HybridFlow: A Flexible and Efficient RLHF Framework

Guangming Sheng
The University of Hong Kong
gmsheng@connect.hku.hk

Chi Zhang
ByteDance
zhangchi.usc1992@bytedance.com

Zilingfeng Ye
ByteDance
yezilingfeng@bytedance.com

Xibin Wu
ByteDance
wuxibin@bytedance.com

Wang Zhang
ByteDance
zhangwang.nozomi@bytedance.com

Ru Zhang
ByteDance
zhangru.1994@bytedance.com

Yanghua Peng
ByteDance
pengyanghua.yanghua@bytedance.com

Haibin Lin
ByteDance
haibin.lin@bytedance.com

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Abstract

Reinforcement Learning from Human Feedback (RLHF) is widely used in Large Language Model (LLM) alignment. Traditional RL can be modeled as a dataflow, where each node represents computation of a neural network (NN) and each edge denotes data dependencies between the NNs. RLHF complicates the dataflow by expanding each node into a distributed LLM training or generation program, and each edge into a many-to-many multicast. Traditional RL frameworks execute the dataflow using a single controller to instruct both intra-node computation and inter-node communication, which can be *inefficient* in RLHF due to large control dispatch overhead for distributed intra-node computation. Existing RLHF systems adopt a multi-controller paradigm, which can be *inflexible* due to nesting distributed computation and data communication. We propose *HybridFlow*, which combines single-controller and multi-controller paradigms in a *hybrid* manner to enable *flexible* representation and *efficient* execution of the RLHF data*flow*. We carefully design a set of hierarchical APIs that decouple and encapsulate computation and data dependencies in the complex RLHF dataflow, allowing efficient operation orchestration to implement RLHF algorithms and flexible mapping of the computation onto various devices. We further design a 3D-HybridEngine for efficient actor model resharding between training and generation phases, with zero memory redundancy and significantly reduced communication overhead. Our experimental results demonstrate 1.53×~20.57× throughput improvement when running various RLHF algorithms using HybridFlow, as compared with state-of-the-art baselines. HybridFlow source code will be available at https://github.com/volcengine/verl

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696075

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords: Distributed systems, Reinforcement Learning from Human Feedback

ACM Reference Format:
Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3689031.3696075

1 Introduction

Large language models (LLMs) such as GPT [11], Llama [73] and Claude [7] have revolutionized various artificial intelligence (AI) applications, ranging from writing [2], searching [52] to coding [63]. LLMs are first pre-trained on trillions of tokens from books, websites, etc,. via next-word prediction to accumulate broad knowledge [11]. Next, LLMs are trained on domain-specific datasets via supervised fine-tuning (SFT), to be able to follow human instructions [11]. Despite the outstanding capabilities of LLMs on natural language tasks after pre-training and SFT, the detrimental and biased contents in the training datasets may still mislead an LLM to generate toxic and undesirable content. Reinforcement Learning from Human Feedback (RLHF) is introduced to further align an LLM to human values, for building helpful and harmless AI applications [7, 55].

RLHF is built upon traditional RL algorithms [4, 68, 78], e.g., Proximal Policy Optimization (PPO) [68] and REINFORCE [78]. The widely adopted PPO-based RLHF system typically consists of four LLMs [7, 55]: an *actor*, a *critic*, a *reference policy* network and a *reward model*. PPO-based RLHF proceeds in iterations, each with three stages: (1) response *generation* using the actor model with a batch of prompts;

(2) *preparation* of training data by scoring the generated responses through a single forward pass of the critic, reference policy, and reward models; (3) *learning* from human preference by updating actor and critic through forward and backward computation. Other RLHF variants [19, 43] follow similar stages but involves different numbers of models and data dependencies among the models.

Traditional RL can be modeled as a dataflow [46], which is a directed acyclic graph (DAG): each node in the RL dataflow represents computation of a neural network (e.g., actor or critic network which can be CNN or MLP); each edge denotes data dependency between NN computations (e.g., output of the critic is used as input to actor training [68].) RLHF dataflow is more complex, with more complicated models involved (e.g., LLMs for the actor/critic/reference/reward models), each running distinct computation, and more diverse data dependencies among them (i.e., multicast between distributed model partitions). Training and generation of an LLM in the RLHF dataflow requires distributed computation (e.g., using tensor/pipeline/data parallelism) [40, 71]. Therefore, each node in the RLHF dataflow is a complex distributed program, corresponding to distributed computation of the respective LLM. Models in different nodes typically use different parallelism strategies as their workloads vary. The edge represents data resharding, which is often a many-to-many multicast. Consequently, ***Flexible*** representation and ***efficient*** execution of the complex and resource intensive RLHF is imperative.

Traditional RL frameworks such as RLLib [45] and RLLib Flow [46] utilize a hierarchical single-controller paradigm to run RL dataflows. A centralized controller assigns nodes in the dataflow to different processes and coordinates their execution order. Each node process can further spawn more workers to perform computation, again following the single-controller paradigm. However, they only provide primitives for data-parallel training and are constrained to neural networks that are at most hundreds of MB in size [45, 46]. In the RLHF dataflow, each node corresponds to an LLM with up to billions of operators, computed using some complex parallelism. A single-controller paradigm is inefficient due to the substantial overhead of dispatching operators to distributed accelerators [1, 9].

Existing RLHF systems adopt a multi-controller paradigm to manage intra-node computation and inter-node data resharding [17, 30, 80]. Each controller independently manages the computation of one device and uses multiple point-to-point operations to coordinate data dependencies between different nodes. This multi-controller paradigm introduces negligible dispatch overhead when performing LLM computation (detailed in §2.2).

However, without central control, it is *inflexible* to implement various RLHF dataflow, as modifying a single node to adapt to different data dependencies requires changing all dependent nodes' implementation, hindering code reuse.

To address these limitations, we propose *HybridFlow*, a flexible and efficient RLHF framework to easily represent and execute diverse RLHF dataflows, attaining high throughput. Our key observation is that utilizing the single-controller paradigm on the inter-node level enables flexible expression of various data dependencies and easy coordination of inter-node data resharding with minimal overhead, while integrating the multi-controller paradigm within intra-node computation enhances computation efficiency substantially. We advocate a hierarchical hybrid programming model to generate RLHF dataflows. At the node level, multiple model classes are provided that encapsulate distributed computation (training, inference and generation) of different LLMs in the dataflow into primitive APIs. These APIs can seamlessly support various parallelism strategies from the existing LLM frameworks, including 3D parallelism [71], ZeRO [59], and PyTorch FSDP [57]), and perform distributed computation under the multi-controller paradigm. Among the nodes, a set of transfer protocols are designed to hide the complexity of data resharding from users, as coordinated by a single controller. This programming model abstracts away the complexity of distributed computing, allowing users to implement an RLHF dataflow in a few lines of code and run RLHF through a single process of the single controller. It also effectively decouples intra-node computation and inter-node data transfer, allowing independent optimization of each model without changing the code of other models in the dataflow.

Training and generation of the actor model represent major computation in the RLHF dataflow. We further design a *3D-HybridEngine* to enable efficient execution of training and generation of the actor model, introducing zero memory redundancy and significantly reduced communication overhead during model parameter resharding between the training and generation stages. Our hybrid programming model also facilitates flexible placement of models onto the same or different sets of GPU devices. This allows us to provide an effective algorithm to optimize GPU allocation and placement of the models, with various model sizes and distinct workloads, for any RLHF dataflow. Our contributions in designing HybridFlow are summarized as follows:

- We propose a hierarchical hybrid programming model for conveniently building the RLHF dataflow. This programming model enables efficient distributed execution of intra-node computation and flexible inter-node data resharding and transfer, for various RLHF algorithms (§4).
- We design a 3D-HybridEngine that executes training and generation of the actor model with high computation efficiency and zero-redundancy transition between the training stage and the generation stage (§5).
- We devise an effective mapping algorithm to automatically identify optimized GPU allocation and placement of each node (model) in the RLHF dataflow (§6).
- We conduct extensive experiments comparing HybridFlow with state-of-the-art RLHF systems [17, 30, 82] under various

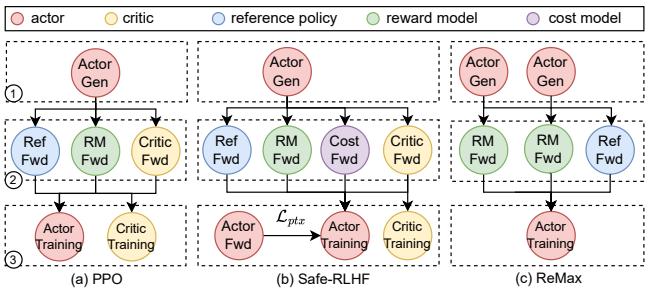


Figure 1. Dataflow graph of 3 RLHF algorithms [19, 43, 55]. Stage ①, ②, ③ represent Generation, Preparation, and Training, respectively.

RLHF algorithms, model sizes and cluster scales. Our evaluation demonstrates 1.53×~20.57× throughput improvements.

