Increasing Sample Throughput for RL Environments

Authors: Bryan Chen, Benny Chen, Adrian Liu

1 Introduction

Although there have been many advances in deep Reinforcement Learning recently, training reinforcement learning (RL) algorithms is a non-trivial problem due to its complex nature. RL algorithms could be divided into on-policy algorithms and off-policy algorithms. Off-policy algorithms like DQN[7], DDPG[5], and SAC[2] do not require the data it is trained on to be produced by the policy it improves. They usually use a cache called replay buffer to store the decision that the agent makes. On-policy reinforcement learning algorithms including A3C[6], TRPO[10], and PPO[11] evaluate and try to improve the policy that produces the action. While the on-policy algorithms are stabler during training, they are also more sample inefficient, often requiring on the order of millions of samples to learn good policies. Even in relatively simple simulated environments, this may take a long time. The question we are trying to answer is: can we utilize the GPU through custom CUDA kernels to simulate common environments in parallel in order to increase sample throughput? If we are successful, our work may speed up training on simple environments and provide the groundwork for major speed ups for more difficult environments that are CPU bottlenecked.

2 Related Work

Stable Baseline's [9] vectorized environment leverages Python's multiprocessing library, which spawns worker processes. Each worker process executes one environment and sends the results to the main process, which introduces overhead from the inter-process communication. RLlib [4] provides remote environments, which create env instances in Ray actors and step them in parallel. These remote processes introduce communication overheads and only help if the environment is very expensive to step / reset. NVIDIA has announced Isaac Gym [8], a physics simulation environment for reinforcement learning research. Isaac Gym enables a complete end-to-end GPU RL pipeline by leveraging NVIDIA's PhysX GPU-accelerated simulation engine, which allows it to gather the experience data required for robotics RL. Isaac Gym also enables observation and reward calculations to take place on the GPU, thereby avoiding significant performance bottlenecks. In particular, costly data transfers between the GPU and the CPU are eliminated. NVIDIA declares that researchers can achieve the same level of success as OpenAI's supercomputer on a single A100 GPU in about 10 hours.

3 Methodology

3.1 Setting

In order to investigate speeding up sample throughput for RL environments, we take a simple and commonly used environment offered by OpenAI Gym [1], CartPole. We briefly introduce the general setting from the perspective of the simulation. The environment moves in discrete steps corresponding to actions from an agent. Thus at a particular timestep t, the agent receives some state information s, takes an action a, and receives a reward r as well as the next state s'. The steps are categorized into "episodes"; an episode might end when the agent reaches some goal or a predetermined amount of steps, upon which the environment is reset to a particular state and the next episode begins. For the purposes of this project, we are *not* aiming to speed up the details of any particular environment. Instead, we aim to increase sample throughput by simulating multiple environments in parallel.

As an illustration, in CartPole the goal of the agent is to balance a pole on a cart. The actions available to the agent are to move the cart left and right. It gets a reward based on how close to balanced the pole is.

3.2 Approaches

We outline the various approaches taken in order to increase sample throughput using various forms of parallelism.

Baseline. The simplest way to induce parallelism would be process-level, and this comes in standard packages like in the stable baselines [9] shown in Figure 1. The idea is to keep p copies of the environment across p processes. Unfortunately, this method is quite ineffective. First, it scales extremely poorly; you can only have as many copies as number of CPU cores; next, the overhead is quite high to synchronize all of the processes. While it is true that you could use multithreading in simpler environments, for more complicated environments this is not possible so we do not examine it.

PyTorch Batching. The first approach we take is to utilize the GPU to induce some parallelism. This most naive way to do this is to reimplement the environment dynamics to do each operation with PyTorch tensors, and extend it in a batched manner so each operation operates over multiple environments. The architecture of this method is shown in Figure 2 While this provides much speedup, there are a couple downsides: first, we have to wait for the slowest thread/environment to complete each operation. Next, the PyTorch operations are black-box and not optimized for the particular calculations we want to do.

