GEAR: A GPU-Centric Experience Replay System for Large Reinforcement Learning ModelsBY

Hanjing Wang [*] [1] [2] Man-Kit Sit [*] [3] Congjie He [3]
Ying Wen [1] Weinan Zhang [1] Jun Wang [4] Yaodong Yang [5] Luo Mai [3]

Abstract

This paper introduces a distributed, GPU-centric experience replay system, GEAR, designed to perform scalable reinforcement learning (RL) with large sequence models (such as transformers). With such models, existing systems such as Reverb face considerable bottlenecks in memory, computation, and communication. GEAR, however, optimizes memory efficiency by enabling the memory resources on GPU servers (including host memory and device memory) to manage trajectory data. Furthermore, it facilitates decentralized GPU devices to expedite various trajectory selection strategies, circumventing computational bottlenecks. GEAR is equipped with GPU kernels capable of collecting trajectories using zero-copy access to host memory, along with remote-directed-memory access over InfiniBand, improving communication efficiency. Cluster experiments have shown that GEAR can achieve performance levels up to 6× greater than Reverb when training state-of-the-art large RL models. GEAR is open-sourced at `https://github.com/bigrl-team/gear`.

1. Introduction

Recent breakthroughs in AI technologies have paved the way for training large sequence models (Vaswani et al., 2017), such as Gato (Reed et al., 2022), DB1 (Wen et al., 2022b), MAT (Wen et al., 2022a) and GPT (Brown et al., 2020), using a reinforcement learning (RL) approach (OpenAI, 2023). These models have set new benchmarks in complex decision-making such as game playing, robotics control, and question answering (Reed et al., 2022; Chen et al., 2021; Wen et al., 2022a; Janner et al., 2021), recommender systems (Geng et al., 2022; Sima et al., 2022) and AI-assisted generated content (Ramesh et al., 2022; Shen et al., 2023).

A pivotal component that makes RL viable for these large sequence models, or ”large RL models,” is the experience replay system. This system archives past experiences, organized as trajectories, for RL agents. These trajectories are gathered by RL policies through on-policy, off-policy, or offline methods (Bellemare et al., 2013; Todorov et al., 2012). For each training batch, the experience replay system chooses trajectories based on a certain strategy, such as first-in-first-out (FIFO), last-in-first-out (LIFO), weighted replay, or prioritized replay. These selected trajectories are then acquired by GPU-based training servers, executing multi-dimensional parallelism for sequence models (Huang et al., 2019; Rasley et al., 2020).

Training large RL models using experience replay systems presents several challenges: (i) Such systems necessitate a substantial number of servers to store hundreds of terabytes of trajectories in memory (Reed et al., 2022). This requirement makes operating these experience replay systems exceedingly expensive. (ii) The complexity of trajectory selection strategies often escalates with the size of the batch (Berner et al., 2019) and the total number of trajectories (Reed et al., 2022), leading to significant computational costs. (iii) Training large RL models invariably involves using large batch sizes, which can result in collecting massive trajectories over the network per training iteration. This process imposes exorbitant communication costs. Given these challenges, it is crucial to consider the storage, computational, and communication costs incurred by an experience reply system when supporting large RL models with experience replay systems.

Existing experience replay systems, unfortunately, fall short in fully addressing the aforementioned challenges. Most of these systems, such as RLlib (Liang et al., 2018), RL-Zoo (Ding et al., 2021), stable-baselines (Hill et al., 2018), rlpyt (Stooke & Abbeel, 2019), tianshou (Weng et al.,

[*]Equal contribution [1]Shanghai Jiao Tong University [2]Digital Brain Lab [3]University of Edinburgh [4]University College London [5]Peking University. Correspondence to: Ying Wen <ying.wen@sjtu.edu.cn>, Weinan Zhang <wnzhang@sjtu.edu.cn>, Luo Mai <luo.mai@ed.ac.uk>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

2022a), TorchOpt-RL (Liu et al., 2022; Ren et al., 2022), sample factory (Petrenko et al., 2020), and envpool (Weng et al., 2022b), are incorporated as part of single-server RL frameworks and fail to offer distributed trajectory storage, selection, and collection. The recent development in distributed experience replay systems, exemplified by Reverb (Cassirer et al., 2021), allows for storing trajectories on memory-optimized servers. Reverb utilizes CPU processors for trajectory selection and collection during training. However, it remains significantly inefficient: to store massive trajectories, Reverb demands the deployment of numerous additional servers, thereby incurring high storage costs. It also struggles with performance issues when executing selection tasks on a large number of trajectories due to the restricted parallelism of CPU processors. Furthermore, Reverb employs RPC libraries (specifically, gRPC) to collect trajectories, leading to excessive memory copies and data serialization. This methodology exhibits low communication efficiency.

In this paper, our objective is to design an experience replay system that can be effectively employed in training large RL models. We have observed that the training servers typically possess vast memory, computation, and communication resources. These servers boast a large amount of host memory (usually ranging from 1 to 4 terabytes) and multiple GPUs (up to 8). These GPUs have high-bandwidth memory and are interconnected with high-bandwidth connectivity including NVlink and InfiniBand. Our primary design strategy, therefore, involves leveraging these training servers to: (i) Store and manage trajectories in their host memory, eliminating the need to use additional servers for trajectory storage. (ii) Speed up trajectory selections with the help of distributed GPUs. (iii) Ensure efficient collection of trajectories using InfiniBand, which provides high bandwidth and zero-copy direct access to remote data.