We have open-sourced HybridFlow and believe that HybridFlow can boost future RLHF research and development.

2 Background and Motivation

2.1 Reinforcement Learning from Human Feedback

RLHF Workflow. RLHF aligns the linguistic space of LLMs with human values, using a set of human-ranked candidates of given prompts [7, 19, 41, 43, 55, 70, 91]. An RLHF system typically consists of multiple models, e.g., an actor, a critic, a reference policy, and one or multiple reward models. The actor and the reference are each pre-trained/fined-tuned LLM (i.e., the LLM that is undergoing RLHF). The critic and reward models can be different LLMs fine-tuned on the human preference dataset, with the language modeling head replaced by a scalar output head [7, 55]. The RLHF workflow can be decomposed into 3 stages (Figure 1) and we take PPO as an example:

•*Stage 1 (Generation):* The actor produces responses from a batch of prompts using auto-regressive generation.

•*Stage 2 (Preparation):* Using prompts and generated responses, the critic computes their values [66, 68], the reference policy computes their reference log probabilities, and the reward model computes their rewards [7, 55], all via a single pass of forward computation of the respective model.

•*Stage 3 (Learning/Training):* The actor and the critic are updated via Adam [38], using the batch of data produced by previous stages and the loss function [55].

Other RLHF algorithms largely follow the 3-stage workflow as well (Figure 1(b)(c)). Safe-RLHF [19] introduces an auxiliary pretrain loss following PPO-ptx [55] and includes an additional cost model to fit human preferences and safety labels simultaneously. ReMax [43] requires an additional generation pass for variance reduction and eliminates the critic model in the dataflow. Researchers are actively exploring novel RLHF algorithms [41, 70, 91] and integrating traditional RL methods into RLHF domains [37]. These variances necessitate a flexible representation of the RLHF dataflow graph to accommodate diverse algorithmic requirements.

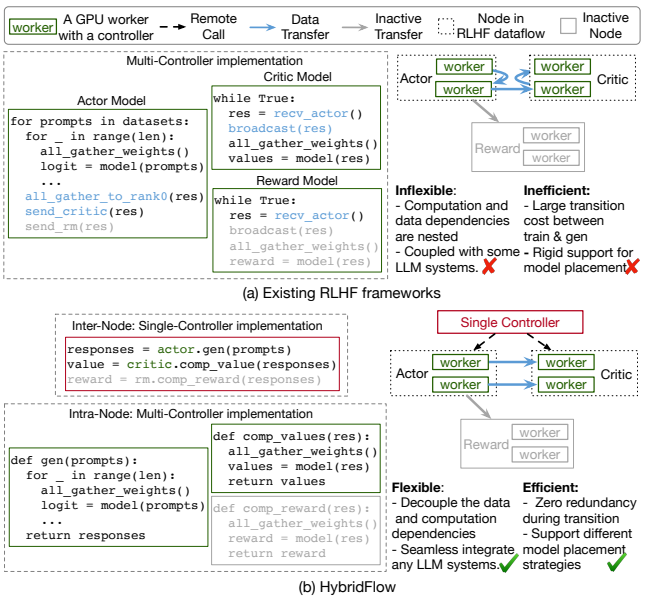


Figure 2. Programming model used in RLHF systems. (a) Existing RLHF systems adopt the *multi-controller* paradigm. (b) HybridFlow utilizes a hybrid programming model: the *single-controller* coordinates models; each model uses *multi-controller* paradigm in distributed computation. Inactive node in grey represents operation not executed at this time.

Parallelism Strategies. LLMs are trained and served with data, pipeline, and tensor parallelism [36, 40, 54]. With data parallelism (DP), the input data is split into multiple subsets; each subset is processed by a separate device (e.g., a GPU) [69]. ZeRO [59] is a memory-optimized solution for DP training, progressively sharding optimizer states, gradients, and model parameters across GPUs. Pipeline parallelism (PP) [32, 53] and tensor parallelism (TP) [71] distribute model parameters, gradients and optimizer states across multiple GPUs. Modern distributed training frameworks like Megatron-LM [71] and MegaScale [36] utilize *3D parallelism* or *PTD parallelism* [54], where P, T, D stand for PP, TP, DP, respectively. In 3D parallelism, PP size represents the number of pipeline stages in model training, TP size refers to the number of shards that a tensor is partitioned into, and DP size is the number of model replicas. LLM serving systems employ 3D parallelism similar to training while only model parameters and KVCache are sharded [16, 29, 40].

LLM models in the RLHF dataflow may perform distinct computations, including training (one forward pass, one backward pass and model update), inference (one forward pass) and generation (auto-regressive generation with multiple forward passes). In particular, training and generation are performed on the actor model, training and inference on the critic, and inference on reference policy and reward models. Distinct parallel strategies can be applied to different models for varied computations to achieve optimal throughput.

2.2 Programming Model for Distributed ML

Single-Controller. It employs a centralized controller to manage the overall execution flow of the distributed program. With centralized control logic, users can build core functionalities of the dataflow as a single process (Figure 2(b)), while the controller automatically generates distributed workers to carry out the computation. With a global view of the hardware and dataflow graph, the single-controller paradigm allows flexible and optimized resource mapping and execution order coordination among dataflow tasks. However, coordination messages are passed from the controller to all workers, incurring significant dispatch overhead when executing expansive dataflow graphs on large clusters [1, 9].

Multi-Controller. Each device (aka worker) has its own controller. State-of-the-art distributed LLM training and serving systems adopt the multi-controller paradigm, due to its scalability and low dispatch overhead (control messaging largely passed from CPU to GPU over fast PCIe links) [36, 40, 60, 71]. As shown in the example that employs multi-controller RLHF implementation in Figure 2(a), a separate program is run for each model, and all workers of one model execute the same program. Each worker only possesses a local view of the system state and requires point-to-point communication between two models (blue code and arrows) to coordinate model execution order. To implement an RLHF workflow in the multi-controller architecture, a user must intricately integrate the code for collective communication, computation, and point-to-point data transfer in the program run at each device. This leads to deeply nested code of computation and data transfer, challenging to develop, maintain, and optimize. In Figure 2(a), each model performs local computation and all_gather operations (black code), while the actor model must explicitly manage send operations to the critic and reward models, and the latter must correspondingly implement receive operations at precise points in their program.

2.3 RLHF Characteristics

Heterogeneous model workloads. The actor, critic, reference and reward models in RLHF may execute training, inference or generation at different stages, with different memory footprint and computation demand. For reference policy and reward models, only their model parameters need to be stored in GPU memory, as they perform only the forward pass computation. For the actor and the critic, their model parameters, gradients, and optimizer states must be stored as they undergo model training. Moreover, a small actor model (e.g., a 7B pre-trained/fine-tuned LLM) can be paired with larger critic and reward models (e.g., 70B LLMs) in RLHF for better alignment [7]. Given such heterogeneity, different parallelism strategies and tailored optimizations are needed for running each model during RLHF.

Unbalanced computation between actor training and generation. In the RLHF dataflow, training and generation of the actor model are represented by two nodes (Figure 1),

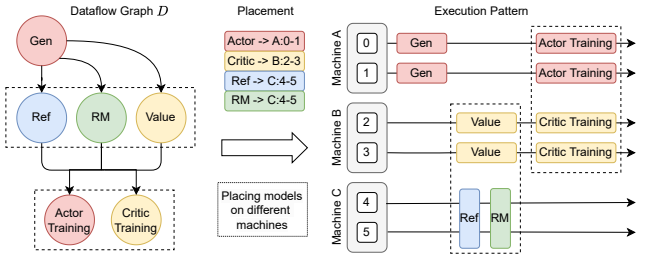


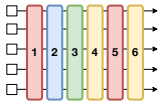
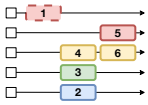
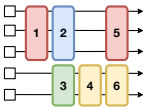
Figure 3. Dataflow execution given a model placement plan. Blocks with numbers represent GPUs. In dashed boxes, the models are placed on different sets of devices and can be concurrently computed. Reference model (blue) and reward model (green) are colocated on the same set of GPUs and executed sequentially.

which often render majority of the workload in each RLHF iteration (e.g., 58.9% of total RLHF time with HybridFlow). Actor training is computation bound [24], often requiring a larger model-parallel (MP) size (i.e., the number of partitions the model is partitioned into) and distributing the workload to more GPUs, e.g., 8 partitions of a 7B model on 8 GPUs. Using the same parallelism strategy (e.g., the same MP size) for generation can lead to underutilization of GPU computation resources due to its memory-bound nature [40]. Previous studies show that combining a larger DP size with a smaller MP size (hybrid data and model parallelism), e.g., partition a 7B model into two and replicate it four times on 8 GPUs, can improve the generation throughput [44, 92]. Although using different parallelism strategies for actor training and generation may optimize throughput in both stages, resharding the actor model weights at runtime between the two stages can incur significant communication and memory overhead. For example, aligning a 70B actor model requires transferring 140GB of model weights from training to generation per RLHF iteration, taking up to 36.4% of an iteration time when the two stages are on different devices [30].