Custom CUDA Kernels Our main approach is to take it a step further and write some custom CUDA Kernels in order to fully take advantage of the GPU. We do this using Numba in Python, as shown in Figure 3. The idea is that since we know the parallelism is over the environments, we assign one thread to each environment, and then tune the block size appropriately. Then each

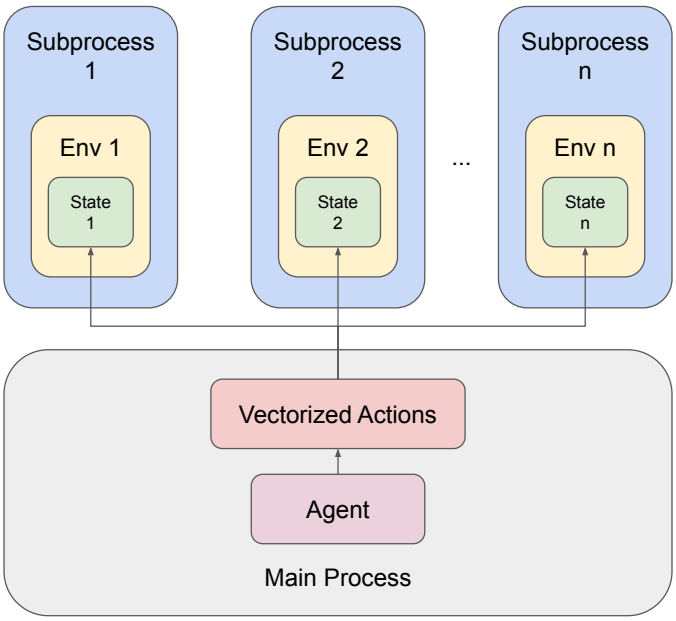


Figure 1: Architecture of stable baseline vectorized environments

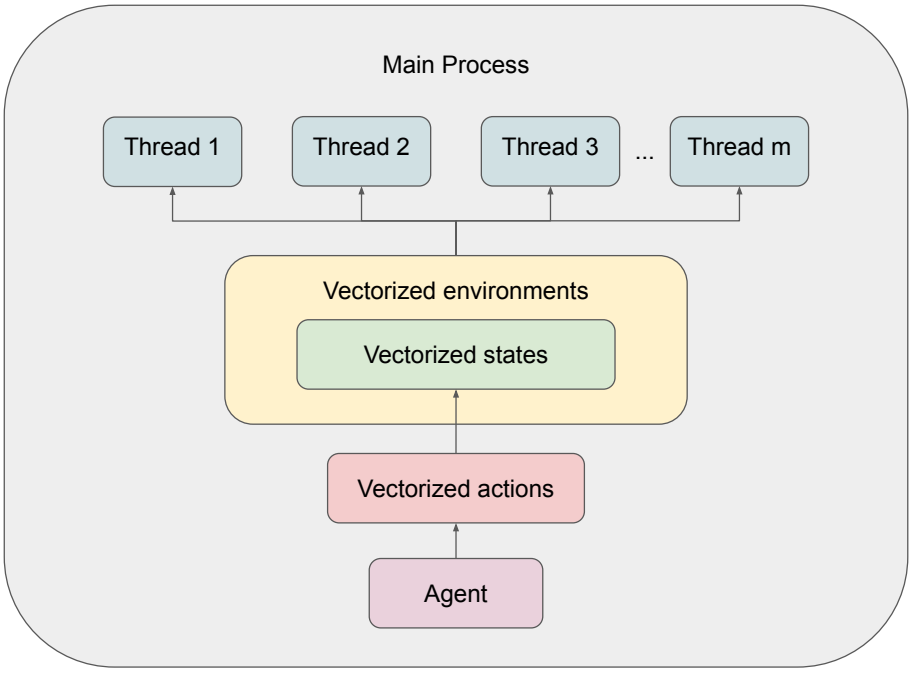


Figure 2: Architecture of vectorized environments based on PyTorch Tensors

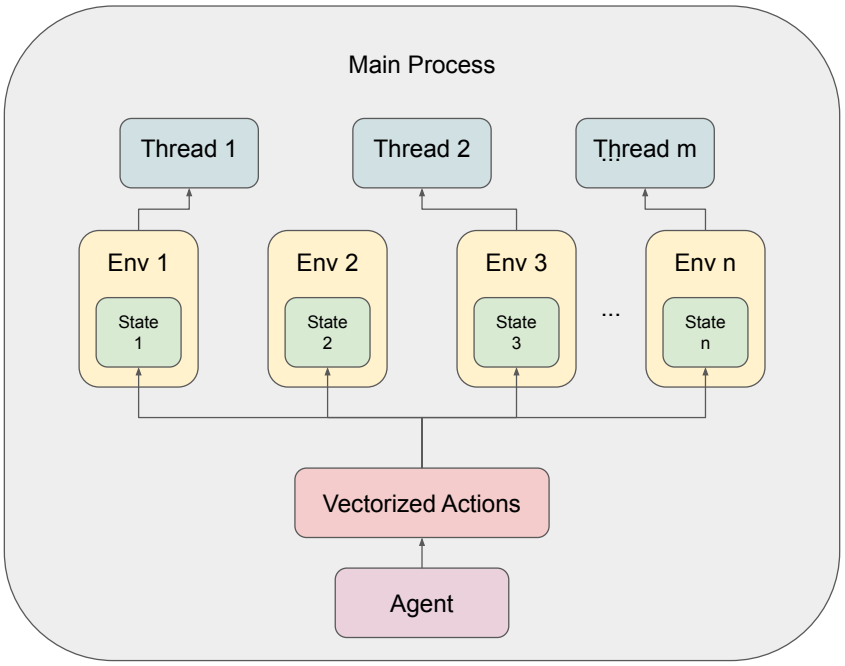


Figure 3: Architecture of vectorized environments based on Numba CUDA Kernel

thread can perform the calculations directly without any synchronization or overhead until the end of the step.

3.3 Further Optimizations

Search over block sizes We know that block sizes should be above 32 to avoid wasted cycles, but what about above that? In order to answer this, we run a search across the block sizes and select the best performing one (with respect to sample throughput.)

Profiling We run a profiler to see whether or not there is a bottleneck during our calculations. However, from the analysis, most of the time is in doing the work and it is unlikely there is further optimization to be found in better sychronization or other details.

4 Results

4.1 Scaling with number of environments

We first compare the runtime of a single step on Stable Baseline's [9] Subprocess Environments (CPU), vectorized environments based on PyTorch Tensor operation, and vectorized environments based on Numba Kernels with different threads per block (TPB) settings. We keep the amount