To realize the above idea, we design and implement GEAR, a novel distributed GPU-centric experience replay system. The design of GEAR makes the following contributions in scaling the training of large RL models:

(1) Trajectory management on training servers. GEAR can divide trajectories into shards and allocate these shards across distributed training servers. The trajectory sharding strategy enhances data locality by factoring in the topology of pipeline parallelism and trajectory priorities during selection. Moreover, GEAR incorporates an optimized trajectory storage format wherein trajectory fields selected together are placed in continuous memory, thereby maximizing data locality and bandwidth utilization.

(2) GPU-optimized distributed trajectory selection. GEAR allows various trajectory selection strategies to benefit from distributed GPUs. It realizes centralized trajectory selection which guarantees deterministic selection results when using distributed GPUs. It further realizes decentralized trajectory selection, thus parallel GPUs can contribute partial trajectory selection results, significantly improving the efficiency of selecting trajectories in extremely large datasets (e.g., those with 100s TB trajectories).

(3) GPU-centric trajectory collection. GEAR has optimized GPU kernels that maximize the communication efficiency in collecting trajectories in GPUs. For trajectories in local host memory, the GPU kernels use zero-copy directed-memory-access (DMA), bypassing CPUs and avoiding data copies and serialization. For trajectories on remote servers, the GPU kernels can launch RDMA send/receive to retrieve the trajectories over InfiniBand.

We evaluated GEAR in a 24-GPU cluster with state-of-the-art large RL models including Gato (Reed et al., 2022) and MAT (Wen et al., 2022a). Experimental results show that GEAR achieves up to better 6x performance (35 GB/s throughput in collecting trajectories) compared to the state-of-the-art Reverb (6 GB/s) with a wide range of configurations (different trajectory sizes, different models and different datasets).

2. Background and Motivation

In this section, we describe the background and motivation for designing GEAR.

2.1. Reinforcement learning for large sequence models

Recent studies have demonstrated the considerable benefits of incorporating RL into large sequence model training. Typically, an RL-based sequence model training system comprises (1) actors, which generate trajectories online through simulation environments such as Atari (Mnih et al., 2013), Mujoco (Todorov et al., 2012), and Google Football (Kurach et al., 2020), and (2) learners, which persistently select a batch of trajectories to train a deep neural network, herein referred to as the model (Liang et al., 2018).

Furthermore, trajectories can also be produced by empirical expert policies offline, allowing large sequence models to mimic or even surpass the performance of these policies. Real-world datasets are another valuable source of trajectories (Janner et al., 2021), such as D4RL (Fu et al., 2020), Minecraft (Fan et al., 2022), and robot manipulation (Jang et al., 2022).

Current large sequence models possess billions or even trillions of parameters. To counter the memory restrictions of a single GPU, developers must employ multi-dimensional parallelism. This involves partitioning and replicating the model, with different model partitions being executed using multi-dimensional parallelism (Zheng et al., 2022): a combination of data parallel (Mai et al., 2020), model parallel (Rajbhandari et al., 2020), and pipeline parallel approaches (Huang et al., 2019).

A pivotal component in large RL model training systems is the experience replay system, which manages massive online and offline produced trajectories. This system enables the model training system to select a batch of trajectories continuously, based on a specific trajectory selection strategy. Following trajectory selection, the training system computes gradients to refine the models. For online RL, the models also return actions to the simulation environments.

2.2. Challenges for experience replay systems

Training large RL models, while promising, presents several significant challenges today. The key challenges include:

(1) High storage costs incurred by trajectory storage. The need for extensive datasets to train large RL models has intensified, incorporating massive offline-generated trajectories by various empirical models and trajectories collected over extended periods from parallel environment simulators. For instance, the dataset used to train the DB1 model (Wen et al., 2022b) encompasses 110 TBs (over 350 billion tokens), mirroring the datasets employed for training the DeepMind Gato (Reed et al., 2022). These trajectories must be retained in server memory for subsequent selection and transport to training servers. However, since commodity servers only offer up to a few TBs of memory, storing the entire dataset necessitates hundreds of servers, leading to substantial memory costs. For example, a memory-optimized server with 1TB of memory in a public cloud can cost several dollars per hour. Thus, using multiple servers to store a dataset of 110 TBs could lead to storage costs amounting to several thousand dollars per hour.

(2) High computation costs associated with trajectory selection. The process of selecting trajectories for the training of large RL models presents considerable computational challenges. Even with the support of auxiliary data structures (Cassirer et al., 2021), such as max heaps and prefix-sum trees, to facilitate the selection process, the need for large batch sizes and the high volume of trajectories often required in training results in a significant number of iterations necessary to complete the selection. For instance, using a prefix-sum tree to conduct priority-based trajectory selection has a time complexity of $O(B \times logN)$, where N is the total number of trajectories and B is the batch size. When B is 1 million and N is 1000 million, the prefix-sum tree still requires more than 10 million iterations to complete trajectory selection. This computational demand can be burdensome for CPUs.

(3) High communication costs due to trajectory collection. Training large RL models often necessitates the collection of an extensive amount of sizeable trajectories within a training batch. A trajectory can vary in size from hundreds of kilobytes to megabytes. This is attributable to (i) the use of large RL models in long-term planning tasks, such as the game of Go (Silver et al., 2016), which often involve trajectories tied to thousands of timestamps (Vinyals et al., 2019; Berner et al., 2019), and (ii) the fact that large RL models may also employ trajectories with multi-modal data (Nair et al., 2022), including text, images, and videos (Fan et al., 2022). The collection of such a vast number of large trajectories places a significant demand on communication bandwidth at the training servers. For instance, gathering one million trajectories, each sized at 100 KBs, necessitates the transfer of more than 100 GBs of data. This rate poses a challenge for current communication technologies (Mai et al., 2015), such as Ethernet and the PCIe bus, which respectively offer 10-40 Gbps and 32 GBs of bandwidth (Koliousis et al., 2019).

