for physics. It enables thousands of environments running in parallel on a single GPU. FinRL [15] is a new finance-related DRL platform, which simulates various markets as training environments that are built on historical data and live trading APIs.

3 Design Principles and Overview

We aim to develop a user-friendly open-source library that fully exploits cloud resources to train DRL agents. The library emphasizes the following design principles:

- **Scaling-out**: scalability and elasticity.
- **Efficiency**: low communication overhead, massively parallel simulations and robustness of agents.
- **Accessibility**: lightweightness and customization.

For algorithm design, GPU-podracer employs a *tournament-based ensemble training* to balance exploration and exploitation. In contrast to Evolutionary Strategies (ES) [20] where a population of agents evolve over generations, our tournament-based ensemble training scheme updates agents asynchronously in parallel, which decouples population evolution and single-agent learning. As

shown in Fig. 2, the key of the tournament-based ensemble training scheme is the interaction between a *leaderboard* and a *training pool*. The training pool contains hundreds of agents (pods) that 1) are trained in an asynchronous manner, and 2) can be initialized with different DRL algorithms for the ensemble purpose. The leaderboard records the agents with high performance and continually updates as more agents (pods) are trained.

As shown in Fig. 2, the tournament-based ensemble training scheme proceeds as follows:

1. An *orchestrator* instantiates a new agent (pod) and put it into a training pool.
2. A *generator* initializes an agent (pod) with networks and optimizers selected from a leaderboard. The *generator* is a class of subordinate functions associated with the leaderboard, which has different variations to support different evolution strategies.
3. An *updater* determines whether and where to insert an agent into the leaderboard according to its performance, after a pod has been trained for a certain number of steps or certain amount of time.

For system design, GPU-podracer follows the *cloud-native* paradigm. GPU-podracer achieves containerization by implementing the tournament-based ensemble training scheme as the synergy of microservices. Such a paradigm allows a lightweight usage via simple APIs and a high degree of customization through the flexible cooperation of microservices.

At a high-level, GPU-podracer has the following capabilities to embody our design principles:

- **Asynchronously distributed training** is made possible through a training pool. GPU-podracer can scale out to hundreds or even thousands of computing nodes and elastically adjust the number of agents according to the available computing resources.
- **Tournament-based ensemble training** is made possible through a leaderboard. Tournament-based training scheme decouples the agent learning and population evolution to achieve low communication overhead between pods. Ensembling many DRL algorithms increases efficiency by exploiting agent robustness and diversity.
- **Cloud-nativity** is made possible with the containerization, microservices, and MLOps adherence. MLOps achieves continuous training/integration/delivery (CT/CI/CD) by exploiting the Kurbernetes (K8s) [3] software for automated cloud orchestration.

4 GPU-Podracer: Scalable and Elastic Cloud-native Library

In this section, we propose a scalable and elastic cloud-native library, called *GPU-Podracer*. We first describe its key components and then present its features.

4.1 Ensemble Training Using Microservices

As shown in Fig. 2, the ensemble training scheme exploits the synergy of the following microservices: orchestrator, leaderboard (including updater and generator), and agents (pods) in the training pool, where each microservice maps to a *container*.

Orchestrator: An *orchestrator* monitors the available computing resources and determines the number of pods in the training pool. When K8s signals that the workload is light, the orchestrator generates a set of new pods and insert them into the training pool. When a training objective (i.e., target rewards) is achieved, the orchestrator will terminate the training process.

Leaderboard: A *leaderboard* records a set of candidate agents with high performance, say cumulative reward. An updater updates the candidate agents to the leaderboard, while a generator instantiates a new pod by referring to candidate agents. The leaderboard may also track other information, such as the covariance matrix, mean, variance, etc, which helps the generator to adaptively allocate computing resources to highly potential candidate agents.

- **Updater**: An *updater* receives a trained agent (pod) and may insert its training files (including actor network, critic network, optimizer parameters, replay buffer (for off-policy algorithms)) into the leaderboard if it has high performance.
- **Generator**: A *generator* generates training files for a newly created pod. The generator may perform a mutation of the candidate agents to increase the diversity.

