ReaLHF: Optimized RLHF Training for Large Language Models through Parameter Reallocation

Zhiyu Mei[1,2,*] Wei Fu[1,2,*] Kaiwei Li[4] Guangju Wang [3] Huanchen Zhang [1,2] Yi Wu [1,2,3]
[1]IIIS, Tsinghua University [2]Shanghai Qi Zhi Institute [3]OpenPsi Inc. [4]Independent Researcher

Abstract

Reinforcement Learning from Human Feedback (RLHF) stands as a pivotal technique in empowering large language model (LLM) applications. Since RLHF involves diverse computational workloads and intricate dependencies among multiple LLMs, directly adopting parallelization techniques from supervised training can result in sub-optimal performance. To overcome this limitation, we propose a novel approach named *parameter ReaLlocation*, which dynamically redistributes LLM parameters in the cluster and adapts parallelization strategies during training. Building upon this idea, we introduce ReaLHF, a pioneering system capable of automatically discovering and running efficient execution plans for RLHF training given the desired algorithmic and hardware configurations. ReaLHF formulates the execution plan for RLHF as an augmented dataflow graph. Based on this formulation, ReaLHF employs a tailored search algorithm with a lightweight cost estimator to discover an efficient execution plan. Subsequently, the runtime engine deploys the selected plan by effectively parallelizing computations and redistributing parameters. We evaluate ReaLHF on the LLaMA-2 models with up to 4×70 billion parameters and 128 GPUs. The experiment results showcase ReaLHF's substantial speedups of 2.0 − 10.6× compared to baselines. Furthermore, the execution plans generated by ReaLHF exhibit an average of 26% performance improvement over heuristic approaches based on Megatron-LM. The source code of ReaLHF is publicly available at https://github.com/openpsi-project/ReaLHF.

1 Introduction

Large Language Models (LLMs) such as ChatGPT [24] have amazed the world with their powerful capabilities. Their success relies on the enormous model sizes, e.g., GPT-3 [5] has 175 billion parameters. Because each graphic processing unit (GPU) has limited memory, to perform supervised training for such an expansive model, the computation along with the model parameters must be distributed across vast GPU clusters [14, 23, 34]. Meanwhile, the critical fine-tuning technique, known as Reinforcement Learning from Human Feedback (RLHF), catalyzed the evolution of GPT-3 into ChatGPT [25, 35, 45]. Despite RLHF's crucial role in production-level LLM applications [1–3, 36], research regarding developing an efficient RLHF system is largely missing.

The workflow of RLHF training is much more complicated than supervised training for LLMs. In RLHF, a primary LLM (the training target, referred to as *Actor*) receives prompts sampled from the dataset and generates responses (i.e., the generation step). These responses are then evaluated by three additional LLMs: a *Reward* model, a *Reference* model, and a *Critic* model (i.e., the inference step). Finally, *Actor* and *Critic* use the evaluation results from the previous step to perform supervised training by iteratively computing gradients and updating parameters (i.e., the training step). In summary, the workflow of RLHF contains four LLMs (referred to as ***models***) with independent parameters and distinct types of computational tasks on GPUs (referred to as ***model function calls***), namely *generation*, *inference*, and *training*.

Existing RLHF systems adopt parallelization techniques directly from supervised training for LLMs [11, 39]. However, we observe two major limitations based on our profiling of the previous systems. First, when the system adopts a *symmetric parallelization* strategy (i.e., models are distributed to every GPU node that applies the same parallelization strategy), it is often *over-parallelized*. Our system profiling in Figure 1 (top) shows that over-parallelization leads to substantial synchronization and communication overhead (the light purple bars), thus compromising the end-to-end system's performance.

Moreover, different computational tasks are better off with different parallelization strategies, as shown in Table 1. A single global parallelization strategy, therefore, is likely to be sub-optimal. An alternative way is to assign different models to different GPU nodes, where models can execute concurrently and apply different parallelization strategies independently. However, our second observation is that such *asymmetric parallelization* 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 the allocation of models on GPU devices is fixed throughout training, which implies a fixed parallelization strategy as well. Therefore, the key idea in this paper is to enable dynamic reallocation of model parameters between GPUs to improve the efficiency of the entire RLHF training process. As shown in Figure 1 (bottom), by first choosing a parallelization strategy tailored for

[*] Equal contribution. Correspond to: meizy20@mails.tsinghua.edu.cn, fuwth17@gmail.com, jxwuyi@gmail.com

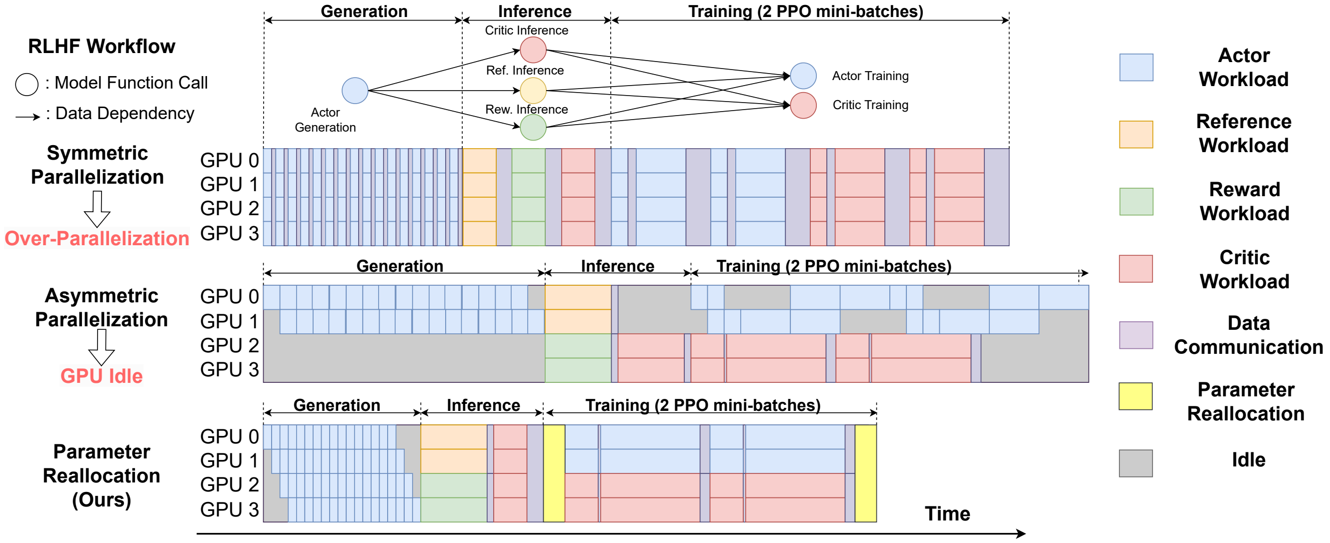

Figure 1. An RLHF iteration breakdown based on the profiling of real systems. The directed graph shows the RLHF workload. Nodes represent model function calls and edges represents their data dependencies. We present timelines to visualize execution plans that employ: [top] the same parallelization strategy for all models, [middle] independent parallelization and fixed device allocation for each model, and [bottom] distinct parallelization strategies for each *model function call* generated by ReaLHF. The plan of ReaLHF considers parameter reallocation.

each model function call (e.g., use pipelining for Generation) and then executing these calls concurrently with a smaller parallelization degree (e.g., Actor and Critic in Training), we can eliminate redundant communication while maximizing GPU utilization, effectively addressing the limitations of prior solution.

