GPU-Accelerated Robotic Simulation for Distributed Reinforcement Learning

Jacky Liang[†‡1]
Robotics Institute
Carnegie Mellon University

Viktor Makoviychuk[†2]
NVIDIA

Ankur Handa[†2]
NVIDIA

Nuttapong Chentanez[2]
NVIDIA

Miles Macklin[2]
NVIDIA
University of Copenhagen

Dieter Fox[2]
NVIDIA

[1]jackyliang@cmu.edu
[2]{vmakoviychuk, ahanda, nchentanez, mmacklin, dieterf}@nvidia.com

Abstract: Most Deep Reinforcement Learning (Deep RL) algorithms require a prohibitively large number of training samples for learning complex tasks. Many recent works on speeding up Deep RL have focused on distributed training and simulation. While distributed training is often done on the GPU, simulation is not. In this work, we propose using GPU-accelerated RL simulations as an alternative to CPU ones. Using NVIDIA Flex, a GPU-based physics engine, we show promising speed-ups of learning various continuous-control, locomotion tasks. With one GPU and CPU core, we are able to train the Humanoid running task in less than 20 minutes, using $10 - 1000\times$ fewer CPU cores than previous works. We also demonstrate the scalability of our simulator to multi-GPU settings to train more challenging locomotion tasks.

Keywords: Deep Reinforcement Learning, GPU Acceleration, Simulation

1 Introduction

Model-free Deep RL has seen impressive achievements [1, 2, 3] in recent years, but many methods and tasks require enormous amount of compute due to the large sample complexity of exploration in high-dimensional state and action spaces. One approach to overcome exploration is by using human demonstrations [4, 5], but collecting human demonstrations remain challenging for many tasks, and it is difficult to scale. Another approach is to vastly scale up RL simulation and training to distributed settings, so large amounts of data can be obtained in a relatively short amount of time. Many recent works of this approach have seen scaling benefits by performing policy training on the GPU while scaling up environment simulation on many CPUs. In this work, we propose using a GPU-accelerated RL simulator to bring the benefits of GPU's parallelism to RL simulation as well.

Using Flex, a GPU-based physics engine developed with CUDA, we implement an OpenAI Gym-like interface to perform RL experiments for continuous control locomotion tasks. We benchmark our simulator on ant and humanoid running tasks as well as their more challenging variations, inspired by ones proposed in OpenAI Roboschool and the Deepmind Parkour environments. They include learning to run toward changing target locations, recovering from falls, and running on complex, uneven terrains. Our choice of tasks is driven by their popularity and the challenges they offer to various Deep RL algorithms. Although our training results are not directly comparable to those obtained in physics simulators used in prior work (*e.g.* MujoCo, Bullet) due to differences in physics simulation, we have endeavoured to do head-to-head comparisons wherever possible. Using

[†] Shared first author
[‡] Work done during internship at NVIDIA

2nd Conference on Robot Learning (CoRL 2018), Zürich, Switzerland.

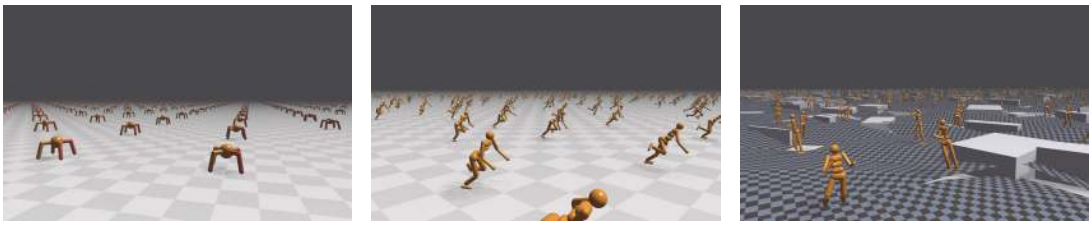


Figure 1: GPU-Accelerated RL Simulation. We use an in-house GPU-accelerated physics simulator, to concurrently simulate hundreds to thousands of robots for Deep RL of continuous control locomotion tasks. Here we show the Ant, Humanoid, and Humanoid Flagrun Harder on Complex Terrain tasks benchmarked in our work. Using a single machine (1 GPU and CPU core), we are able to train humanoids to run in less than 20 minutes.

our GPU-accelerated RL framework to simulate and train hundreds to thousands of agents at once on a single GPU, we were able to achieve faster training results than previous works which used large CPU clusters. In addition, the scale and speed-ups achieved through our simulator, especially in the more challenging tasks, make GPU-accelerated RL simulation a viable alternative to CPU ones.

We summarize our key contributions below:

1. A GPU-accelerated RL simulator built with an in-house GPU-based physics engine. We plan to release our simulator in the near future to facilitate fast RL training to the community.
2. Experiments on massively distributed simulations of hundreds to thousands of locomotion environments on single and multiple GPU settings.
3. Improvements in training speed for various challenging locomotion tasks, learning the humanoid running task in less than 20 minutes on a single machine. See our trained policies at `https://sites.google.com/view/accelerated-gpu-simulation/home`.

We note that in this paper our focus is on the application of our GPU-based physics engine to RL simulation, and not on comparisons and benchmarks of the physics engine itself.