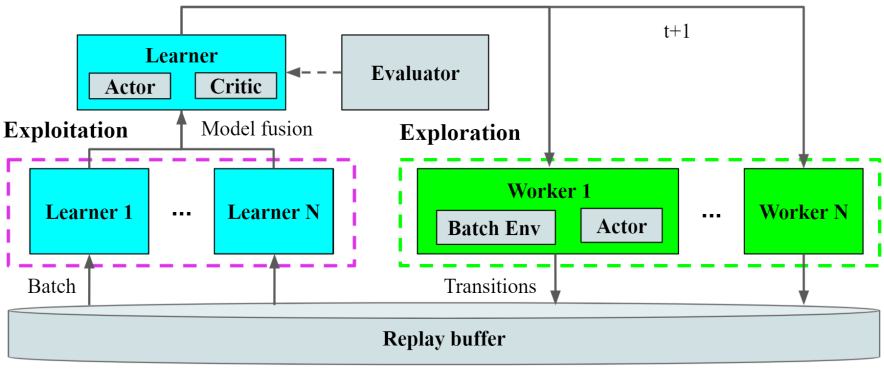

Figure 3: An agent (pod) is split into three types of microservices: worker, learner and evaluator.

4.2 Agent Learning Using Microservices

GPU-podracer maps the training process of an agent to a *pod*, which is the smallest deployable unit in K8s [3]. As shown in Fig. 3, GPU-podracer separates an agent learning into three microservices: worker, learner and evaluator.

Worker: A *worker* generates batches of transitions from interactions between the agent and the batched environment. A batched environment consists of multiple independent sub-environments. Each actor collects transitions from the sub-environments of the batched environment in parallel to accelerate the data collection process.

Learner: A *learner* fetches a batch of transitions from the replay buffer to train the neural networks. GPU-podracer trains multiple learners in parallel and fuses the networks by aggregating **network parameters**. In this way, GPU-podracer experiments much less communication overhead than distributed SGD in RLlib [13].

Evaluator: An *evaluator* continuously evaluates a pod and records its performance and corresponding networks during the training process. Commonly used performance metrics are the mean and variance of the episodic reward. Note that the evaluator effectively mitigates the performance loss caused by either overfitting or early stopping.

4.3 Features of GPU-Podracer

GPU-podracer achieves several features that facilitate the implementation of a lightweight and powerful training process on a GPU cloud.

Scalable parallelism: The multi-level parallelism of GPU-podracer leads to high scalability.

- **Agent parallelism**: The agents in the training pool are parallel, thus can easily scale out to a large number. The asynchronous training of parallel agents can also reduce the frequency of agent-to-agent communication.
- **Learner parallelism**: An agent employs multiple learners to train the neural networks in parallel, and then fuse the networks parameters to obtain a result agent, instead of using distributed SGD. Such a model fusion through network parameters involves a much lower frequency communication as the fusion process only happens at the end of an epoch.
- **Worker parallelism**: An agent utilizes multiple rollout workers to sample trajectories in parallel.

Elastic resource allocation: The elasticity is critical for cloud-level applications as it helps users adapt to the changes in cloud resources and prevent over-provisioning and under-provisioning of resources [17, 11]. GPU-podracer can elastically allocate the number of agents (pods) by employing an orchestrator to monitor the available computing resources and the current training status.

Cloud-oriented optimizations: GPU-podracer co-locates microservices on GPUs to accelerate the parallel computation on both data collection and model training. For the data transfer and storage, GPU-podracer represents data as tensors to speedup the communication and allocates the shared replay buffer on the contiguous memory of GPUs to increase the addressing speed.

Continuous training (CT) pipeline: Continuous training, which is a part of the MLOps practice, seeks to automatically and continuously retrain the model to adapt to changes that might occur in the data. GPU-podracer performs the CT of a DRL agent by automating a lightweight DRL training pipeline, which is composed of microservices. Users can conduct different experiments and hyper-parameter search by modifying workers, learners and other microservices.