2.3. Limitations of existing systems

Existing experience replay systems such as RLlib (Liang et al., 2018), stable-baseline (Hill et al., 2018), rlpyt (Stooke & Abbeel, 2019), tianshou (Weng et al., 2022a), sample factory (Petrenko et al., 2020), and envpool (Weng et al., 2022b) have been predominantly designed for relatively smaller RL models and their implementations are confined to single-server contexts. Consequently, they are not equipped to handle the distributed trajectory storage, selection, and collection necessary for training large RL models.

Recently, DeepMind presented Reverb (Cassirer et al., 2021), a distributed experience replay system and a key component of their Acme research framework (Hoffman et al., 2020). Reverb employs a collection of memory-optimized servers for trajectory storage and relies on CPU processors for trajectory selection. Subsequently, the chosen trajectories are conveyed to training servers via gRPC, which uses network sockets.

However, despite its strengths, Reverb falls short of fully tackling the challenges inherent in training large RL models. To store datasets featuring hundreds of terabytes of trajectories, Reverb necessitates a considerable number of CPU servers, leading to elevated storage costs. Furthermore, the limited parallelism offered by CPU processors translates to lengthy selection times (spanning several seconds), which is starkly contrasting to the time required to complete a batch of GPU training (typically in the order of hundreds of milliseconds) (Shoeybi et al., 2019). In addition, the use of network sockets for trajectory transfer involves multiple data copies, such as moving trajectories from user space to the operating system kernel, and necessitates the serialization and deserialization of trajectory data. This restricts

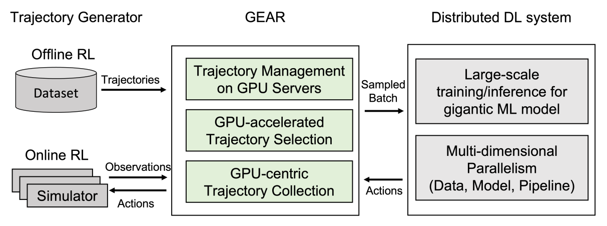

Trajectory Generator
GEAR
Distributed DL system
Offline RL
Dataset
Trajectories
Online RL
Simulator
Observations
Actions
Trajectory Management on GPU Servers
GPU-accelerated Trajectory Selection
GPU-centric Trajectory Collection
Sampled Batch
Actions
Large-scale training/inference for gigantic ML model
Multi-dimensional Parallelism (Data, Model, Pipeline)

Figure 1. Overview of GEAR

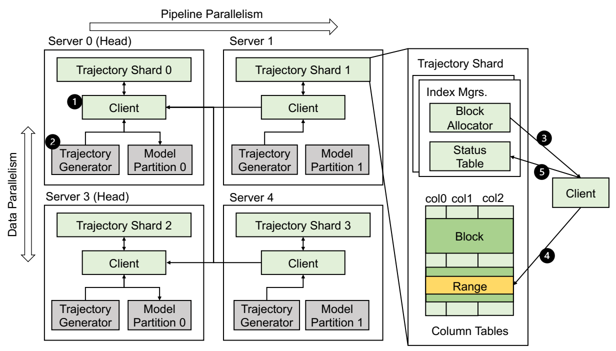


Figure 2. Trajectory management on distributed training servers

Reverb's throughput to a few gigabytes per second, which is an order of magnitude less than the throughput (hundreds of gigabytes per second) required by large RL models.

3. GEAR Design and Implementation

In this section, we delve into the design and implementation of GEAR. Our design approach stems from an observation that modern training servers for large RL models are typically equipped with substantial memory resources (such as terabyte-scale server memory and SSDs), computation accelerators (for instance, 8-16 GPU devices), and high-bandwidth networks (including 40 GB/s InfiniBand and 600 GB/s GPU NVLink).

We utilize these training servers to: (i) leverage the host memory for storing and managing trajectories, thereby eliminating the need for additional storage servers, (ii) employ parallel CUDA kernels to hasten the process of trajectory selection, and (iii) incorporate GPU-centric data communication methods to gather trajectories through PCIe and high-bandwidth networks.

3.1. Overview

Figure 1 offers an overview of GEAR. Within GEAR, trajectory generators both write trajectories to and read trajectories from the system. GEAR is responsible for managing and transporting these trajectories between the trajectory generator and the distributed DL system. Ultimately, GEAR operates in tandem with a distributed DL system that trains and infers large sequence models using multi-dimensional parallelism.

GEAR is designed to support two training scenarios for large sequence models: *offline RL* and *online RL*. (i) In the offline RL scenario, GEAR ingests trajectories from training datasets that have been pre-collected offline. These trajectories are then inserted into GEAR's trajectory storage. The DL system subsequently samples trajectory batches from GEAR in order to update the model parameters. (ii) In the online RL scenario, GEAR employs environment simulators to generate observations. These observations are recorded in GEAR and subsequently transferred to the DL system, which responds with actions. These actions are relayed back to the simulators to continue the simulation process.

3.2. Trajectory management on training servers

Enabling trajectory management on distributed training servers exhibits several unique challenges: (i) The training servers realize pipeline parallelism, and based on their assigned roles in the pipeline, the servers need to be assigned with different shards of trajectories (e.g., only the first server in a training pipeline will need to collect trajectories), (ii) The trajectory sharding scheme must optimize data locality (that is, the majority of the trajectories collected by the servers reside in local memory), and (iii) The trajectory storage format needs to optimize for the unique data access pattern of trajectories (that is, trajectories are collected based on their priorities, a range of timestamp and a sub-group of fields specified by the developers of large RL models).