Diverse model placement requirements. Strategic device placement of models in the RLHF dataflow is necessary, according to computation workloads and data dependencies of the models. Figure 3 gives an example model placement plan and the corresponding RLHF execution flow. Models placed on different sets of devices can be executed in parallel if no data dependencies exist. Models placed on the same set of GPUs, referred to as *colocated models*, share the GPU memory and are executed sequentially in a time-sharing manner, as out-of-memory (OOM) error may easily happen if colocated LLMs execute concurrently.

We observe a compromise: placing models on different devices permits parallel processing but may inevitably lead to some GPU idle time, given staged model execution in RLHF. In Figure 3, actor and critic are placed separately, performing training in parallel, but incurring 1/3 of their GPU time being idle, during other RLHF stages. Supporting various

Table 1. Comparison of RLHF frameworks. Figures illustrate execution of one PPO iteration. Numbers 1-6 represent response generation, reward model inference, reference model inference, critic inference, actor training, and critic training, respectively.

RLHF system	DeepSpeed-Chat	OpenRLHF	NeMo-Aligner	**HybridFlow**
Parallelism	Training: ZeRO Generation:TP	Training: ZeRO Generation:TP	3D Parallelism for both training and generation	**Training: 3D, ZeRO, FSDP** **Generation: 3D Parallelism**
Actor weights in training & generation	Model resharding from ZeRO to TP	Using two copies of actor weights for the two stages	Using identical model partition in two stages (shared weights)	**Zero-redundancy** **model resharding**
Model Placement	Colocate all models on the same set of devices	Each model placed on separate devices	Actor/Ref colocated on some GPUs Critic/RM colocated on other GPUs	**Support various** **model placement**
Execution Pattern Actor GPU Process Critic Reward model Reference Policy				**Support various** **execution patterns**

placement strategies and maximizing device utilization are crucial for optimizing RLHF performance at any model size and cluster scale.

2.4 Limitations of existing RLHF systems

Inflexible support for various RLHF dataflow graphs. Existing RLHF systems adopt the multi-controller paradigm for dataflow implementation [17, 30, 80, 82]. To implement various RLHF algorithms, a user must navigate and manage code that mixes collective communication, model computation (potentially using various distributed training/serving frameworks), and point-to-point data transfer. This code structure lacks modularity/function encapsulation, making the RLHF systems tightly coupled with specific LLM training and serving frameworks. Consequently, a user needs to implement and optimize different RLHF dataflows case-by-case [46], hindering code reuse and increasing the risk of making mistakes. Existing RLHF frameworks only support the PPO algorithm. In addition, limited parallel strategies are supported due to implementation complexity. For example, to incorporate 3D parallelism for LLM training and generation in DeepSpeed-Chat [82], one may have to re-implement the whole system due to the mixed code structure.

Inefficient RLHF execution. Table 1 summarizes parallelism strategies, model placement, and execution patterns adopted by the existing RLHF systems. DeepSpeed-Chat [82] and OpenRLHF [30] adopt ZeRO-3 for actor training and TP for actor generation. OpenRLHF uses different copies of the actor model on different devices for training and generation, incurring redundant memory usage and frequent weight synchronization among devices. DeepSpeed-Chat maintains the same copy of actor model on the same set of devices for training and generation, and reshards model weights between training and generation (due to different parallelisms used in the two stages), which may still incur substantial memory and communication overhead for large models (detailed in §5.4). NeMo-Aligner [17] uses the same 3D parallelism configurations in actor training and generation, experiencing low generation throughput (§8.4).

Existing RLHF frameworks are limited to one model placement plan and hence one RLHF execution pattern, as shown in Table 1. Implementing a different placement is difficult, requiring changing the inner logic of model initialization and inter-node data transfer as highlighted in blue in Figure 2. OpenRLHF and NeMo-Aligner allow concurrent model computation in the preparation and learning stages; in the generation stage, models except the actor are idle, wasting the GPUs they occupy. DeepSpeed-Chat colocates all models on the same set of devices, and each device runs each model sequentially according to the RLHF dataflow. With unbalanced workloads among the models, such a placement can be inefficient in resource utilization (evaluated in §8.3).

2.5 Design Considerations

To tackle limitations of existing systems, the key question is - **How to design a flexible and efficient programming model to implement RLHF dataflow?** A single-controller design is particularly advantageous at the inter-node level due to its flexibility in coordinating data transfer, execution order, and resource virtualization among distributed computation of different models [9, 50]. The RLHF dataflow graph typically consists of only a few nodes. Dispatching control messages to different nodes from the single-controller incurs negligible overhead as compared to distributed computation required for nodes (models) in the dataflow. The multi-controller paradigm, known for its low latency in dispatching operators to accelerators [20], can be leveraged in distributed computation of each model. With these insights, we propose a hierarchical hybrid programming model for RLHF dataflow implementation. Our key design principle is to combine single-controller and multi-controller paradigms in a hybrid manner. This design ensures flexible expression and efficient execution of RLHF dataflow, maintaining low control overhead at both inter-node and intra-node levels. As shown in Figure 2(b), this paradigm decouples intra-node distributed computation and inter-node data transfer, allowing each model to focus solely on local computation without managing inter-node communication.

3 HybridFlow Overview

Figure 4 depicts the architecture of HybridFlow, which consists of three major components: *Hybrid Programming Model*,

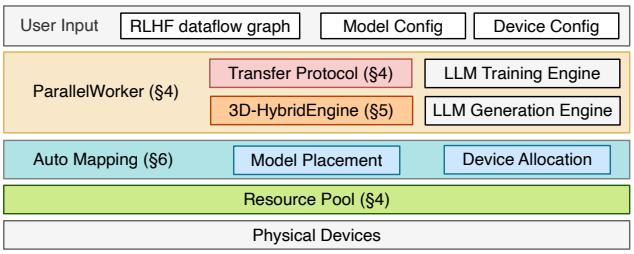


Figure 4. Architecture of HybridFlow.

3D-HybridEngine and *Auto-Mapping algorithm*. The hybrid programming model includes a set of hierarchical APIs to enable flexible expression of the RLHF dataflow and efficient computation of models in the dataflow (§4). The 3D-HybridEngine is particularly designed for efficient training and generation of the actor model, allowing different 3D parallel configurations in the two stages and enabling zero memory redundancy and minimized communication overhead during the transition between two stages (§5). The auto-mapping algorithm determines optimized device placement of each model to maximize the throughput of RLHF (§6).

The workflow of our RLHF system goes as follows. A user provides the following inputs to start the RLHF system: (i) model specifications, including the architecture and size of the actor/critic/reference policy/reward models in the RLHF dataflow; (ii) device placement of the models in the dataflow, as obtained by running the auto-mapping algorithm under given GPU cluster configurations; (iii) parallelism strategy for running each model in each stage, e.g., a tuple of (p, t, d) for 3D parallelism, where p, t, d represent PP size, TP size and DP size, respectively. The single controller program takes these inputs to initialize models in the RLHF dataflow and virtualized resource pool, dispatches operations/models to devices according to the placement plan, and invokes functions run by the multiple controllers on devices to carry out distributed computation of each model.

The multi-controller program implements the ParallelWorker class: it constructs parallel groups of each model among allocated devices according to its parallelism strategies, invokes the 3D-HybridEngine for actor training and generation, and can be integrated seamlessly with existing LLM engines [40, 57, 60, 71] for training, inference and generation of other models. The transfer protocols are coordinated by the single controller program to support resharding of data (including prompts, responses, and other model outputs in RLHF) between models with distinct parallelism strategies. The data resharding of the actor between training and generation is handled by 3D-HybridEngine.

4 Hybrid Programming Model

4.1 Hierarchical APIs

Intra-node: encapsulating distributed program. For distributed computation of each model in different RLHF stages, we provide a base class, `3DParallelWorker`. Given allocated

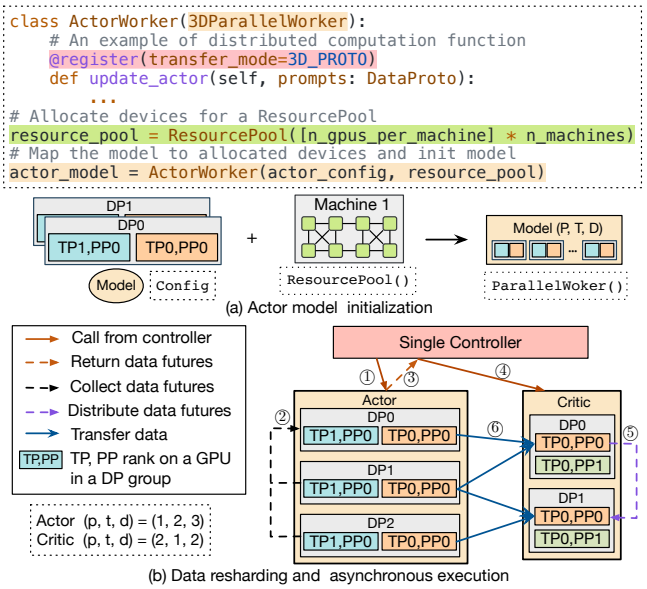


Figure 5. An illustration of hierarchical APIs. (a) Model with 3D parallel configuration, resource allocation, and `3DParallelWorker` initialization. (b) Asynchronous data resharding between two models with collect and distribute functions in `3D_PROTO`.

devices, it facilitates distributed model weight initialization and establishes 3D parallel groups for each model. A parallel group includes a set of GPUs to host a specific parallel dimension of the model, e.g., different tensor shards in TP and different model replicas in DP. Figure 5(a) illustrates initialization of the actor model with our APIs, while initialization of other models is similar.