Continuous integration/delivery (CI/CD): GPU-podracer enables a robust CI/CD for users to explore new ideas by modifying existing microservices or build new microservices. The microservices are loosely coupled, such that the change of one microservice will not break existing ones. Also modularity allows for more comprehensive search over the experiment space, e.g., instead of designing one experiment at a time, we could theoretically be able to test $c_1 \times c_2 \times \cdot \times c_n$ experiments in an automated fashion, where n is the number of components for a DRL algorithm and c_i is the number of optional microservices for component i, $i = 1, ..., n$:

- **Environment variation**: GPU-podracer supports any environment written in gym-style and provides a class *PreprocessVecEnv* to convert a normal environment to a batch mode.
- **Algorithm variation**: GPU-podracer can realize different DRL algorithms through combinations of worker and learner variants. Currently, GPU-podracer supports fine-tuned standard DRL algorithms, including DQN-series [18, 27], DDPG [14], TD3 [8], SAC [10], and PPO [21]. New algorithms may be used as long as they adhere to the agent interface.
- **Evolution variation**: GPU-podracer allows users to customize the evaluator, updater and generator in the leaderboard to decide *how to evaluate*, *where to update*, and *what to generate*.

5 Performance Evaluation

In the section, we describe the cloud platform in our experiments and the performance of GPU-podracer on various tasks from robotic control to stock trading task.

5.1 Experiment Platform: GPU Cloud

All experiments were executed using NVIDIA DGX-2 servers [6] in an NVIDIA DGX SuperPOD cloud [25], a cloud-native infrastructure. Each NVIDIA DGX-2 server contains 8 NVIDIA A100 GPUs and 320 GB GPU memory in total, and also has 128 CPU cores of Dual AMD Rome 7742 running at 2.25GHz. Among the 8 NVIDIA A100 GPUs in one NVIDIA DGX-2 server, any two A100 GPUs are connected with each other through 12 NVLinks, providing 600 Gbps of full-duplex bandwidth [6].

5.2 Robotic Control Tasks

Ant and humanoid are two canonical robotic control tasks that simulate an ant and a humanoid, respectively, where each task has both MuJoCo [26] version and Isaac Gym [16] version, as shown in Fig. 4. The ant task is a simple environment to simulate due to its stability in the initial state, while the humanoid task is often used as a testbed for locomotion learning. Even though the implementations of MuJoCo [26] and Isaac Gym [16] are slightly different, the objective of both is to have the agent move forward as fast as possible. The state space, action space and reward function are given in Table 1. We select the same tasks from the two platforms in order to show that 1) GPU-podracer can support different simulator platforms, and 2) the potential of massively parallel simulations in the DRL training by comparing the CPU-based MuJoCo [26] with the GPU-based Isaac Gym [16].

Compared methods: On one DGX-2 server, we compare GPU-podracer with RLlib [13], since both support multiple GPUs. We used PPO [21] in GPU-podracer, while in RLlib [13] we used the Decentralized Distributed Proximal Policy Optimization (DD-PPO) [28] algorithm that scales well to multiple GPUs. For fair comparison, we keep all adjustable parameters and computing resources the same, such as the depth and width of neural networks, total training steps, number of workers, and GPU and CPU resources.

Performance metrics: We employ two different metrics to evaluate the agent's performance:

Tasks	State space $\mathcal{S}$	Action space $\mathcal{A}$	Reward $r(s, a, s')$
Ant [26, 16]	Body height and rotation	8 controllable joints	Alive bonus
	Velocity and angular velocity		Running speed
	Joint angles		Standing and Heading
	Forces, etc.		Contact forces
Humanoid [26, 16]	Body height and rotation	17 joints for MuJoCo	Alive bonus
	Velocity and angular velocity	21 joints for Isaac gym	Running speed
	Joint angles		Standing and Heading
	Forces, etc.		Contact forces
Stock trading [15]	Balance, Shares	Buy	Change of account value
	Close prices	Sell	
	Technical indicators	Hold	

Table 1: The state space, action space and reward function of ant, humanoid and stock trading tasks.

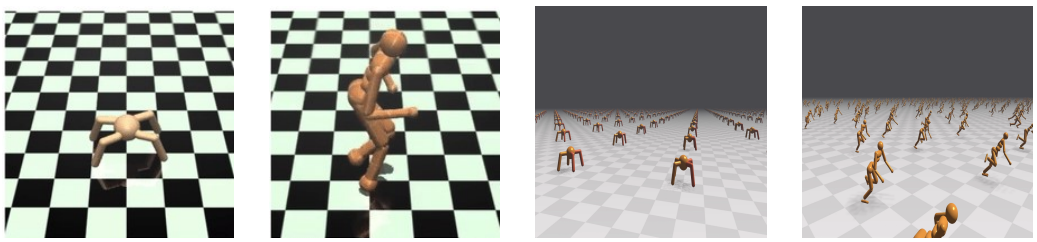

Figure 4: Snapshots of robotic control environments. From left to right, the ant and humanoid tasks from MuJoCo [26], and the ant and humanoid tasks from Isaac Gym [16].

