LARGE BATCH SIMULATION FOR DEEP REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce a reinforcement learning system that trains agents in photo-realistic 3D simulated environments at over 19,000 frames of experience per second on a *single* GPU (and up to 72,000 frames per second on a single eight-GPU machine) – more than 100× faster than prior work in the same environments. Our core innovation is a custom simulator design that accepts and executes large batches of simulation requests simultaneously. Batch simulation allows in-memory storage of scene assets, rendering work, and synchronization costs to be shared/amortized across many simulation requests, dramatically improving the number of simulated agents per GPU and overall simulation throughput. To keep up with faster simulation, we build a computationally efficient policy DNN that maintains high task performance with reduced inference and training costs, and modify training algorithms to maintain sample efficiency when training with large mini-batches. By combining batch simulation, inference, and training performance optimizations with multi-GPU parallelism, we demonstrate that PointGoal navigation agents can be trained in photo-realistic 3D environments on a single GPU in 1.5 days to 97% of the accuracy of agents trained on a prior state-of-the-art system using a 64-GPU cluster over three days. This increase in performance enables rapid exploration of reinforcement learning algorithms in regimes that were previously prohibitively slow and computationally expensive.

1 INTRODUCTION

Speed matters. It is now common for modern reinforcement learning (RL) algorithms leveraging deep neural networks (DNNs) to require *billions* of samples of experience from simulated environments (Wijmans et al., 2020; Petrenko et al., 2020; OpenAI, 2018; Silver et al., 2017; Vinyals et al., 2019). For embodied AI tasks such as visual navigation, where the ultimate goal for learned policies is deployment in the real world, learning from *realistic simulations* is important to ensure policies are transferable to physical robots. In these cases simulators must render detailed 3D scenes and simulate agent interaction with complex environments (Kolve et al., 2017; Dosovitskiy et al., 2017; Savva et al., 2019; Xia et al., 2020; Gan et al., 2020).

Evaluating and training a DNN on billions of simulated samples is computationally expensive and therefore slow without parallelization. For instance, the DD-PPO system (Wijmans et al., 2020) used 64 GPUs over 3 days to learn from 2.5 billion frames of experience and achieve near-perfect PointGoal navigation in photo-realistic 3D scans of indoor spaces (Gibson dataset (Xia et al., 2018)). Unfortunately, experiments at this scale are out of reach for most researchers and this problem will only grow worse as the field explores more complex tasks in more detailed simulated environments.

Recent efforts to scale RL focus on system architectures that improve the efficiency of DNN evaluation and training – *e.g.*, 'centralizing' computations to facilitate efficient, batch execution on GPUs or TPUs (Espeholt et al., 2020; Petrenko et al., 2020) or parallelizing across GPUs (Wijmans et al., 2020). However, most RL platforms still accelerate simulation by running *many copies of off-the-shelf, unmodified simulators* – *e.g.*, simulators designed for video game engines (Bellemare et al., 2013; Kempka et al., 2016; Beattie et al., 2016) – on large numbers of CPUs or GPUs. This approach is a simple and productive way to improve simulation throughput, but it makes inefficient use of computation resources. For example, when rendering complex environments (Kolve et al., 2017; Savva et al., 2019; Xia et al., 2020), a single simulator instance might consume gigabytes of GPU memory, limiting the number of simulation instances to far below the parallelism afforded

Figure 1: We train agents to perform PointGoal navigation in visually complex Gibson (Xia et al., 2018) and Matterport3D (Chang et al., 2017) environments such as the ones shown here. These environments are detailed scans of real-world locations composed of up to 600K triangles and high-resolution textures. Using batch simulation, our system is able to train agents using 64×64 depth sensors (a high-resolution example is shown on the left) in these environments at 19900 frames per second, and agents with 64×64 RGB cameras at 13300 frames per second on a *single* GPU.

by a machine. Further, running many simulator instances (potentially distributed across machines) can introduce overhead in synchronization and communication with other components of the RL system. We contend that inefficient simulation is a major reason RL platforms typically require scale-out parallelism to achieve high end-to-end system throughput.

In this paper, we crack open the simulation black box and explore *batch simulation* for RL: designing high-throughput simulators that accept large batches of requests as input (aggregated across different environments and potentially with different assets) and efficiently execute the entire batch at once. Exposing work en masse facilitates a number of optimizations: we reduce memory footprint by sharing scene assets (geometry and textures) across rendering requests (enabling orders of magnitude more environments to be rendered simultaneously on a single GPU), amortize rendering work using GPU commands that draw triangles from multiple scenes at once, hide latency of scene I/O, and exploit batch transfer to reduce data communication and synchronization costs between the simulator, DNN inference, and training. To achieve end-to-end RL speedups, DNN evaluation and inference must be optimized to match the high rate of simulation, so we design a computationally efficient policy DNN that maintains high task performance with reduced cost. Large-batch simulation increases the number of samples collected per training iteration, so we also employ techniques from large-batch supervised learning to maintain sample efficiency in this regime.

