ReaL: Efficient RLHF Training of Large Language Models with Parameter Reallocation

Zhiyu Mei[*1 2] Wei Fu[*1 2] Kaiwei Li[3] Guangju Wang[4] Huanchen Zhang[1 2] Yi Wu[1 2 4]

Abstract

Reinforcement Learning from Human Feedback (RLHF) is a pivotal technique for empowering large language model (LLM) applications. Compared with the supervised training process of LLMs, the RLHF training process is much more sophisticated, requiring a diverse range of computation workloads with intricate dependencies between multiple LLM instances. Therefore, simply adopting the fixed parallelization strategies from supervised training for LLMs can be insufficient for RLHF and result in low training efficiency. To overcome this limitation, we propose a novel technique named *parameter* ReaL*location*, which dynamically adapts the parallelization strategies for different workloads during training by redistributing LLM parameters across the training cluster. Building upon this idea, we introduce ReaL, a pioneering system for efficient RLHF training. ReaL introduces the concept of an *execution plan*, which defines a fine-grained resource allocation and parallelization strategy particularly designed for RLHF training. Based on this concept, ReaL employs a tailored search algorithm with a lightweight run-time estimator to automatically discover an efficient execution plan for an instance of RLHF experiment. Subsequently, the runtime engine deploys the selected plan by effectively parallelizing computations and redistributing parameters. We evaluate ReaL on the LLaMA models with up to 70 billion parameters and 128 GPUs. The experimental results demonstrate that ReaL achieves speedups of up to 3.58× compared to baseline methods. Furthermore, the execution plans generated by ReaL exhibit an average of 81% performance improvement over heuristic approaches based on Megatron-LM in the long-context scenario. The source code of ReaL is publicly available at `https://github.com/openpsi-project/ReaLHF`.

1 Introduction

Large Language Models (LLMs) such as ChatGPT (OpenAI, 2022) have amazed the world with their powerful capabilities. Their success relies on the enormous model sizes, e.g., GPT-3 (Brown et al., 2020) has 175 billion parameters. Because each graphic processing unit (GPU) has limited memory, to train such an expansive model, the computation along with the model parameters must be distributed across a vast GPU cluster. Recent literature has proposed a wide range of parallelization strategies (Huang et al., 2019; Shoeybi et al., 2019; Rajbhandari et al., 2020; Narayanan et al., 2021; Jiang et al., 2024) specifically designed for the supervised training paradigms, such as pretraining and supervised fine-tuning (Dong et al., 2023; Zhang et al., 2024). Meanwhile, another remarkable training paradigm for LLMs, known as Reinforcement Learning from Human Feedback (RLHF), is the foundation technique for the success of ChatGPT-like models (Ziegler et al., 2019; Stiennon et al., 2020; Ouyang et al., 2022; Anil et al., 2023; Antropic, 2023; Touvron et al., 2023; Bai et al., 2022; OpenAI, 2024). The workflow of RLHF training is much more complicated than supervised training. However, most existing RLHF systems adopt parallelization techniques directly from supervised training (Yao et al., 2023b; Hu et al., 2024; Shen et al., 2024), which could lead to sub-optimal training efficiency.

The typical workflow of RLHF, which is often based on Proximal Policy Optimization (PPO) (Schulman et al., 2017) algorithm, involves three distinct types of computational tasks on four LLMs with independent parameters. In each RLHF training iteration, a primary LLM (the training target, referred to as the *Actor* model) receives prompts and generates responses (i.e., the generation tasks). These responses are evaluated by three additional LLMs: the *Reward* model, the *Reference* model, and the *Critic* model (i.e., the inference tasks). Then, *Actor* and *Critic* use the evaluation results to compute gradients and update their parameters (i.e., the training tasks).

To identify the drawbacks of the existing RLHF systems,

[*] Equal contribution [1] Institute for Interdisciplinary Information Science, Tsinghua University, Beijing, China [2] Shanghai Qi Zhi Institute, Shanghai, China [3] Independent Researcher [4] OpenPsi Inc.. Correspondence to: Zhiyu Mei <meizy20@mails.tsinghua.edu.cn>, Wei Fu <fuwth17@gmail.com>, Yi Wu <jxwuyi@gmail.com>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

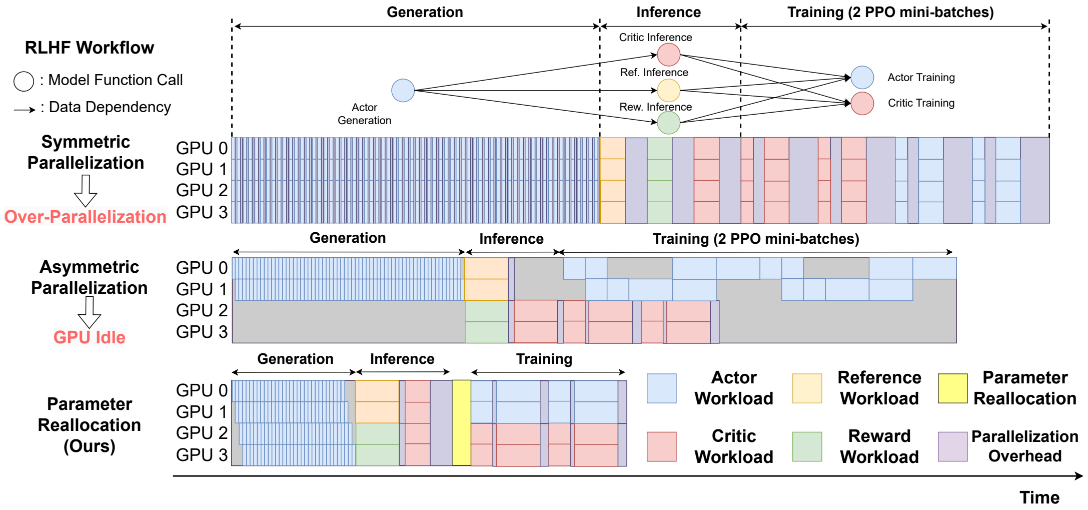


Figure 1: An RLHF iteration breakdown based on the profiling of real systems (Table 6). The directed acyclic graph shows the RLHF workload. Nodes represent model function calls and edges represents their data dependencies. We present timelines to visualize execution plans that employ: [top] the same parallelization strategy that spreads across the entire cluster for all LLMs, [middle] independent resource allocations and parallelization strategies for each LLM, and [bottom] distinct resource allocations and parallelization strategies for each *model function call* generated by ReaL. The plan of ReaL considers parameter reallocation for the actor and critic model.

we conduct a thorough profile and discover two major limitations. First, we note that many systems apply the same parallelization strategy that spreads across the entire GPU cluster for all LLMs. We name this a *symmetric parallelization* strategy, which often leads to *over-parallelization*. Our system profiling in Figure 1 (top) shows that over-parallelization leads to substantial synchronization and communication overheads (the light purple bars), thus compromising the end-to-end system performance. Moreover, different computational tasks are better off with different parallelization strategies (Lei et al., 2024). A single global parallelization strategy, therefore, is likely to be sub-optimal. Accordingly, some other systems choose to allocate different LLMs to different sets of GPUs with different parallelization strategies. In this way, tasks from different LLMs could be executed concurrently. We call this an *asymmetric parallelization* strategy. However, our second observation is that such a strategy often causes under-utilization of the GPUs (e.g., the gray areas in Figure 1 (middle)) because of the dependencies between tasks.

The crux of the above inefficiencies is that resource allocations and parallelization strategies for LLMs are fixed throughout training. Therefore, we propose to enable *dynamic reallocation of model parameters* between GPUs, allowing fine-grained resource allocations and parallel strategies at the task level to improve the efficiency of the entire RLHF training process. For clarity, we refer to an individual task on an LLM as a *model function call*. As shown in Figure 1 (bottom), by first choosing a parallelization strategy tailored for each model function call (e.g., Actor generation and training) and then executing these calls concurrently with a smaller parallelization degree (e.g., Actor and Critic training), we can reduce the communication overhead caused by parallelization while maximizing GPU utilization. Parameter reallocation effectively addresses the limitations of prior solutions and can lead to a significant end-to-end throughput improvement, as we show in Figure 2.

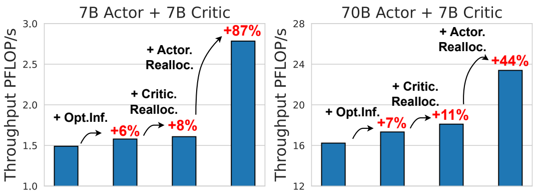


Figure 2: The optimization opportunity over a 3D parallelism execution plan inspired by pre-training. We show the sequential improvement by optimizing inference parallelization strategy, reallocating critic workloads (inference/training), and reallocating actor workloads (generation/training).

Based on the key idea of parameter reallocation, we developed ReaL, a pioneering system for efficient RLHF training. ReaL consists of two components, i.e., an execution plan generator and a runtime engine. An execution plan specifies the resource allocations and parallelization strategies for every model function call in the RLHF training workflow under a specific algorithmic and hardware configuration. The execution plan generator performs Markov Chain Monte Carlo (MCMC) sampling to search for efficient execution plans using an extremely lightweight profiling-assisted runtime estimator. After a sufficiently good execution plan is obtained, the runtime engine deploys the derived plan by effective parallelization and parameter redistribution.