2 Related Works

2.1 Distributed Deep RL

Many prior works have explored parallelizing environment simulation and policy training on CPUs. Nair et al. [6] proposed the first massively parallelized method for training RL agents. Their method, Gorila DQN, has separate learners, actors, and parameter servers. Simulating and training with hundreds of CPU cores it was able to achieve superhuman performance in most Atari games in a few days. Following Gorila DQN, Mnih et al. [1] proposed the Asynchronous-Actor-Critic-Agents (A3C) algorithm, and with 16 CPU cores they were able to compete with Gorila DQN in Atari games with about the same training time.

Babaeizadeh et al. [7] extended A3C by proposing a CPU/GPU hybrid variation of A3C, GA3C, which moves the policy to the GPU. This enabled GPU-accelerated policy inference and training. Their algorithm dynamically adjusts the number of parallel agents, trainers, and parameter servers to maximize training speed. On a single machine with 16 CPU cores and 1 Tesla K40 GPU GA3C achieved more than $4\times$ speed-up over a CPU-only implementation of A3C. Adamski et al. [8] furthered scaled up A3C training by using larger batchsizes and a well-tuned Adam Optimizer. Their method allowed them to learn many Atari games with hundreds of CPU cores, with no GPU, in less than an hour (*e.g.* Breakout with 768 CPU cores in 21 minutes).

Salimans et al. [9] explored using evolutionary strategies (ES) for RL. The authors scaled up ES to use as much as 720 CPU cores to learn Atari games in about one hour. They also performed learning experiments with MuJoCo locomotion tasks, and with 1440 CPU cores they were able to train a humanoid to walk in 10 minutes. Such et al. [10] applied Genetic Algorithms (GA), a

gradient-free optimization method, to RL. GA is also amenable to parallelization, and it was able to learn Atari games also in one hour with 720 CPU cores. In humanoid locomotion tasks however, the algorithm trained much slower than ES, and was not able to achieve comparable performance in the same time frame.

Recent advances in parallel computation tools such as Horovod [11] and Ray have enabled researchers to easily scale up machine learning to distributed settings. Ray RLLib [12] is a distributed RL library using the Ray framework. In their benchmarks, the authors were able to scale ES with Ray RLLib to more than 8000 CPU cores, learning the humanoid walking task in just 3.7 minutes. Mania et al. [13] also used Ray but for their proposed algorithm, Augmented Random Search (ARS). ARS learned the humanoid walking task in 21 minutes with 48 CPU cores, while using $15\times$ less CPU time than ES.

Other previous works aimed to improve the efficiency of off-policy learning from the large amount of data generated by many parallel actors. Espeholt et al. [14] proposed IMPALA, which applies large-scale, distributed learning systems to solving multi-task RL problems. IMPALA is inspired by A3C, but the actors don't compute and send policy gradients to the learners - they send the sampled trajectories instead. The authors scaled IMPALA to use 500 CPU actors and 8 GPU learners, and it learned the benchmarked tasks (DMLab-30, a suite of multi-task video game-like environments) in less than a day. Horgan et al. [15] introduced Distributed Prioritized Experience Replay, an off-policy RL algorithm that uses a novel technique to sample more important trajectories in its replay buffer for learning. This work uses many CPU cores for simulating the RL environment, and 1 GPU for training. With 360 actors, the method learned most Atari games in a few hours, and with 32 actors it trained a humanoid to walk in 1 hour, run in 4 hours. Stooke and Abbeel [16] explored optimizing existing RL algorithms for fast performance on a multi-GPU system. The authors used an entire NVIDIA DGX-1, which contains 8 NVIDIA Tesla V100 GPUs. Running 256 parallel simulations on 40 CPU cores and performing training on all 8 V100s, the authors report being able to train many Atari games in less than 10 minutes. Recently, OpenAI massively scaled up Proximal Policy Optimization [3] to use more than 6000 CPU cores to simulate in-hand manipulation tasks [17] and more than $100,000$ for playing Dota [1].

2.2 Locomotion and Physics Simulation

We focus our attention to continuous control locomotion tasks first proposed in MuJoCo [18], a popular CPU-based physics simulator. DART and Bullet are other notable alternatives, but MuJoCo remains by far the most popular physics simulator in the Deep RL community [9, 10, 12, 13, 15, 19, 3, 20] for its efficient simulation and relatively stable contact models. Duan et al. [19] first benchmarked different RL algorithms on various continuous control tasks such as cartpole swing-up and humanoid walking forward. Later, Schulman et al. [3] introduced more complex locomotion tasks, such as humanoid flagrun where the agent must learn to change and run toward different directions. Heess et al. [20] take this one step further and train humanoid agents to walk and run on uneven and dynamic terrains. Taking inspiration from these works, we use the humanoid running task and its more challenging variations for benchmarking. Owing to the humanoid's high degree of freedom control space, its tasks require most Deep RL algorithms to use a significant number of samples to learn, which provide opportunities for improving learning speed via reduction in simulation time.

Many previous works on distributed RL have focused on discrete control problems such as Atari games, which do not require physics simulation. Moreover, the works in continuous control tasks have only used CPU-based simulations. While GPU-accelerated physics simulations have been applied in scientific computing [21, 22] and healthcare [23, 24], they have yet to be applied in robotics. To achieve state-of-the-art performance, previous works often had to scale environment simulation to hundreds, if not thousands of CPU cores. In our work, we explore using GPU-accelerated simulation as an alternative to CPU-based ones. Using a single GPU, we can simulate hundreds to thousands of robots and achieve state-of-the-art results in locomotion tasks on a single machine, learning the humanoid running task in less than 20 minutes.