Based on the key idea of parameter reallocation, we developed ReaLHF, a pioneering system for efficient RLHF training. ReaLHF consists of two components, i.e., an execution plan generator and a runtime engine. An execution plan is formulated as an augmented dataflow graph, which specifies a particular execution strategy of the RLHF training workflow given the desired algorithmic and hardware configurations. The execution plan generator performs Markov Chain Monte Carlo (MCMC) sampling to search for the most efficient plan using an extremely lightweight profiling-assisted cost estimator. After a sufficiently good execution plan is obtained, the runtime engine deploys the derived plan by effective parallelization and model parameter redistribution.

Our experimental evaluation entails RLHF training on LLaMA-2 models ranging from 7 to 70 billion parameters across 8 to 128 Nvidia A100 GPUs. Results showcase that ReaLHF is able to achieve a speedup ranging from 2.0 to 10.6 times over baseline systems. Furthermore, we demonstrate that the performance of ReaLHF's searched execution plans surpasses heuristic plans based on Megatron by 26% in average and up to 80% in particular cases.

In summary, our contributions are as follows:

- We propose to dynamically reallocate model parameters during training for efficient RLHF training.
- We introduce a general formulation and an effective search algorithm to discover rapid RLHF execution plans.
- We design and implement ReaLHF, an RLHF training system that can automatically discover and run a fast execution plan with high end-to-end throughput.
- We conduct comprehensive evaluations of ReaLHF with detailed ablations and case studies. Moreover, ReaLHF achieves 2.0-10.6× higher throughput than the baseline systems.

Type	Full Pipeline Parallelism	Full Tensor Parallelism
Generation	**29.82s**	37.05s
Training	6.28s	**5.50s**

Table 1. The generation and training time of 7B LLaMA on 8 GPUs.

2 Background

2.1 Introduction to RLHF

This paper adheres to the common practice of RLHF, focusing on GPT-like decoder-only transformer-based neural networks (GPT-like LLMs) [5, 27, 36] and the Proximal Policy Optimization (PPO) algorithm [31].

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 prompt phase and

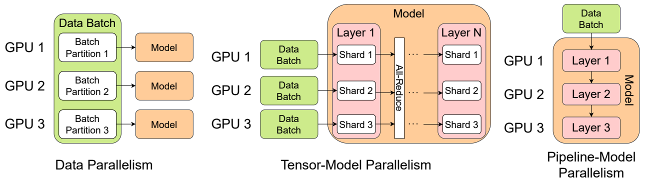

Figure 2. An illustration of existing parallelization approaches.

a decoding phase. The prompt 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 [25], as illustrated in Figure 1. 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. RLHF usually requires multiple consecutive training trials, and each trial is composed of multiple training iterations. For example, Touvron et al. [36] perform 4 × 400 RLHF iterations to build LLaMA-2 series. Meta reports that a single RLHF iteration over the 70B model in their proprietary system consumes about 330 seconds, resulting in about 150 hours of training in total.

2.2 Parallelization of Large Language Models

Classical parallelization approaches for LLMs encompass data, tensor-model, and pipeline-model parallelism. We first discuss them independently and then illustrate how to effectively combine them. Figure 2 presents visualizations of these parallelization methods.

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. DP will consume a large amount of GPU memory due to duplicated parameter storage. As a result, practitioners usually combine them with model parallelism in practice.

Tensor-model Parallelism (TP) partitions model parameters (i.e., weight matrices) and distributes matrix multiplications across multiple GPUs. Each layer processes the entire data batch 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 individual layer of the LLM, TP leads to substantial data communication overhead when scaling to more GPUs and deeper models.

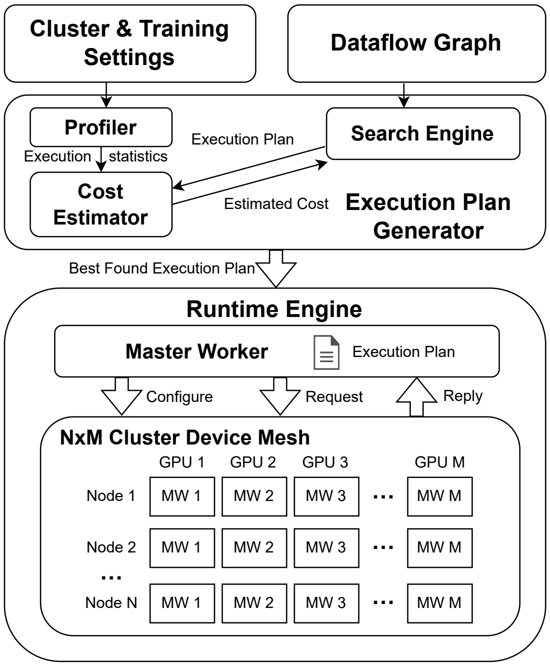

Figure 3. An overview of the architecture of ReaLHF. "MW" is the abbreviation of "Model Worker".

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. Compared to collective communications like all-reduce, send-receive operations entail less communication overhead. Due to the sequential nature of computation, a straightforward implementation of PP may result in significant GPU idle time. To improve the efficiency of PP, a common approach is to divide the data into multiple pipeline micro-batches, allowing different GPUs to process different micro-batches simultaneously.

Since the above parallelization approaches are mutually independent, Megatron-LM [34] integrates them as *3D Paralleism* to perform LLM supervised training at scale. A *parallelization strategy* 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 [23, 43].