Our experimental evaluation entails RLHF training on LLaMA models (Touvron et al., 2023; Dubey et al., 2024) ranging from 7 to 70 billion parameters across 8 to 128 Nvidia H100 GPUs. Results showcase that ReaL is able to achieve a speedup up to 3.58 times over the baseline systems. Furthermore, we demonstrate that the performance of ReaL's searched execution plans surpasses heuristic plans based on Megatron by 54% on average and up to 81% with a longer context length.

In summary, our contributions are as follows:

- We propose to reallocate model parameters dynamically for efficient RLHF training.
- We introduce execution plans at the model function call level and propose an efficient search algorithm to identify fast plans.
- We design and implement ReaL, an RLHF training system that can automatically discover and run a fast execution plan with a high training throughput.
- We conduct comprehensive evaluations with detailed breakdowns and ablation studies. ReaL achieves up to 3.58× higher throughput compared to the baselines.

2 Background

2.1 Introduction to RLHF

For the ease of illustration, this section adheres to the common practice of RLHF, focusing on GPT-like LLMs (Radford et al., 2019; Brown et al., 2020) and the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017). However, we remark that ReaL can also support other RLHF algorithms, as we will discuss in Section 4.

An RLHF training iteration involves six model function calls on four LLMs: Actor generation, Reward inference, Critic inference, Reference inference, Actor training, and Critic training. Their dependencies are shown in Figure 1 (top). In these model function calls, *Generation* is composed of multiple forward passes. It involves a prefill phase and a decoding phase. The prefill phase is a single forward pass, which consumes all prompt tokens to sample the first generated token. The decoding phase repeatedly inputs the (single) latest generated token and produces the subsequent token until termination. *Inference* is a forward pass over the combination of prompts and generated responses. *Training* is an ordinary supervised training iteration, composed of a forward pass, a backward pass, and a parameter update. The next RLHF iteration then applies the updated Actor and Critic for generation and inference.

Notably, training the Actor and Critic with PPO can incorporate multiple minibatches (Ouyang et al., 2022). For each minibatch, the parameter update must occur before the subsequent forward pass, distinguishing this approach from gradient accumulation that performs a single parameter update across minibatches.

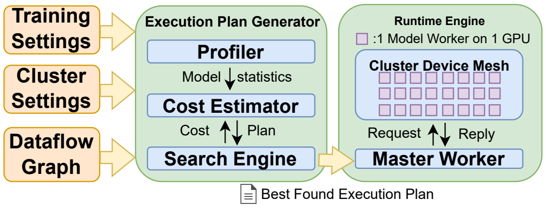


Figure 3: An overview of the architecture of ReaL.

2.2 Parallelization of Large Language Models

Classical parallelization approaches for LLMs encompass data, tensor-model, and pipeline-model parallelism.

Data Parallelism (DP) partitions data along the batch dimension and dispatches each partition to a model replicate for independent computations. After the backward pass during training, all DP peers should perform an all-reduce over gradients before applying them for parameter update.

Tensor-model Parallelism (TP) partitions model parameters and distributes matrix multiplications across multiple GPUs. Each TP rank processes the same data and produces a partial intermediate value. Then, all TP peers perform an all-reduce over this value to obtain the full result and pass it to the next layer. Since all TP peers should perform the all-reduce operation in each layer of the LLM, TP leads to substantial data communication overhead when scaling to more GPUs and deeper models.

Pipeline-model Parallelism (PP) clusters adjacent layers into several *pipeline stages*. PP peers transfer intermediate results among stages for a complete forward or backward pass, which entails less communication overhead than TP. To improve the efficiency of PP, a common approach is to divide the data into micro-batches, allowing different GPUs to process different micro-batches simultaneously.

Since the above parallelization approaches are mutually independent, Megatron-LM (Narayanan et al., 2021) integrates them as *3D Paralleism* to perform LLM supervised training at scale. A *parallelization strategy* S is denoted by three integer values (dp, tp, pp), representing the degrees of DP, TP, and PP, respectively. Each coordinate in this grid represents a process running on an independent GPU. 3D parallelism entails near-optimal parallelization for GPT-like language models, which has been extensively experimented in previous studies (Zheng et al., 2022).

3 Overview

ReaL is a system capable of automatically planning and executing RLHF training workflows given algorithm and cluster specifications. The key idea behind the design of ReaL is *parameter reallocation*: dynamically reallocating model parameters across GPUs and assigning different GPU resources with a suitable parallelization strategy to each model function call. Parameter reallocation enables ReaL

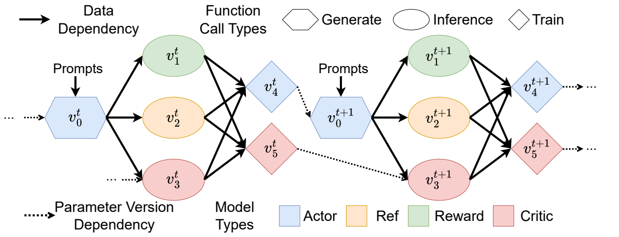


Figure 4: The dataflow graph of two consecutive RLHF iterations. Each ***model*** is an independent LLM. Each ***model function call*** is computational task of the model.

to carry out the fine-grained orchestration of model function calls. Specifically, REAL can allocate distinct groups of resources to different model function calls, allowing them to execute concurrently on different sets of GPUs. REAL also chooses a tailored parallelization strategy for each model function call. In this way, REAL reduces communication overhead and improves GPU utilization without exceeding the memory limitation of the devices. By exploiting parameter reallocation, REAL adopts a comprehensive design that addresses various training scenarios, seizing more optimization opportunities throughout the RLHF training workflow compared to existing systems (Yao et al., 2023b; Hu et al., 2024; Shen et al., 2024).

We summarize the steps of running REAL as follows. First, REAL parses the RLHF workflow into a dataflow graph at the granularity of model function calls. Then, REAL adopts an efficient search algorithm to produce a fast execution plan that includes parallelization strategies and intermediate data/parameter communications. Finally, REAL runs this plan with an efficient worker-based runtime engine.

As demonstrated in Figure 3, there are two major components in the system, the ***Execution Plan Generator*** and the ***Runtime Engine***. The search engine in the execution plan generator continuously searches for execution plans with the Markov Chain Monte Carlo (MCMC) algorithm. A lightweight estimator calculates the approximate time cost of the proposed plan by exploiting execution statistics of profiling. After reaching the search time limit, the fastest execution plan obtained is presented to the runtime engine for deployment.

The runtime engine is composed of a centralized master worker and multiple model workers. The master worker resolves task dependencies and sends requests to the corresponding model workers for task execution. Model workers act as RPC servers and respond to the master worker to update dependencies for subsequent requests. The interaction between the master worker and model workers repeats until the execution plan finishes.

4 PROBLEM FORMULATION

REAL is designed to accelerate RLHF workflows with GPT-like LLMs. To achieve this goal, we introduce the concept of execution plans at the model function call level. We formulate the problem as: taking training configurations (e.g., model size and batch size) and cluster specifications as inputs, search for an optimized execution plan that is able to be executed on the given distributed cluster. In this section, we introduce our detailed terminology definitions in our formulation of the execution plan search problem.

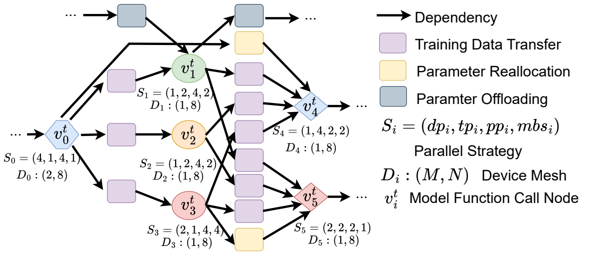


Figure 5: An augmented dataflow graph $\mathcal{G}_p$ of an execution plan instance p in the t-th RLHF iteration.

Dataflow Graph. REAL considers the workflow of RLHF training as a dataflow graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, as demonstrated in Figure 4. A node $v_i^t \in \mathcal{V}$ represents the i-th model function call at the t-th training iteration. An edge $(v, v') \in \mathcal{E}$ indicates a data or parameter version dependency. We emphasize that $\mathcal{G}$ represents the concatenated graph of all the iterations throughout the entire training process. By operating on $\mathcal{G}$, we can potentially overlap computations with no mutual dependencies across training iterations.

Device Mesh. A device mesh D is defined as a two-dimensional grid of GPUs. The shape of D is denoted as (N, M) if it covers N nodes equipped with M devices. Note that device meshes with the same shape could have different locations. We assume all devices in the cluster have the same computing capability with identical intra-node bandwidths, and inter-node bandwidths.

Execution Plan. An execution plan p of a dataflow graph $\mathcal{G}$ assigns a device mesh D_i and parallelization strategy S_i for the i-th *individual function call* in $\mathcal{G}$. We express an execution plan p in the form of an ***augmented dataflow graph*** $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$, as visualized in Figure 5. It also involves data transfer, parameter redistribution, and offloading (Ren et al., 2021) across function calls. They are represented as a set of extra nodes in $\mathcal{G}_p$ (rounded squares in Figure 5).

Search Space. We make several assumptions to make the generation and deployment of execution plans practically feasible. First, we assume that D_i either covers several entire hosts or a consecutive portion that is capable of dividing the number of devices on one host, e.g., (1, 1), (1, 2), (1, 4), (1, 8), (2, 8), $\cdots$, $(N, 8)$ in a cluster of $(N, 8)$. This ensures that multiple device meshes can fully cover the entire cluster, eliminating sub-optimal execution plans with idle GPUs (Zheng et al., 2022). Second, S_i considers the 3D parallelism degrees (dp_i, tp_i, pp_i) and the number of micro batches mbs_i. Data will be divided into mbs_i portions and passed to the function call sequentially. This feature provides an option to avoid the out-of-memory issue with a large batch size and context length.