[1] https://openai.com/five/

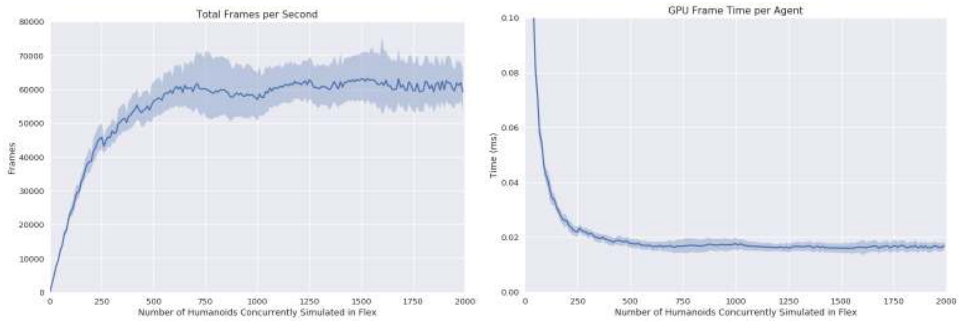


Figure 2: GPU Simulation Speed. We measure the speed of GPU simulation for the humanoid task as we increase the number of concurrent humanoids simulated. The total simulations per second peaks at around 60KHz for 750 humanoids, and the best mean GPU simulation frame time per agent is less than $0.02ms$. The simulation time grows much slower than the number of humanoids because of the constant CUDA kernels launch overhead, which dominates in total step time when only a few humanoids are available.

3 GPU-Accelerated Robotics Simulation

3.1 GPU-based Physics Engine

Our in-house GPU-based physics engine uses a non-smooth Newton method for its rigid body solver and a maximal coordinate formulation. Like with environments in MuJoCo and Bullet, we use torque control as the actuation model. Potential collisions and contacts among bodies are detected speculatively and are modeled through unilateral constraint functions with a smooth isotropic Coulomb friction model. We use sliding friction coefficient of 1.0, the same as in MuJoCo [25]. Restitution coefficient is 0.0 and gravity is $9.8\frac{m}{s^2}$ downward. For time-stepping we use an implicit time-discretization also like [25], and the time step used is $\frac{1}{120}$s. Each Newton iteration is solved by a sparse iterative Krylov linear solver with the minimum number of linear iterations such that simulation is stable for our experiments. We found Krylov methods allowed sufficient stiffness to achieve realistic humanoid gaits, while relaxation methods like Projected Gauss-Seidel were less effective, especially when paired with a maximal coordinate representation.

We develop a GPU-accelerated robotics simulation framework for RL that can simulate many robot agents in parallel for a variety of tasks. To simulate multiple robots performing the same task in parallel, we load all robots and task-related objects into the *same* simulation. This is unlike previous works that parallelize simulation by using multiple CPU processes or threads, each simulating an individual environment. The parallelism and the speed-up in our simulation is achieved by performing the physics computations on the GPU. We note that in our simulations, agents are able to interact with each other, which is not possible for running multiple simulation processes of 1 agent each.

3.2 GPU Simulation Speed

To illustrate the typical performance of our simulator on a single-machine setting, we measured the GPU simulation frame time for the humanoid task as we increase the number of humanoids concurrently simulated in the environment. The results are obtained on an NVIDIA Tesla V100 GPU. The GPU simulation frame time does not include the time needed for calculating rewards, applying actions, and transfer RL-related data back and forth from the Python client, as that speed varies based on implementation of the RL framework. In Figure 2 we report two plots, the total simulation frames generated per second, calculated by multiplying the number of agents by the frames per second of the entire simulation, and the GPU frame time per agent. We note that both values converge with around 750 simulated humanoids, where it can generate 60K frames per second, and the mean frame time per agent is below 0.02.

We observe in our learning experiments that although the total simulation frames generated per second peaks around 750 agents, this is not the optimal number of agents for minimizing learning speed. The number of agents used affects how the learning algorithm and policy explores the state and action spaces, and the threshold for optimizing learning speed depends on the specific task.

4 Experiments

To evaluate the performance of our GPU-accelerated physics simulator for RL, we perform a set of learning experiments on various locomotion tasks. We first measure the learning performance on a single GPU as we vary the number of parallel agents simulated on that GPU. Then we measure how well learning performance scales with multiple GPUs and nodes as we fix the number of agents simulated per GPU.

4.1 Tasks

We perform learning experiments on the following 4 tasks, 3 of which are shown in Figure 1.

Ant. We use the ant model commonly found in MuJoCo locomotion environments. It has 4 legs and 8 controllable joints that form the action space. The goal of the Ant task is to have the agent move forward as fast as possible. Ant is relatively easy to learn, because the initial state of the task is stable. This makes it a useful task for sanity checks and debugging.

Humanoid. Like [20, 26], we use the humanoid model with 28 degrees of freedom and 21 actuated joints. This humanoid is more complex than the 24-DoF humanoid model with 17 actuated joints used in [3, 9, 12, 13]. The 28-DoF humanoid has 4 additional joints to control the ankle angles of the robot, whereas the 24-DoF one has ball-shaped feet that cannot be rotated. We choose the more complex humanoid for benchmarking, because the additional ankle joints allow the humanoid to learn more realistic running and turning motions. The goal of the Humanoid task is to have the agent move forward as fast as possible. This task is often used in the literature for benchmarking locomotion learning performance.