3 Overview

ReaLHF is a system capable of automatically planning and executing RLHF training workflows given algorithm and cluster specifications. The key idea behind the design of ReaLHF is exploring the possibility of *parameter reallocation*. This facilitates ReaLHF to produce an execution plan that

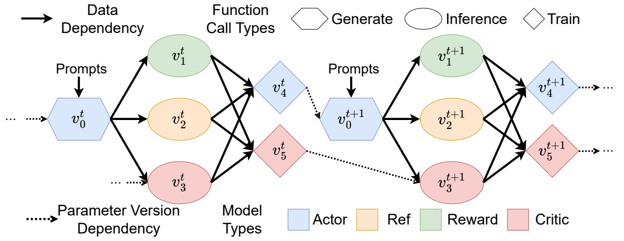

Figure 4. The dataflow graph of two consecutive RLHF iterations. Each ***model*** is an independent LLM. Each ***model function call*** is computational task of the model.

effectively eliminates redundant communication and maximizes GPU utilization. Specifically, this execution plan assigns independent parallelization strategies and device locations to each function call. It then dynamically redistributes parameters at runtime to maximize the overall efficiency.

We summarize the steps of running an RLHF training workflow in REALHF as follows. First, REALHF parses the RLHF workflow into a dataflow graph at the level of model function calls. Then, a specialized search algorithm produces a fast execution plan to decide the parallelization strategies of model function calls and intermediate data/parameter communications. Finally, REALHF runs this fast execution plan on the distributed cluster with an efficient implementation of the 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. The estimated time cost of the searched plan is calculated via a light-weight estimator, which exploits execution statistics obtained by profiling. After reaching search time limit, the fastest discovered execution plan 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 resides on a CPU. It resolves the dependencies of communication and computation tasks in the execution plan. Once a task is ready, the master worker will send requests to the corresponding model workers for its execution. Each model worker is hosted on a single GPU, but it can hold multiple LLM handles (e.g., both Actor and Reward). Model workers act as RPC servers and handle requests from the master worker. After completing the requested task, the model worker responds 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.

Figure 5 shows an example of the API for an REALHF 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 [40]. 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.

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

The rest of the paper is organized as follows. Section 4 introduces our problem formulation and related definitions. Section 5 explains the details of the methods exploited by the execution plan generator to discover an optimized execution plan. Section 6 mainly introduces the implementation details of the runtime engine in REALHF. Section 7 discusses the advantages and limitations of REALHF. Section 8 shows our experiment results and ablation studies. The final two sections discuss the related works and conclude the paper.

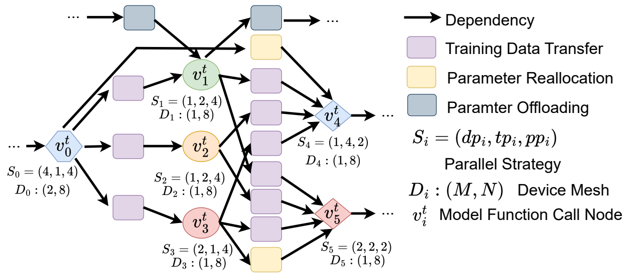

Figure 6. An augmented dataflow graph $\mathcal{G}_p$ of an execution plan instance p in the t-th RLHF iteration.

4 Problem Formulation

ReaLHF aims to find a fast execution plan for RLHF, given the training configurations (e.g., size for each model and training batch size) and the cluster specifications. In this section, we introduce our detailed terminology definitions and the formulation of the execution plan search problem.

Dataflow Graph. ReaLHF 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 this graph, we can effectively overlap computations with no mutual dependencies across training iterations, thus improving the overall training efficiency.

Device Mesh. A device mesh D is the unit for executing an individual function call. It is defined as a two-dimensional grid of GPUs located in the cluster. 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. Different device meshes can overlap if they share some devices; otherwise, they are disjoint. We assume all devices in the cluster have the same computing capability. We characterize communication within device meshes with two types of bandwidth: intra-node (i.e., within one cluster node) communication bandwidth and inter-node (i.e., between cluster nodes) communication bandwidth. We assume bandwidth values of the same type are identical. Typically, intra-node communication bandwidth (e.g., NVLink) is higher than inter-node communication (e.g., InfiniBand or other network interfaces).

Execution Plan. An execution plan of a dataflow graph $\mathcal{G}$ concretely assigns a device mesh and parallelization strategy for every *individual function call* in $\mathcal{G}$. It also appends required data and parameter communication. We express an execution plan p in the form of an ***augmented dataflow graph*** $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$, as illustrated in Figure 6.

Since sub-graphs across training iterations are identical, the i-th model function call over training iterations is assigned the same device mesh and 3D parallelization strategy, which we denote as D_i and S_i respectively. In $\mathcal{G}_p$, each original node v_i^t is tagged with S_i and D_i. S_i will be used to estimate the time cost of this function call and D_i will be used for computing the global time cost of $\mathcal{G}_p$ in Section 5. 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, avoiding sub-optimal execution plans with idle GPUs [43].

Additionally, we introduce an extra type of nodes to represent the transfer of either data or parameters between function call nodes. In particular, for two dependent nodes $v_i^{t_1}$ and $v_j^{t_2}$, we denote the transfer node between them as $u_{ij}^{t_1 t_2}$. Data transfer nodes generally follow the data dependency in the original workflow $\mathcal{G}$, while parameter transfer nodes are applied between the consecutive function calls from the same model, which can be either from the same training iteration or possibly across two consecutive iterations. The device mesh attached to $u_{ij}^{t_1 t_2}$ is the union of D_i and D_j, while it does not retain a parallelization strategy attribute. In our system, data transfer contains device-device communication only, while parameter transfer could be either host-device communication or device-device communication. Host-device parameter transfer resembles offload [30], which copies model parameters to local CPU memory when they are temporarily not needed. Device-device communication is conducted over two device meshes that are either overlapped or disjoint, mapping one 3D parallelization strategy to another.

5 Execution Plan Generator

The Execution Plan Generator takes a dataflow graph, the training configurations, and the cluster specifications as input 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 cost 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 Cost 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 cost estimator assisted by profiling. Profiling the statistics in a single experiment takes only minutes, while evaluating

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

```
Data: 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.EndTime
completed_set = Set() // Contains completed nodes
for $v \in \mathcal{V}_p$ do
    if $v.parents = \emptyset$ then
        ready_queue.push($v$)
    end
end
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
    end
    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
    end
end
return max{$v$.EndTime | $v \in \mathcal{V}_p$}
```

the cost for a candidate execution plan requires only hundreds of microseconds, as opposed to several minutes for profiling a single plan in the real world. In the subsequent paragraphs, we denote the estimated values of time cost and runtime memory of an execution plan as *TimeCost*($\mathcal{G}_p$) and *MaxMem*($\mathcal{G}_p$), respectively.