Beyond PPO. The example shown in Figure 4 and Figure 5 represents the typical RLHF algorithm, PPO. Meanwhile, we emphasize that our formulation is inherently expressive for training algorithms whose workflow could be decomposed into the function calls and represented as a dataflow graph. Experiments in Section 8.3 demonstrate the capability of ReaL to accelerate other prevalent RLHF algorithms including DPO (Rafailov et al., 2023), ReMax (Li et al., 2024) and GRPO (Shao et al., 2024).

5 Execution Plan Generator

The execution plan generator takes the dataflow graph, the training configurations, and the cluster specifications as inputs to automatically search for a rapid execution plan in the form of an augmented dataflow graph. This generator comprises two primary components. First, a lightweight runtime estimator predicts the time and memory cost of any execution plan, leveraging statistical results from profiling. Second, a search engine refines the proposed execution plan using a Markov Chain Monte Carlo (MCMC) search algorithm based on the preceding cost estimation.

5.1 Estimation

The architecture of LLMs is typically a stack of identical layers, exhibiting clear computation patterns. Hence, we can profile the time cost of operations on individual layers and estimate the total cost of each model function call through arithmetic operations. We present a lightweight runtime estimator assisted by profiling. Profiling the statistics in a single experiment takes only minutes, while evaluating the cost for a candidate execution plan requires only hundreds of microseconds, as opposed to several minutes for profiling a single plan on a real run. In the subsequent paragraphs, we denote the estimated values of the time cost and the runtime memory of an execution plan as *TimeCost*$(\mathcal{G}_p)$ and *MaxMem*$(\mathcal{G}_p)$.

Time Cost. We first estimate the time cost for each node $v \in \mathcal{V}_p$. For model function call nodes, ReaL profiles the cost of forward, backward, and associated communication (e.g., all-reduce) of individual layers across a set of data input sizes. The range of this set is decided by the configured batch size, the number of devices in the cluster, and the minimum batch size on each device according to parallelization strategies. We only profile sizes that are powers of two in this range. If the data input size for v falls outside the profiling set, ReaL estimates the time cost using a linear interpolation of the existing profiling statistics. We estimate the costs of data and parameter transfer by running a simulation to the algorithm outlined in Section 6. We approximate the time with the data size and the bandwidth instead of running a real NCCL operation.

Next, we derive *TimeCost*$(\mathcal{G}_p)$ from the cost of each node. The calculation can be much more complex than simple summation because different nodes can be executed concurrently on disjoint device meshes. We employ an algorithm to find the shortest path from source nodes to sink nodes in $\mathcal{G}_p$, with the constraint that nodes assigned to overlapped device meshes cannot execute simultaneously. The algorithm assigns each node $v \in \mathcal{G}_p$ with attributes *StartTime*, *EndTime*, and *ReadyTime*. Each device mesh D tracks the last completed node from all devices within D as $D.last$. The algorithm maintains a priority queue containing all nodes that have been ready for execution but not yet completed. The priority queue iteratively selects the node with the minimum ready time, marks it as completed, updates $D.last$ for all D, and adds new ready nodes to the queue. When the priority queue becomes empty, all nodes in $\mathcal{G}_p$ should be completed, and the maximal *EndTime* of all nodes yields the final result of *TimeCost*$(\mathcal{G}_p)$. The details of the simulation algorithm is shown by Algorithm 1 in Appendix C.

Maximum Memory Allocated. An execution plan p is executable only if its maximum runtime memory does not exceed device limitations. We categorize the runtime memory into the *static memory* and the *active memory*. The static memory consists of the gradients and optimizer states, which will not be freed or transferred until the entire experiment finishes. The active memory is only stored in GPU when it is required, including the KV cache, intermediate activations, and reallocable parameters, etc. We first calculate the static memory and the peak active memory allocated for each function call according to their parallelization strategies. Afterwards, we calculate the peak memory during an RLHF iteration for each device and take the maximum to obtain *MaxMem*$(\mathcal{G}_p)$.

5.2 Execution Plan Search

An execution plan p assigns a device mesh D_i and a parallelization strategy S_i for the i-th model function call. The number of choices grows exponentially with the number of devices in the cluster. For instance, in a cluster of shape $(8, 8)$, there are over 500 options for each model function call, and over 10^{16} execution plans in total, rendering brute-force enumeration practically infeasible. Therefore, ReaL employs an efficient MCMC-based search algorithm tailored for this problem setting.

We associate each execution plan with a cost defined by

$$\begin{aligned} \textit{cost}(\mathcal{G}_p) = I\left(MaxMem(\mathcal{G}p) < mem_d\right) \cdot TimeCost(\mathcal{G}_p) \\ + \left(1 - I\left(MaxMem(\mathcal{G}p < mem_d\right)\right) \cdot \alpha \cdot TimeCost(\mathcal{G}_p), \end{aligned}$$

where mem_d is the device memory capacity, I is an OOM indicator, and α is a large integer representing the OOM penalty. We then define an energy-based distribution $P(p) \propto \exp(-\beta \cdot cost(\mathcal{G}_p))$, where β is the sampling temperature. Lower-cost execution plans have higher probabilities of being sampled from P. Hence, the searching process for a fast execution plan becomes drawing samples from the target distribution P, where MCMC techniques come into play.

We employ the Metropolis-Hastings algorithm (Metropolis et al., 1953; Hastings, 1970) for drawing samples from P. The sampling process begins with a greedy solution p_0 minimizing the summation of time costs of all function calls. Notably, this execution plan can be sub-optimal due to the excessive memory allocation on devices and the lack of overlap between different model function calls. Subsequently, we construct a Markov Chain comprising execution plans $p_0, p_1, \cdots$. We alter D_i and S_i of a random function call i and accept this transition with probability

$$P_{acc}(p_n \to p_{n+1}) = \min\left(1, \frac{P(p_{n+1})}{P(p_n)}\right),$$

This process repeats until a terminating condition, such as when a constant time limitation is met. Finally, the execution plan with the minimum $TimeCost(\mathcal{G}_p)$ throughout the entire searching process is selected as the output of the execution plan generator.

6 RUNTIME ENGINE

In this section, we introduce the runtime engine, including the implementation details of workers, redistributing parameters, and transferring data among function calls.

Workers. The master worker resides on a CPU and executes several `asyncio` coroutines to manage the each function call. The coroutine awaits the completion of all the parent function calls and dispatches requests via sockets upon the function call is ready. These messages do not transfer the associated data. Instead, the data is retained locally in the GPUs of model workers. The master worker communicates the data locations to the model workers in requests to initiate data transfers. Each model worker acts as an RPC server on a GPU. It polls requests from the socket for each local LLM handle (e.g., Actor and Reward) in a round-robin manner. Received requests are put in a FIFO queue for sequential execution and responding.

Redistributing Parameters encompasses host-device (e.g., offload) and device-device communications. Host-device communication utilizes an additional CUDA stream for asynchronous memory copying. Device-device communication involves mapping one 3D parallelization strategy to another, e.g., from (dp_1, tp_1, pp_1) to (dp_2, tp_2, pp_2). We regard the remapping as a hierarchical process consisting of an outer loop (Figure 6 left) and an inner loop (Figure 6 right). Initially, we focus on remapping pipeline stages from pp_1 to pp_2. Each stage $i \in [pp_1]$ holds a group of layers distributed in a device mesh specified by (dp_1, tp_1). For each stage pair (i, j), where $i \in [pp_1]$ and $j \in [pp_2]$, we transfer the parameters of common layers between device meshes specified by (dp_1, tp_1) and (dp_2, tp_2). We denote the devices in (dp_1, tp_1) as source GPUs and (dp_2, tp_2) as destination GPUs. For each destination GPU, we greedily assign a source GPU with the lowest communication cost (e.g., a local GPU has a lower cost than remote GPUs). Once assigned, the source GPUs broadcast parameters to the destinations in parallel. This process iterates until all stage pairs (i, j) are covered.

Data Transfer Among Function Calls. Model function calls produce disjoint data partitions along the DP dimension, while replicating the data along the TP dimension. This mirrors the communication pattern of redistributing parameters in the right part of Figure 6, but with reversed TP-DP dimensions. Therefore, we employ the same broadcast-based algorithm for data transfer.

Remark: Zhuang et al. (2022) explored a similar problem to data transfer in REAL. In our paper, we do not focus on developing an optimal communication algorithm in such scenarios, as long as the cost is minor compared to other workloads in RLHF, as we will show in Figure 11.

7 DISCUSSIONS

This section discusses the advantages and limitations of REAL and clarifies the contexts where REAL can be applied. REAL is a system that is applicable on accelerating RLHF workflows composed of training, inference, or generation function calls with GPT-like LLMs. Apart from its superb performance, REAL has following advantages (■) and limitations (◇):

■ REAL's method is orthogonal to advanced techniques for accelerating individual function calls (e.g., Paged-attention (Kwon et al., 2023)) or fusing different function calls (e.g., RLHFuse (Zhong et al., 2024b)). These techniques can be integrated for better performance.

■ REAL can generalize beyond the workflow for PPO. It can also significantly accelerate various other prevalent RLHF algorithms, such as DPO (Rafailov et al., 2023).