Inheriting from the `3DParallelWorker` class, several model classes, for actor, critic, reference, and reward model, respectively, are provided. Each of these model classes encapsulates APIs to implement the model's distributed forward and backward computation, auto-regressive generation, and optimizer updates, decoupling the distributed computation code with data dependencies with other models. These APIs can be easily implemented by reusing the computation scripts from existing LLM systems. For example, the computation involved in `update_actor` function of `ActorWorker` (the class for the actor model) is similar to the pre-training scripts in Megatron-LM [71]. A model class encapsulates fundamental operations for implementing various RLHF algorithms, e.g., `generate_sequences` in the actor model class for generating responses based on the prompts and `compute_reward` in the reward model class for evaluating responses through a forward pass. (More APIs are detailed in Appendix A).

Besides base class `3DParallelWorker` that implements 3D parallelism, we further provide base classes for PyTorch FSDP (`FSDPWorker`) and ZeRO (`ZeROWorker`), and the corresponding model classes inheriting each base class, to support different parallelism strategies in model computation. ParallelWorker in Figure 4 denotes one of these base classes.

Inter-node: unifying data resharding implementation between models. Many-to-many multicast is involved for data transfer between models employing different parallelism strategies on different devices. We unify this data transfer implementation by associating each operation in each model class with a transfer protocol, using `@register`. Each transfer protocol consists of a collect function and a distribute function, to aggregate output data and distribute input data according to the parallelism strategy of each model. In the example in Figure 5(a), `update_actor` operation is registered to transfer protocol `3D_PROTO`, as 3D parallelism is used for actor training. In `3D_PROTO`, the collect function gathers all the output data of corresponding model function (e.g., the loss scalar return from the `update_actor`) in each DP group to the single controller, and the distribute function distributes the input data to the registered function (e.g., advantages for the `update_actor`) to each DP group. Data resharding is enabled using the source model's output collect function and the destination model's input distribute function. Figure 5(b) illustrates data resharding between the actor (generation) and the critic (inference), where computation of the models adopts different 3D parallelism strategies. The single controller gathers data futures using the collect function in `3D_PROTO` of actor (steps ①-③) and sends it to critic (step ④); critic distributes the received data futures to each DP group using the distribute function in its `3D_PROTO` (step ⑤). Then remote data is retrieved from actor to critic, with each of critic's GPUs only fetching the required local batch of the actor's output data according to its DP rank (step ⑥). The actual data transfer only occurs between GPUs, avoiding any central bottleneck.

We provide 8 transfer protocols, including `3D_PROTO`, `DP_PROTO`, `ONE_TO_ALL`, etc., that cover most data resharding scenarios (detailed in Appendix B). A user can further extend the transfer protocols through implementing customized collect and distribute functions.

Facilitating flexible model placement. We provide a `ResourcePool` class that virtualizes a set of GPU devices. When applying a `ResourcePool` instance to a model class (Figure 5(a)), distributed computation of the model will be mapped to the devices. Models utilizing the same `ResourcePool` instance are colocated on the same set of GPUs; models are placed on different sets of GPUs when different `ResourcePool` instances are applied in their model classes. We assume no overlap between different `ResourcePool` instances.

Asynchronous dataflow execution. When models are placed on separate sets of devices, their execution is triggered automatically as soon as their inputs become available [50]. In Figure 5(b), the data future from actor is immediately returned after the controller's call (steps ①-③); the controller then initiates a new call to critic and distributes the futures following the transfer protocol (steps ④-⑤). When some models are placed on the same set of devices, they are executed sequentially based on the calling order. With our programming model, HybridFlow is flexible in supporting diverse distributed execution patterns without any code change of the RLHF algorithm (Figure 6).

```
# Initialize cost model by reusing the RewardWorker
cost = RewardWorker(cost_config, resource_pool)
...  # omit other models initialization
algo_type = "Safe-RLHF" # specify different RLHF numerical computation.
# Examples of PPO and Safe-RLHF
for (prompts, pretrain_batch) in dataloader:
    # Stage 1: Generate responses
    batch = actor.generate_sequences(prompts)
    batch = actor.generate_sequences(prompts, do_sample=False)
    # Stage 2: Prepare experience
    batch = critic.compute_values(batch)
    batch = reference.compute_log_prob(batch)
    batch = reward.compute_reward(batch)
    batch = cost.compute_cost(batch)
    batch = compute_advantages(batch, algo_type)
    # Stage 3: Actor and critic training
    critic_metrics = critic.update_critic(batch, loss_func=algo_type)
    pretrain_loss = actor.compute_loss(pretrain_batch)
    batch["pretrain_loss"] = pretrain_loss
    actor_metrics = actor.update_actor(batch, loss_func=algo_type)
```

is added for ReMax
Not necessary in ReMax
is added for Safe-RLHF

Figure 6. Implementation of PPO [55], ReMax [43], and Safe-RLHF [19]. Users can adapt to different RLHF algorithms by simply adding or deleting a few lines of code.

4.2 Implementation of different RLHF algorithms

Our APIs enable streamlined development of various RLHF algorithms (dataflows). Users can implement an RLHF algorithm in a few lines of code as a single process program to run on the single controller, that involves a sequence of primitive API calls to invoke distributed computation of models. Examples of PPO, ReMax, and Safe-RLHF are given in Figure 6. PPO can be implemented in just 8 lines by invoking model operations including `compute_values` and `generate_sequences`, which are executed under the multi-controller paradigm on multiple GPUs. To adapt to Safe-RLHF which integrates an additional cost model to evaluate safety preferences and the pre-taining loss for actor, only 5 more lines of code are added on top of PPO implementation. To adapt to ReMax, one additional call to actor generation is needed, and the critic-related code can be removed.

Achieving flexible. This flexibility of extension is crucial for researchers to explore different RLHF algorithms: they can reuse distributed computation encapsulated in each model class and simply adjust the code for numerical computations according to specific algorithms, such as GAE [67] and KL divergence in `compute_advantage` and loss functions of actor and critic. The streamlined development can be attributed to the hybrid programming model. Our modular API design simplifies development, facilitates extensive code reuse, and enables directly incorporating the codebase of existing LLM training/serving frameworks. It also decouples model computation and data transfer among models. Any change in the distributed frameworks does not affect the code of the RLHF algorithm (Figure 6), enabling individualized optimization for each model's execution (§5). Flexible placement of models with diverse workloads is supported, enabling optimized mapping of RLHF dataflow onto various devices (§6).

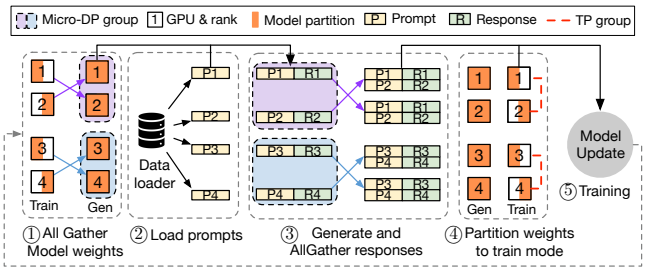


Figure 7. 3D-HybridEngine workflow in one RLHF iteration. 4 GPUs are used for actor training and generation. 1-2-2 (p-t-d) parallel groups are used in training and 1-1-2-2 (p_g-t_g-d_g-d) parallel groups are used in generation.

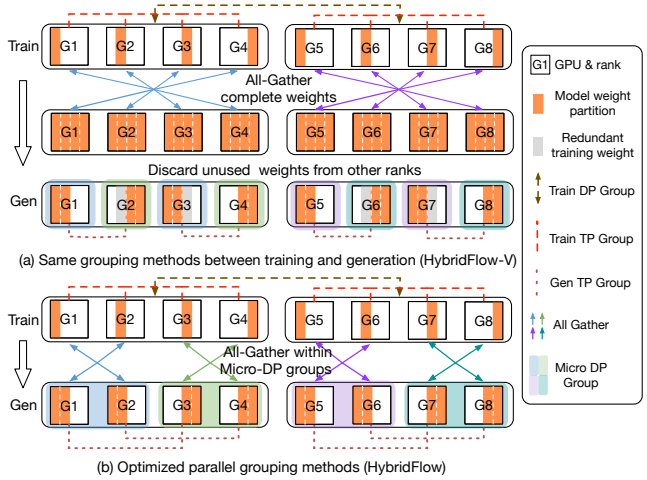


Figure 8. Model weights resharding. 2 machines each with 4 GPUs are used for actor training and generation.