Time Cost. We first estimate the time cost for each node $v \in \mathcal{V}_p$. For model function call nodes, ReaLHF 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, ReaLHF estimates the time cost using a linear interpolation of the existing profiling statistics. We estimate the costs of data and parameter transfer by running the algorithm outlined in Section 6. We approximate the time with the data size and the bandwidth instead of running a real data transfer 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, detailed in Algorithm 1, 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*($\mathcal{G}_p$).

Static Memory	**Dynamic Memory**
Gradients	KV Cache
Optimizer States	Intermediate Activations
Freezed Parameters	Reallocable Parameters

Table 2. The types of dynamic and static memory considered in the cost estimation.

Maximum Memory Allocated. An execution plan p is executable only if its maximum runtime memory does not exceed device limitations. Memory allocation in ReaLHF follows these principles:

1. For a model designated for training, the gradients and optimizer states cannot be redistributed to other devices alongside the parameters.
2. Model parameters can be redistributed to the CPU memory or a different device mesh, freeing the memory occupied in their original location.
3. Memory for intermediate results, including KV cache, logits in model outputs, and intermediate activations, is dynamically allocated during execution.
4. The buffer memory required for data transfer is negligible compared to other memory costs.

Consequently, we categorize runtime memory into *static memory* and *dynamic memory*, as illustrated in Table 2. For each GPU, we find the associated models via querying all the allocated device meshes. Then, we calculate the maximum runtime memory as the summation of static memory and the peak dynamic memory during an RLHF iteration. We can precisely calculate the memory from the parameter sizes of each model, the shapes of training data, and the chosen parallelization strategies. Finally, *MaxMem*($\mathcal{G}_p$) represents the largest maximum memory across all devices.

Validity of the Estimation. The validity holds under the following assumptions. First, the communication and computation operations considered in the estimation are deterministic and have low variance in the real-world time cost. This ensures predictability in the time cost of nodes. Furthermore, the runtime engine incurs negligible time and memory overhead when executing the plan. The time cost for updating dependencies and dispatching tasks is minor compared to the estimated time cost. In ReaLHF, both of these assumptions are upheld due to the fixed computational workflows of RLHF training in different iterations and our efficient implementation of the runtime engine.

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, ReaLHF employs an efficient MCMC-based search algorithm tailored for this problem setting.

We associate each execution plan with a cost defined by

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

where mem_d is the device memory capacity, I is an OOM indicator, and α is a large integer representint 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 [10, 22] 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 is often 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

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

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 worker-based runtime engine in ReaLHF, including the implementation details of workers, redistributing parameters, and transferring data among model function calls.

Workers. For each node in $\mathcal{G}_p$, the master worker executes an `asyncio` coroutine to send requests to the model workers. The coroutine awaits the completion of all the parent model function calls and dispatches requests via sockets upon satisfying dependencies. These messages do not transfer the required or produced data by function calls. Instead, the data is retained locally in the GPUs of model workers. The master worker maintains the global information about data locations. It communicates this information to the model workers in requests to initiate data transfers. Each model worker acts as an RPC server. 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.

Redistributing Parameters. 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 7 left) and an inner loop (Figure 7 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 7, but with reversed TP-DP dimensions. Therefore, we employ the same broadcast-based algorithm for data transfer. Given that data occupies far less GPU memory than parameters, we additionally maintain a local cache to store the received data, reducing redundant communication.

Remark: Zhuang et al. [44] explored a similar problem to data transfer in ReaLHF. They further analyzed the efficiency of a broadcast-based approach over send-receive and gather-scatter alternatives, validating the rationality of our implementation. In our paper, we do not focus on developing

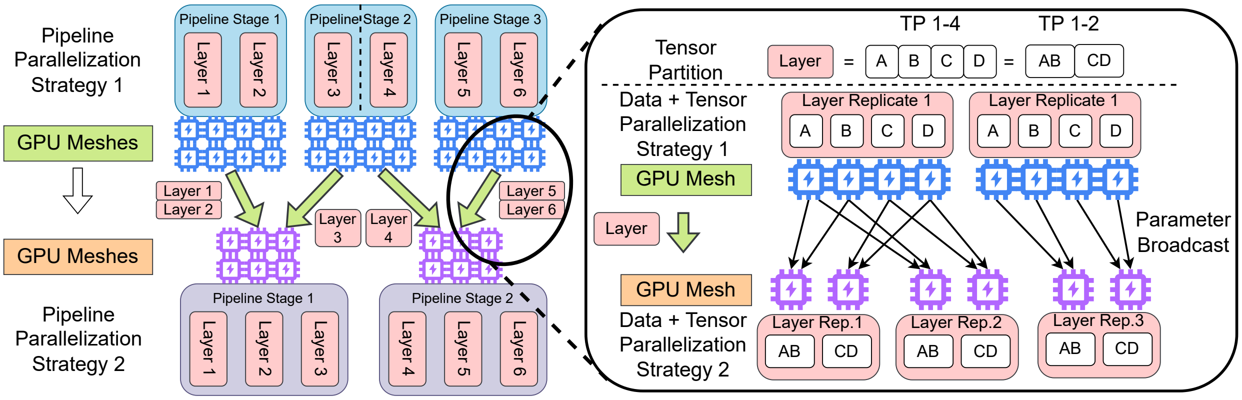

Figure 7. Parameter redistribution is a hierarchical procedure. In the outer loop (left), each pair of pipeline stages communicates the parameters of their common layers. These parameters are distributedly stored in a DP plus TP device mesh. In the inner loop (right), layers are remapped from one DP 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.

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 Section 8.

7 Discussions

This section discusses the advantages and limitations of ReaLHF and clarifies the contexts where ReaLHF can be applied. ReaLHF is a system that is applicable on accelerating RLHF training workflow on GPT-like large language models. ReaLHF has following advantages (■) and limitations (⋄):

- ■ ReaLHF supports 3D parallelism and automatic execution for RLHF, which largely improves system throughput and eliminates human efforts in production. However, neither of them was supported in prior RLHF systems.
- ■ ReaLHF explores a novel technique, parameter reallocation, in LLM training workflows, which can introduce a wide range of new optimization opportunities.
- ■ ReaLHF's method is orthogonal to advanced optimization techniques for model function calls on single LLMs (e.g., Paged-attention [15] for generation). These techniques can be integrated into ReaLHF for better performance.
- ⋄ ReaLHF does not consider parallelization strategies that goes beyond 3D parallelism, which could lead to inferior performance on deep learning models other than LLMs.
- ⋄ ReaLHF requires predictable function calls to ensure the validity of cost estimation and searching. An unstable cluster or dynamic dataflow graph can violate this assumption.
- ⋄ The searching of ReaLHF does not guarantee optimality despite producing plans close to optimal ones.