Pipeline-aware trajectory sharding. Figure 2 gives an overview of the management system design in pipeline parallel training servers. In this figure, we have 4 training servers, where server 0 and 3 are the head servers in pipeline parallelism. The trajectory storage is partitioned into equal-sized shards to allow for storing very large amounts of trajectories. Each shard is hosted by one machine. As each GPU server contains up to several TB of system memory, the shards are stored in the system memory of the GPU servers.

The client (❶) can write to the local shard and read from the remote shards. Trajectory generators (❷) write trajectories to the store through the client, which can be either offline (from datasets) or online (from simulators). The placement controller decides the placement of the shards to improve data locality. The placement controller considers both the type of trajectory generator and parallelism. With pipeline parallelism, only the head servers which run the input layer needs to read the shards. Therefore, for offline generators, we place the data with a higher probability (i.e. higher priority) to be selected in the head machines. For an online generator, the trajectories are written to the local shard where the trajectories are generated to reduce write overheads.

Shard storage format. Clients typically need to read only certain fields of trajectories (such as observation, action, or reward) and write to the storage field by field. To facilitate this access pattern, we chose a column-based storage system, which stores the same type of fields together in a consecutive manner. Each shard is composed of two main components: column tables and index managers. These components are allocated in shared memory, allowing direct access by client processes without the overhead of inter-process communication (IPC).

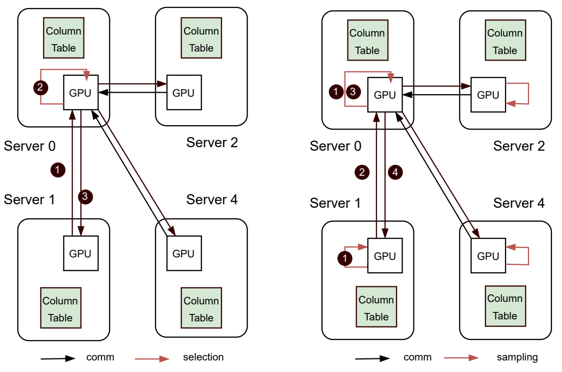
A column table holds a specific field type of a trajectory. Since the shape of the fields is defined by the users at creation, the tables are implemented as continuous arrays divided into blocks of equal size. Each block represents a field of a trajectory and serves as the basic unit of read/write operations. The fields within a block are stored as a sequential chain of flattened tensors. The storage capacity is defined by the number of blocks in a table, and all tables share the same capacity, representing the maximum number of trajectories that a shard can accommodate.

Index manager. The Index Manager comprises a Block Allocator and a Status Table. Each block within a table can be uniquely identified by an index, with a row of blocks (i.e., blocks sharing the same index across different tables) forming a complete trajectory.

The Block Allocator is tasked with block allocation and the subsequent status updates. To manage the indices of free blocks – those that are either empty or contain outdated trajectories awaiting eviction – the allocator employs a single queue. The processes of allocation and release involve dequeue and enqueue operations, respectively. The Status Table, on the other hand, maintains essential statistics such as priorities and timestamps that reflect the usability of indices and their associated trajectories. The priorities of indices that represent ongoing or evicted trajectories are set to zero to avoid their selection. When allocation or eviction events occur, the updates on block status are consolidated and committed to the Status Table.

GEAR utilizes a *multi-controller* architecture, where client programs are duplicated and run to manage exclusive portions of hardware resources, such as GPU devices. For preventing data corruption, GEAR directs clients and index managers to periodically synchronize with each other, ensuring clients have up-to-date information on block allocation.

In line with the principles of a multi-controller architecture,



(a) Centralized selection (b) Decentralized selection

Figure 3. Centralized and decentralized trajectory selection

trajectory shards are further divided and managed by individual local index managers. Each local index manager is tied to a specific client process to minimize local synchronization overhead. This arrangement enables the Index Manager to smoothly integrate with existing deep learning libraries, including Torch-DDP and DeepSpeed.

Trajectory insertion and deletion. To insert a trajectory into the store, the client initiates an `allocate` operation, which generates a buffer containing memory views of the blocks. As demonstrated in Figure 2, this operation involves several steps: ❸ The client requests a free index from the block allocator. ❹ The client retrieves the memory locations of the indexed blocks from each column table. These memory locations are encapsulated into a buffer, which the client then fills with trajectory data. After writing the blocks, the client ❺ `commits` the buffer, triggering an update in the Status Table. This update designates the index as available for selection. All write operations are performed in place to avoid unnecessary memory copying.

Releasing a row requires the block allocator to enqueue the freed index and designate it as unavailable for selection. This occurs when either (1) the block allocator receives a request but no index is available, or (2) the number of selectable indices reaches the preset maximum capacity. In these scenarios, the allocator automatically selects a victim index to be released according to a user-defined removal strategy (e.g., FIFO, LIFO).

3.3. Trajectory selection with distributed GPUs

GEAR facilitates efficient trajectory selection using distributed GPUs on training servers. It addresses key challenges, such as (i) ensuring deterministic selection results for consistent training outcomes of large RL models; (ii) maintaining consensus among all participants on the selected indices to avoid data corruption; and (iii) controlling communication overheads for effective scaling.