5 3D-HybridEngine

We design the *3D-HybridEngine* to support efficient training and generation of the actor model, targeting significant RLHF throughput improvement.

5.1 Parallel Groups

To eliminate redundant actor model copies, we advocate deploying actor training and generation stages on the same set of devices, N_a GPUs allocated to the actor, and execute them sequentially on the same copy of actor model weights. Nonetheless, actor training and generation may well adopt different 3D parallelism strategies, i.e., the generation stage typically requires smaller TP and PP sizes but a larger DP size, than the training stage (§2.3). 3D-HybridEngine enables efficient model parameter resharding between actor training and generation across the same set of devices in this context.

Let p-t-d denote 3D parallel groups constructed for actor training, corresponding to the set of GPUs to host p pipeline stages, t tensor shards, and d model replicas [54]. 3D-HybridEngine builds different parallel groups for actor training and generation, according to their different 3D parallelism strategies, respectively. We use p_g, t_g, and d_g to denote the size of generation pipeline parallel group, generation tensor parallel group, and micro data parallel group, respectively, in the generation stage. d_g indicates the ratio of model replica number in generation over that in training, i.e., each DP replica in training becomes d_g micro DP replicas, to process d_g microbatches of prompts and responses. We have N_a=p×t×d=p_g×t_g×d_g×d such that $d_g = \frac{pt}{p_g t_g}$. The micro DP groups are employed exclusively in actor generation stage to render a larger DP size for full device utilization. The generation parallel groups are denoted by p_g-t_g-d_g-d.

5.2 3D-HybridEngine Workflow

Between actor training in iteration i of RLHF and actor generation in iteration $i+1$, the actor model parameters need to be resharded and prompts data to be distributed, following the parallel group configurations in the two stages. In iteration $i+1$ of RLHF, 3D-HybridEngine gathers the actor model parameters updated in iteration i (step ① in Figure 7), for generation within each micro DP group. Then, the batch of prompts are loaded to each model replica (step ②), which generates responses (Generation stage of RLHF). Following this, 3D-HybridEngine performs an all-gather operation on the generation results within each micro DP group (step ③), and re-partitions model parameters according to the 3D parallelism for actor training (step ④). With model weights, prompts and responses correctly re-distributed, the loss of the actor model is computed and actor model weights are updated following the RLHF algorithm (step ⑤) - actor training stage of iteration $i+1$.

5.3 Zero redundancy model resharding

Parallel grouping methods in 3D parallelism are typically as follows: PP and TP groups are formed by assigning consecutive ranks to pipeline stages and tensor shards, respectively; DP groups are constructed by selecting ranks at regular intervals, determined by the product of PP size and TP size. In Figure 8(a), actor training uses 3D parallel groups, 1-4-2: there is one PP group for all GPUs (for illustration clarify); the TP groups are [G1, G2, G3, G4], [G5, G6, G7, G8], and the DP groups are [G1, G5], [G2, G6], [G3, G7], [G4, G8]. Suppose the same parallel grouping methods are used but with different parallel sizes, e.g., 1-2-2-2 for generation in Figure 8(a). During the transition from training to generation, 3D-HybridEngine applies all-gather operations among the model parallel groups to aggregate all parameters, and then retain only a subset of model weights on each device for its generation, according to the parallel groups the device belongs to. On some GPUs (e.g., G2, G3, G6, G7), there is no overlap between training and generation model weights, and separate memory is needed to maintain weights for subsequent training as well (grey boxes in Figure 8(a)).We call

Table 2. Transition overhead between training & generation

	DS-Chat	HybridFlow-V	HybridFlow
Comm. Vol	$\frac{tpd-1}{tpd}M$	$\frac{tp-1}{tp}M$	$\frac{tp-t_gp_g}{t_gp_gtp}M$
Peak Mem.	M	M	$\frac{1}{t_gp_g}M$
Redundancy	$\frac{1}{tpd}M$	$\frac{1}{tp}M$	0

the system HybridFlow-V, when 3D-HybridEngine uses the above vanilla parallel grouping methods in the two stages.

We further design a new parallel grouping method for 3D-HybridEngine to use in the generation stage, that eliminates the redundancy in weights storage and leads to minimal memory footprint and communication due to actor model resharding between training and generation. Specifically, we form generation TP and PP groups by selecting ranks at regular intervals, determined by $\frac{t}{t_g}$ and $\frac{p}{p_g}$, and construct micro DP groups by sequentially assigning ranks along the generation TP or PP dimensions. In Figure 8(b), 1-2-2-2 parallel groups are used in generation: the generation TP groups are [G1, G3], [G2, G4], [G5, G7], [G6, G8]; and the micro DP groups are [G1, G2], [G3, G4], [G5, G6], [G7, G8]. This strategic rearrangement of generation parallel groups leads to overlap between training and generation model weights on each device, enabling reuse of training weights during generation and *zero redundancy* in device memory usage due to model resharding. In addition, 3D-HybridEngine conducts several all-gather operations concurrently, one within each micro DP group, leading to significantly reduced communication overhead.

5.4 Transition overhead

In Table 2, we compare communication overhead and memory footprint during the transition between training and generation stages, among different actor engine designs. We assume model size of the actor is M and N_a GPUs are used for its training and generation. The actor engine in DeepSpeed-Chat conducts an all-gather operation across all GPUs during transition; HybridFlow-V performs this all-gather within training TP and PP groups. The communication volumes for these operations are $\frac{N_a-1}{N_a}M = \frac{tpd-1}{tpd}M$ for DeepSpeed-Chat and $\frac{tp-1}{tp}M$ for HybridFlow-V, calculated following [13]. Both engines aggregate all model parameters in each GPU's memory before subsequently partitioning model states according to the generation parallel groups, resulting in a peak memory usage of model parameters M. As they cannot reuse training weights during generation on some GPUs, training weights need to be maintained on them, amounting to $\frac{1}{tpd}$ and $\frac{1}{tp}$ redundant memory consumption, respectively.

With our parallel grouping method for the generation stage, HybridFlow confines the all-gather operation within each micro DP group. The communication overhead is reduced to $\frac{d_g-1}{tp}M = \frac{tp-t_gp_g}{t_gp_gtp}M$. Each GPU only needs to collect remote parameters within its micro DP group and can reuse the training weights in generation. Therefore, the peak memory usage of model parameters in HybridFlow precisely matches the model partition size on each GPU in generation, eliminating any redundancy in GPU memory usage.

6 Auto Device Mapping

Our hybrid programming model requires users to input the following configurations, which are referred to as a *mapping* of the RLHF dataflow to the given devices: (a) device placement of the models in the dataflow; (b) the corresponding parallelism strategy for running each model in each stage.

We provide an efficient algorithm (Algorithm 1) for users to identify the optimized mapping of executing the RLHF dataflow on a given cluster of devices, that minimizes the end-to-end latency of each RLHF iteration. Given a dataflow D, we first explore all possible placement plans $\mathcal{P}$ for the models in the given cluster (Line 3). For example, the PPO algorithm involves four models, resulting in 15 possible placements (from the Bell partition problem [10, 62]), ranging from a completely standalone placement where all models are placed on different devices (e.g., OpenRLHF's placement) to colocating all models on the same set of devices (e.g., DeepSpeed-Chat's placement). We refer to colocated models on the same set of GPUs as a colocated *set*. Models in a colocated set can employ different parallelism strategies across the same set of GPUs. We identify the smallest number of GPUs to be allocated to each of the colocated model sets, A_{min}, based on memory consumption of colocated models, ensuring no out-of-memory errors (Line 9).

Next, starting from the minimal GPU allocation in A_{min}, we enumerate all feasible device allocations to each colocated model set (Lines 10-12). Given device allocation A to the colocated set and computation workload W of models in the set, we explore optimized parallelism strategies for each model in the `auto_parallel` module, that minimizes model execution latency. The workload W includes input and output shapes and computation (training, inference or generation) of each model. In `auto_parallel`, we utilize a simulator module `simu` to estimate the latency of different parallel strategies, following previous research [42, 84, 90, 92] (outline in Appendix. C).