The observations for Ant and Humanoid include the agent's height, velocity, and joint angles, among others. See Appendix B for a detailed comparison of observations used in our and previous works.

Humanoid Flagrun Harder (HFH). In the HFH task, a humanoid must learn to not only run forward but also to turn and run toward different target locations. The agent must also learn how to recover from falling by standing up. This is a much more challenging task than vanilla Humanoid, and it takes more time to train. The action space of this task is the same as that in the Humanoid task. We observe that training a humanoid for HFH leads to more robust and symmetric walking gaits *e.g.* humanoids can maintain their stand-up and running skills even with as much as 50% higher or lower gravity.

Humanoid Flagrun Harder on Complex Terrain. In this task, the agent must learn to run and change directions on uneven terrain with static rectangular obstacles. The dimensions, location, and orientation of the obstacles are randomly sampled from a uniform distribution. The action space of this task is the same as that in the Humanoid task. To help the humanoid navigate complex terrain and overcome obstacles, we augment the observation space with a 2D, 15×11 rectangular height map that follows the humanoid's center of mass. Similar to [20], our height map is denser near the humanoid.

For Ant and Humanoid, an episode terminates if the agents fall below a threshold. For the two HFH tasks, we allow the agents to fall below a threshold for a certain time period (160 frames) before terminating the agent. This enables the agents to learn to stand up and recover from a fall. The Flagrun targets for the HFH tasks change every 200 frames to a random location within $100m$ of the agent, or earlier if the humanoid reaches within $1m$ of the target. For all tasks, the maximum episode length for both training and evaluation is 1000 frames.

Rewards. Similar to previous works, the reward function for all tasks reward the current speed toward the desired targets and penalize excessive torque applied to the joints. Our reward functions however, are not immediately comparable to previous works, due to a smaller alive bonus and the addition of other terms that we empirically found to lead to more natural, symmetric running gaits. We note that the locomotion rewards used in MuJoCo and Bullet are also different, arising from the vagaries of implementation details, such as the solver and the number of iterations used for optimisation. See Appendix C, D for the exact rewards used and a comparison of our rewards with those used in previous work.

A common reward threshold for solving the humanoid running forward task is 6000 [9, 12, 13], but this threshold is for the 24-DoF humanoid, and to our knowledge there is no widely used reward

threshold for the 28-DoF humanoid and for the HFH tasks. For the Humanoid task, we chose a reward threshold of 3000 for walking and 5000 for running. 3000 roughly corresponds to a forward moving speed of 2 *m/s*, and 5000 for 4 *m/s*, which is about the same speed as Roboschool's example 24-DoF humanoid running policy. For Ant, we use 3000 as the reward threshold for running, and 7000 for final reward. For the HFH task, we use 2500 as an intermediary reward threshold, around which the agents first learn to stand up from the ground, and 4000 as the final reward.

Initial State and Perturbations. Unlike parallel simulations on CPUs that simulate multiple agents in their own environments, usually one environment per CPU core, we simulate all agents in one environment on the GPU. The initial positions, velocities, joint angles, and joint angular velocities of agents are perturbed slightly during initialization. We also exert random forces onto the agents for all 4 tasks every 200 to 300 frames, for a few Newtons each time. These external perturbations help the agent to learn more robust policies. We also enabled inter-agent collisions for the HFH tasks. The initial spacing of the humanoids affect the collision frequency, and the occasional collisions help the agents to explore states where they must learn to balance recover from falls, leading to more robust policies.

4.2 Learning Algorithm

We use a variation of Proximal Policy Optimization (PPO) [3] for all our experiments to benchmarks locomotion tasks. We adapted the open source OpenAI Baseline [2] implementation of PPO to work with our simulation environment, where a single environment step simulates multiple agents concurrently. Similar to [3, 20], we also use an adaptive-learning rate based on the KL divergence between the current and previous policies. Additionally, for stability we whiten the current observations by maintaining online statistics of mean and standard deviation from the history of past observations.

Our policy and value functions share the same feed-forward network architectures. As in the Baselines implementation of PPO, we use scaled exponential linear units (SELU [27]) for the activation function. SELU implicitly encourages normalized activations, and in our hyperparameter search, policies with SELU learned faster and achieved higher rewards than those with ReLU and Tanh.

For our multi-GPU benchmarks, we implemented two variants of Distributed PPO. Our first variant uses multiple GPUs to generate rollouts but trains the policy and value function on a single GPU. This is often the case with CPU based implementations, where each CPU worker generates rollouts, and one master CPU trains. Our second variant is similar to Distributed PPO [20]. Both are synchronized algorithms, where at every iteration, gradients from all workers are applied to a central policy, and the worker policies are updated with the new policy parameters and this is what we use for all our experiments. We found that the first variant was not scalable to multiple nodes. We also experimented with averaging parameters, but we found that it performed significantly worse than averaging gradients. We use Horovod [3] for distributed simulation and training across multiple GPUs and nodes where each GPU runs its own simulation and training instance in Tensorflow. The weight parameters are updated by averaging gradients across multiple GPU workers using efficient allreduce by NCCL [4] and broadcasting from a master GPU. Importantly, since the *env.step()* function is implemented directly on GPU for multiple agents, we are able to leverage the parallelism offered by GPUs to obtain observation-action pairs concurrently for hundreds to thousands of agents.