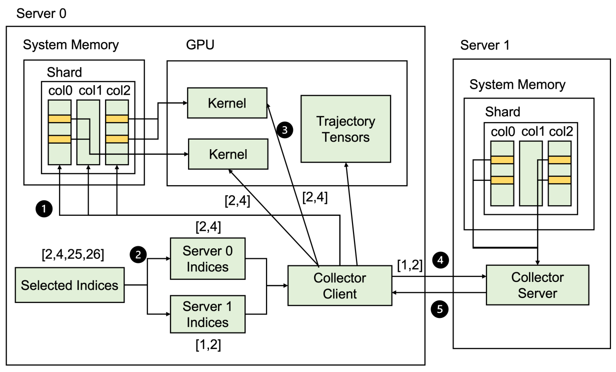


Figure 4. Trajectory collection on distributed GPUs

Centralized trajectory selection. Figure 2(a) illustrates the process of centralized trajectory selection, which consists of the following steps: ❶ The GPU in the central server conducts a global gathering operation to collect selectable indices and weights from all servers. ❷ A selection algorithm then picks from these collected indices and generates a list of selected indices. ❸ The central server broadcasts this list of selected indices to the other servers.

GEAR supports both uniform sampling and weighted sampling as centralized selection algorithms. We've implemented an optimized CUDA kernel for weighted sampling. This kernel first computes a prefix sum array using the decoupled look-back algorithm and then performs binary searching to locate the bins associated with uniformly generated random numbers. The prefix sum calculation and the searching procedure require $\frac{k \times \log N}{s}$ steps, where s is the degree of parallelism, k is the number of samples, and N is the sample size.

Decentralized trajectory selection. GEAR provides FIFO and TopK selection implementation, which are deterministic in decentralized selection scenarios. Therefore, all servers can perform a local scan to generate k samples before the global gathering operation, which can significantly reduce the communication overhead from $O(n)$ to $O(mk)$ when $k \ll n$, where m is the parallel world size of servers.

As depicted by Figure 2(b), the decentralized trajectory selection in GEAR will first run GPU kernels in each GPU to compute the partially selected trajectories (e.g., the top-K priority trajectories) (❶). The partially selected trajectories will be sent to the central GPU (❷) to compute the global top-K selected trajectories. The global top-K selected trajectories will be broadcast to all GPUs and allow these GPUs to retrieve their trajectories (❸).

3.4. GPU-centric trajectory collection

GEAR aims to eliminate data copies and serialization, utilizing high-bandwidth networks for GPUs, thereby reducing latency when collecting trajectories for training servers. However, we identify two key research gaps: (i) Modern GPUs can directly read from shared memory, negating the need for data copying and serialization over CPU-managed memory. This capability is yet to be fully exploited in current machine learning frameworks. (ii) GPU servers often have InfiniBand, providing high bandwidth and efficient hardware-assisted data copy and serialization. However, in existing machine learning frameworks: InfiniBand is primarily used for synchronizing the gradients for RL models, leaving it underutilized in collecting trajectories.

Index translation. As the selected list of indices is global indices, it is translated into lists of local indices for each server. Since the shards have equal size, it is simple to determine if an index belongs to which machine by dividing the index by the capacity. The translated lists of local indices are sent to the collector.

Trajectory collection. GEAR enables GPUs to directly collect trajectory in host memory by facilitating one-sided data accesses. The collector consists of a collector client and a collector server. For local collection, the collector utilizes the zero-copy access feature of NVIDIA GPUs, which allows GPU threads directly access the host memory without the help of the CPU. Compared to the more common DMA-based data transfer, it is more suitable for sparse data accesses (Tan et al., 2023), as it allows GPUs to send more fine-grained memory requests to the system memory directly through PCIe. To enable zero-copy, the memory of the tables is pinned to page-lock the table data (❶ in Figure 4). The collector client launches one CUDA kernel per table to collect trajectories from host memory to GPU memory. For remote trajectory collection, the collector utilizes NCCL to copy trajectories from remote shards through InfiniBand.

Collection example. For example, in Figure 4, a client wants to collect field `"col0"` and `"col1"` of indices 2, 4, 25, 26 from the store, it calls `collect([2, 4, 25, 26], ["col0", "col1"])`. The indices are translated into [2, 4] and [1, 2] which are the local indices of server 0 and server 1 respectively (❷). The collector client launches 2 CUDA kernels to read locally from each of the requested tables (❸). The client sends [1, 2] to the collector server in server 1 to request the trajectories remotely (❹). After the collector server collected the trajectories, it sends them to the collector client (❺). The local and remote collected trajectories are concatenated and converted to Pytorch Tensor.

3.5. Implementation details

The current implementation of GEAR comprises 4,300 lines of C/C++ code and 2,000 lines of Python code. GEAR supports large RL models authored in PyTorch and parallelized by DeepSpeed, a popular distributed training and inference library for sizable transformers. It facilitates the import of offline PyTorch and TensorFlow datasets and accommodates data generated by a diverse assortment of environment simulators, such as Atari, Google Football, Starcraft II, and Robot Arms.

Multi-framework support. GEAR is designed as a framework-independent library, enabling its integration with various machine learning frameworks. Currently, it exposes trajectory data via the PyTorch Tensor interface, facilitating seamless interaction and data exchange between GEAR and PyTorch models.

Moreover, GEAR is incorporated into DeepSpeed to support multi-dimensional parallelism when training large sequence models. Distributed communication and additional data placement optimizations that hinge on pipeline parallelism can be facilitated by DeepSpeed's parallel topology unit.

Failure recovery. GEAR allows for trajectory shards to be checkpointed on local SSDs. Developers frequently trigger checkpointing processes at data epoch boundaries which are aligned with model parameter checkpoint boundaries (Mai et al., 2020). Since GEAR is incorporated into the DeepSpeed framework, it relies on DeepSpeed to detect failures and recover trajectory shards.

4. Experiments

In this section, we outline our experiment settings and results to assess the performance and accuracy of GEAR.