◇ REAL requires predictable function calls to ensure the validity of cost estimation. An unstable cluster or dynamic workflow (e.g., the generation length varies significantly during training) can violate this assumption.

◇ The searching of REAL does not guarantee optimality despite producing plans that are fast and efficient in practice.

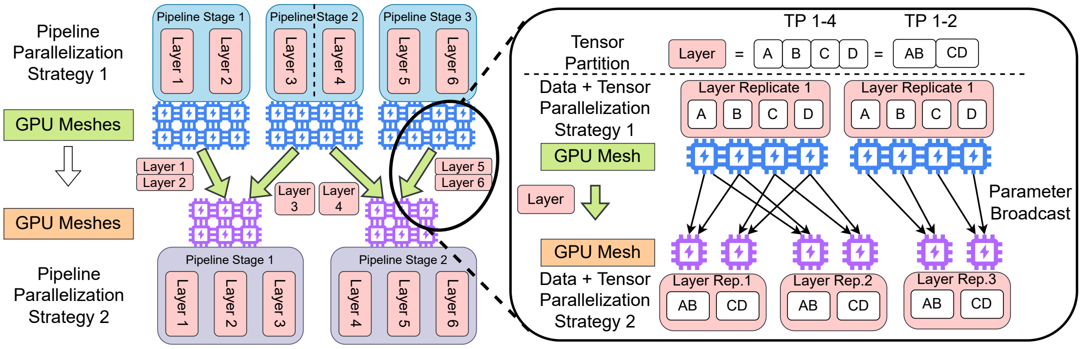


Figure 6: The parameter redistribution is a hierarchical procedure. In the outer loop (left), each pair of pipeline stages communicates the parameters of their common layers. These parameters are distributedly stored in a DP plus TP device mesh. In the inner loop (right), layers are remapped from one DP plus TP mesh to another. Each destination GPU is assigned with a source that has the lowest communication cost. All assigned sources broadcast TP partitions required by destination GPUs in parallel.

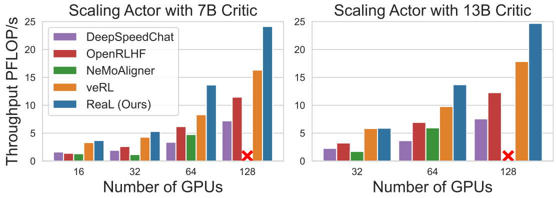


Figure 7: An end-to-end throughput comparison with baseline systems. Red cross denotes instability and OOM errors caused by scalability issues.

8 EXPERIMENTS

REAL is implemented in Python (41k LoC) and C++ (2.5k LoC). The search engine and simulator components are written in C++, while the remaining modules leverage PyTorch (Paszke et al., 2019). Our experiments are conducted on a cluster of 128 H100 GPUs, interconnected via NVLink for the intra-node communication and RoCE with a 3.2Tbps bandwidth for the inter-node communication. We adopt the most advanced LLaMA-3 models (Dubey et al., 2024) for our experiments. Since the vocabulary size of LLaMA-3 is large (128k), resulting a 250GB memory usage during computing softmax[1], we are only able to train a 70B actor with a 13B critic model under the resource constraint.

Our evaluation comprises five key components. First, we benchmark REAL's end-to-end performance against two open-source RLHF systems and a heuristic baseline. Second, we present a detailed performance breakdown to identify key improvements. Third, we conduct an ablation study of the execution plan generator. Fourth, we demonstrate REAL's compatibility with and acceleration of various RLHF algorithms beyond PPO. Finally, we analyze REAL's strong scaling characteristics and provide suggestions for the practical usage.

8.1 Comparison with Baselines

Baselines. We evaluate REAL against four prominent open-source systems: DeepSpeed-Chat (Yao et al., 2023b) (commit f73a6ed with DeepSpeed v0.15.1 as backend), veRL (Hybrid Flow) (Sheng et al., 2024) (v0.2.0.post2 with vLLM v0.6.3 and FSDP on pytorch v2.4.0), NeMo-Aligner (Shen et al., 2024) (v0.4.0 with TRT-LLM v0.10.0 and Megatron v0.8.0) and OpenRLHF (Hu et al., 2024) (v0.4.2 with vLLM v0.4.2 and DeepSpeed v0.15.0). Addtional details of the baseline systems are listed in Appendix D.

DeepSpeed-Chat employs a symmetric parallelization strategy using the ZeRO-3 data parallelism (Rajbhandari et al., 2020) across all RLHF models. Its *Hybrid Engine* temporarily redistributes ZeRO-3 partitions to TP during the generation task, reverting afterward. Beyond this mechanism, DeepSpeed-Chat does not support TP or PP implementations.

OpenRLHF implements an asymmetric parallelization strategy, dividing GPUs into three GPU groups. The groups hold the actor/reference model, the critic/reward model, and a generation engine using vLLM (Kwon et al., 2023). The vLLM engine is only responsible for the actor generation. It remains idle during the actor training, awaiting parameter updates before proceeding to the next RLHF iteration.

Similarly, NeMoAligner divides GPUs into two disjoint GPU groups. Unlike OpenRLHF, it locates actor training and generation on the same GPU group. veRL (HybridFlow) is a concurrent work to REAL that supports colocating models on GPUs and split placement of models on different GPU groups, including the strategies adopted by three previous systems.

Additionally, we evaluate REAL-Heuristic, a pre-training-inspired approach (Narayanan et al., 2021) that implements a symmetric 3D parallelization across all models. This strategy combines the intra-node TP with the inter-node PP and DP, maximizing the DP degree within memory constraints.

[1] VocabSize × BatchSize × CtxLen × BytesPerParam = $128e3 \times 512 \times 2048 \times 2 = 250$GB

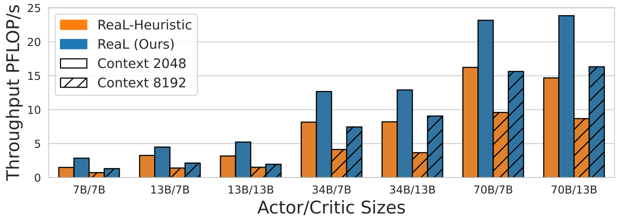


Figure 8: Throughput comparisons with the heuristic execution plan with different context lengths.

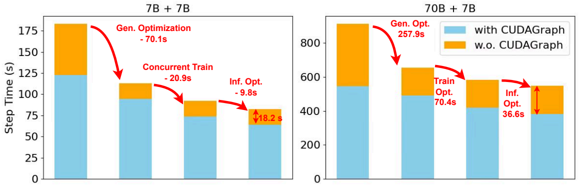


Figure 9: The wall time of one training step of 7B + 7B and 70B + 7B settings, with different levels of optimization. The left most and right most bars show the performance of REAL-Heuristic and REAL. From left to right, the optimized resource allocation and parallelization strategies of generation, training and inference are applied in order.

Settings. We evaluate the *weak scaling* characteristics of REAL, where the model size and the batch size both increase proportionally with the number of devices. Our experimental configuration follows InstructGPT (Ouyang et al., 2022) with more details in Appendix A.

Evaluation Metrics. Since the dataflow graph dependencies ensure consistent convergence properties, we focus on the total training throughput as our primary performance metric. Measurements are taken over 20 consecutive training iterations following appropriate warm-up periods. The observed throughput variation across trials is negligible, thus error bars are omitted from our figures.

Results. Throughput comparisons presented in Figure 7 and Figure 8 demonstrate REAL's superior performance. Compared to the baseline systems, REAL achieves at most 3.58× higher throughput. The search-generated execution plan outperforms REAL-Heuristic by an average of 54%. This advantage further increases to 81% when extending the context length from 2048 to 8192 tokens, highlighting REAL's particular effectiveness in long-context scenarios.

8.2 Breakdown Analysis

To illustrate the source of REAL's performance improvement, we break down and analyze the time cost per training iteration across several representative experimental settings.

Function-call Level Breakdown. First, we analyze the wall time of model function calls in two representative cases: a 7B actor with a 7B critic, and a 70B actor with a 7B critic. These cases respectively feature identical/similar and different sizes for the actor and critic models. Figure 9 shows the wall time per training step in REAL with progressively applied optimizations:

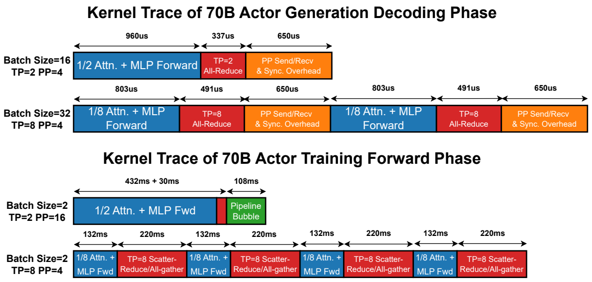


Figure 10: Simplified kernel traces of a transformer layer completing the same amount of decoding or training computation. The top trace in each sub-figure represents REAL, while the bottom trace represents REAL-Heuristic. REAL's parallelization strategies reduce memory I/O by invoking fewer computation kernels and minimize communication overhead caused by excessive tensor or pipeline parallelism.

- CUDAGraph Generation.
- Generation parallelization
- Training parallelization & concurrent execution.
- Inference parallelization & concurrent execution.