The `d_cost` module estimates the end-to-end latency of the RLHF dataflow under given model placement and parallelism strategies, by iterating through all stages in the dataflow graph and summing up latencies of all stages (Lines 17, 25). For models in the same colocated set and involving computation in the same stage (such as actor and critic both performing model update in RLHF training stage), their execution latencies are summed up (Line 32). For models in different colocated sets, their execution within the same stage

Algorithm 1 Device Mapping for an RLHF Dataflow

1: **Input:** RLHF dataflow graph D, LLMs in RLHF dataflow L=$[l_1, l_2, \ldots, l_k]$, workload W of LLMs in RLHF dataflow, total # of GPUs N, memory capacity per GPU Q
2: **Output:** device mapping of models in RLHF dataflow
3: $\mathcal{P} \leftarrow$ get_placements(D, L, N)
4: $C^* \leftarrow \infty$
5: $best\_mapping \leftarrow \emptyset$
6: **for all** $plm \in \mathcal{P}$ **do**
7: $\quad C_{plm} \leftarrow \infty$
8: $\quad best\_plm\_alloc \leftarrow \emptyset$
9: $\quad A_{min} \leftarrow$ get_min_alloc(plm, Q, N)
10: $\quad$ **for all** $A \in$ enum_alloc(N, A_{min}) **do**
11: $\quad\quad \widehat{L} \leftarrow []$
12: $\quad\quad$ **for all** set $\in plm$ **do**
13: $\quad\quad\quad$ **for all** $l \in$ set **do**
14: $\quad\quad\quad\quad \widehat{l} \leftarrow$ auto_parallel(A, A_{min}, l, W)
15: $\quad\quad\quad\quad \widehat{L}$.append$(\widehat{l})$
16: $\quad\quad plm$.update$(\widehat{L})$
17: $\quad\quad C_{alloc} \leftarrow$ d_cost(D, plm, W)
18: $\quad\quad$ **if** $C_{alloc} < C_{plm}$ **then**
19: $\quad\quad\quad C_{plm} \leftarrow C_{alloc}$
20: $\quad\quad\quad best\_plm\_alloc \leftarrow (plm, A)$
21: $\quad$ **if** $C_{plm} < C^*$ **then**
22: $\quad\quad C^* \leftarrow C_{plm}$
23: $\quad\quad best\_mapping \leftarrow best\_plm\_alloc$
24: **return** *best_mapping*
25: **Procedure** d_cost(D, plm, W):
26: $\quad s \leftarrow$ number of stages in D
27: $\quad c \leftarrow [0] \times s$ *// Initialize latency for each stage to 0*
28: $\quad$ **for all** set $\in plm$ **do**
29: $\quad\quad c_g \leftarrow [0] \times s$
30: $\quad\quad$ **for all** $i \in \{0, ..., s-1\}$ **do**
31: $\quad\quad\quad$ **for all** $\widehat{l} \in$ set **do**
32: $\quad\quad\quad\quad c_g[i] \leftarrow c_g[i] +$ simu$(\widehat{l}, W[i])$
33: $\quad\quad\quad c[i] \leftarrow max\{c[i], c_g[i]\}$
34: $\quad$ **return** sum(c)

can be parallelized, and the latency of the stage is determined by the maximum execution time among different sets (Line 33). We identify the best device placement of the models with their corresponding parallelism strategies, achieving minimal execution time per RLHF iteration (Lines 18-23).

The complexity of Algorithm 1 is $O(\frac{(N-1)!}{(k-1)!(N-k)!})$, where k is the number of models in the dataflow and N is the total number of devices to run the dataflow. This is the worst-case complexity for enumerating all possible device allocations for a placement strategy (i.e., the standalone placement), calculated by assigning N devices to k models (known as the integer partition problem [6]). For better efficiency, we cache parallelism strategies identified for each model on a number of devices A, to eliminate redundant searches for the same parallelism strategies when the model is placed on different sets of A GPUs in different placement strategies.

Though we assume N homogeneous GPUs when running the auto mapping algorithm, Algorithm 1 can be readily extended for optimizing model mapping over heterogeneous devices, by considering heterogeneous devices in `simu` and `auto_parallel` modules [88].

7 Implementation

HybridFlow is implemented in around 12k lines of Python code (LoC).

Hybrid programming model. The hierarchical APIs are implemented with 1.8k LoC. The centralized single controller is built on top of Ray [50] and uses Remote Process Calls (RPC) to coordinate the execution order of different models and transfer data between models following the dataflow. These intermediate data are stored in TensorDict [57]. In our multi-controller paradigm for distributed computation, each model function runs on a separate process across various devices, with control messages relayed from each controller's CPU process to the corresponding GPU. Our implementation supports Megatron-LM, PyTorch FSDP, and DeepSpeed as the LLM training and inference engines, and vLLM for auto-regressive generation. In vLLM, we replace the centralized KVCache manager with a distributed manager to align with the multi-controller paradigm.

3D-HybridEngine. Its main logic is implemented with 2.4k LoC on top of Megatron-LM and vLLM. We store actor model weights for training and generation stages on separate memory buffers, offload generation weights to the CPU memory during training, reload generation weights back to GPU memory during the transition, and use both buffers in generation. We use NCCL communication primitives [35] to collect and concatenate model parameters in each micro DP group during the transition between training and generation. We offload KVCache to CPU memory after generation and reload it back to GPU in the next iteration.

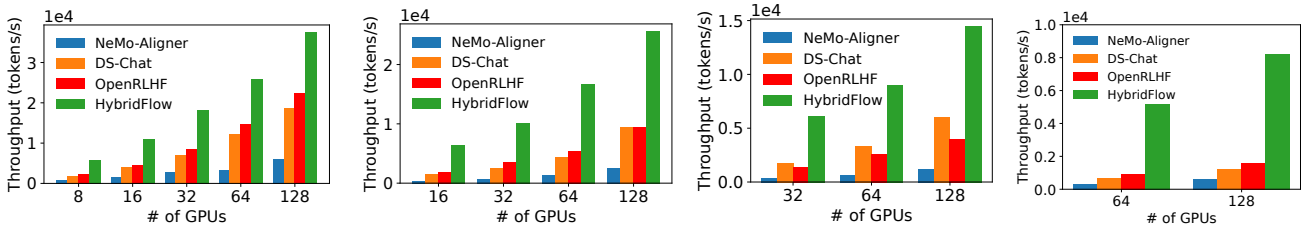
Auto-Mapping Algorithm is implemented with 1.9k LoC, together with three simulators for training, inference, and generation workloads. The algorithm is run before starting the RLHF dataflow on CPU, to generate device mapping and parallelism strategies for dataflow initialization.

8 Evaluation

8.1 Experimental Setup

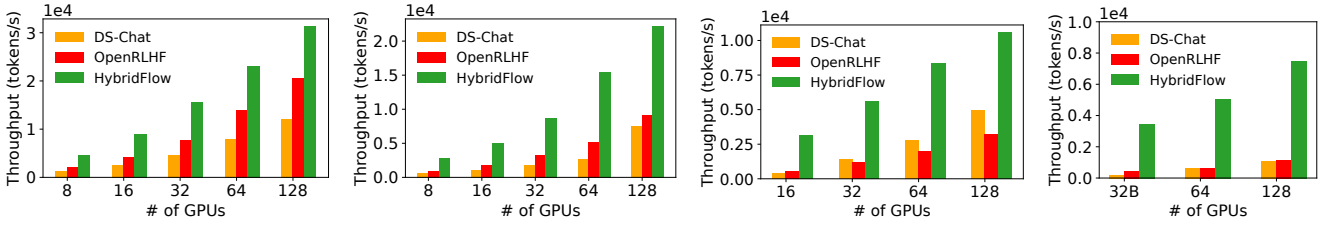
Testbed. We deploy HybridFlow on a cluster of 16 machines (128 GPUs). Each machine is equipped with 8 NVIDIA A100-80GB GPUs inter-connected with 600GB/s NVLink. The inter-machine bandwidth is 200Gbps. Our experiments use the following software versions: CUDA12.1, PyTorch 2.1.2, Megatron-core 0.6.0, NCCL 2.18.1, and vLLM 0.3.1.

Models and RLHF algorithms. We run the RLHF dataflow (Figure 1) of PPO [68], ReMax [43] and Safe-RLHF [19] algorithms. PPO is one of the most popular algorithms for RLHF [7, 55], consisting of actor, critic, reference policy, and



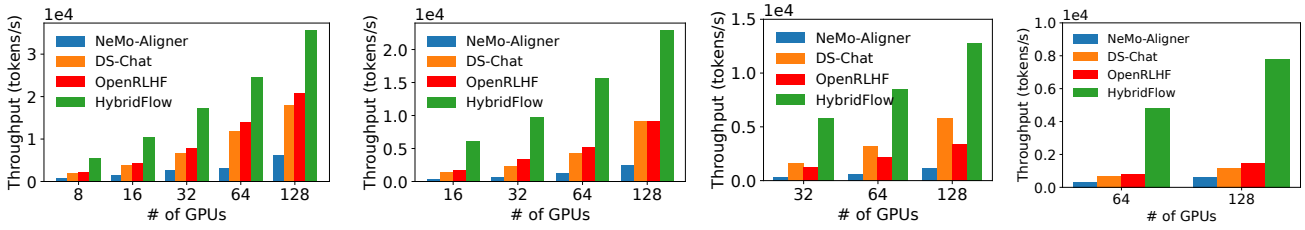
(a) 7B (1.68×~8.63×) (b) 13B (2.70×~18.96×) (c) 34B (2.41×~20.57×) (d) 70B (5.17×~17.98×)

Figure 9. PPO throughput. Numbers in parentheses are HybridFlow speedups compared with baselines.