4.3 Hardware

All experiments were done using NVIDIA Tesla V100 GPUs on NVIDIA's internal compute cluster. Each single-GPU experiment uses 1 CPU core of a 20-Core Intel Xeon E5-2698 v4 processor running at 2.2 GHz. For multi-GPU experiments, we scale the number of CPU cores used to match the number of GPUs used.

4.4 Single-GPU Simulation and Training

We first performed hyperparameter grid-search on the number of frames used per PPO optimization iteration and the network architectures for training 1024 parallel agents on the Ant and Humanoid

[2] https://github.com/openai/baselines

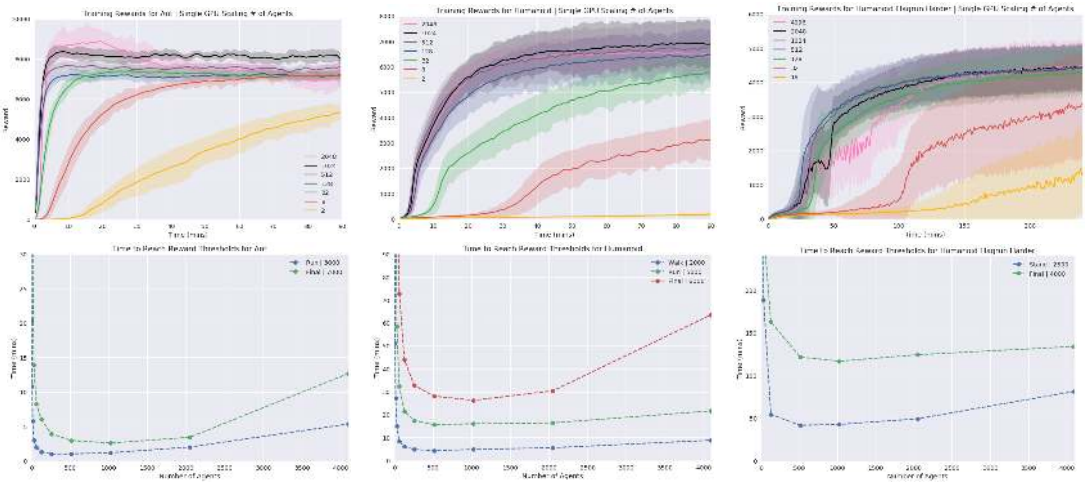
[3] https://github.com/uber/horovod

[4] https://developer.nvidia.com/nccl

tasks. For both tasks, we found the best policy and value network architectures to have 3 hidden layers decreasing in size. See Appendix E for the specific architectures and hyperparameters used.

We found the best frames per iteration within our searched values for Ant and Humanoid to be 32 frames per iteration $\times 1024$ agents = 32768 frames. This number is kept constant as we scale the number of parallel agents simulated up and down. For example, for 512 agents, we use 64 frames per iteration, and for 2048 we use 16. Keeping the frames per iteration constant helps us to show differences in learning speed as caused by improvements in simulation speed, and not by performance of the learning algorithm. We note that for high agent counts, the small number of frames used still enable learning, because our tasks use dense rewards.

We report the time needed to reach certain reward thresholds for the Ant, Humanoid, and HFH tasks as we vary the number of agents in Figure 3. Because we fix the amount of experience used per PPO update constant, we are able to observe a trade-off between increasing the number of agents but collecting less frames per agent and decreasing the number of agents but collecting more frames per agent. The point of diminishing return varies across task and reward thresholds.



(a) Ant (b) Humanoid (c) Humanoid Flagrun Harder

Figure 3: Single GPU Experiments. We show the reward curves and wall time needed for training various tasks using increasing numbers of simulated agents to reach certain reward thresholds. The number of agents we evaluated vary in the powers of 2. The reward thresholds were chosen for significant behavior changes. For example - at around 2500 reward the Humanoid Flagrun Harder agents learn to stand up from sitting positions and begin walking. We keep the amount of experience used per PPO update iteration constant across evaluations by decreasing the frames used per agent as the number of agents increase. Using 512 agents we were able to train the Humanoid agent to run in about 16 minutes. All experiments were running against the same set of seeds for consistent comparison.

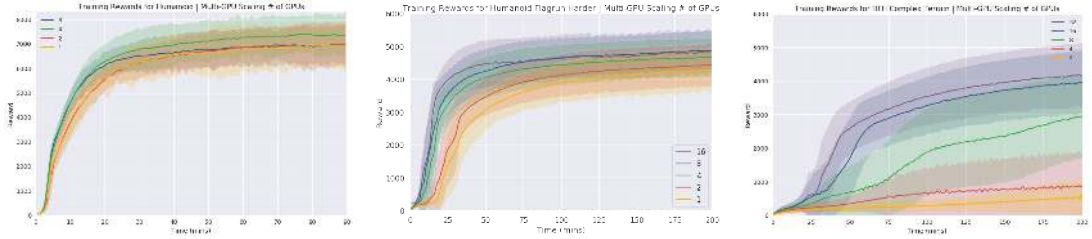
We also note the short time needed to learn these tasks using GPU-accelerated simulations. We list training times and resources used for the humanoid task in prior works in Table 1, all of which used CPU-based physics simulation. With one GPU and CPU core, the Humanoid agents were able to run in less than 20 minutes while using 10$\times$ to a 1000$\times$ less CPU cores than previous works [13, 9].