We evaluate our system on the task of PointGoal navigation (Anderson et al., 2018) in 3D photo-realistic Gibson and Matterport3D environments, and demonstrate that end-to-end optimization of batched simulation, inference, and training yields over a 110× speedup over state-of-the-art prior systems (running on the same GPU), while achieving 97% of the task performance for `Depth` agents and 91% for `RGB` agents. Specifically, we demonstrate sample generation and training at over 19,000 frames of experience per second on a single GPU.[1] In real-world terms, a single GPU is capable of training a virtual agent on 26 years of experience in a single day.[2] This new performance regime significantly improves the accessibility and efficiency of RL research in realistic 3D environments, and opens new possibilities for more complex embodied tasks in the future.

2 Related Work

Systems for high-performance RL. Accelerating experience collection – the inference-simulation loop – has been a subject of significant interest. Espeholt et al. (2018) and Horgan et al. (2018) proposed a system where processes asynchronously collect experience that is sent to a centralized learner. Espeholt et al. (2020) and Petrenko et al. (2020) built upon this idea and introduced inference workers that centralize network inference, thereby allowing it to be accelerated by GPUs or TPUs. Wijmans et al. (2020) proposed a synchronous distributed system for similar purposes. A number of efficient implementations of these ideas have been proposed as part of RL frameworks or in other deep learning libraries (Liang et al., 2018; Stooke & Abbeel, 2019; Küttler et al., 2019).

[1] Samples of experience used for learning, not 'frameskipped' metrics typically used in Atari/DMLab.

[2] Calculated based on wall-clock time a physical robot (LoCoBot (Carnegie Mellon University, 2019)) would need to collect the same number of observations, operating constantly at maximum speed (0.5 m/s) capturing 1 frame every 0.25m.

Despite differences in approach, task, RL methodology, *etc*., all these works have one commonality – they treat a simulator designed for efficient *single* agent simulation as a black box. We take an orthogonal approach and crack open this black box to improve its efficiency under RL workloads. Our developed improvements to simulation are synergistic with distributed training. A high-throughput, efficient, large-batch simulator is a drop-in replacement for existing (non-batched) simulation workers, reducing the number of processes and therefore communication overhead needed for asynchronous methods and eliminating simulation worker processes all together for synchronous methods. We demonstrate this by combining our system with DD-PPO (Wijmans et al., 2020).

Large mini-batch optimization. A consequence of large batch simulation is that more experience is collected between gradient updates. This provides the opportunity to accelerate learning via large mini-batch optimization. In supervised learning, large mini-batch optimization typically decreases the generalization performance of models (Keskar et al., 2017). Goyal et al. (2017) first demonstrated that this issue can be circumvented by scaling the learning rate proportionally with the batch size and 'warming-up' the learning rate at the start of training. You et al. (2017) proposed an optimizer modification, LARS, that adaptively scales the learning rate at each layer and applied it to SGD to improve generalization further. In reinforcement learning and natural language processing, the Adam optimizer (Kingma & Ba, 2015) is often used instead of SGD. You et al. (2020) proposed to combine LARS (You et al., 2017) with Adam (Kingma & Ba, 2015).

We do not find that large mini-batch optimization harms generalization in reinforcement learning, but we do find it decreases sample efficiency. We adapt the techniques proposed above – learning rate scaling (You et al., 2017) and the Lamb optimizer (You et al., 2020) – to improve sample efficiency.

Simulators for machine learning. Several simulation platforms focused on realistic environments have emerged in recent years. These fall into two broad categories: those built on top of pre-existing game engines (Kolve et al., 2017; Dosovitskiy et al., 2017; Lee et al., 2019; Gan et al., 2020), and those built from scratch using open-source 3D graphics and physics libraries (Savva et al., 2017; 2019; Xia et al., 2018; 2020; Xiang et al., 2020). While performance has been a focus of this line of work, it has been evaluated in a narrow sense (*i.e*. frame rate benchmarks for predetermined agent trajectories), not accounting for performance of end-to-end RL training. We instead take a holistic approach to co-design rendering and simulation modules and their interfaces to the RL training system, obtaining significant gains in end-to-end performance over the state of the art.