- **Episodic reward vs. training time (wall-clock time)**: we measure the episodic reward at different training time, which can be affected by the convergence rate, communication overhead, scheduling efficiency, etc.
- **Episodic reward vs. training step**: from the same testings, we also measure the episodic reward at different training steps. This result can be used to investigate the massive parallel simulation capability of GPUs, and also check the algorithm's convergence rate.

During the training process, we store the snapshots of the policy networks at different training time, say approximately every 800 seconds. Then, for each snapshot of a policy network, we evaluate it 32 times to obtain 32 episodic rewards and report the corresponding average episodic reward and standard deviation.

Performance analysis: For the four tasks in Fig. 4, we terminate the training processes at $10,000$s (≈ 2.8 hours), $40,000$s (≈ 11.1 hours), $22,000$s (≈ 6.1 hours) and $28,000$s (≈ 7.8 hours), respectively. Correspondingly, GPU-podracer has run 2.4×10^7 steps, 3.3×10^7 steps, 18×10^7 steps and 28×10^7 steps, respectively, while RLlib has run 1.6×10^7 steps, 2.3×10^7 steps, 5.4×10^7 steps and 11.5×10^7 steps, respectively.

As shown in Fig. 5, we can see that given the same training time, GPU-podracer achieves higher episodic reward than RLlib. For the Isaac Gym environments in particular, the corresponding episodic rewards have been tripled. Take a closer look at Fig. 6, we can see that RLlib has not converged yet within the given training time (due to the limitation of cloud usage), e.g, ≈ 6.1 hours and ≈ 7.8 hours, respectively. However, we believe it is not practically viable to use over 20 hours to train an RLlib agent for those two tasks.

5.3 Stock Trading Task

Finance is a promising and challenging real-world application of DRL algorithms [15]. We apply GPU-podracer to a stock trading task as an example to show its potential in quantitative finance.

Stock trading task: we aim to train a DRL agent that decides *where to trade*, *at what price and what quantity* in a stock market, thus the objective of the problem is to maximize the expected return and

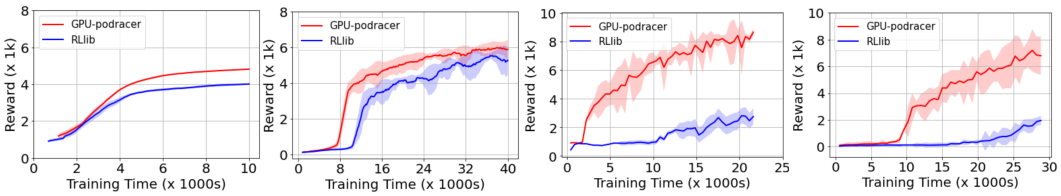

Figure 5: Episodic reward vs. training time (wall-clock time) for the four tasks in Fig. 4.

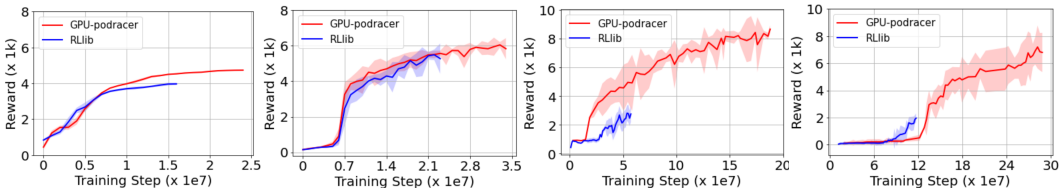

Figure 6: Episodic reward vs. training steps for the four tasks in Fig. 4.

minimize the risk. We model the stock trading task as a Markov Decision Process (MDP) as in FinRL [15], where the state space, action space and reward function are given in Table 1.