Algorithm	CPU Cores	GPUs	Time (mins)
Evolution Strategies [9]	1440	-	10
Augmented Random Search [13]	48	-	21
Distributed Prioritized Experience Replay [15]	32	1	240
Proximal Policy Optimization w/ GPU Simulation (Ours)	1	1	16

Table 1: Resources and Times for Training a Humanoid to Run. Prior works all used CPU-based physics simulations. In this table, we do not include the original PPO paper [3] - it used 128 CPU cores for the humanoid task but did not report training time. We also did not include the Distributed PPO paper [20] - their humanoid training took more than 40 hours to converge, but they did not report the number of CPU cores used.

4.5 Multi-GPU Distributed Simulation and Training

We extend our method to distribute GPU simulations across multiple GPU workers to see how learning speed can be improved on the Humanoid, HFH, and HFH on Complex Terrain tasks. For these experiments, we run a simulation and training instance on each GPU, and we use Horovod for distributed gradients averaging. We also normalize the advantage estimates across all GPUs and distribute them back to each GPU worker at every iteration, ensuring that advantages across all GPUs share a consistent global mean and standard deviation. The number of agents simulated per GPU for Humanoid and HFH is 1024. We use a smaller number of 512 agents per GPU for HFH on Complex Terrain to keep memory usage and simulation speed reasonable, as the addition of the height map significantly increases the dimensionality of the observations. Results are reported in Figure 4. We observe only limited scaling effects for the Humanoid task, which hit diminishing returns after 4 GPUs. In the more complex tasks however, we observed noticeable speed-ups. For the HFH task, the 1 GPU run reached 4000 rewards in about 2 hours, while the 8 GPU run reached it in about 30 minutes, and 16 GPUs in about 15. For the HFH on Complex Terrain task, we observe more apparent scaling benefits with multiple GPU simulation and training. On average, the 16 and 32 GPU runs learn the task faster than the 2, 4, and 8 GPU runs, while the large overlap in standard deviations for 8 GPUs with 16 and 32 shows the diminishing returns of using more agents in learning.



(a) Humanoid (b) Humanoid Flagrun Harder (HFH) (c) HFH Complex Terrain

Figure 4: Multi-GPU Simulation and Training. We show how our GPU-accelerated RL simulation can be scaled to simulation and training with multiple GPUs. The overlap of standard deviations indicates that there are little scaling effects for the Humanoid task, and the benefit of multi-GPU simulation and training is only apparent for more complex tasks and with a greater difference in the number of GPUs used. All experiments were running against the same set of seeds for consistent comparison.

5 Conclusion and Future Work

In this work, we used an in-house GPU-accelerated physics simulator to concurrently simulate hundreds to thousands of robots for Deep RL of continuous-control, locomotion tasks. In contrast to prior works that trained locomotion tasks on CPU clusters, with some using hundreds to thousands of CPU cores, we are able to train a humanoid to run in less than 20 minutes on a single machine with 1 GPU and CPU core, making GPU-accelerated RL simulation a viable alternative to CPU-based ones. Our RL simulation framework can also be scaled to multi-GPU and multi-node settings, and we observed that multi-GPU simulation and training shows greater learning speed improvements for more complex locomotion tasks. Given the recent successes of sim2real transfer learning, from grasping in clutter [28], quadruped locomotion [29], to dexterous manipulation [30], all of which used Bullet to generate simulation data to train policies that worked in the real world, we believe our simulator can provide valuable speed-ups for similar applications in the future.

In future work, we plan to experiment with more complex humanoid environments by allowing the humanoid to actively control the orientation of the rays used to generate the height map. This may enable the humanoids to navigate dynamic obstacles and obstacles in mid-air. We also plan to use our simulator for manipulation tasks with robots such as the Fetch, Baxter, and YuMi. In this work, we've considered locomotion tasks with full state information. For many tasks in manipulation and navigation however, training from vision data is preferred. For such tasks, we note the potential of zero-copy training - directly feeding simulation data generated by a GPU-based simulator and the task's states and rewards into a deep learning framework without the data leaving the GPU. Zero-copy training eliminates the need to communicate data from the GPU to the CPU, and can further improve training speed.

Acknowledgments

We thank Phil Rogers, Vikrama Ditya, Christopher Lamb, Nathan Luehr, David Addison, Hari Sundararajan, Sivakumar Arayandi Thottakara and many others who manage the NVIDIA GPU infrastructure for all the kind help they provided in carrying out the experiments on the GPU clusters.

References

[1] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529: 484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

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

[4] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[5] A. Rajeswaran, V. Kumar, A. Gupta, J. Schulman, E. Todorov, and S. Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *CoRR*, abs/1709.10087, 2017.

[6] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[7] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz. Reinforcement learning through asynchronous advantage actor-critic on a gpu. 2016.

[8] I. Adamski, R. Adamski, T. Grel, A. Jedrych, K. Kaczmarek, and H. Michalewski. Distributed deep reinforcement learning: Learn how to play atari games in 21 minutes. *arXiv preprint arXiv:1801.02852*, 2018.

[9] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[10] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[11] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[12] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica. Ray rllib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.

[13] H. Mania, A. Guy, and B. Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[14] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[15] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[16] A. Stooke and P. Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[17] OpenAI. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2017.

[18] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi:10.1109/IROS.2012.6386109.