Performance improves incrementally from REAL-Heuristic (leftmost bar) to REAL (rightmost bar), with each step adding one optimization. The orange and blue bars demonstrate the impact of CUDAGraph generation, a key contributor to performance improvement. The primary difference between the two settings lies in training phase optimization. In the 7B+7B configuration, REAL concurrently executes actor and critic training on separate devices. Since their training times are similar, this creates perfect overlap, maximizing performance. In contrast, for configurations with large model size disparities like 70B+7B, REAL executes training sequentially on the global device mesh. The significant computational imbalance makes concurrent execution inefficient, so REAL instead employs tailored parallelization strategies for each model to optimize overall performance. The details of the execution plans and wall time breakdown are presented in Tables 2 to 6.

Kernel Level Breakdown. To understand the enhancement of transforming parallelization strategies, we further examine the CUDA kernel traces, with a simplified example shown in Figure 10. During decoding, REAL prioritizes TP over PP to avoid the significant synchronization overhead between pipeline stages caused by numerous small decoding steps. Additionally, REAL maximizes the DP degree within available GPU memory constraints. This reduces memory I/O and P2P communication overheads with less kernel invocations. For the compute-bounded training phase,

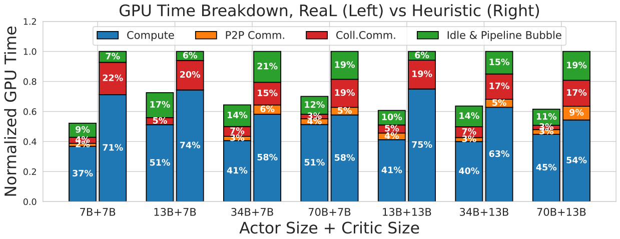


Figure 11: The CUDA kernel time statistics of an RLHF iteration for REAL (left) and REAL-Heuristic (right). REAL effectively eliminates the overhead of parallelization, i.e., the collective communication of TP and the P2P communication of PP, and reduces the memory IO time in compute kernels.

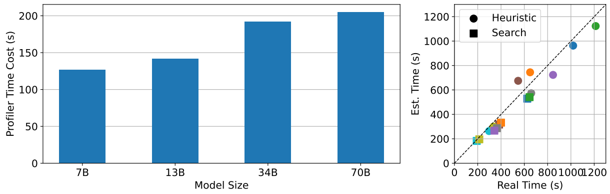


Figure 12: (Left) The time of profiling before cost estimation. We consider batch sizes ranging from 1 to 512 and sequence lengths limited to 256, 512, and 1024. (Right) The estimated time cost produced by the estimator and the real time cost of execution plans used in experiments. Two data points of a same color denote the searched and heuristic execution plan in one experiment setting.

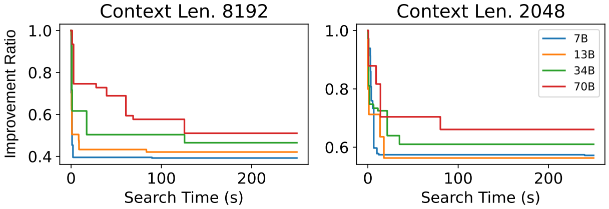


Figure 13: The time cost of the best discovered execution plan compared to the initial one as the searching process proceeds. We name this metric as **improvement ratio**.

REAL utilizes a larger PP degree with a large number of micro-batches. Consequently, it minimizes the TP-induced collective communication overhead with a minimal bubble time increase.

We further validate these observations by decomposing the GPU time per training iteration into three CUDA kernel types, as illustrated in Figure 11. REAL demonstrates a similar trend of the kernel time decreasing across all scenarios. We also note that the broadcasts of data transfer and parameter reallocation take much less GPU time than visualized types, so we omit them from the figure. To conclude, the improvement of REAL stem from two key aspects of our execution plan design. First, given a fixed device count, REAL optimizes the parallelization strategies to minimize redundant memory IO and communication overheads from excessive TP or PP degrees. Second, by executing function calls concurrently across different device subsets, REAL reduces per-function communication overheads through decreasing parallelization degrees.

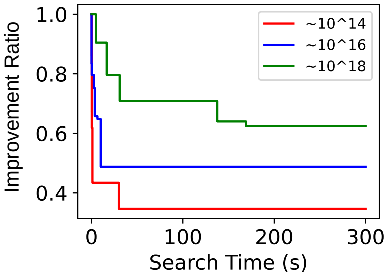


Figure 14: The performance of the MCMC-based search algorithm with pruning in an experiment setting with 1024 GPUs. Three lines show performance with the search spaces that are pruned to 10^{14}, 10^{16}, and 10^{18} execution plans.

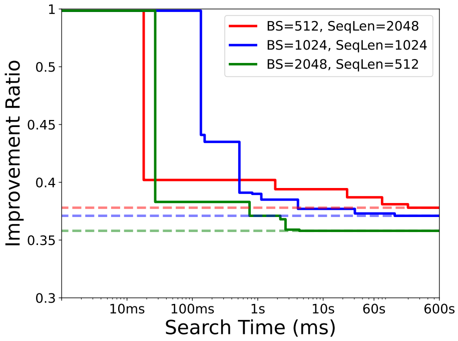


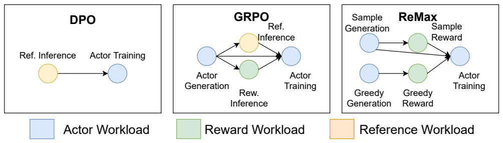
Figure 15: The performance of execution plans produced by MCMC-based search in 10 minutes, in the setting of 7B+7B model sizes and 8 GPUs and three different settings of batch sizes and sequence lengths. The x-axis is log scaled, and the dotted lines mark the optimal performance produced by brute-force search.

We evaluate three key aspects of the execution plan generator in REAL, including the time cost of the profiler, the accuracy of the runtime estimator, and the performance of the search engine.

Profiler. The profiler requires less than 4 minutes to collect a model's complete statistics, as shown in Figure 12 (left). Our experiments only profile statistics of individual layers and inter/intra-node bandwidths. These profiled statistics are reusable across experiments within the same model family.

Runtime Estimator Accuracy. The comparison between the real and estimated time costs, presented in Figure 12 (right), shows relative differences consistently below 25%. Crucially, the estimated costs maintain the same relative ordering as the real costs across different execution plans, ensuring the reliability of searched execution plans.

Search Engine. Figure 13 tracks the estimated RLHF training cost throughout the searching process, using settings from the throughput experiments in Figure 8. The search engine is able to identify execution plans with significant throughput improvements within 150 seconds across all experimental configurations.



Throughput (PetaFLOP/s)	DPO	GRPO	ReMax
ReaL-Heuristic	32.84	60.96	2.49
ReaL (Ours)	50.38	71.12	7.23
Improvement (%)	+53.4%	+16.67%	+190.84%

Figure 16: Throughput comparison with the heuristic execution plan on three prevalent RLHF algorithms other than PPO. Their dataflow graph representations are shown in the upper part.

As the number of GPUs increases, the search space grows at a high-degree polynomial rate. When the number of GPUs reaches over 1000, the entire search space has more than 10^{24} execution plans. In this case, the efficiency of MCMC sampling can degrade badly. To address this, we employ an effective heuristic pruning technique that eliminates suboptimal execution plans likely to cause excessive GPU idle time, out-of-memory errors, or high communication overhead. For instance, we discard parallelization strategies where the tensor parallelization degree exceeds the number of GPUs per node, as these incur significant communication bottlenecks due to limited inter-node bandwidth. We also prune execution plans where the model function calls do not fully utilize the device mesh, as such configurations inevitably lead to GPU idle time and suboptimal performance.

In Figure 14, we present an ablation study that shows the relationship between the size of pruned search space and the efficiency of MCMC sampling in an experiment setting with 1024 GPUs. The results show that our algorithm can still find a fast execution plan within 5 minutes by pruning the search space. Furthermore, while our demonstration uses a single-threaded implementation, the search process can be further accelerated through a multi-core parallelization.

The Optimality of MCMC-based Search. Figure 15 demonstrates a comparison between the execution plan produced by the MCMC-based search algorithm and the optimal ones produced by brute-force search, in the setting of 7B+7B model sizes and 8 GPUs. The result in Figure 15 shows that, in this setting, our search algorithm could achieve more than 95% of the best performance in 5 seconds. Moreover, our search algorithm could produce the optimal execution plans within 10 minutes.

8.3 RLHF Algorithms Beyond PPO

ReaL can naturally incorporate any RLHF algorithms representable as a directed acyclic graph (DAG) with generation, inference, and training function calls. We examine three concrete examples, including DPO (Rafailov et al., 2023), GRPO (Shao et al., 2024), and ReMax (Li et al., 2024).

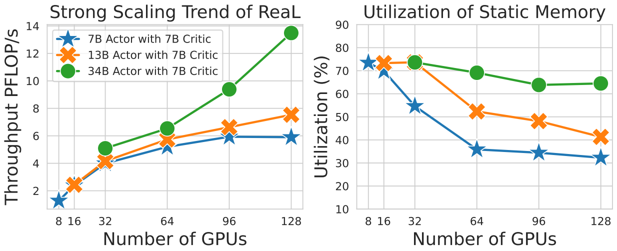


Figure 17: The throughput and memory utilization in strong scaling experiments. ReaL can achieve (super-)linear scaling when the computation budget is tight by parallelizing computation and trading memory for communication. For a small model, the performance will hit the plateau due to the memory IO overhead of auto-regressive generation.