3 System Design & Implementation

Batch simulation accelerates rollout generation during RL training by processing many simulated environments simultaneously in large batches. Figure 2 illustrates how batch simulation interacts with policy inference to generate rollouts. Simulation for sensory-motor agents, such as the Point-Goal navigation task targeted by our implementation, can be separated into two tasks: determining the next environment state given an agent's actions and rendering its sensory observations. Therefore, our design utilizes two components: a batch simulator that performs geodesic distance and navigation mesh computations on the CPU, and a batch renderer that renders the complex Gibson (Xia et al., 2018) and Matterport3D (Chang et al., 2017) environments on the GPU.

During rollout generation, batches of requests are passed between these components—given N agents, the simulator produces a batch of N environment states. Next, the renderer processes the batch of environment states by simultaneously rendering N frames and exposing the result directly in GPU memory. Agent observations (from both the simulator and the renderer) are then provided as a batch to policy inference to determine the next actions for the N agents.

The key idea is that implementations of the batch simulation and rendering (in addition to policy inference and training) *take responsibility for their own parallelization*. Large batch sizes (values of N on the order of hundreds to thousands of environments per batch) provide opportunities for implementations to efficiently utilize parallel execution resources (*e.g*. GPUs) as well as amortize processing, synchronization, and data communication costs across many environments. The remainder of this section describes the design and key implementation details of our system's batch simulator and batch renderer, as well as contributions that improve the efficiency of policy inference and optimization in this regime.

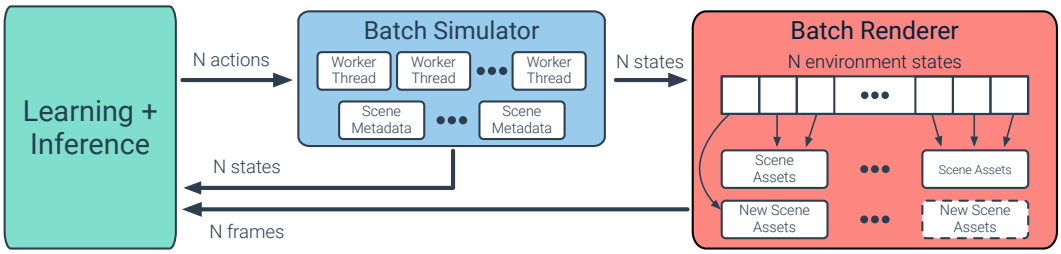

Figure 2: The batch simulation and rendering architecture. The batch renderer maintains a set of unique scene assets in GPU memory, which are shared by subsets of the N simulated environments to amortize per-environment memory usage. Asynchronously, the renderer loads new assets, which can be used to instantiate new environments as loading completes (shown above as the leftmost environment). The simulator and renderer are decoupled by separating scene asset data into simulator metadata such as navigation meshes versus geometry and textures used by the renderer.

3.1 Batch Environment Simulation

Our CPU-based batch simulator executes geodesic distance and navigation mesh computations in parallel for a large batch of environments. Due to differences in navigation mesh complexity across environments, the time to perform simulation may differ per environment. This variance is the source of workload imbalance problems in parallel synchronous RL systems (Wijmans et al., 2020; Savva et al., 2019) (and one motivation for recent asynchronous designs (Petrenko et al., 2020; Espeholt et al., 2020; 2018)). To ensure good workload balance, our batch simulator operates on large batches that contain *significantly more* environments than the number of available CPU cores, and dynamically schedules work onto cores using a pool of worker threads (simulation for each environment is carried out sequentially). Worker threads record simulation results into a designated per-environment slot in a results buffer. When simulation of all environments is complete, the results buffer is efficiently communicated to the renderer as a single batched request.

3.2 Batch Rendering

A high-performance renderer for performing RL in scanned real-world environments such as Gibson (Xia et al., 2018) and Matterport3D (Chang et al., 2017) must efficiently produce many low-resolution renderings (e.g., 64×64 pixels) of scenes that contain high-resolution textures and meshes with up to 600K triangles. Low-resolution output presents challenges for GPU acceleration. Rendering images one at a time produces too little rendering work to efficiently utilize a modern GPU rendering pipeline's parallel processing resources. Rendering many environments concurrently but individually (e.g., from different worker threads or processes) exposes more rendering work for the GPU, but still suffers from the overhead of sending the GPU many (small) rendering commands to draw a large number of environments.

To address the problem of rendering many small images efficiently, our renderer combines GPU rendering commands for all N environments in a simulation batch into a *single rendering request* to the GPU – effectively drawing the entire batch as a single large frame (individual renderings are tiles in the large output image). This approach amortizes GPU pipeline configuration and rendering overhead over an entire batch while also exposing large amounts of rendering work to the GPU. Our implementation makes use of modern GPU pipeline features (Khronos Group, 2017) that allow rendering tasks accessing different texture and mesh assets to proceed as part of a single large operation without triggering GPU pipeline flushes due to pipeline state reconfiguration.