Cluster. We conduct benchmarking tests and RL training tasks on a three-server cluster, each being a standard NVIDIA DGX-A100 server. Each server houses dual AMD Rome 7742 processors, eight NVIDIA A100 GPUs, 1000GB of RAM, and high-bandwidth NVLink and IB connections. This setup enables us to evaluate GEAR's scalability for distributed training and its effectiveness as a seamless substitute for the existing RL codebase without impacting model convergence.

Baseline. We choose Reverb, a widely adopted distributed experience replay system, as our comparison baseline. We conduct extended benchmarks to measure the average sampling throughput and compare GEAR's performance against Reverb. It's worth noting that Reverb relies heavily on TensorFlow, complicating its direct integration with our PyTorch-based benchmark kits. To tackle this, we develop mock clients for both GEAR and Reverb that merely collect and instantly discard data, bypassing the conversion procedure and neutralizing potential benchmarking biases due to different data interfaces.

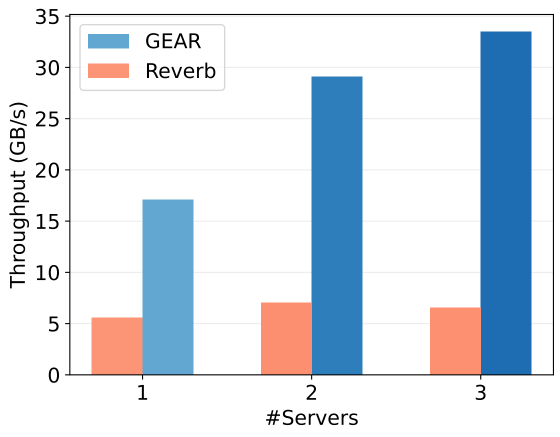


Figure 5. End-to-end throughput comparison with Reverb

4.1. Trajectory throughput

In our study, we compare the trajectory selection throughput between GEAR and Reverb. This involves the implementation of clients that concurrently generate selection requests and gather data from a central server process—a common practice in Single-Program-Multiple-Data (SPMD) parallel programs.

In the Reverb setup, the server process is hosted on server 0. In contrast, for GEAR, global aggregations and selection are managed by GPU 0 on server 0. Additionally, each GPU within GEAR is allocated to a unique client process. All these processes then form a data parallel group, enabling synchronized data retrieval and consumption. The purpose of this experimental setup is to push both GEAR and Reverb to their maximum potential throughput. This offers a rigorous assessment of their performance when tasked with handling heavy workloads.

The results depicted in Figure 5 illustrate that, under single-node settings, Reverb reaches its peak single-node throughput at 6.73 GB/s with 64 parallel clients. Conversely, GEAR achieves a maximum throughput of 17.1 GB/s with only 8 clients involved in globally synchronized sampling loops.

In multi-server experiments, GEAR attains a total throughput of 29.1 GB/s with 16 clients spread across two nodes, and a total throughput of 33.0 GB/s with 24 clients spanning three nodes. This demonstrates that GEAR exceeds Reverb's performance by a factor of five.

We observed that GEAR's total throughput increases more

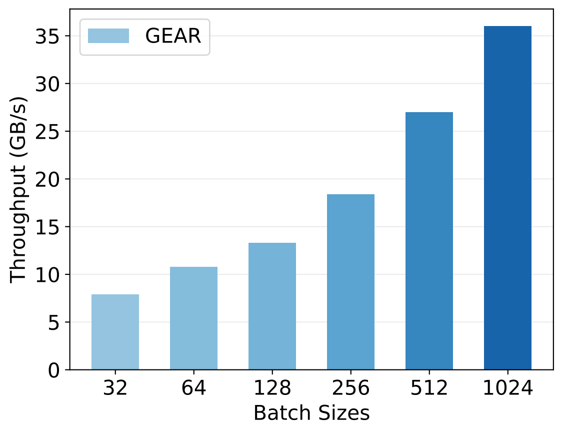


Figure 6. Trajectory collection throughput with varied batch sizes

slowly than the scale, particularly when expanding from two nodes. This is primarily due to the strict global synchronization policy that was adopted to align with Reverb's behavior. This effect could be mitigated by implementing parallel sampling techniques such as reservoir sampling and divide-and-conquer sampling. These methods can be executed prior to global synchronization, reducing communication overhead. However, it's important to note that these techniques could have side effects in distributed scenarios. For instance, they might provide less strict theoretical guarantees or might adversely affect the reproducibility performance of downstream algorithms. Therefore, we leave the decision to the user as to whether or not to implement these techniques.

4.2. Batch size

In our experiment, we investigate the impact of varying batch size (Mai et al., 2019) on both the computation cost and the communication cost of trajectory selection and collection in GEAR. We carried out a set of tests with GEAR using a range of batch sizes: 32, 64, 128, 256, 512, and 1024. The results of this experiment are presented in Figure 6. As observed from the figure, GEAR can achieve a linear scalability in trajectory collection performance with increasing batch sizes: its throughput starts at 8 GB/s using a small batch size 32 and the throughput jumps to 36 GB/s when a large batch size of 1024 is employed. This showcases GEAR's ability to effectively scale its performance with larger batch sizes.

4.3. Model convergence

We evaluate the correctness of trajectory selection of GEAR by showing the convergence results of using GEAR for training two popular large RL models: Gato (Reed et al., 2022) and MAT (Wen et al., 2022a). The former covers the scenarios where trajectories are produced offline and the latter covers the cases where trajectories are produced by environment simulators online.