Data pre-processing: We select the NASDAQ-100 constituent stocks as our stock pool and use the minute-level dataset for our experiment. For the data preparation, we download the raw data from the Compustat database through the Wharton Research Data Services (WRDS) [22]. Next, we process it to an open-high-low-close-volume (OHLCV) format, and extract technical indicators. Finally, we follow a training-backtesting pipeline and split the dataset into two sets: the data from 01/01/2016 to 05/12/2019 for training, and the data from 05/13/2019 to 05/26/2021 for backtesting.

Performance metrics: We first evaluate training performance and testing performance, respectively. Then, five common metrics are used to quantify the trading performance:

- **Cumulative return**: the ratio between the final capital and the initial capital.
- **Annual return**: the geometric average amount of money earned by the agent each year.
- **Annual volatility**: the annual standard deviation of the return.
- **Sharpe ratio**: the difference between the annualized risk-free rate and annualized return, standardized (divided) by the annualized volatility.
- **Max drawdown**: the maximum percentage loss during the trading period.

For the training performance, we use the same metrics with the robotic control problems in the Section 5.2. For the stock trading problem, we reserve a time period that is not used for training, but only testing to evaluate generalization performance. Since the agent cannot access the testing dataset during the training, we store the model snapshots at different training times, say every 100 seconds. We then later we use each snapshot model to perform inference on the testing dataset to obtain the cumulative return.

Compared methods: We compare GPU-podracer with a vanilla implementation without ensemble training, FinRL [15], Stable Baseline3 [19], and RLlib [13]. Invesco QQQ ETF is used to represent the market performance. There are in total 80 A100 GPUs assigned to our usage. For the ensemble training scheme of GPU-podracer, we use three DRL algorithms, PPO [21], SAC [10] and TD3 [8] with ratio $1:1:1$. We used PPO [21] in vanilla GPU-podracer. For RLlib [13], we used the Decentralized Distributed Proximal Policy Optimization (DD-PPO) [28] algorithm that scales well to multiple GPUs. For fair comparison, we keep all adjustable parameters and computing resources the same, such as the depth and width of neural networks, total training steps, number of workers, and GPU and CPU resources.

Trading performance: From Fig. 7, all DRL agents are able to achieve a better performance than the market baseline with respect to the cumulative return, demonstrating the algorithm's effectiveness. According to Table 2, we observe that GPU-podracer has cumulative return 362.408%, annual return 111.549%, and Sharpe ratio 2.42, which outperforms other methods in all metrics in terms of

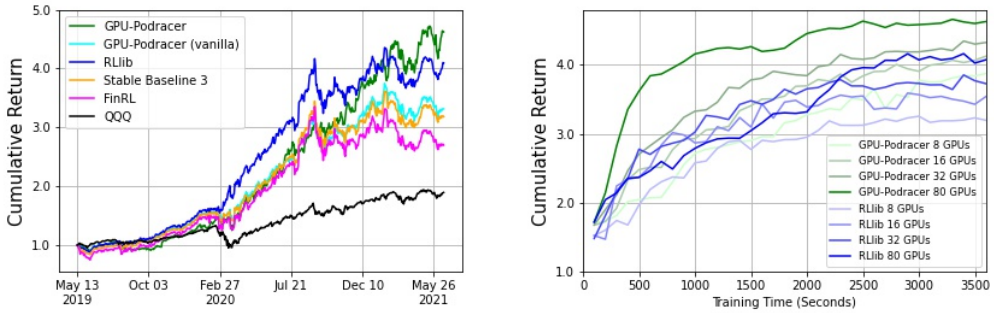

Figure 7: Left: cumulative return on minute-level NASDAQ-100 dataset during 05/13/2019 to 05/26/2021 (initial captial $\$1,000,000$, transaction cost 0.2%). Right: generalization performance using the model snapshots of GPU-podracer and RLlib [13] at different training time (wall-clock time).

	Cumul. return	Annual return	Annual volatility	Max drawdown	Sharpe ratio
GPU-Podracer (**Ours**)	**362.408%**	**111.549%**	33.427%	**-15.874%**	**2.42**
GPU-Podracer (vanilla)	231.747%	79.821%	31.024%	-21.002%	2.05
RLlib [13]	309.540%	99.347%	31.893%	-22.292%	2.33
Stable Baseline3 [19]	218.531%	76.280%	34.595%	-23.750%	1.82
FinRL [15]	169.975%	62.576%	42.908%	-27.267%	1.35
Invesco QQQ ETF	89.614%	36.763%	**28.256%**	-28.559%	1.25

Table 2: Stock trading performance on NASDAQ-100 constituent stocks with minute-level data.

profitability. Moreover, GPU-podracer also shows outstanding stability during the backtesting period: it achieves a low annual volatility 33.427% and max drawdown -15.874%. The high profitability and stability of GPU-podracer demonstrate the effectiveness of our tournament-based ensemble training, which can train a more profitable and robust agent in the complex stock market.