A challenge of batch rendering is that large N (we use up to 1024) is necessary to efficiently utilize a GPU, but geometry and texture assets for a single environment may require gigabytes of GPU memory. Naively using a unique set of scene assets per environment prevents use of large N due to GPU memory limitations. Our solution is to allow multiple environments in a batch to reference the same 3D scene assets (geometry and textures) in GPU memory. By materializing K unique assets in GPU memory, and constructing batches of N environments that reference these assets ($K \ll N$), it is possible to render large batches of environments without exceeding GPU memory. Asset reuse decreases the diversity of training experiences in a batch. To preserve diversity we limit the ratio of N to K in any one batch to 32, and continuously modify the set of K assets maintained in GPU memory. Specifically, in parallel with rendering (and the larger RL system), the renderer asynchronously loads new scene assets onto the GPU. Once new assets are loaded, new rollout environments can reference the newly loaded assets. Assets no longer referenced by environments

are flagged for removal from GPU memory, resulting in a continuously rotating set of K active assets. This design allows policy optimization to learn from an entire dataset of assets, while staying within GPU memory footprint and masking the latency of asset loading behind rollout generation.

When rendering detailed geometry to low-resolution images, most scene triangles are subpixel in size, making geometry processing, not pixel processing, the dominant rendering workload. As a result, rendering performance is determined by the rate at which the GPU's rasterization hardware can process triangles. To reduce the number of triangles the GPU pipeline must process, the renderer discards geometry that lies outside the agent's view—a process known as frustum culling (Akenine-Möller et al., 2018). Our implementation employs GPU compute shaders to perform frustum culling on batches of triangles. Frustum culling the geometry for one environment is pipelined with triangle rendering of other environments in a batch. This design both reduces rendering work and increases GPU utilization by using processing by the GPU's programmable cores (for culling) to reduce the amount of work performed by the GPU pipeline's triangle processing hardware.

3.3 Policy DNN Architecture

High-throughput batch simulation creates a need for high-performance policy DNN inference. Therefore, we develop a policy DNN architecture designed to achieve an efficient balance between high task performance and low computational cost. Prior work in PointGoal navigation (Wijmans et al., 2020) used a policy DNN design where a visual encoder CNN processes an agent's visual sensory information followed by an LSTM (Hochreiter & Schmidhuber, 1997) that determines the policy's actions. We adopt this core design for our policy DNN while significantly reducing computational complexity compared to prior work.

First, we reduce resolution from 256×256 (Habitat, 2019) to 64×64, reducing computation considerably. Beyond this simple optimization, we choose a relatively shallow visual encoder – a nine-layer ResNet (ResNet18 with every other block removed) – followed by a two-layer LSTM. Additionally, we use a `SpaceToDepth` stem (Ridnik et al., 2020) as this performs equally to the standard `Conv+MaxPool` stem while using less GPU memory and compute. To counteract the lower capacity, all stages include Squeeze-Excite (SE) blocks (Hu et al., 2018) with r=16.

Finally, we avoid use of normalization layers in the CNN as these require spatial reductions over the feature maps, preventing layer-fusion optimizations. Instead, the CNN utilizes Fixup Initialization (Zhang et al., 2019) to improve training stability. Fixup initialization replaces expensive normalization layers after each convolution by cheap elementwise multiplication and addition.

3.4 Large Mini-Batch Policy Optimization

In on-policy reinforcement learning, policy optimization utilizes trajectories of experience to reduce bias and for backpropogation-through-time. When generating trajectories of length L with a simulation batch size of N, a rollout will have $N \times L$ steps of experience. Therefore, a consequence of simulation with large N is that more experience is collected per rollout.

Large N presents the opportunity to utilize large mini-batches to improve throughput of policy optimization. However, in supervised learning, large mini-batch training is known to hurt generalization (Keskar et al., 2017). While we do not see evidence of this in reinforcement learning for PointGoal navigation, we find that it does harm sample efficiency. Consider an agent learning its initial skills; if these skills are prerequisite to learning the task, *e.g.* not continuously bumping into a wall, then it is paramount this is learned with few samples. To counteract this, we adapt techniques from the large mini-batch optimization literature.