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

[20] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[21] C. Freniere, A. Pathak, M. Raessi, and G. Khanna. The feasibility of amazon's cloud computing platform for parallel, gpu-accelerated, multiphase-flow simulations. *Computing in Science & Engineering*, 18(5):68–77, 2016.

[22] J. Spiechowicz, M. Kostur, and L. Machura. Gpu accelerated monte carlo simulation of brownian motors dynamics with cuda. *Computer Physics Communications*, 191:140–149, 2015.

[23] A. L. Blumers, Y.-H. Tang, Z. Li, X. Li, and G. E. Karniadakis. Gpu-accelerated red blood cells simulations with transport dissipative particle dynamics. *Computer physics communications*, 217:171–179, 2017.

[24] J. Wu, C. K. Chui, and C. L. Teo. A software component approach for gpu accelerated physics-based blood flow simulation. In *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, pages 2465–2470. IEEE, 2015.

[25] E. Todorov. Implicit nonlinear complementarity: A new approach to contact dynamics. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2322–2329. IEEE, 2010.

[26] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[27] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 972–981, 2017.

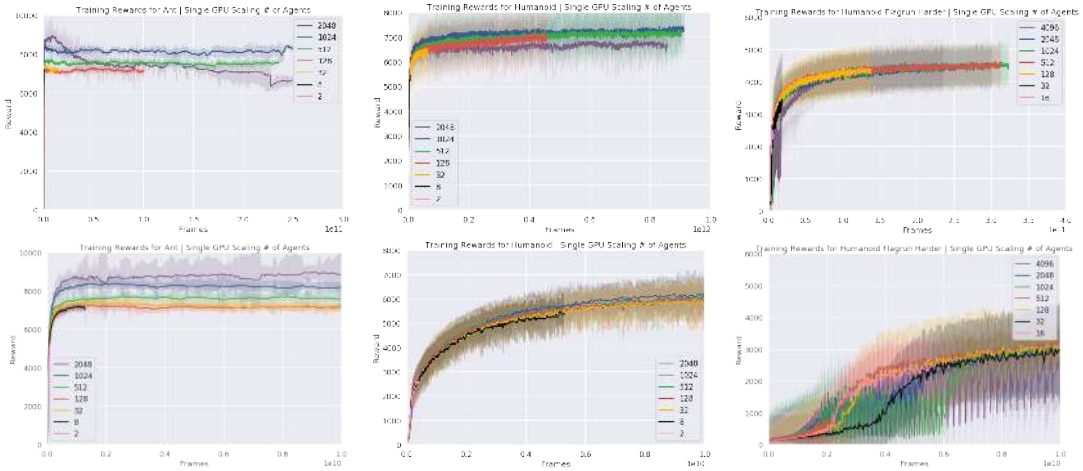
[28] J. Mahler and K. Goldberg. Learning deep policies for robot bin picking by simulating robust grasping sequences. In *Conference on Robot Learning*, pages 515–524, 2017.

[29] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[30] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. Learning dexterous in-hand manipulation, 2018.

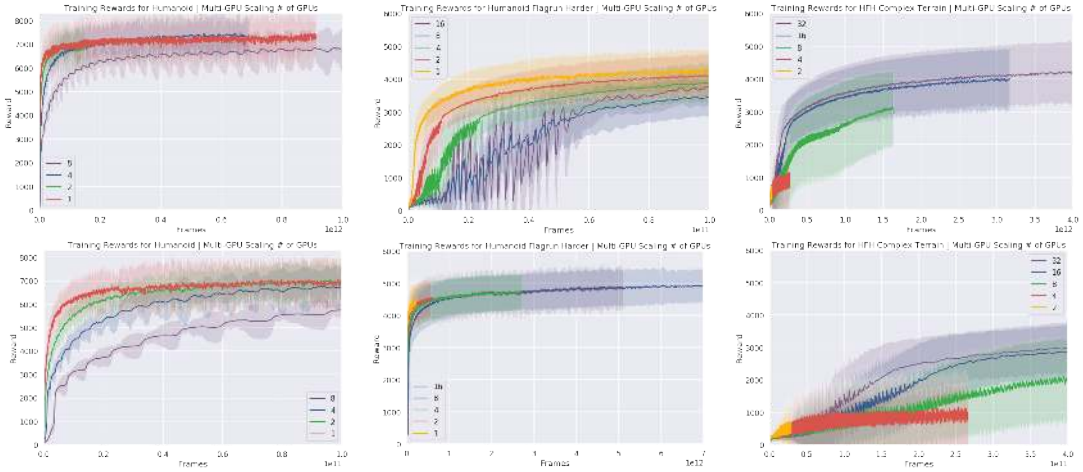
A Rewards vs Frames

We plot the reward vs frames curves for single-GPU experiments in Figure 5 and multi-GPU experiments in Figure 6. A zoomed-in version of each plot is shown on the second row. The difference in the number of total frames for different number of agents is due to the fact that we stop training based on a fixed amount of time.



(a) Ant (b) Humanoid (c) Humanoid Flagrun Harder

Figure 5: Reward vs Frames for Single GPU Experiments.



(a) Humanoid (b) Humanoid Flagrun Harder (HFH) (c) HFH Complex Terrain

Figure 6: Reward vs Frames for Multi-GPU Experiments.

B Comparison of Observations

Table 2 compares the different observations for the Humanoid running task used in MuJoCo, Roboschool, Control Suite, and our environments.