In Figure 16, we compare ReaL with ReaL-Heuristic using a 70B Actor and 7B Critic on 16 nodes and observe an average throughput improvement of 87%. Among these three algorithms, ReMax achieves the highest gain by execution its two generation calls concurrently rather than sequentially. Conversely, GRPO shows more modest improvements due to its grouped generation technique. GRPO increases the batch size by 8× and makes the workload much more compute-bounded, which diminishes the benefits of reducing memory IO or TP/PP overheads.

8.4 Strong Scaling Trend

We analyze the *strong scaling* performance by measuring throughput for fixed problem sizes across increasing device counts. Figure 17 reveals a sub-linear scaling for 7B actors but a super-linear scaling for 34B actors.

Analysis. The scaling behavior can be understood by examining the generation and training patterns, which dominate the RLHF iteration time. With limited computational resources, both operations are compute-bounded. Additional resources enable computation parallelization and can trade off more overall memory usage for less communication, leading to linear or super-linear gains. However, as resources increase, generation becomes a bottleneck due to the inherent sequential nature of autoregressive processing. It requires iterative KV cache loading, creating an irreducible overhead that leads to a diminishing scaling return.

Practical Suggestions. Due to its larger algorithm design and hyperparameter searching space, RLHF implementations face a fundamental trade-off between faster training with more resources and broader configuration exploration with fewer GPUs. The observed sub-linear scaling indicates the reduced resource efficiency at larger scales. However, a minimal resource allocation can impede production by extending experiment duration. Our results suggest identifying the transition point from super-linear to sub-linear scaling for each training configuration as the optimal device allocation. We recommend using static memory utilization as a heuristic metric, with utilization below 60% indicating diminishing returns from additional GPU resources, as demonstrated in Figure 17 (right).

9 Related Work

9.1 Systems for Training and Serving LLMs

Significant efforts have been made to develop distributed LLM training systems (Narayanan et al., 2021; Chowdhery et al., 2023; Jiang et al., 2024) that leverage efficient data (Zhao et al., 2023a; Rajbhandari et al., 2020), tensor-model (Lepikhin et al., 2021; Wang et al., 2019), and pipeline-model parallelism (Huang et al., 2019; Li et al., 2021). Concurrently, research has focused on the efficient serving of pre-trained LLMs for generation (Sheng et al., 2023b;a; Yu et al., 2022; Zhong et al., 2024a). However, integrating dependent workloads for training, inference, and generation, as in the case of RLHF, presents a challenge that extends beyond these individual efforts.

Previous RLHF systems (Yao et al., 2023a; Hu et al., 2024; Shen et al., 2024) typically employ hand-crafted parallelization strategies with limited flexibility. While they incorporate techniques like concurrent execution and ad hoc parameter resharding, these approaches remain inefficient and fail to adapt to diverse RLHF training scenarios. Several concurrent works (Xiao et al., 2023; Lei et al., 2024; Sheng et al., 2024) explore RLHF system design and are conceptually similar to our paper. In comparison, REAL identifies parameter reallocation as the key to addressing this challenge, a factor overlooked by these works. Moreover, our problem formulation and search-based solution offers more generalization and optimization opportunities. Another concurrent work (Zhong et al., 2024b) is an orthogonal extension to our paper. It proposes to fuse pipeline stages of actor and critic training and balancing the data skewness during generation, which targets on some special cases in the workflow of PPO.

9.2 GPU Memory Management for Distributed Training

Previous work on GPU memory management has primarily focused on reducing runtime memory usage, rather than improving training throughput. Techniques such as gradient checkpointing, ZeRO-3 optimization (Rajbhandari et al., 2020), and parameter offloading (Ren et al., 2021; Rajbhandari et al., 2021; Lv et al., 2023; Wu et al., 2024) trade computation or communication to save memory.

Model parameter communication has been explored in parameter server architectures (Li et al., 2014) and large-scale reinforcement learning systems (Berner et al., 2019; Mei et al., 2023). These systems replicate the same set of parameters across multiple devices for concurrent job execution, with periodic synchronization for parameter updates. OpenRLHF (Hu et al., 2024) follows this pattern as well. Parameter synchronization is a specific case of parameter reallocation, where the source and destination devices are disjoint. However, in the context of LLMs, this technique often results in GPU underutilization, making it inefficient.

The concept most related to parameter reallocation is the HybridEngine in DSChat (Yao et al., 2023b) and HybridFlow (Sheng et al., 2024). However, HybridEngine was limited to the actor model on the same device mesh. Parameter reallocation generalizes this approach, allowing it to be applied to any models within the algorithm, whether devices are disjoint or overlapping, leading to further throughput improvements, as demonstrated in Table 6.

9.3 Automatic Parallelization of DL Models

Given the substantial effort required to manually design a parallelization strategy, numerous studies have focused on automating the parallelization of deep learning models (Zheng et al., 2022; Jia et al., 2019; Fan et al., 2020; Harlap et al., 2018; Wang et al., 2019). Notably, Alpa (Zheng et al., 2022) and FlexFlow (Jia et al., 2019) offer general solutions for deep learning models that can be parsed into tensor operator graphs. Alpa leverages dynamic programming, while FlexFlow employs a custom search algorithm.

In theory, the entire RLHF training workflow could be represented as a tensor operation graph and automatically parallelized using prior methods. However, they are sub-optimal for two key reasons. First, parameter reallocation introduces significant optimization opportunities in RLHF, but is unnecessary in supervised training. Consequently, previous methods do not consider parameter reallocation at runtime, resulting in subpar performance. Second, RLHF involves four different LLMs, which are highly operator-intensive. Searching through the entire tensor operator graph would be prohibitively expensive. In contrast, REAL accounts for parameter reallocation and operates at the granularity of model function calls. Our approach not only enhances end-to-end training performance but also explores a smaller solution space, significantly speeding up the search process.

10 Conclusion

In this paper, we present REAL, the first system capable of automatically finding and executing a fast execution plan for RLHF training with parameter reallocation. We evaluate the performance of REAL against prior RLHF systems to demonstrate its superior performance. We believe that REAL will not only democratize the powerful RLHF training algorithm but also encourage the development of novel algorithms on LLMs in the future.

References

Anil, R., Borgeaud, S., Wu, Y., Alayrac, J., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023. doi: 10.48550/ARXIV.2312.11805. URL https://doi.org/10.48550/arXiv.2312.11805.

Antropic. Claude, Jul 2023. URL https://claude.ai/chats.

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., Drain, D., Fort, S., Ganguli, D., Henighan, T., Joseph, N., Kadavath, S., Kernion, J., Conerly, T., Showk, S. E., Elhage, N., Hatfield-Dodds, Z., Hernandez, D., Hume, T., Johnston, S., Kravec, S., Lovitt, L., Nanda, N., Olsson, C., Amodei, D., Brown, T. B., Clark, J., McCandlish, S., Olah, C., Mann, B., and Kaplan, J. Training a helpful and harmless assistant with reinforcement learning from human feedback. *CoRR*, abs/2204.05862, 2022. doi: 10.48550/ARXIV.2204.05862. URL https://doi.org/10.48550/arXiv.2204.05862.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019. URL http://arxiv.org/abs/1912.06680.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., and et al. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., and et al. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023. URL http://jmlr.org/papers/v24/22-1144.html.

Dong, H., Xiong, W., Goyal, D., Zhang, Y., Chow, W., Pan, R., Diao, S., Zhang, J., Shum, K., and Zhang, T. Raft: Reward ranked finetuning for generative foundation model alignment, 2023. URL https://arxiv.org/abs/2304.06767.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., Diao, L., Liu, X., and Lin, W. DAPPLE: A pipelined data parallel approach for training large models. *CoRR*, abs/2007.01045, 2020. URL https://arxiv.org/abs/2007.01045.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., and Gibbons, P. B. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018. URL http://arxiv.org/abs/1806.03377.

Hastings, W. K. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57 (1):97–109, 1970. ISSN 00063444. URL http://www.jstor.org/stable/2334940.

Hu, J., Wu, X., Wang, W., Zhang, D., Cao, Y., et al. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 103–112, 2019.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019. URL https://proceedings.mlsys.org/book/265.pdf.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., Jia, Y., He, S., Chen, H., Bai, Z., Hou, Q., Yan, S., Zhou, D., Sheng, Y., Jiang, Z., Xu, H., Wei, H., Zhang, Z., Nie, P., Zou, L., Zhao, S., Xiang, L., Liu, Z., Li, Z., Jia, X., Ye, J., Jin, X., and Liu, X. Megascale: Scaling large language model training to more than 10, 000 gpus. *CoRR*, abs/2402.15627, 2024. doi: 10.48550/ARXIV.2402.15627. URL https://doi.org/10.48550/arXiv.2402.15627.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In Flinn, J., Seltzer, M. I., Druschel, P., Kaufmann, A., and Mace, J. (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Lei, K., Jin, Y., Zhai, M., Huang, K., Ye, H., and Zhai, J. PUZZLE: efficiently aligning large language models through light-weight context switch. In Bagchi, S. and Zhang, Y. (eds.), *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 127–140. USENIX Association, 2024. URL https://www.usenix.org/conference/atc24/presentation/lei.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B. Scaling distributed machine learning with the parameter server. In Flinn, J. and Levy, H. (eds.), *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pp. 583–598. USENIX Association, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

Li, Z., Zhuang, S., Guo, S., Zhuo, D., Zhang, H., Song, D., and Stoica, I. Terapipe: Token-level pipeline parallelism for training large-scale language models. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 6543–6552. PMLR, 2021. URL http://proceedings.mlr.press/v139/li21y.html.