First, we scale the learning rate by $\sqrt{\frac{B}{B_{\text{base}}}}$, where B_{base}=256. We find it beneficial to use the scaled learning rate immediately instead of 'warming-up' the learning rate (Goyal et al., 2017). Second, we use and adapt the Lamb optimizer (You et al., 2020). Lamb is a modification to Adam (Kingma & Ba, 2015) that applies LARS (You et al., 2017) to the step direction estimated by Adam to better handle high learning rates. Since the Adam optimizer is often used with PPO (Schulman et al., 2017), Lamb is a natural choice. Given the Adam step direction $s_t^{(k)}$ for weights $x_t^{(k)}$,

$$x_{t+1}^{(k)} = x_t^{(k)} - \eta_t r_t^{(k)}(s_t^{(k)} + \lambda x_t^{(k)}) \qquad r_t^{(k)} = \frac{\phi(||x_t^{(k)}||)}{||s_t^{(k)} + \lambda x_t^{(k)}||} \tag{1}$$

where η_t is the learning rate and λ is the weight decay coefficient. We set $\phi(||x_t^{(k)}||)$ as $\min\{||x_t^{(k)}||, 10.0\}$ and introduce an additional clip on the trust ratio $r_t^{(k)}$:

$$r_t^{(k)} = \min\left\{\max\left\{\frac{\phi(||x_t^{(k)}||)}{||s_t^{(k)} + \lambda x_t^{(k)}||}, \rho\right\}, \frac{1}{\rho}\right\} \tag{2}$$

We find the exact value of ρ to be flexible (we observed similar training with $\rho = \{10^{-2}, 10^{-3}, 10^{-4}\}$) and also observed that this clip is only influential at the start of training, suggesting that there is an initialization scheme where it is unnecessary.

4 RESULTS

Simulation of detailed 3D environments such as Gibson (Xia et al., 2018) and Matterport3D (Chang et al., 2017) is computationally expensive and is a compelling target for acceleration. The fastest published end-to-end training performance in these environments is achieved with a synchronous RL system; specifically, the implementation of PointGoal navigation presented with DD-PPO (Wijmans et al., 2020). Therefore, to evaluate the impact of our contributions on end-to-end training speed and task performance in these environments, we compare three synchronous PPO-based implementations.

4.1 SYSTEM SETUP

Batch Processing Simulator (BPS). We provide an RL system for learning PointGoalNav built around the batch simulation techniques and system-wide optimizations described in Section 3. The remainder of the paper refers to this system as BPS (Batch Processing Simulator). In addition to optimizations presented in Section 3, BPS uses half-precision inference and mixed-precision training.

Baseline. The primary baseline for this work is the open-source Habitat-Lab implementation of PointGoalNav, which uses Habitat-Sim (Savva et al., 2019) – the prior state-of-art in high-performance simulation of realistic environments such as Gibson. Subsequent sections refer to the Habitat-Lab implementation using a ResNet50 visual encoder as Baseline. Unlike BPS, Baseline simulates multiple environments simultaneously by using separate worker processes.

Baseline++. This implementation uses the same policy DNN (including performance optimizations) as BPS, but otherwise uses the same system design and simulator as Baseline (with a minor modification to not load textures for Depth agents). Baseline++ serves to isolate the impact of two components of BPS: first, the low-level DNN efficiency improvements, and, more importantly, the performance of batch simulation versus Baseline's independent simulation worker design.

Multi-GPU training. To support multi-GPU training, all three systems replace standard PPO with DD-PPO (Wijmans et al., 2020). DD-PPO scales rollout generation and policy optimization across all available GPUs, scaling the number of environments simulated and the number of samples gathered between training iterations proportionally. We report results with eight GPUs.

Determining batch size. The per-GPU batch size, N, controls a trade-off between memory usage, sample efficiency, and speed. For BPS, N determines the batch size for simulation, inference, and training. For Baseline and Baseline++, N determines the batch size for inference and training, as well as the number of simulation processes. Baseline sets N=4 for consistency with Wijmans et al. (2020). To maximize performance of single-GPU runs, BPS chooses the largest batch size that fits in GPU memory, subject to the constraint that no one scene asset can be shared by more than 32 environments in the batch. In eight-GPU configurations, DD-PPO scales the number of parallel rollouts with the number of GPUs, so to maintain reasonable sample efficiency BPS limits per-GPU batch size to N=128. Baseline++ Depth experiments use N=64 (limited by system memory due to N separate processes running Habitat-Sim). Batch size in Baseline++ RGB experiments is limited by GPU memory (N ranges from 6 to 20 depending on the GPU). We give the batch sizes used in all experiments in Appendix A.

4.2 EXPERIMENTAL SETUP

PointGoal navigation. We train and evaluate agents via the same procedure as Wijmans et al. (2020): agents are trained for PointGoalNav (Anderson et al., 2018) with either a Depth sensor

Sensor	System	Tesla V100		RTX 2080TI		RTX 3090
		1 GPU	8 GPUs	1 GPU	8 GPUs	1 GPU
Depth	BPS	12600	46900	12900	72000	19900
	Baseline++	2100	13100	2800	9300	2800
	Baseline	200	1360	230	1600	180
RGB	BPS	9000	37750	8400	43000	13300
	Baseline++	1500	8400	860	4600	990
	Baseline	190	1320	OOM	OOM	140

Table 1: **System performance.** Average frames per second (FPS, measured as samples of experience processed per second) achieved by each system. BPS achieves a speedup of 110× over Baseline on Depth experiments and 95× on RGB experiments on an RTX 3090 GPU. OOM (out of memory) indicates that the RTX 2080TI could not run Baseline with the published DD-PPO system parameters due to insufficient GPU memory.