Training efficiency: We compare the training efficiency of GPU-podracer with RLlib [13] on a varying number of GPUs, i.e., 8, 16, 32, and 80 GPUs. In Fig. 7, both GPU-podracer and RLlib [13] can achieve a higher cumulative return at the same training time as the number of GPUs increases, which directly demonstrates the advantage of cloud computing resources on the DRL training. For GPU-podracer, the speed-up of increasing the numbers of GPUs scales near **linearly**. For GPU-podracer with 80 GPUs, it has a much steeper curve and reaches a cumulative return of 4.0 at 800 seconds. GPU-podracer with 32 and 16 GPUs need $2,200$ seconds and $3,200$ seconds to achieve the same cumulative return. Such a linear scaling demonstrates the high scalability of GPU-podracer and the effectiveness of our cloud-oriented optimizations. For the experiments using RLlib [13], increasing the number of GPUs produces relatively similar cumulative return curves and not much speed-up.

6 Discussion and Conclusion

In this paper, we have introduced *GPU-podracer*, a scalable and elastic library for cloud-native deep reinforcement learning. To efficiently utilize millions of GPU cores for DRL training, we first propose a tournament-based ensemble training scheme to orchestrate the training process on hundreds of GPUs, and then enable massively parallel simulation on thousands of GPU cores in a single GPU. Moreover, we follow the cloud-native paradigm to schedule the training of DRL agents by adhering to containerization, microservices, and MLOps. Thus, GPU-podracer realizes the design principles in the respect of **scaling-out**, **efficiency**, and **accessibility**.

By presenting GPU-podracer to the DRL community, we hope that GPU-podracer can help address the data collection bottleneck using the manycore GPU architecture and apply DRL algorithms to complex real-world problems.

References

[1] OpenAI spinning up. `https://spinningup.openai.com`, 2018.

[2] Armin Balalaie, A. Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33:42–52, 2016.

[3] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[5] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.

[6] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. NVIDIA A100 tensor core GPU: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[7] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI baselines. `https://github.com/openai/baselines`, 2017.

[8] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[9] Dennis Gannon, R. Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Comput.*, 4:16–21, 2017.

[10] Tuomas Haarnoja, Aurick Zhou, P. Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[11] N. Herbst, Samuel Kounev, and Ralf H. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, 2013.

[12] 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.

[13] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[14] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

[15] Xiao-Yang Liu, Hongyang Yang, Qian Chen, Runjia Zhang, Liuqing Yang, Bowen Xiao, and Christina Dan Wang. FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance. *Deep Reinforcement Learning Workshop, NeurIPS*, 2020.

[16] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac Gym: High performance GPU-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

[17] P. Mell and T. Grance. The NIST definition of cloud computing. 2011.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[19] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. *GitHub repository*, 2019.

[20] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[21] J. Schulman, F. Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[22] Wharton Research Data Service. Standard & Poor's compustat, 2015. Data retrieved from Wharton Research Data Service.

[23] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[24] Adam Stooke and Pieter Abbeel. rlpyt: A research code base for deep reinforcement learning in pytorch. *arXiv preprint arXiv:1909.01500*, 2019.

[25] NVIDIA DGX A100 system reference architecture. *NVIDIA DGX SuperPOD: Scalable infrastructure for AI leadership*. NVIDIA Corporation, 2020.

[26] 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*, pages 5026–5033. IEEE, 2012.

[27] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[28] Erik Wijmans, Abhishek Kadian, Ari S. Morcos, Stefan Lee, Irfan Essa, Devi Parikh, M. Savva, and Dhruv Batra. DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *ICLR*, 2020.

[29] Tengteng Zhang and Hongwei Mo. Reinforcement learning for robot research: A comprehensive review and open issues. *International Journal of Advanced Robotic Systems*, 18(3):17298814211007305, 2021.