8 Experiments

We implement ReaLHF with 36k lines of Python code and 2k lines of C++ code. The search engine and simulator are implemented in C++, while the profiler, frontend, and model implementations are based on Python and PyTorch [26]. We assess ReaLHF by executing RLHF with the LLaMA-2 model series [36], currently the most widely used open-source LLM. LLaMA-2 models vary in the parameter counts, with options of 7B, 13B, 34B, and 70B. Our experiments are conducted on a cluster comprising 16 nodes and 128 A100 GPUs. Each node features 128 CPU cores and 1TB memory. Intra-node communication utilizes NVLink, while inter-node communication employs IB with a 200Gbps bandwidth.

Our experimental design unfolds as follows. Initially, we compare the end-to-end performance of ReaLHF with two open-source RLHF systems. We present a breakdown study to delve into the performance improvement of ReaLHF. Subsequently, we ablate and analyze the execution plan generator. Finally, we present a case study showcasing the execution plans devised by ReaLHF.

8.1 End-to-End Performance

Baselines. Since production-level RLHF systems are mostly proprietary, we compare ReaLHF against two open-source solutions: DeepSpeed-Chat [39] and OpenRLHF [11]. Another open-source system, ColossalChat [18], fails to run in distributed environments with more than two nodes, so we omit it from our experiments.

DeepSpeed-Chat (DSChat) employs symmetric parallelization (depicted in Figure 1, top), which parallelizes models using ZeRO-3 data parallelism [28]. Similar to DP, ZeRO-3 performs synchronized forward and backward passes on different data partitions. It additionally partitions model parameters, gradients, and optimizer states across GPUs. Each GPU scatters parameters to peers when peers require them for computation. However, this introduces significant overhead during the decoding phase of generation. DSChat mitigates this with a customized *Hybrid Engine*, which rearranges ZeRO-3 partitions to TP during generation and reverts afterward. DSChat lacks support for TP and PP beyond the Hybrid Engine. OpenRLHF adopts asymmetric parallelization (shown in Figure 1, middle). It partitions devices into five disjoint subsets. Four of them are used to allocate LLM

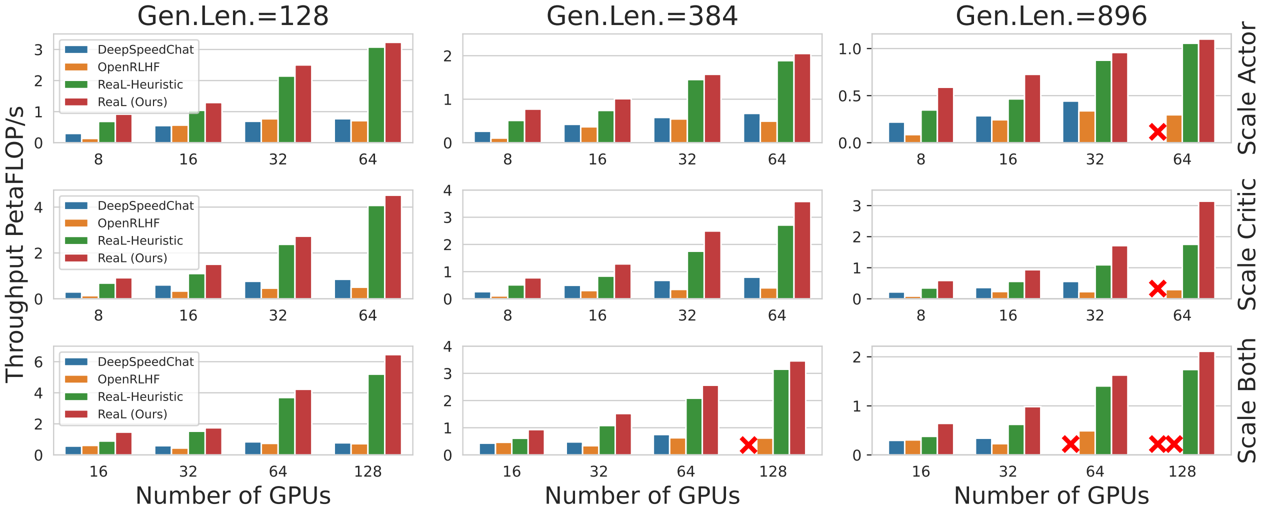

Figure 8. The results of end-to-end throughput comparison. Each row depicts a distinct model size setting: "Scale Actor" maintains the sizes of Critic and Reward at 7B while increasing the sizes of Actor and Reference with the number of GPUs. "Scale Critic" follows the opposite approach, and "Scale Both" increases sizes of all models proportionately. Each column represents a different generation setting, resulting in varied computation workload patterns. ReaLHF-Heuristic denotes a heuristic parallelization strategy on ReaLHF without redistributing parameters and searching. Red crosses indicate that the configured batch size is smaller than the minimum requirement on each GPU according to the parallelization strategy, rendering training infeasible.

models in RLHF, and the last is used to allocate a generation engine based on vLLM [15]. As such, the generation engine is solely responsible for Actor generation, and the original Actor model is solely responsible for training. Parameters are synchronized after each RLHF iteration. Similar to DSChat, it also lacks TP and PP support for individual models.

It is noteworthy that the customized optimizations in DSChat and OpenRLHF represent special cases of our execution plans. However, our search engine may overlook these solutions due to high estimated cost. For baselines, we explore all feasible configurable options and device partitions. Finally, we report the best performance achieved without out-of-memory errors.

We also consider executing a manually crafted execution plan based on a heuristic strategy, which we denote ReaLHF-Heuristic. ReaLHF-Heuristic disables the search engine and does not redistribute parameters. It applies TP for intra-node parallelization and PP for inter-node parallelization across all models, similar to Megatron-LM [34].