Gato. We implement the GATO model using PyTorch and reproduce similar loss performance consistent with what they reported in their paper. The GATO model has 1 billion parameters and it is trained with a dataset that has 100 TBs trajectories. We report a minimal GATO experiment and convergence performance with GEAR in the D4RL mujuco hopper task pretrained with the D4RL expert dataset. As we can see from Figure 5, the episode length convergence to $1k$ and episode return converged to 3000 These results show that GEAR has been correctly integrated with existing pipeline parallelism libraries and it can correctly select trajectories that make SOTA RL models to converge.

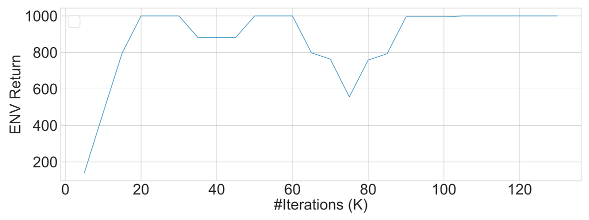
MAT. We also implement the MAT model, the state-of-the-art online multi-agent model that uses transformers as the backbone. The muli-agent RL model actually has a more challenging requirement for trajectory selection. Through this experiment, we show that GEAR enables many multi-agent researchers to work on large RL models. The MAT model can be scalable to larger sizes flexibly. In this case, to illustrate GEAR does not affect model convergence, we use a MAT model of 1 block of hidden size 64, which is aligned We report the MAT performance with GEAR in the StartCraftII tasks, namely $2c\_vs\_64zg$, $3s5z$, $27m\_vs\_30m$, which are correspondingly the "easy", "hard" and "super hard" tasks in SC2 environments. As we can see from Figure 6, GEAR integrates MAT converged to 20 in terms of environment returns, which is the desired convergence result of SC2.

4.4. Performance Breakdown

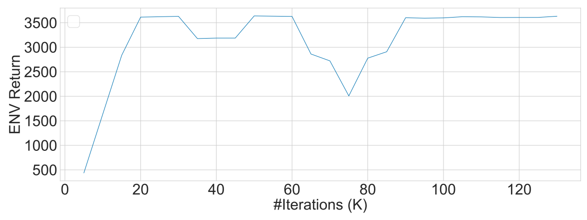
To provide a holistic understanding of GEAR's performance, we carry out evaluations to analyze the individual contributions of each component and to quantify the enhancements they offer:

Computation. We contrast GPU trajectory sampling with its CPU equivalents used in Reverb. Our results indicate that distributed GPU sampling is typically 100-1000 times quicker than the CPU versions.

Memory access. We assess the GPU kernel that enables direct access to trajectories in host memory. This exhibits an impressive throughput of 17 GB/s, significantly outperforming the 2 GB/s achieved when a CPU thread is used to orchestrate data movement, as is the current practice in PyTorch and TensorFlow. This amounts to a performance increase by a factor of 8.5.

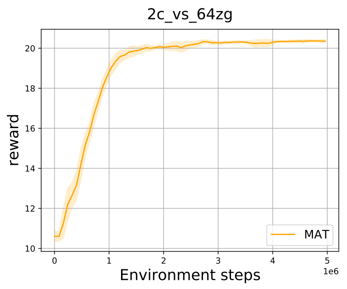


(a) Episode length

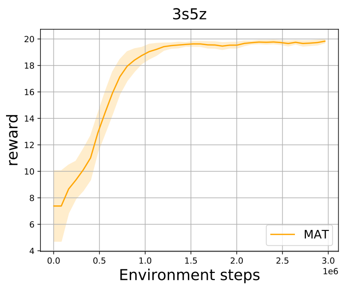


(b) Episode return

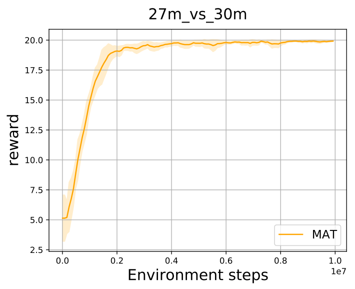
Figure 7. Convergence experiment of GATO with GEAR. The x-axis is the number of iterations and the y-axis is the environment return.



(a) $2c_vs_64zg$



(b) $3s5z$



(c) $27m_vs_30m$

Figure 8. Convergence experiment of MAT with GEAR. The x-axis is the environment steps and the y-axis is the reward.

Communication. With respect to InfiniBand-based distributed trajectory collection, we juxtapose GEAR and Reverb, which employs gRPC for trajectory communication. The evaluation reveals that GEAR exhibits superior scalability when compared to Reverb. When utilizing four machines, GEAR reaches a communication throughput of 35 GB/s, outpacing Reverb, which caps at 5 GB/s.

5. Conclusion

This paper presents GEAR, a distributed GPU-centric experience replay system designed to enhance the efficiency of large RL model training. GEAR explores a novel design approach, employing a GPU-centric architecture for experience replay systems. It stores trajectories on distributed training servers, utilizes distributed GPUs to expedite trajectory selection, and enables GPUs to efficiently gather trajectories through directed memory access technologies. Our experimental evaluations demonstrate that GEAR significantly enhances experience replay performance compared to leading systems: Reverb. We anticipate GEAR playing a pivotal role in facilitating the training of large and complex RL models and await future advancements in this field.

Limitations. However, it's important to acknowledge several existing constraints in our proposed system. Firstly, while GEAR is specifically designed for large RL models, further research is needed to examine its scalability with increasingly large models and datasets. Secondly, our current focus is on the experience replay segment of the RL pipeline. Consequently, additional investigations are required to understand how our system could be seamlessly integrated with other pipeline segments such as model training and evaluation.