Li, Z., Xu, T., Zhang, Y., Lin, Z., Yu, Y., Sun, R., and Luo, Z. Remax: A simple, effective, and efficient reinforcement learning method for aligning large language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=Stn8hXkpe6.

Lv, K., Yang, Y., Liu, T., Gao, Q., Guo, Q., and Qiu, X. Full parameter fine-tuning for large language models with limited resources. *CoRR*, abs/2306.09782, 2023. doi: 10.48550/ARXIV.2306.09782. URL https://doi.org/10.48550/arXiv.2306.09782.

Mei, Z., Fu, W., Wang, G., Zhang, H., and Wu, Y. SRL: scaling distributed reinforcement learning to over ten thousand cores. *CoRR*, abs/2306.16688, 2023. doi: 10.48550/ARXIV.2306.16688. URL https://doi.org/10.48550/arXiv.2306.16688.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equation of state calculations by fast computing machines. 3 1953. doi: 10.2172/4390578. URL https://www.osti.gov/biblio/4390578.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on GPU clusters using megatron-lm. In de Supinski, B. R., Hall, M. W., and Gamblin, T. (eds.), *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, pp. 58. ACM, 2021. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

Nvidia. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2024.

OpenAI. Introducing chatgpt, Nov 2022. URL https://openai.com/blog/chatgpt.

OpenAI. Introducing openai o1-preview, Sep 2024. URL https://openai.com/index/introducing-openai-o1-preview/.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. In Cuicchi, C., Qualters, I., and Kramer, W. T. (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 20. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: breaking the GPU memory wall for extreme scale deep learning. In de Supinski, B. R., Hall, M. W., and Gamblin, T. (eds.), *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, pp. 59. ACM, 2021. doi: 10.1145/3458817.3476205. URL https://doi.org/10.1145/3458817.3476205.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Gupta, R., Liu, Y., Tang, J., and Prakash, B. A. (eds.), *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pp. 3505–3506. ACM, 2020. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. In Calciu, I. and Kuenning, G. (eds.), *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pp. 551–564. USENIX Association, 2021. URL https://www.usenix.org/conference/atc21/presentation/ren-jie.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL https://doi.org/10.48550/arXiv.2402.03300.

Shen, G., Wang, Z., Delalleau, O., Zeng, J., Dong, Y., Egert, D., Sun, S., Zhang, J., Jain, S., Taghibakhshi, A., et al. Nemo-aligner: Scalable toolkit for efficient model alignment. *arXiv preprint arXiv:2405.01481*, 2024.

Sheng, G., Zhang, C., Ye, Z., Wu, X., Zhang, W., Zhang, R., Peng, Y., Lin, H., and Wu, C. Hybridflow: A flexible and efficient RLHF framework. *CoRR*, abs/2409.19256, 2024. doi: 10.48550/ARXIV.2409.19256. URL https://doi.org/10.48550/arXiv.2409.19256.

Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J. E., and Stoica, I. S-lora: Serving thousands of concurrent lora adapters. *CoRR*, abs/2311.03285, 2023a. doi: 10.48550/ARXIV.2311.03285. URL https://doi.org/10.48550/arXiv.2311.03285.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single GPU. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 2023b. URL https://proceedings.mlr.press/v202/sheng23a.html.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

Stiennon, N., Ouyang, L., Wu, J., Ziegler, D. M., Lowe, R., Voss, C., Radford, A., Amodei, D., and Christiano, P. F. Learning to summarize with human feedback. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1f89885d556929e98d3ef9b86448f951-Abstract.html.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., and et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. doi: 10.48550/ARXIV.2307.09288. URL https://doi.org/10.48550/arXiv.2307.09288.

Wang, M., Huang, C., and Li, J. Supporting very large models using automatic dataflow graph partitioning. In Candea, G., van Renesse, R., and Fetzer, C. (eds.), *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pp. 26:1–26:17. ACM, 2019. doi: 10.1145/3302424.3303953. URL https://doi.org/10.1145/3302424.3303953.

Wu, X., Rao, J., and Chen, W. ATOM: asynchronous training of massive models for deep learning in a decentralized environment. *CoRR*, abs/2403.10504, 2024. doi: 10.48550/ARXIV.2403.10504. URL https://doi.org/10.48550/arXiv.2403.10504.

Xiao, Y., Wu, W., Zhou, Z., Mao, F., Zhao, S., Ju, L., Liang, L., Zhang, X., and Zhou, J. An adaptive placement and parallelism framework for accelerating RLHF training. *CoRR*, abs/2312.11819, 2023. doi: 10.48550/ARXIV.2312.11819. URL https://doi.org/10.48550/arXiv.2312.11819.

Yao, Z., Aminabadi, R. Y., Ruwase, O., Rajbhandari, S., Wu, X., Awan, A. A., Rasley, J., Zhang, M., Li, C., Holmes, C., Zhou, Z., Wyatt, M., Smith, M., Kurilenko, L., Qin, H., Tanaka, M., Che, S., Song, S. L., and He, Y. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales, 2023a.

Yao, Z., Aminabadi, R. Y., Ruwase, O., Rajbhandari, S., Wu, X., Awan, A. A., Rasley, J., Zhang, M., Li, C., Holmes, C., Zhou, Z., Wyatt, M., Smith, M., Kurilenko, L., Qin, H., Tanaka, M., Che, S., Song, S. L., and He, Y. Deepspeed-chat: Easy, fast and affordable RLHF training of chatgpt-like models at all scales. *CoRR*, abs/2308.01320, 2023b. doi: 10.48550/ARXIV.2308.01320. URL https://doi.org/10.48550/arXiv.2308.01320.

Yoo, A. B., Jette, M. A., and Grondona, M. SLURM: simple linux utility for resource management. In Feitelson, D. G., Rudolph, L., and Schwiegelshohn, U. (eds.), *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, volume 2862 of *Lecture Notes in Computer Science*, pp. 44–60. Springer, 2003. doi: 10.1007/10968987_3. URL https://doi.org/10.1007/10968987_3.

Yu, G., Jeong, J. S., Kim, G., Kim, S., and Chun, B. Orca: A distributed serving system for transformer-based generative models. In Aguilera, M. K. and Weatherspoon, H. (eds.), *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pp. 521–538. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zhang, S., Dong, L., Li, X., Zhang, S., Sun, X., Wang, S., Li, J., Hu, R., Zhang, T., Wu, F., and Wang, G. Instruction tuning for large language models: A survey, 2024. URL https://arxiv.org/abs/2308.10792.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch FSDP: experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023a. doi: 10.14778/3611540.3611569. URL https://www.vldb.org/pvldb/vol16/p3848-huang.pdf.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch FSDP: experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023b. doi: 10.14778/3611540.3611569. URL https://www.vldb.org/pvldb/vol16/p3848-huang.pdf.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In Aguilera, M. K. and Weatherspoon, H. (eds.), *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pp. 559–578. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In Gavrilovska, A. and Terry, D. B. (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 193–210. USENIX Association, 2024a. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Zhong, Y., Zhang, Z., Wu, B., Liu, S., Chen, Y., Wan, C., Hu, H., Xia, L., Ming, R., Zhu, Y., and Jin, X. Rlhfuse: Efficient RLHF training for large language models with inter- and intra-stage fusion. *CoRR*, abs/2409.13221, 2024b. doi: 10.48550/ARXIV.2409.13221. URL https://doi.org/10.48550/arXiv.2409.13221.

Zhuang, Y., Zhao, H., Zheng, L., Li, Z., Xing, E. P., Ho, Q., Gonzalez, J. E., Stoica, I., and Zhang, H. On optimizing the communication of model parallelism. *CoRR*, abs/2211.05322, 2022. doi: 10.48550/ARXIV.2211.05322. URL https://doi.org/10.48550/arXiv.2211.05322.

Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P. F., and Irving, G. Fine-tuning language models from human preferences. *CoRR*, abs/1909.08593, 2019. URL http://arxiv.org/abs/1909.08593.

A Experiment Details

Our base setting is adopted from Ouyang et al. (2022), which utilizes a global batch size of 512, context length 2048, and a maximum prompt length of 1024. The global batch is divided into 8 mini-batches for PPO training.

We emphasize that the prompt and generation length may vary for different models, datasets or tasks, algorithm implementation, and even during RLHF training. To eliminate this effect and perform a fair comparison, we synthesize random data with the maximum prompt length and terminate generation only after the maximum length is reached.

We create LLaMA models of four different sizes with their detailed configurations shown in Table 1. For weak scaling experiments, we increase the model size and batch size proportionally to the number of devices. In particular, for 16, 32, 64, and 128 GPUs, the model sizes are 7B, 13B, 34B, and 70B, and batch sizes are 512, 1024, 2048, 4096, respectively. For experiments with a longer context length, we fix the number of tokens in the global batch. For instance, when the context length increases from 2048 to 8192, the global batch size decreases by a factor of 4. In experiments for strong scaling and dditional RLHF algorithms, we adopt the base setting with 70B actor/reference models and 7B critic/reward models on 16 nodes.

We show the execution plans of wall time breakdown examples (Table 6) in Tables 2 to 5.