	Sensor	System	Validation		Test	
			SPL	Success	SPL	Success
1	Depth	BPS	94.4±0.7	99.2±1.4	91.5	97.3
2		Baseline	95.6±0.3	99.9±0.2	94.4	98.2
3	RGB	BPS	88.4±0.9	97.6±0.4	83.7	95.7
4		Baseline	92.9	99.1	92.0	97.7

Table 2: **Policy performance.** SPL and Success of agents produced by BPS and Baseline. The performance of the BPS agent is within the margin of error of the Baseline agent for Depth experiments, and within four percent on RGB.

or an RGB camera. Depth agents are trained on Gibson-2plus (Xia et al., 2018) and RGB agents are additionally trained on Matterport3D (Chang et al., 2017). RGB camera simulation requires textures for the renderer, increasing the GPU memory consumed by each scene significantly. Both classes of agent are trained on 2.5 billion simulated samples of experience.

Agents are evaluated on the Gibson dataset (Xia et al., 2018). We use two metrics: Success, whether or not the agent reached the goal, and SPL (Anderson et al., 2018), a measure of both Success and efficiency of the agent's path. We perform policy evaluation using Habitat (Savva et al., 2019), unmodified for direct comparability to prior work.

Benchmark evaluation. We report end-to-end performance benchmarks in terms of average frames per second (FPS) achieved by each system. We measure FPS as the number of samples of experience processed over 16000 inference batches divided by the time to complete rollout generation and training for those samples. Results are reported across three models of NVIDIA GPUs: Tesla V100, GeForce RTX 2080TI, and GeForce RTX 3090 (details in Appendix B).

4.3 End-to-End Training Speed

Single-GPU performance. On a single GPU, BPS trains agents 45× to 110× faster than Baseline (Table 1). The greatest speedup was achieved using the RTX 3090, which trains Depth agents at 19900 FPS and RGB agents at 13300—a 110× and 95× increase over Baseline, respectively. A reduction in BPS simulation batch size from Depth to RGB is the primary cause of the decrease in performance between tasks (further detail in Section 4.5).

For Depth agents, BPS outperforms Baseline++ by 4.5× to 7.8×. Under several configurations, Baseline++ achieves an order of magnitude speedup over Baseline. This increase is due to Baseline++'s ability to use up to 8× more simulation instances (and correspondingly, use larger batch size for policy inference) than Baseline because it does not load RGB textures. When training RGB agents, each Baseline++ simulation process must load textures, keeping the maximum number of simulation workers low. As a result, BPS outperforms Baseline++ by 6× to 13× on RGB.

Multi-GPU performance. BPS outperforms Baseline by 29× to 34× on the Tesla V100s and by 45× on RTX 2080TIs. Although using more GPUs yields high end-to-end throughput (up to 72000 FPS for Depth on eight RTX 2080TIs), BPS exhibits imperfect multi-GPU scaling because per-GPU batch size is lower in eight-GPU configurations due to sample efficiency concerns. As a result, GPU efficiency in eight-GPU experiments is lower than in the single-GPU experiments. Multi-GPU scaling for Depth is lower on the Tesla V100 (3.7×) compared to the 2080TI (5.6×) because larger batch sizes are needed to utilize the large number of parallel compute units on the Tesla V100.

4.4 Policy Task Performance

We evaluate the performance of our trained agents to understand how BPS's system design and policy DNN architecture impact final agent performance. For Depth agents, the reduction in CNN depth does not result in a significant reduction in Success and only a 1% reduction in SPL on Val. On Test, a 3 SPL and 0.9 Success reduction (Table 2, row 1 vs. 2). For RGB agents, BPS suffers a performance loss of 3.8/1.3 SPL/Success on Val and 8.3/2.0 SPL/Success on Test (Table 2, row 3 vs. 4). Despite this performance reduction, the RGB agent trained by BPS would have won the 2019 Habitat challenge by 4 SPL and is only beaten by Baseline's ResNet50-based policy on the Habitat test split.

4.5 Runtime Breakdown

Figure 3 provides a breakdown of time spent in each of the main components of the BPS system (μs per frame). Although BPS renders complex environments and uses a small, low-cost policy DNN, on the RTX 3090 GPU nearly 60% of runtime (for both Depth and RGB) is spent in DNN inference and training. This result demonstrates the high degree of simulation efficiency achieved by BPS.