Acknowledgements

The authors from Shanghai Jiao Tong University are supported by the National Key R&D Program of China (2022ZD0114804), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102), Shanghai Sailing Program (21YF1421900), and National Natural Science Foundation of China (No. 62106141 & 62076161).

References

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners.

Advances in neural information processing systems, 33: 1877–1901, 2020.

Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T., and Kroiss, M. Reverb: a framework for experience replay. *arXiv preprint arXiv:2102.04736*, 2021.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

Ding, Z., Yu, T., Zhang, H., Huang, Y., Li, G., Guo, Q., Mai, L., and Dong, H. Efficient reinforcement learning development with rlzoo. In *Proceedings of the 29th ACM International Conference on Multimedia*, pp. 3759–3762, 2021.

Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *arXiv preprint arXiv:2206.08853*, 2022.

Fu, J., Kumar, A., Nachum, O., Tucker, G., and Levine, S. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

Geng, S., Liu, S., Fu, Z., Ge, Y., and Zhang, Y. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM Conference on Recommender Systems*, pp. 299–315, 2022.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jang, E., Irpan, A., Khansari, M., Kappler, D., Ebert, F., Lynch, C., Levine, S., and Finn, C. Bc-z: Zero-shot task generalization with robotic imitation learning. In *Conference on Robot Learning*, pp. 991–1002. PMLR, 2022.

Janner, M., Li, Q., and Levine, S. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34: 1273–1286, 2021.

Koliousis, A., Watcharapichat, P., Weidlich, M., Mai, L., Costa, P., and Pietzuch, P. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proceedings of the VLDB Endowment*, 12(11), 2019.

Kurach, K., Raichuk, A., Stańczyk, P., Zajac, M., Bachem, O., Espeholt, L., Riquelme, C., Vincent, D., Michalski, M., Bousquet, O., et al. Google research football: A novel reinforcement learning environment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4501–4510, 2020.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., and Stoica, I. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pp. 3053–3062. PMLR, 2018.

Liu, B., Feng, X., Ren, J., Mai, L., Zhu, R., Zhang, H., Wang, J., and Yang, Y. A theoretical understanding of gradient bias in meta-reinforcement learning. *Advances in Neural Information Processing Systems*, 35:31059–31072, 2022.

Mai, L., Hong, C., and Costa, P. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association, 2015.

Mai, L., Koliousis, A., Li, G., Brabete, A.-O., and Pietzuch, P. Taming hyper-parameters in deep learning systems. *ACM SIGOPS Operating Systems Review*, 53(1):52–58, 2019.

Mai, L., Li, G., Wagenländer, M., Fertakis, K., Brabete, A.-O., and Pietzuch, P. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 937–954, 2020.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Nair, S., Mitchell, E., Chen, K., Savarese, S., Finn, C., et al. Learning language-conditioned robot behavior from offline data and crowd-sourced annotation. In *Conference on Robot Learning*, pp. 1303–1315. PMLR, 2022.

OpenAI. Gpt-4 technical report, 2023.

Petrenko, A., Huang, Z., Kumar, T., Sukhatme, G., and Koltun, V. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pp. 7652–7662. PMLR, 2020.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., and Chen, M. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Reed, S., Zolna, K., Parisotto, E., Colmenarejo, S. G., Novikov, A., Barth-Maron, G., Gimenez, M., Sulsky, Y., Kay, J., Springenberg, J. T., et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Ren, J., Feng, X., Liu, B., Pan, X., Fu, Y., Mai, L., and Yang, Y. Torchopt: An efficient library for differentiable optimization. In *OPT: Optimization for Machine Learning (NeurIPS 2022 Workshop)*, 2022.

Shen, S., Hou, L., Zhou, Y., Du, N., Longpre, S., Wei, J., Chung, H. W., Zoph, B., Fedus, W., Chen, X., Vu, T., Wu, Y., Chen, W., Webson, A., Li, Y., Zhao, V., Yu, H., Keutzer, K., Darrell, T., and Zhou, D. Flan-moe: Scaling instruction-finetuned language models with sparse mixture of experts, 2023.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Sima, C., Fu, Y., Sit, M.-K., Guo, L., Gong, X., Lin, F., Wu, J., Li, Y., Rong, H., Aublin, P.-L., et al. Ekko: A {Large-Scale} deep learning recommender system with {Low-Latency} model update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 821–839, 2022.

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

Tan, Z., Yuan, X., He, C., Sit, M.-K., Li, G., Liu, X., Ai, B., Zeng, K., Pietzuch, P., and Mai, L. Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness. *arXiv preprint arXiv:2305.10863*, 2023.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Wen, M., Kuba, J. G., Lin, R., Zhang, W., Wen, Y., Wang, J., and Yang, Y. Multi-agent reinforcement learning is a sequence modeling problem. *arXiv preprint arXiv:2205.14953*, 2022a.

Wen, Y., Wan, Z., Zhou, M., Hou, S., Cao, Z., Le, C., Chen, J., Tian, Z., Zhang, W., and Wang, J. On realization of intelligent decision-making in the real world: A foundation decision model perspective. *arXiv preprint arXiv:2212.12669*, 2022b.

Weng, J., Chen, H., Yan, D., You, K., Duburcq, A., Zhang, M., Su, Y., Su, H., and Zhu, J. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022a. URL `http://jmlr.org/papers/v23/21-1127.html`.

Weng, J., Lin, M., Huang, S., Liu, B., Makoviichuk, D., Makoviychuk, V., Liu, Z., Song, Y., Luo, T., Jiang, Y., et al. Envpool: A highly parallel reinforcement learning environment execution engine. *arXiv preprint arXiv:2206.10558*, 2022b.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.