B The API of ReaL

Figure 18 shows an example of the API for an REAL experiment. Users define the dataflow graph of the algorithm (e.g., RLHF) using a list of `ModelFunctionCallDef` objects. These objects encapsulate the model configuration and the function call type, along with specifying input and output data dependencies. Models sharing the same `model_name` must have identical architectures (e.g., `llama7b`). They form parameter version dependencies, such that the inference and generation must wait for the training in the previous iteration. The experiment configuration is then wrapped by the `auto` decorator, which initiates the search engine to derive an efficient execution plan. This plan is transformed into a scheduling configuration for launching workers, each assigned to a specific GPU or CPU via SLURM (Yoo et al., 2003). The search engine and launcher both run under the hood. Users are free to provide distinct interface implementations to implement a diverse range of training workflows.

C Simulation Algorithm

The simulation algorithm in show in Algorithm 1.

```python
# auto is a decorator that generates worker
# scheduling configs in the cluster.
@auto(nodelist="com[01-08]", batch_size=256)
@dataclasses.dataclass
class Experiment:
    seed: int = 1
    ppo: PPOHyperparameters

    @property
    def rpcs(self) -> List[ModelFunctionCallDef]:
        return [
            ModelFunctionCallDef(
                model_name="actor",
                model_type="llama7b",
                interface_type=GENERATE,
                input_data=["prompts"],
                output_data=["seq", "logp"],
            ),
            ModelFunctionCallDef(
                model_name="reward",
                model_type="llama7b-critic",
                interface_type=INFERENCE,
                input_data=["seq"],
                output_data=["r"],
            ),
            ModelFunctionCallDef(
                model_name="actor",
                interface_type=TRAIN_STEP,
                input_data=["seq", "r", ...],
            ),
            # ref inference, critic inference,
            # and critic training
            ...,
        ]
```

Figure 18: An example of the user interface of REAL. Given the dataflow graph (represented by a list of `ModelFunctionCallDef` objects), the training batch size, and cluster specifications, REAL will automatically derive an execution plan via the `auto` decorator.

D Baselines

In Figure 7, we show the performance comparison between REAL and 4 baseline RLHF systems: DeepSpeed-Chat (Yao et al., 2023b), OpenRLHF (Hu et al., 2024), NeMoAligner (Shen et al., 2024) and veRL (HybridFlow (Sheng et al., 2024)). The first three baselines are previous works of REAL, and veRL is concurrent to REAL. In this section, we will briefly introduce the implementation of these baseline systems. We also list the version and backend of baseline systems used in our experiments.

DeepSpeedChat is developed using modules from a popular training backend DeepSpeed (Rasley et al., 2020). It supports sequential execution of model function calls, and uses TP for the generation task, ZeRO-3 DP for the training and inference task. It also implements HybridEngine, a technique that reshards parameters between actor training and generation.

OpenRLHF exploits vLLM (Kwon et al., 2023) as their generation backend and DeepSpeed ZeRO-3 DP as their training backend. It divides GPUs into three groups, holding the actor/reference model, the critic/reward model and

Identifier	7B	13B	34B	70B
HiddenSize	4096	5120	8192	8192
IntermediateSize	14336	13824	22016	28672
NumLayers	32	40	48	80
NumAttentionHeads	32	40	64	64
NumKVHeads	8	40	8	8
VocabSize	128256	128256	128256	128256
MaxPositionEmbeddings	8192	8192	8192	8192
TotalParamCount	8030261248	14001525760	35321028608	70553706496
ParamCount w./o. Output Embedding	7504924672	13344855040	34270355456	69503033344

Table 1: The LLaMA-3 model configurations used in experiments. Because critic models have a smaller output embedding layer than the actor (i.e., the output dimension is 1 for the critic), we use the embedding-less parameter count as the identifier.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-16]	2	4	16	4	185.1
RewInf	trainer[01-16]	1	8	16	4	5.6
RefInf	trainer[01-16]	1	8	16	16	35.6
CriticInf	trainer[01-16]	1	8	16	16	5.6
CriticTrain	trainer[01-16]	8	4	4	2	20.8
ActorTrain	trainer[01-16]	2	16	4	2	108.0

Table 2: Device allocations and parallelization strategies for the 70B Actor and 7B critic searched case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-16]	8	4	4	8	241.8
RewInf	trainer[01-16]	8	4	4	8	12.6
RefInf	trainer[01-16]	8	4	4	8	63.5
CriticInf	trainer[01-16]	8	4	4	8	12.5
CriticTrain	trainer[01-16]	8	4	4	8	35.7
ActorTrain	trainer[01-16]	8	4	4	8	163.4

Table 3: Device allocations and parallelization strategies for the 70B Actor and 7B critic heuristic case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-02]	2	2	4	1	16.3
RewInf	trainer01	2	1	4	16	6.0
RefInf	trainer02	1	2	4	16	8.0
CriticInf	trainer[01-02]	1	2	8	8	4.7
CriticTrain	trainer02	4	2	1	2	28.1
ActorTrain	trainer01	2	4	1	2	26.6

Table 4: Device allocations and parallelization strategies for the 7B Actor and 7B critic searched case in Table 6.

	DeviceMesh	TP	PP	DP	#Micro-Batches	Time
ActorGen	trainer[01-02]	8	1	2	4	44.2
RewInf	trainer[01-02]	8	1	2	4	7.3
RefInf	trainer[01-02]	8	1	2	4	7.6
CriticInf	trainer[01-02]	8	1	2	4	6.8
CriticTrain	trainer[01-02]	8	1	2	4	24.3
ActorTrain	trainer[01-02]	8	1	2	4	24.7

Table 5: Device allocations and parallelization strategies for the 7B Actor and 7B critic heuristic case in Table 6.

Time (s)	7B + 7B		70B + 7B	
	REAL	Heuristic	REAL	Heuristic
ActorGen (with CUDAGraph)	16.3	44.2	185.1	241.8
ActorGen (w.o. CUDAGraph)	34.5	104.6	185.1	241.8
RewInf	6.0	7.3	5.6	12.6
RefInf	8.0	7.6	35.6	63.5
CriticInf	4.7	6.8	5.6	12.5
CriticTrain	28.1	24.3	20.8	35.7
ActorTrain	26.6	24.7	108.0	163.4
End2End (with CUDAGraph)	64.0	122.6	383.1	546.8
End2End (w.o CUDAGraph)	82.2	183.0	547.4	912.3

Table 6: The RLHF wall time breakdown of two most common and representative cases. REAL reduces the end-to-end time by accelerating individual model function calls as well as concurrently executing independent computations.

the vLLM generation engine separately. It allows the concurrent execution of actor and critic training. However, the generation and training phase can not be executed concurrently due to data and parameter dependencies. This results in a significant GPU idle time.

Similarly, NeMoAligner divides GPUs into 2 disjoint GPU groups. Unlike OpenRLHF, it locates actor training and generation on the same GPU group. It splits the computations into micro batches and pipeline them to reduce the GPU idle time. It exploits TRT-LLM (Nvidia, 2024) (supports TP and resharding) as generation backend and Megatron-LM (Shoeybi et al., 2019) as training backend (supports 3D parallelization).

veRL supports colocating models on GPUs and split placement of models on different GPU groups, including the strategies adopted by three previous systems. It provides different choices for the generation (SGLang (Zheng et al., 2024) and vLLM (Kwon et al., 2023)) and training backend (Megatron-LM (Shoeybi et al., 2019) and Pytorch FSDP (Zhao et al., 2023b)) to support different parallelization strategies.

We list the version and backend of baseline systems used in our experiments in Table 7. We remark that in this experiment, REAL uses its own generation backend, model and pipeline parallelization, and adopts tensor parallelization and optimizer implementation from Megatron-LM. In a more recent version of REAL, we also support vLLM and SGLang as generation backend, which is not included in the experiments in this paper.

Algorithm 1 Calculate TimeCost($\mathcal{G}_p$)

```
Require: The augmented dataflow graph $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$, device
  meshes $D \in \mathcal{D}$ where $\mathcal{D}$ contains all valid device meshes in the
  cluster.
  ready_queue = PriorityQueue()// Sorted by v.ReadyTime
  completed_set = Set() // Contains completed nodes
  for $v \in \mathcal{V}_p$ do
    if $v$.parents=$\emptyset$ then
      ready_queue.push($v$)
    end if
  end for
  while !ready_queue.empty() do
    Node $v$ = ready_queue.pop()
    DeviceMesh $D$ = $v$.device_mesh
    // $D$.last record the last completed node from all devices
    within $D$
    $v$.StartTime = max$\{v$.ReadyTime, $D$.last.EndTime$\}$
    $v$.EndTime = $v$.StartTime + TimeCost($v$)
    completed_set.add($v$)
    for $D' \in \mathcal{D}$ do
      if overlap($D$, $D'$) and $D'$.last.EndTime $\leq$ $D$.last.EndTime
      then
        $D'$.last = $v$
      end if
    end for
    for $u \in v$.children do
      $u$.ReadyTime = max$\{u$.ReadyTime, $v$.EndTime$\}$
      if $w \in$ completed_set for all $w \in u$.parents then
        ready_queue.push($u$)
      end if
    end for
  end while
  return max$\{v$.EndTime $|v \in \mathcal{V}_p\}$
```

System	Version	Generation Backend	Training Backend
DeepSpeedChat	commit f73a6ed	DeepSpeed v0.15.1	DeepSpeed v0.15.1
OpenRLHF	v0.4.2	vLLM v0.4.2	DeepSpeed v0.15.0
NeMoAligner	v0.4.0	TRT-LLM v0.10.0	Megatron-LM v0.8.0
veRL	0.2.0.post2	vLLM v0.6.3	Pytorch FSDP v2.4.0

Table 7: The version, generation backend and training backend used in our baseline experiments.