Settings. We consider three model size settings and three generation length settings, forming a 3 × 3 grid of experiments. For the model size setting, we first consider the classical setting [25], which scales the Actor and Reference model when the number of GPUs increases and maintains the size of the Critic and Reward at 7B. Then, we consider a mirror setting that scales Critic and Reward, representing applications in weak-to-strong alignment [6]. Finally, we scale all models to test the scalability of ReaLHF. In the former two settings, the numbers of GPUs used for 7B, 13B, 34B, and 70B models are 8, 16, 32, and 64, respectively. The number of GPUs is doubled for the last setting due to the twice larger overall parameter storage. Different generation lengths represent different computation workload patterns of RLHF. With a fixed prompt length of 128, we vary the length of generation in $128, 384, 896$, with the corresponding batch size set to $512, 256, 128$ to maintain the total number of tokens for training at 2^{17}. Actor generation terminates after reaching the maximum generation sequence length. We split the whole batch into 4 PPO mini-batches following [25].

System	DSChat	OpenRLHF	ReaLHF-Heuristic	ReaLHF
Time (hrs)	141.5	152.8	21.2	**17.0**

Table 3. The estimated training time for 4 × 400 [36] RLHF iterations with 70B Actor, 70B Critic, and generation length 128.

Evaluation Metrics. The PPO algorithm implementation in all systems is based on the one in DeepSpeed-Chat. Since both the dependencies in the dataflow graph and the algorithm implementation remain unchanged, the convergence property will not be affected. Therefore, we measure the performance of systems in terms of total training throughput. We record throughput over 20 consecutive training iterations with three warm-up iterations. The variation is small across trials (less than 1%), and we omit error bars in the figures.

Results. We present a comparison of the throughput in Figure 8, as well as the estimated total training time in Table 3.

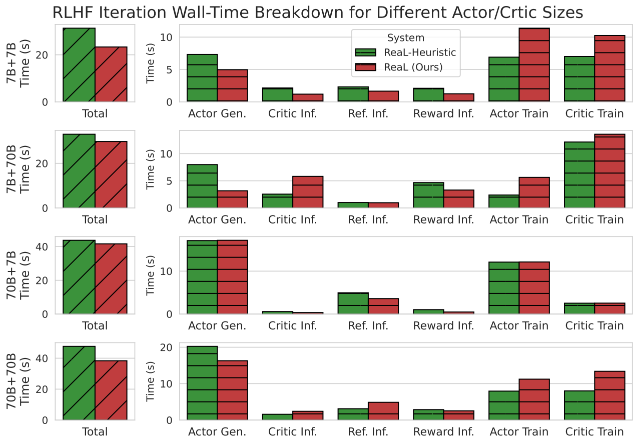

Figure 9. The elapsed wall-time of individual function calls. ReaLHF tends to optimize generation throughput with proper parallelization strategies (e.g., the first two rows). Additionally, for ReaLHF, the summation of times for individual function calls is larger than that of a training iteration. This indicates that ReaLHF concurrently executes different function calls.

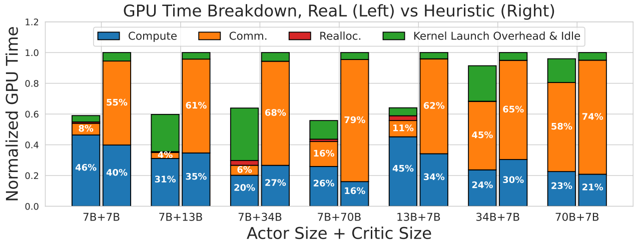

Figure 10. The CUDA kernel execution time of an RLHF iteration for ReaLHF (left) and ReaLHF-Heuristic (right) with a generation length of 896. ReaLHF effectively reduces communication overhead caused by improper parallelization. Bars are normalized according to the execution time of ReaLHF-Heuristic in each setting.

Note that the batch size on each GPU must be at least the number of PPO mini-batches. Since both baseline systems only employ DP or ZeRO-3, they may fail to scale on more GPUs due to violating this condition (red crosses).

In all scenarios, both ReaLHF and ReaLHF-Heuristic outperform baselines significantly, with a training throughput increase of 2.0 - 10.6×. Particularly, because baseline systems are only optimized for scaling the Actor and Reference, they fail to efficiently run when also scaling the Critic and Reward models. Compared with ReaLHF-Heuristic, our searched execution plan leads to a relative improvement of 26.5% on average throughput per GPU. In the following section, we will conduct a breakdown analysis of this performance improvement.

Model Size	Total	Actor Gen.	Critic Inf.	Ref. Inf.	Reward Inf.	Actor Train	Critic Train
7B+7B	8	8	8	8	8	4	4
7B+70B	64	8	56	16	56	8	56
70B+7B	64	64	64	64	64	64	64
70B+70B	128	64	56	24	48	64	64

Table 4. The number of GPUs for each function call in the execution plan produced by ReaLHF across different model size settings. ReaLHF-Heuristic always uses GPUs in the entire cluster for all function calls.

8.2 Breakdown Analysis

To delve deeper into the performance enhancement of ReaLHF, we analyze both wall time and GPU time in an RLHF iteration and compare it with the heuristic plan.

The breakdown of wall time is depicted in Figure 9, where we opt for representative model size settings to ensure clarity. We also show the number of GPUs used by individual model function calls in Table 4. ReaLHF's execution plan tends to reduce function call durations on the critical path, such as Actor generation in the 7B Actor plus 7B Critic setting. Given that both ReaLHF and the heuristic plan can use the same number of GPUs for this function call (see Table 4), the improvement is achieved by tailoring a parallelization strategy with lower communication cost than the TP plus PP heuristic. For ReaLHF-Heuristic, the total iteration time precisely sums up the individual function call durations because it performs identical parallelization across the entire cluster for all models. In contrast, ReaLHF's cumulative function call time surpasses the total iteration time. We believe that ReaLHF overlaps the function calls with the efficient parallelization strategies on disjoint device subsets, which notably reduces the overall communication overhead.

To corroborate this hypothesis, we dissect the GPU time of one training iteration into CUDA kernels of three types as shown in Figure 10. Here, communication specifically pertains to the overhead introduced by 3D parallelism, like all-reduces in DP/TP. The GPU time of data transfer is negligible and omitted. It is noteworthy that diverse execution plans may entail data with varying batch sizes, potentially resulting in differences in compute kernel execution times. In Figure 10, the communication kernels dominate the GPU time for ReaLHF-Heuristic, primarily due to the over-parallelization of models. Correspondingly, the time reduction of ReaLHF mainly originates from the decreased communication cost. Furthermore, the overhead of parameter transfer remains minimal, occupying an average of 2.2% of the total GPU time.