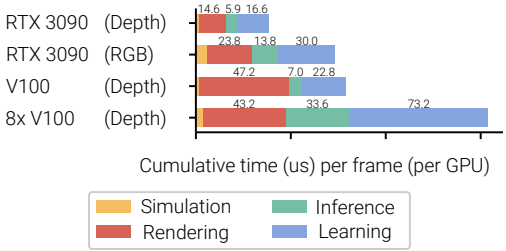

Figure 3: BPS runtime breakdown. Inference represents policy evaluation cost during rollout generation. Learning represents the total cost of policy optimization.

DNN operations on the RTX 3090 are nearly twice as fast for Depth as for RGB, even though these tasks use the same policy DNN. The reason is that RGB (N=256) must use a smaller batch size than Depth (N=1024) to fit texture assets in GPU memory. The larger batch size improves GPU utilization for all system components. A similar effect is visible when comparing the single-GPU and eight-GPU Depth experiments on the V100. BPS reduces the per-GPU batch size from N=1024 to N=128 in eight-GPU experiments (to maintain an aggregate batch size of 1024 for sample efficiency). Further work in policy optimization to address this learning limitation would improve multi-GPU scaling by allowing larger aggregate batch sizes.

5 Discussion

We demonstrated that architecting an RL training system around the idea of batch simulation can accelerate learning in complex 3D environments by one to two orders of magnitude over prior work. With these efficiency gains, agents can be trained with billions of simulated samples from complex environments in about a day using only a single GPU. We believe the fast turnaround times enabled by this level of performance stand to make RL in realistic simulated environments accessible to a broad range of researchers, increase the scale and complexity of tasks and environments that can be explored, and facilitate new studies of how much visual realism is needed to learn a given task (e.g., dynamic lighting, shadows, custom augmentations). We intend to open source our simulation system to the community to facilitate such efforts.

More generally, this work demonstrates the value of building RL systems around components that have been specifically designed for RL workloads, not repurposed from other application domains. We believe this philosophy should be applied to other components of future RL systems, in particular to new systems for performing physical simulation in complex environments.

REFERENCES

Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2018.

Peter Anderson, Angel Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, et al. On Evaluation of Embodied Navigation Agents. *arXiv:1807.06757*, 2018.

Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv:1612.03801*, 2016.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 2013.

Carnegie Mellon University. Locobot: An open source low cost robot. https://locobot-website.netlify.com/, 2019.

Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. In *International Conference on 3D Vision (3DV)*, 2017. MatterPort3D dataset license available at: http://kaldir.vc.in.tum.de/matterport/MP_TOS.pdf.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1–16, 2017.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Chuang Gan, Jeremy Schwartz, Seth Alter, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, Megumi Sano, et al. Threedworld: A platform for interactive multi-modal physical simulation. *arXiv:2007.04954*, 2020.

Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv:1706.02677*, 2017.

Habitat. Habitat Challenge 2019 @ Habitat Embodied Agents Workshop. CVPR 2019. https://aihabitat.org/challenge/2019/, 2019.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997.

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. *arXiv:1807.11205*, 2018.

Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, 2016.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

Khronos Group. The Vulkan specification. 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3D environment for visual AI. *arXiv:1712.05474*, 2017.

Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. Torchbeast: A pytorch platform for distributed rl. *arXiv:1910.03552*, 2019.

Youngwoon Lee, Edward S Hu, Zhengyu Yang, Alex Yin, and Joseph J Lim. IKEA furniture assembly environment for long-horizon complex manipulation tasks. *arXiv:1911.07246*, 2019.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

OpenAI. OpenAI five. https://blog.openai.com/openai-five/, 2018.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3D control from pixels at 100000 fps with asynchronous reinforcement learning. *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.

Tal Ridnik, Hussam Lawen, Asaf Noy, and Itamar Friedman. Tresnet: High performance gpu-dedicated architecture. *arXiv:2003.13630*, 2020.

Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. MINOS: Multimodal indoor simulator for navigation in complex environments. *arXiv:1712.03931*, 2017.

Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2019.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

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

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), 2017.

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

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782), 2019.

Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. DD-PPO: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Fei Xia, Amir R Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson env: Real-world perception for embodied agents. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. Gibson dataset license agreement available at https://storage.googleapis.com/gibson_material/Agreement%20GDS%2006-04-18.pdf.

Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Edmond Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive Gibson benchmark: A benchmark for interactive navigation in cluttered environments. *IEEE Robotics and Automation Letters*, 5(2), 2020.