(a) 7B (1.53×~2.56×) (b) 13B (2.49×~3.66×) (c) 34B (2.14×~4.80×) (d) 70B (6.46×~9.78×)

Figure 10. ReMax throughput. Numbers in parentheses are HybridFlow speedups compared with baselines



(a) 7B (1.71×~12.87×) (b) 13B (2.49×~18.47×) (c) 34B (2.20×~19.76×) (d) 70B (4.89×~16.86×)

Figure 11. Safe-RLHF throughput. Numbers in the parentheses are HybridFlow speedups compared with the baselines

reward models. Each model is a Llama [73] model with sizes ranging from 7B to 70B. Safe-RLHF has an additional cost model whose architecture and size are the same as the reward model and ReMax eliminates the critic model. We use mixed precision for actor and critic training, i.e., BF16 for model parameters and FP32 for gradient and optimizer states, with Adam [38] optimizer in all experiments. BF16 is used in model inference and auto-regressive generation. If not specified, the experiment results are obtained from PPO.

Baselines. We compare HybridFlow with state-of-the-art RLHF systems including DeepSpeed-Chat [82] v0.14.0, OpenRLHF [30] v0.2.5, and NeMo-Aligner [17] v0.2.0 (detailed in Table 1). NeMo-Alginer doesn't support ReMax algorithm. We do not compare HybridFlow to other frameworks such as Trlx [27], HuggingFaceDDP [79], and Collosal-Chat [15] as they are less representative and slower than the above baselines (as reported in [82]).

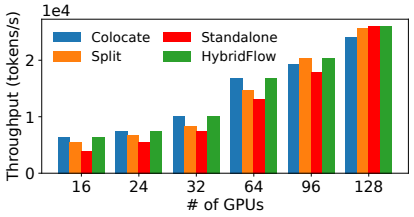
We use RLHF throughput (tokens/sec) as the performance metric, computed by dividing the total number of tokens in prompts and responses in a global batch by one RLHF iteration time. All reported performance numbers are averaged over 5 training iterations after a warm-up of 10 iterations.

Datasets and hyperparameters. We perform RLHF on "Dahoas/ful-hh-rlhf" dataset [7] of HuggingFace, which is widely used for LLM alignment [64, 85]. As the baseline systems may not incorporate continuous-batching optimization [83] during generation, for a fair comparison, we enforce the same length on all responses to be generated. In each experiment, the input prompt length and the output response length are both 1024 and the global batch size of input prompts to the actor model is 1024. The number of PPO epochs is 1 and the number of PPO update iterations per epoch is 8, aligning with previous RLHF research [31, 55, 81].

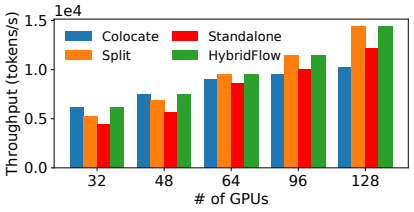
8.2 End-to-End performance

Figures 9, 10, and 11 show RLHF throughput when running PPO, ReMax, and Safe-RLHF respectively. The actor, critic, reference, and reward models in this set of experiments are of the same size, following previous practice [7, 55, 82]. The number of GPUs used in experiments of different model sizes ranges from the smallest number of GPUs to run RLHF without OOM to 128 GPUs. We do not enable offloading optimizer states [61] in the experiments for fair comparison.

Overall performance. We observe that HybridFlow consistently outperforms the baselines across all model scales. In



(a) 13B



(b) 34B

Figure 12. Throughput of HybridFlow under different placements

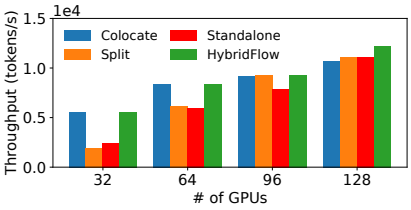


Figure 13. Placement comparison under 13B actor and reference policy & 70B critic and reward model.

Figure 9 for PPO, HybridFlow outperforms DeepSpeed-Chat, OpenRLHF and NeMo-Aligner by 3.67× (up to 7.84×), 3.25× (up to 5.93×) and 12.52× (up to 20.57×), respectively. This is mainly because HybridFlow effectively executes generation, inference, and training in all RLHF stages by sharding the models with different parallelism strategies to fit various computation workloads. HybridFlow achieves the highest average speedup of 9.64× when training 70B models, as HybridFlow reduces the transition overhead by up to 71.2% and 89.1% compared to DeepSpeed-Chat and OpenRLHF, which also incurs large inter-machine communication when training with ZeRO-3. Due to the lack of KVCache in generation engine, NeMo-Aligner's main performance bottleneck lies in the generation stage, which accounts for up to 81.2% of its RLHF iteration time. Similar results can be observed in Figures 10, 11 validating the efficiency of HybridFlow on running various RLHF algorithms.

Scalability. HybridFlow achieves at least 2.09× speedup on 8 GPUs. With increasing GPUs, the strong scaling efficiency of HybridFlow on various model scales is 66.8%, computed by dividing $\frac{\text{throughput in largest scale}}{\text{throughput in smallest scale}}$ by $\frac{\text{max. \# of GPUs}}{\text{min. \# of GPUs}}$ [5], averaging over three algorithms and all model scales. Scaling to a large number of GPUs with a fixed global batch size results in smaller local batch sizes for each worker, potentially causing GPU underutilization. Running 7B models on 128 GPUs, HybridFlow still outperforms the best baseline OpenRLHF for 1.68×, 1.53×, and 1.71× on PPO, ReMax, and Safe-RLHF respectively. This can be attributed to HybridFlow's ability to adapt the best placement strategies for different models and cluster sizes to minimize RLHF time. OpenRLHF performs better in a larger GPU cluster but less efficiently on smaller ones.

8.3 Model Placement

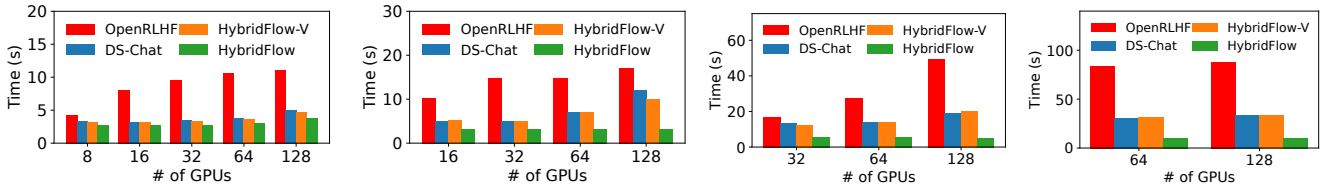
In this experiment, we implement various model placements of the PPO algorithm in HybridFlow, under the same model and cluster settings as in Sec. 8.2: (i) *colocate*, the placement strategy in DeepSpeed-Chat; (ii) *standalone*, that in OpenRLHF and; (iii) *split*, NeMo-Aligner's colocation placement (actor and reference policy on the same set of devices and critic and reward model on another); (iv) *hybridflow*, the optimized placement obtained by Algorithm 1.

Comparison of different model placements. Figure 12 reveals that optimized placement of HybridFlow under different numbers of GPUs varies. From 16 to 64 GPUs, colocating all models on the same set of devices yields the best performance. For 96 to 128 GPUs with 34B models and 96 GPUs with 13B models, the split strategy becomes optimal. The split strategy divides GPUs evenly between the two sets of models, as their sizes are equal. For 13B models on 128 GPUs, the standalone strategy achieves the highest throughput. In this case, HybridFlow allocates 64 GPUs for the actor, 32 for the critic, and 16 each for the reference and reward model. In smaller clusters, computation of all models can fully utilize GPU resources; the colocate strategy ensures maximum GPU usage in different RLHF stages. In larger clusters, RLHF throughput under colocate placement fails to scale up linearly as the batch size is fixed and the computation-to-communication ratio decreases with a larger DP size on more GPUs. Standalone and split strategies place models on different devices with a smaller DP size for each model in larger clusters, facilitating parallel execution of different models in the same stages. In all cases, our Algorithm 1 produces the best placement with the highest training throughput.

Larger critic and reward model. We further evaluate model placements when running PPO with a 13B actor and reference policy and 70B critic and reward models (larger critic and reward models are expected to produce better alignment [7]). Figure 13 shows that the colocate strategy still outperforms others by 44.8% on average with up to 64 GPUs. The split strategy achieves higher throughput with 96 GPUs. When scaling to 128 GPUs, the best placement obtained by Algorithm 1 colocates actor, reference, and reward models on 64 GPUs while allocating the remaining 64 GPUs to critic. On the same number of GPUs, actor and reference policy's computation time is much smaller than critic and reward model, and colocating the reward model with actor and reference policy reduces the GPU idle time in the experience preparation stage. In general, distributing actor and critic on different devices for parallel execution in the training stage leads to higher throughput in large clusters.