Combining the above findings, we conclude that the reduction of communication overhead for ReaLHF originates from two aspects in the parallelization strategy design. First, with a consistent device count, ReaLHF can tailor an optimal

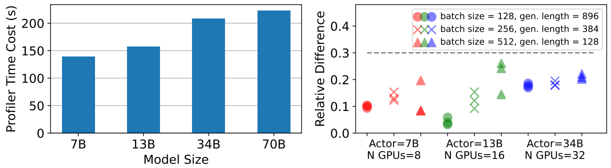

Figure 11. (Left) The time of profiling before cost estimation. We consider batch sizes ranging from 1 to 512 and sequence lengths limited to 256, 512, and 1024. (Right) The relative difference between the estimated time cost and the actual execution time cost using different execution plans.

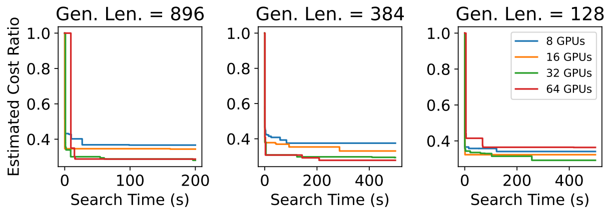

Figure 12. The normalized cost of the best discovered execution plan as the searching process proceeds.

parallelization strategy for specific function calls to minimize the time costs. Second, ReaLHF concurrently executes function calls across different device subsets, reducing the communication overhead for each function call due to less parallelization degrees. In cases where the tailored strategy yields marginal benefits or the concurrent execution proves infeasible, ReaLHF resorts to a strategy akin to the heuristic approach, resulting in fewer advantages.

8.3 Ablations of the Execution Plan Generator

In this section, we will demonstrate the time required for profiling, the accuracy of the cost estimator, and the performance gain brought by the search engine in ReaLHF.

Profiler. Throughout our experiments, the execution statistics of each type of model and inter- and intra-node bandwidth are profiled once by the profiler. As shown in Figure 11 (left), it takes less than 4 minutes to profile the whole set of statistics of one model, which could be repeatedly used across different experiments with the same model type.

Estimator Accuracy. In this experiment, we demonstrate the relative differences between the estimated time cost and the real end-to-end execution time of different execution plans. Since it is expensive to run and profile RLHF in the real world, we randomly sample three execution plans that do not violate the memory constraints from different training configurations and compare their estimated time cost and real execution time of five training iterations. The results in Figure 11 (right) show that relative differences in all 27 trials are at most 28%. The search engine is capable of producing execution plans that optimize the real end-to-end execution time within this acceptable range of difference, as proved by our previous experiments.

N GPUs	8 (7B)	16 (13B)	32 (34B)	64 (70B)
Est. Save (hrs)	5.39 ± 0.13	10.34 ± 0.14	6.62 ± 0.12	6.80 ± 0.23

Table 5. The estimated time saved in a typical RLHF training experiment described in [36] (around 4 × 400 training iterations) with **2 minutes** of searching on a single-threaded search engine. Each result is sampled 10 times.

Function Call	Allocation	3D Parallelization Strategy (dp, tp, pp)
Actor Gen.	Node[1-2]	(8, 1, 2)
Actor Train	Node[1-2]	(2, 1, 8)
Critic Train	Node[3-4]	(1, 4, 4)
Critic Inf.	Node[3-4]	(2, 2, 4)
Reward Inf.	Node[1-2]	(1, 2, 8)
Reference Inf.	Node[1-2]	(4, 4, 1)

Table 6. An execution plan produced by ReaLHF for the setup featuring 7B Actor plus 34B Critic, with a generation length of 896.

Search Engine. Figure 12 shows how the estimated time cost changes as the search process proceeds. In these experiments, the Critic model size is fixed at 7B, and the Actor model size scales from 7B to 70B, with the number of GPUs scaling from 8 to 64. The search engine runs for 15 minutes, and the time required to find the best-executed plan is under 5 minutes for all experiments.

Table 5 displays the estimated time saved in an end-to-end RLHF experiment described in the report of LlaMA-2 [36]. With just two minutes of searching, our search engine can generate an execution plan that saves several hours compared to the heuristic plan. We conduct 10 repetitions of the search for each setting, affirming its ability to reproduce stable outcomes. Note that our search algorithm could be further accelerated with a multi-threaded implementation.

8.4 Case Study

In this section, we showcase an execution plan devised by ReaLHF for the 7B Actor plus 34B Critic setup. The GPU execution timeline is depicted in Figure 13, and parallelization strategies are elaborated in Table 6. In this case, ReaLHF strategically allocates the Actor and Critic to disjoint devices, such that Actor generation and Critic training can be *overlapped across consecutive RLHF iterations* to improve the end-to-end throughput. Besides, ReaLHF redistributes parameters in the local device mesh (see Table 6) to further accelerate individual function calls. Despite GPU idle time introduced by non-perfect overlaps (shown in the third bar of Figure 10), ReaLHF achieves a 42% throughput improvement over heuristic parallelization and at least a 2.7x enhancement over the baselines. This case serves as an ideal example to show that the details of an efficient execution plan could

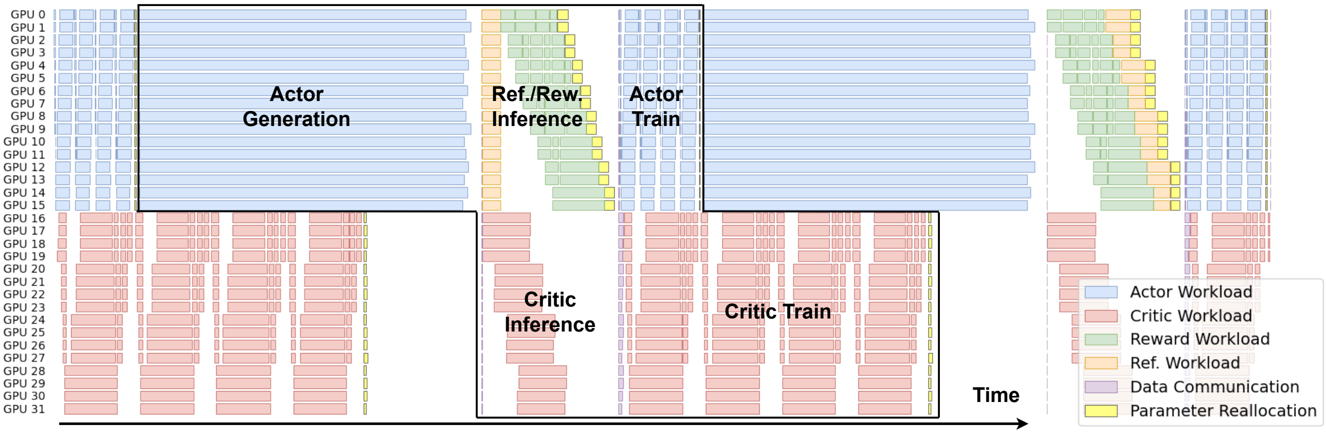

Figure 13. The GPU execution timeline of REALHF in the 7B Actor plus 34B Critic setting with a generation length of 896, obtained by real-time profiling. Black lines mark an RLHF training iteration. The searched execution plan overlaps computations across multiple RLHF training iterations to achieve a higher end-to-end throughput.

be counter-intuitive and opaque for manual design, highlighting the importance of automatic methods capable of discovering such plans.