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. SAPIEN: A simulated part-based interactive environment. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32K for ImageNet training. *arXiv:1708.03888*, 2017.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

A Experiment and Training Additional Details

Complete PointGoal navigation description. We train and evaluate agents via the same procedure as Wijmans et al. (2020). Specifically, agents are trained for `PointGoalNav` (Anderson et al., 2018) where the agent is tasked with navigating to a point specified relative to its initial location. Agents are equipped with a GPS+Compass sensor (providing the agent with its position and orientation relative to the starting position) and either a `Depth` sensor or `RGB` camera. The agent has access to 4 low-level actions, `forward` (0.25m), `turn_left`($10°$), `turn_right`($10°$), and `stop`.

Agents are evaluated on the Gibson dataset (Xia et al., 2018). We use two metrics to evaluate the agents, Success, whether or not the agent called `stop` within 0.2m of the goal, and SPL (Anderson et al., 2018), a measure of both Success and efficiency of the agent's path. During evaluation, the agent does not have access to reward.

Half-precision inference and mixed-precision training. We perform inference in half precision for all components except the action distribution. We train in mixed precision (Jia et al., 2018) utilized the Apex library in O2 mode. We use half precision for all computations expect the action distribution and losses. The optimizer however utilizes single precision for all computations and applies gradients to a single-precision copy of the weights.

Training hyper-parameters Our hyper-parameters for eight-GPU runs are given in Table 3. We additionally employ a gradual learning rate decay where we decay the learning rate from its scaled value back to the base value over the first half of training. We use a cosine schedule.

We find it necessary to set ρ=1.0 for the bias parameters, fixup parameters, and layer-norm parameters of the network, making the optimizer for these parameters equivalent to AdamW (Kingma & Ba, 2015; Loshchilov & Hutter, 2018). We also use L2 weight-decay both to add back regularization lost by removing normalization layers and to stabilize Lamb; we use λ=10^{-2}.

We find one epoch of PPO with two mini-batches to be sufficient (instead of two epochs with two mini-batches), thus effectively doubling the learning speed. We also evaluated one mini-batch, but found two to be beneficial while also having little penalty on overall training speed.

B Benchmarking Additional Details

Pretrained benchmarking. A pretrained DNN is used when benchmarking to avoid frequent environment resets at the start of training.

PPO Parameters	
PPO Epochs	1
PPO Mini-Batches	2
PPO Clip	0.2
Clipped value loss	No
Per mini-batch advantage normalization	No
γ	0.99
GAE-λ (Schulman et al., 2016)	0.95
Learning rate	2.5×10^{-4}
Learning rate scaling	$\sqrt{\frac{B}{B_{\text{base}}}}$
B_{base}	256
Max gradient norm	1.0
Weight decay	0.01
Lamb ρ	0.01
Per GPU parameters	
Number of unique scenes (K)	4
Simulation batch size/Number of Environemnts (N)	128
Rollout length (L)	32

Table 3: Hyper-parameters used for BPS training on 8 GPUs.

Sensor	System		Tesla V100		RTX 2080TI		RTX 3090
			1 GPU	8 GPUs	1 GPU	8 GPUs	1 GPU
Depth	BPS	PPO Epochs			1		
		Rollout length (L)			32		
		Number of Environments (N)	1024	128	512	128	1024
	Baseline++	PPO Epochs			1		
		Rollout length (L)			32		
		Number of Environments (N)			64		
	Baseline	PPO Epochs			2		
		Rollout length (L)			128		
		Number of Environments (N)			4		
RGB	BPS	PPO Epochs			1		
		Rollout length (L)			32		
		Number of Environments (N)	512	128	128	128	256
	Baseline++	PPO Epochs			1		
		Rollout length (L)			32		
		Number of Environments (N)	20	20	6	6	16
	Baseline	PPO Epochs			2		
		Rollout length (L)			128		
		Number of Environments (N)			4		

Table 4: System configuration parameters for Table 1.

Benchmarking hyper-parameters. Table 4 shows the setting for hyper-parameters that impact system throughput.

GPU details We report FPS results on three models of NVIDIA GPU: Tesla V100, GeForce RTX 2080 TI, and GeForce RTX 3090. We demonstrate scaling to multiple GPUs with eight GPU configurations for all but the RTX 3090. Single GPU and eight GPU results are benchmarked on the same machines; however single GPU configurations are limited to 12 cores and 64 GB of RAM as this is a reasonable configuration for a single GPU workstation.

CPU details. Each GPU configuration also uses different CPU configurations based on hardware access. V100 benchmarking was done with 2x Intel Xeon E5-2698 v4 (a DGX-1 station). 2080 Tis benchmarking with 2x Intel Xeon Gold 6226. 3090 benchmarking with Intel i7-5820k. On all CPUs,

we disable Hardware P-State (HWP) (where applicable) and put software P-State in performance mode. Our CPU load on simulation worker cores is inherently sporadic and we find that CPU is unable to change clock frequencies fast enough to not incur a considerable performance penalty if it is allowed to enter a power saving state.