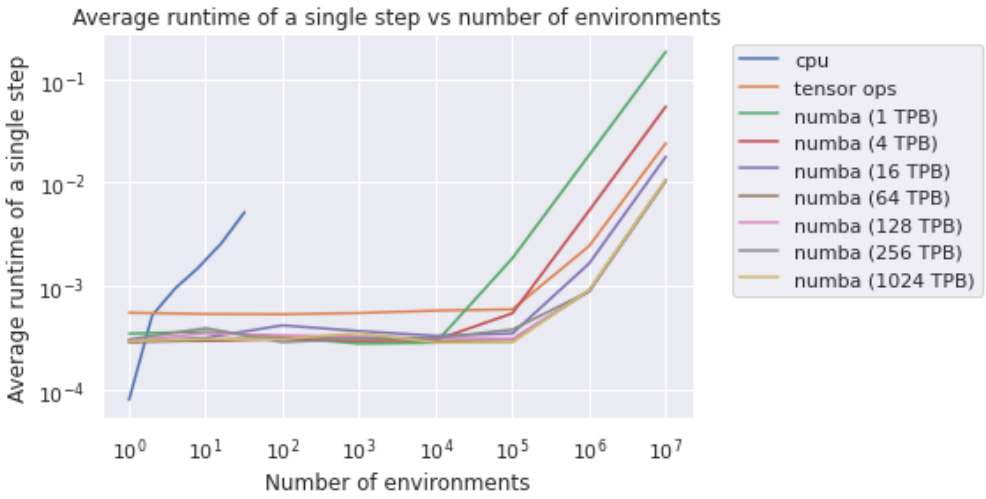


Figure 4: Average runtime of a singe environment step with different number of environments and threads per block

of work same across different environment instances and vary the number of environments. The results are shwon in Figure 4.

The number of total environments are limited to 32 in Stable Baseline, whereas there is no limitation when using GPUs. We find that the Stable Baseline implementation only has the best performance when the number of environments is 1. With a proper TPB, the Numba implementation could run faster than the PyTorch implementation. The number of threads per block can be any value between 1 and 1024 but gives the best performance when it is at least 64. Increasing number of threads per block does not affect performance when it is greater than 64. We use *turning point* to refer to the number of environments where the runtime starts to increase linearly. The turning point is around 10^4 when TPB is low and could reach 10^5 when TPB is higher.

4.2 Scaling with work

We also compare the runtime of different workloads in each settings. We modify the workload in each environment step by repeating the calculation in one step. The results are shown in Figure 5 and Figure 6. We choose the TPB for the Numba Kernel approach to be 64, which gives a better performance than the PyTorch Tensor approach. While the runtime of PyTorch approach always increases linearly with the amount of workload, Numba Kernel's only increases sublinearly when the number of environments is smaller than 10^6.