	MuJoCo [5]	Roboschool [6]	Control Suite [7]	Ours
Root Body Height	1	1	1	1
Root Body Rotation	4	2	3	2
Root Body Velocity	-	3	3	3
Root Body Angular Velocity	-	-	-	3
Root Body Heading Direction	-	2	-	2
Joint Angles	17	17	21	21
Joint Angle Velocities	23	17	-	21
Positions of Hands and Feet	-	-	4×3	-
Velocities of Bodies	14×3	-	27	-
Angular Velocities of Bodies	14×3	-	-	-
Inertia Tensor and Mass	14×10	-	-	-
Actuator Forces	23	-	-	21
External Forces on Bodies	14×6	-	-	-
Whether Feet in Contact w/ Ground	-	2	-	2
Total	376	44	67	76

Table 2: Observation and dimensionality comparison for Humanoid running task. The Root Body Heading Direction for Roboschool and our agents are represented by the sine and cosine values of the angular difference between our agent's heading and the direction to the target (which for the humanoid running task is just forwards from the starting location). The Root Body Rotation is represented as quaternions for MuJoCo, roll and pitch angles for Roboschool and ours, and the z-projection of the rotation matrix for Control Suite.

C Rewards

The reward function we used for all four tasks are as follows:

$$R = R_{\text{alive}} + S + 0.5R_{\text{heading}} + 0.05R_{\text{standing}} - 4\frac{1}{\tau_{\text{max}}}||\tau||_1 - 0.5||u||_2^2 - 0.2N_{\text{joints}} - N_{\text{feet}} \quad (1)$$

$$R_{\text{heading}} = \begin{cases} 1 & \cos(\theta_{\text{target}}) > 0.8 \\ \cos(\theta_{\text{target}})/0.8 & \cos(\theta_{\text{target}}) <= 0.8 \end{cases} \quad (2)$$

$$R_{\text{standing}} = \mathbb{1}\{\cos(\theta_{\text{vertical}}) > 0.93\} \quad (3)$$

θ_{target} is the angle from the robot's current heading to the angle toward the target location. θ_{vertical} is the angle of the robot's torso from the vertical-axis (i.e. if the humanoid is standing up right, this angle would be 0). R_{alive} is the alive bonus, and it is 0.5 for Ant and 2 for humanoids. S is the speed toward the current target. τ is the vector of motor torques applied at each joint, with τ_{max} being the maximum that can be applied. u is the current action. N_{joints} is the number of joints at joint limits, and N_{feet} is the number of feet that is in collision with ground.

[5] https://github.com/openai/gym/wiki/Humanoid-V1

[6] https://github.com/openai/roboschool/blob/master/roboschool/gym_forward_walker.py

[7] https://github.com/deepmind/dm_control/blob/master/dm_control/suite/humanoid.py

D Comparison of Reward Functions

Table 3 compares the coefficients for the summands in the reward function for the Humanoid running task used in MuJoCo, Roboschool, and our environments.

	MuJoCo [8]	Roboschool [9]	Ours
Alive Bonus	5	2	2
Running Speed Bonus	0.25	1	1
Heading Bonus	-	-	0.5
Standing Bonus	-	-	0.05
Control Cost	-0.1	$-\frac{0.1}{N_d}$	-0.5
Electricity (Torque) Cost	-	-4.25	$-\frac{4}{\tau_{\max}}$
Joints at Limits Cost	-	-0.2	-0.2
Feet Contact Cost	-	-1	-1
External Forces Cost	-5×10^{-6}	-	-

Table 3: Reward function comparison for Humanoid Running task. N_d is the dimension of the controls. The External Forces Cost for MuJoCo is the multiplier of the sum of squares of external forces on all bodies. The reward function for the humanoid walking task in Control Suite is very different in structure - it is the product of the running speed with two coefficients that depend on the magnitude of the controls and how upright the humanoid is. See here [10] for details.

E Hyperparameters

In table 4 we give the hyperparameters used during training for Ant, Humanoid, and Humanoid Flagrun Harder tasks. The timesteps per batch is given relative to a specific amount of parallel agents simulated - 1024, and this is scaled across different experiments. For example, if the timesteps per batch is 32, then we used 64 with the experiment that has 512 agents, and 16 with the experiment that has 2048 agents. The desired KL specifies the target KL value used for adapting the Adam step size at each iteration.

The value and policy networks in our PPO algorithm have the same feed-forward architectures, and they are specified in a list format, where the nth value gives the size of the nth layer.

Hyperparameter	Ant	Humanoid	HFH	HFH Terrain
Timesteps per Batch	32	32	64	64
Num. Epochs	20	20	10	20
Minibatch Size per Agent	16	32	32	8
Desired KL	0.01	0.02	0.01	0.01
Neural Net Architecture	[128, 64, 32]	[256, 128, 64]	[512, 256, 128]	See Caption

Table 4: Hyperparameters used in different tasks. For the HFH Terrain neural network, we pass the 15×11 height map into two fully-connected (FC) layers of sizes $[256, 128]$, the other observations through one layer of size 512, then finally pass their concatenated outputs through two more FC layers of sizes $[256, 128]$ before outputting the controls.

[8] https://github.com/openai/gym/blob/master/gym/envs/mujoco/humanoid.py

[9] https://github.com/openai/roboschool/blob/master/roboschool/gym_forward_walker.py

[10] https://github.com/deepmind/dm_control/blob/master/dm_control/suite/humanoid.py