9 Related Work

9.1 Systems for Training and Serving LLMs

Significant efforts have been invested in developing distributed LLM training systems [7, 14, 23] that employs efficient data [28, 42], tensor-model [16, 37], and pipeline-model parallelism [12, 19]. Concurrently, ongoing researches investigate the efficient serving of pre-trained LLMs for generation [32, 33, 41]. However, the integration of both training and generation workloads, as in the case of RLHF, poses a challenge beyond the scope of these individual endeavors. In this paper, rather than optimizing the throughput of individual function calls like generation or training, we aim to reduce the end-to-end latency of RLHF. We identify parameter reallocation as the key to address this challenge, which is an aspect overlooked by prior works.

9.2 GPU Memory Management for Distributed Training

Previous works on GPU memory management primarily aim to reduce runtime memory usage when training large models rather than improving training throughput. These methods trade computation or communication for reducing memory consumption, such as gradient checkpointing, ZeRO-3 optimization [28], and parameter offload [20, 29, 30, 38]. We incorporate these methods to conserve GPU memory whenever feasible during the evaluation of REALHF and baselines.

The communication of model parameters for small models is investigated by parameter server architectures [17] and large-scale reinforcement learning systems [4, 21]. These systems replicate the same set of parameters on different devices for concurrent job execution, with periodic synchronization for parameter updates. OpenRLHF [11] also follows this pattern. The parameter synchronization is a special case of parameter reallocation, where the source and destination occupy disjoint devices. In the context of RLHF for LLM, this technique is usually inefficient due to GPU underutilization.

The most relevant work to parameter reallocation is perhaps the Hybrid Engine in DSChat [39]. It rearranges the layer-wise partitioned parameters from ZeRO-3 to a TP strategy during generation. However, this remains an ad-hoc solution and exhibits poor scalability. Our solution space with parameter reallocation effectively consolidates this approach, although it will not be the ultimate output of the search engine due to its high communication overhead.

9.3 Automatic Parallelization of DL Models

Because of the substantial engineering effort required to hand-craft a parallelization strategy, numerous studies focus on the automatic parallelization of deep learning models [8, 9, 13, 37, 43]. Among them, Alpa [43] and Flexflow [13] propose general solutions suitable for deep learning models that can be parsed into tensor operator graphs. Specifically, Alpa [43] exploits dynamic programming, while FlexFlow [13] proposes a customized search algorithm.

Theoretically, the entire RLHF training workflow could be represented as a tensor operation graph and automatically parallelized by previous methods. However, these methods are sub-optimal for RLHF due to the following two reasons. First, parameter reallocation introduces significant optimization opportunities to RLHF training, while unnecessary in traditional supervised training. Therefore, neither previous methods consider parameter reallocation at runtime, leading to inferior performance. Second, RLHF incorporates four different large language models, which are operator-intensive. It is unacceptably expensive to search over the entire tensor operator graph of RLHF. In comparison, REALHF takes parameter reallocation into consideration and operates on the granularity of model function calls. For RLHF, our method not only improves the end-to-end training performance but also explores a smaller solution space, significantly accelerating the searching procedure.

10 Conclusion

In this paper, we present ReaLHF, the first system capable of automatically finding and executing a fast execution plan for RLHF training with parameter reallocation. We first propose a new problem formulation that characterizes execution plans, considering parameter reallocation. Based on this formulation, we design a search algorithm based on MCMC sampling to find a fast execution plan that can be executed on our efficient runtime engine. We evaluate the performance of ReaLHF against prior RLHF systems to demonstrate its superior performance. We believe that ReaLHF will not only democratize the powerful RLHF training algorithm but also encourage the development of novel algorithms on LLMs in the future.

References

[1] R. Anil, S. Borgeaud, Y. Wu, J. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican, 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.

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

[3] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. E. Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. B. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan. 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.

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

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *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/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[6] C. Burns, P. Izmailov, J. H. Kirchner, B. Baker, L. Gao, L. Aschenbrenner, Y. Chen, A. Ecoffet, M. Joglekar, J. Leike, I. Sutskever, and J. Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. *CoRR*, abs/2312.09390, 2023. doi: 10.48550/ARXIV.2312.09390. URL https://doi.org/10.48550/arXiv.2312.09390.

[7] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, 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.

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

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

[10] W. K. Hastings. 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.

[11] J. Hu, X. Wu, Xianyu, C. Su, L. Qiu, D. Jiang, Q. Wang, and W. Wang. Openrlhf: A ray-based high-performance rlhf framework. https://github.com/OpenLLMAI/OpenRLHF, 2023.

[12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html.

[13] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *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.

[14] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu. 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.

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

[16] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. 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.

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

[18] S. Li, H. Liu, Z. Bian, J. Fang, H. Huang, Y. Liu, B. Wang, and Y. You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 766–775, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708435. doi: 10.1145/3605573.3605613. URL https://doi.org/10.1145/3605573.3605613.

[19] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In M. Meila and T. Zhang, editors, *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*, pages 6543–6552. PMLR, 2021. URL http://proceedings.mlr.

press/v139/li21y.html.

[20] K. Lv, Y. Yang, T. Liu, Q. Gao, Q. Guo, and X. Qiu. 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.

[21] Z. Mei, W. Fu, G. Wang, H. Zhang, and Y. Wu. 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.

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

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

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

[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *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. URL http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

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

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

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

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

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

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

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

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

[35] N. Stiennon, L. Ouyang, J. Wu, D. M. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano. Learning to summarize with human feedback. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *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.

[36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, 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.

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

[38] X. Wu, J. Rao, and W. Chen. 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.

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

[40] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: simple linux utility for resource management. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *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*, pages 44–60. Springer, 2003. doi: 10.1007/10968987_3. URL https://doi.org/10.1007/10968987_3.

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

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

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

[44] Y. Zhuang, H. Zhao, L. Zheng, Z. Li, E. P. Xing, Q. Ho, J. E. Gonzalez, I. Stoica, and H. Zhang. 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.

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