4.3 Training RL agents using the environments

We train a number of reinforcement learning agents using PPO2 on our environments as well as Stable baseline's vectorized environment. The average rewards are plotted against the training

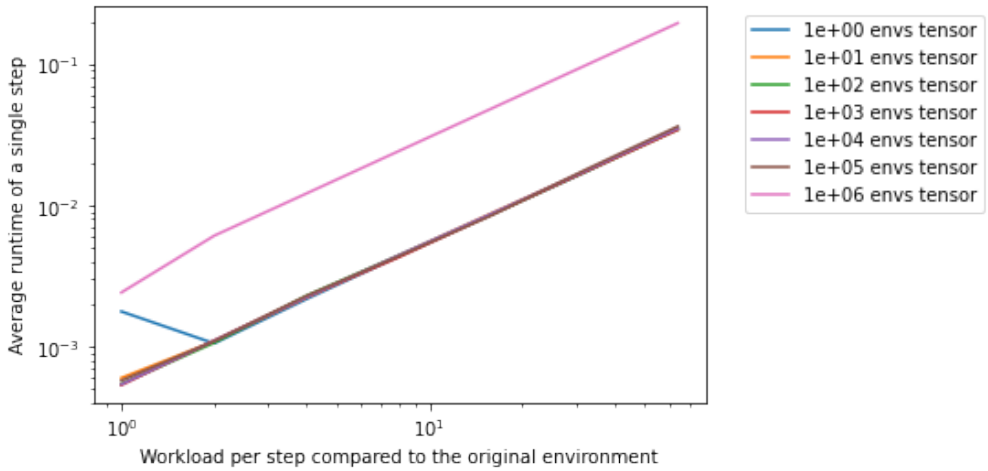


Figure 5: Average runtime of one step with varying workloads and number of environments using PyTorch Tensor

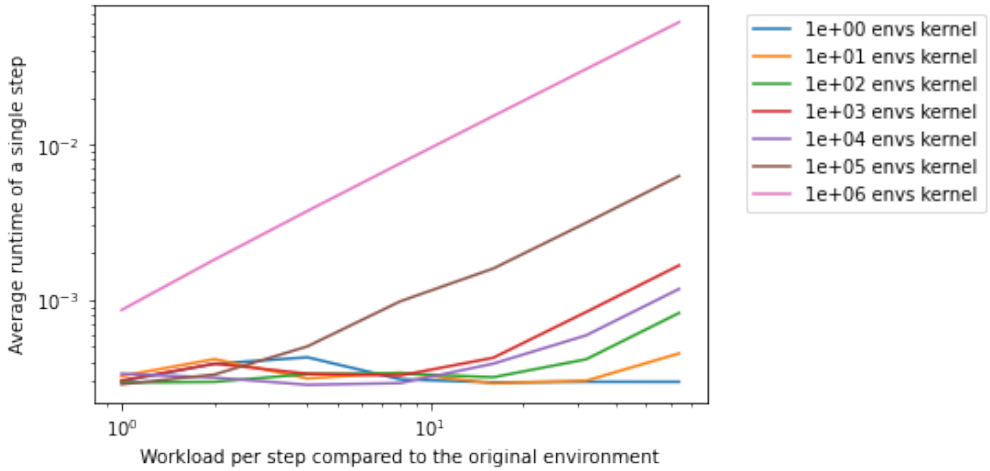
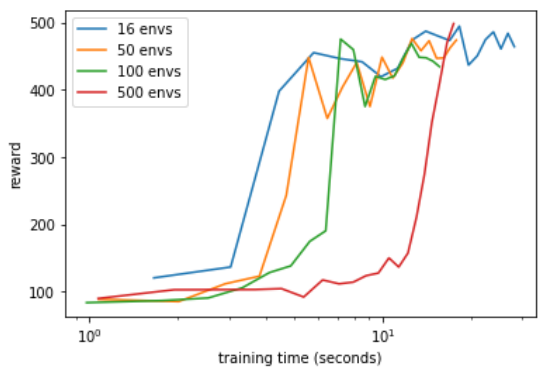
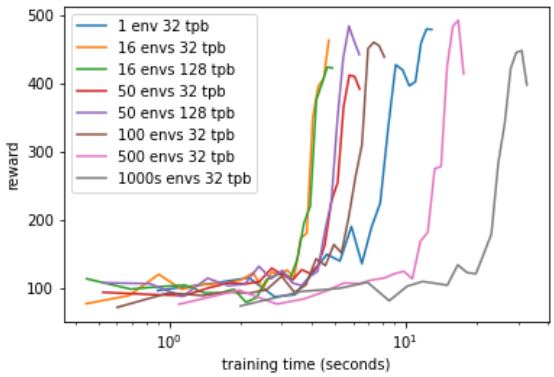


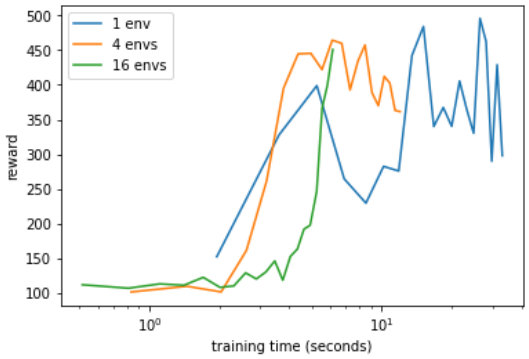
Figure 6: Average runtime of one step with varying workloads and number of environments using Numba Kernel



(a) Pytorch tensor



(b) Numba Kernel



(c) Stable Baselines (CPU)

Figure 7: Average reward against training time

time of the agents, and shown in Figure 7. The agents are evaluated on the original environment and the reported numbers are averages of 5 experiments. The learning curve of RL algorithms can change depending on the number of environments run in parallel. Therefore, we adjusted the number of training steps for each setting such that they would all learn a near-optimal policy in a comparable amount of training time. The results show that our environments can be used to train good policies, but increase in the number of training steps would often offset the faster sampling rate. This suggests further modification to RL algorithms might be needed in order to fully utilize the higher sampling throughput.

4.4 Profiling Diagram

As seen in Figure 8, we use the python code profiler in order to breakdown where the time is being spent during 100 calls to environment steps. The table is produced by snakeviz [3]. As shown, the majority of time (that isn't part of the IPython overhead) is spent quite equally among various function calls to calculations. If there were a bottleneck, we might expect some calls to take much longer than others, or for individual calls among the 100 to vary. It suggests that for our current

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
100	2.028	0.02028	2.336	0.02336	<ipython-input-6-92f1107768ad>:45(step)
101	0.09084	0.0008994	0.1802	0.001784	<ipython-input-6-92f1107768ad>:89(reset)
101	0.06236	0.0006175	0.06236	0.0006175	~:0(<built-in method where>)
100	0.06224	0.0006224	0.06224	0.0006224	~:0(<built-in method rsub>)
100	0.03858	0.0003858	0.03858	0.0003858	~:0(<built-in method sin>)
100	0.02658	0.0002658	0.02658	0.0002658	~:0(<built-in method cos>)
101	0.0263	0.0002604	0.0263	0.0002604	~:0(<built-in method rand>)
1	0.000947	0.000947	2.338	2.338	<ipython-input-8-f4f670a36c0c>:2(the_run)
101	0.000727	7.198e-06	0.000727	7.198e-06	~:0(<method 'unsqueeze' of 'torch._C._TensorBase' objects>)
100	0.000285	2.85e-06	0.06256	0.0006256	tensor.py:525(__rsub__)
1	0.000244	0.000244	0.000244	0.000244	~:0(<built-in method full>)

Figure 8: Table showing runtime breakdown during 100 steps.

approach, it is adequately using the resources and there is no large opportunity for speeding it up.

5 Conclusions

In this work, we present a study of increasing sample throughput for simple RL environments by utilizing parallelism on the GPU. We demonstrate this by utilizing tensor operations and then custom CUDA kernels for an empirical evaluation. Our results are able to outperform the baseline of a simple multiprocessing approach, measured by the sample throughput as we increase the number of environments and as we move to regimes where the environment work increases. We also demonstrate that our environments are usable with modern RL algorithms by training good policies with them. Future work includes moving to environments with sophisticated physics simulations, as well as optimizing for further parallelism speedups in the environments themselves instead of just sample throughput.

References

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

[2] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, August 2018. arXiv: 1801.01290.

[3] Jiffyclub. Snakeviz. `https://jiffyclub.github.io/snakeviz/`.

[4] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray rllib: A composable and scalable reinforcement learning library. *CoRR*, abs/1712.09381, 2017.

[5] 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. *arXiv:1509.02971 [cs, stat]*, September 2015. arXiv: 1509.02971.

[6] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, June 2016. arXiv: 1602.01783.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. page 9.

[8] NVIDIA.

[9] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[10] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, April 2017. arXiv: 1502.05477.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. page 12.