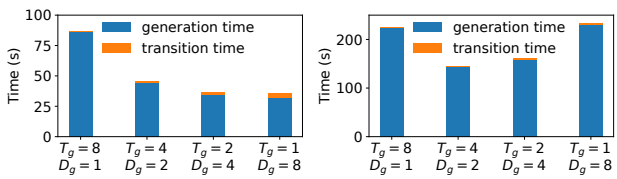
8.4 3D-HybridEngine

Transition time comparison. Figure 14 shows the transition time between actor training and generation stages on



(a) 7B (T_g=2, P_g=1, T=8,P=1) (b) 13B (T_g=4, P_g=1, T=8,P=1) (c) 34B (T_g=8, P_g=1, T=8,P=4) (d) 70B (T_g=8, P_g=1, T=8,P=8)

Figure 14. Transition time between actor training and generation.



(a) 7B (b) 13B

Figure 15. Time breakdown on different generation parallel sizes of the actor model on 16 GPUs.

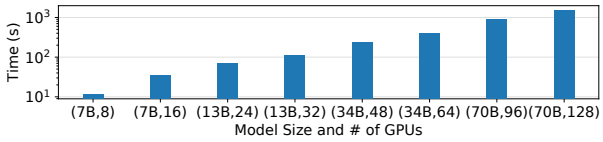


Figure 16. Runtime of device mapping algorithm. The model size and # of GPUs are simultaneously scaled.

various model scales, which is the time to reshard model weights from training to generation, under the same settings in §8.2. OpenRLHF's transition time includes weight synchronization time between two copies of the actor model on different devices. HybridFlow reduces the transition time by 55.2% (11.7s) on average and the transition overhead by up to 89.1% (78.2s) with 70B models, while maintaining consistent overhead across different cluster scales. This is attributed to our new parallel grouping method for the generation stage (§5.4). In baseline methods, all model parameters must be collected during transition, necessitating layer-by-layer collections multiple times to prevent OOM. HybridFlow enables zero memory redundancy during transition and requires only one all-gather operation per micro DP group.

Transition and generation time We further validate the need to use different parallel sizes in actor training and generation in HybridFlow. In this experiment, all models are colocated on the same set of GPUs, and the KVCache for generation is allocated using the remaining GPU memory (i.e., best-effort allocation). Figure 15 gives the transition and generation time when running RLHF on 16 GPUs with 7B and 13B models, respectively, with training parallel groups 1-8-2 (following p-t-d convention) and varying generation TP group size t_g from 1 to 8. The generation PP group size remains constant at p_g=1 and the micro DP group size d_g is computed as $\frac{8}{t_g}$. We observe that applying a smaller generation TP group size, t_g=2, for 7B models and t_g=4 for 13B models reduces the generation latency by 60.3% and 36.4%, respectively. Conversely, using the same TP size as training (t_g=8), following the NeMo-Aligner approach, results in the largest generation latency due to GPU underutilization. Further reducing t_g fails to achieve higher speedup, as a smaller t_g necessitates maintaining a larger KVCache per GPU.

8.5 Algorithm Runtime

Figure 16 shows the running time of Algorithm 1, which is significantly shorter than days of actual RLHF training. A linear growth of running time is exhibited, revealing good scalability of the device mapping algorithm with model size and cluster size. Most of the running time is spent on estimating the execution latency of each model's parallel strategies. More parallelism strategies are available for a larger model, requiring more simulations to identify the optimal one for each placement plan. Our caching of optimal parallelism strategies of the models to be reapplied across different placements reduces the search time for the best placement to at most half an hour.

9 Discussions

Fault Tolerance. HybridFlow is orthogonal to existing fault-tolerance approaches [22, 34, 49, 76, 93] and already incorporates checkpointing. Failures can be detected by NCCL errors and silent-data-corruption by checksums. Our programming model enables the single controller to coordinate checkpoint operations via RPC, allowing the saving of model states within each ParallWorker Group. This includes saving parameters of actor/critic models, dataloader IDs, and Random Number Generator (RNG) states to ensure system-wide consistency. Moreover, HybridFlow can also employ redundancy-based fault-tolerance methods, such as broadcast parameters and CPU checkpoint, for fast recovery if enough healthy model replicas are available [76, 93].

Placement Insights. We conclude three main insights for model placement and GPU allocation in RLHF training. **1)** Allocating more GPUs to the actor model can reduce the time-consuming generation latency, which cannot be parallelized with other models. **2)** When each model computation can fully utilize GPU resources, colocating all the models is most effective when training on relatively small-scale clusters. **3)** When scaling up to large-scale clusters (i.e., strong scaling), distributing the actor and critic models on different devices

for parallel execution in the training and preparation stages would help achieve higher throughput.

Resource multiplexing. HybridFlow enables colocation of models on shared devices by utilizing time-sharing for GPU computation. Recent research in DNN task scheduling has developed fine-grained resource multiplexing techniques, primarily aimed at achieving the service-level objectives of individual tasks [8, 18, 26, 26, 47, 56, 77]. Although the `ResourcePool` implementation supports parallel execution of collocated models, HybridFlow generally adheres to sequential execution to prevent GPU resource contention or OOM issues as discussed in Section 2.3. Applying GPU sharing and heterogeneous resources in RLHF training poses distinct challenges, as it seeks to balance the computation workload and manage complex data dependencies among various tasks. Investigating fine-grained auto-mapping algorithms for GPU sharing in RLHF training, coupled with model offload optimization and integration of heterogeneous devices, would be a promising direction for future research.

From alignment to reasoning. In RLHF for LLM alignment, the reward signal is generated by the reward model. Besides alignment tasks, similar algorithms (e.g., PPO and GRPO [70]) can be applied to other domains, such as code generation and mathematical reasoning. For these tasks, a ground truth may exist for each prompt, which can be determined by assessing the correctness of the output value for each code test case and verifying the accuracy of mathematical results. Therefore, the reward model can be replaced by non-neural-network reward modules, such as a sandbox environment [87] for evaluating generated code or a reward function [14, 65] to validate mathematical results. HybridFlow can seamlessly integrate these reward modules by wrapping them as remote functions and orchestrating their execution within the single-process script, providing a flexible and efficient framework for diverse reinforcement learning applications.

10 Related Work

RL frameworks. There have been plenty of frameworks for RL, ranging from general-purpose RL systems design for small-scale DNNs [12, 25, 28, 39, 45, 46] to RLHF systems specifically optimized for LLMs [15, 17, 30, 80, 82]. We have thoroughly examined closely related work in §2 and we discuss more RL frameworks in this section. These RL frameworks [12, 25, 28, 39, 74], similar to recent RLHF systems, use a hodgepodge of multi-controller frameworks to implement their algorithms. They establish multiple long-running distributed programs with each component coordinating the execution order with hard-coded data synchronization. Gear [74] further optimized the experience replay segment of the RL pipeline. However, all these frameworks fail to support LLM training, inference, and generation in RLHF.

LLM training and serving systems. TorchDDP [57] and Horovod [69] support data parallel training. ByteScheduler [58] and DeepSpeed [60] extend data parallelism with communication and memory optimizations. Numerous systems [23, 36, 48, 54, 71, 75, 89] optimized large model training through model parallelisms such as tensor parallelism and pipeline parallelism to partition models across devices. LLM serving systems [3, 16, 40, 72, 83, 92] also adopts data and model parallelism to accelerate auto-regressive generation with specialized optimizations like continuous-batching [83] and chunked-prefill [3]. Note that all the above frameworks adopt multi-controller paradigm for efficient computation.

Dataflow systems. Dataflow systems like MapReduce [21], Spark [86], Dryad [33], and Naiad [51] are popular for analytics and ML workloads but they lack support for dynamic task graphs. Ray [50] unifies task-parallel and actor programming models in a single dynamic task graph and implements a scalable distributed scheduler and a global control store, which is adopted by many RL frameworks [45, 46]. Pathways [9], a closed-source project for TPUs, are designed to easily express complex parallelism patterns and fine-grain control flow within a single DNN model, such as pipeline parallelism and Mixture-of-Experts with sparse computation. It employs an asynchronous distributed dataflow design that enables parallel control plane execution despite data dependencies, reducing the dispatch overhead from single-controller paradigm. Its main focus lies on single-model training, requiring complex compilations of each sub-network of a DNN model. HybridFlow can integrate Pathways as a submodule to implement the computation of models in the RLHF dataflow.

11 Conclusion

HybridFlow is an RLHF framework that enables flexible representation and efficient execution of diverse RLHF algorithms. We propose a hybrid programming model that allows users to easily build RLHF dataflow in a few lines of code by encapsulating distributed computation of different LLMs into primitive APIs and hiding the complexity of data resharding among nodes. Our 3D-HybridEngine ensures efficient execution of training and generation of the actor model, with zero memory redundancy and significantly reduced communication overhead for model parameter resharding. Furthermore, our effective mapping algorithm optimizes GPU allocation and placement of models in the RLHF dataflow. Extensive experiments demonstrate that HybridFlow achieves 1.53× to 20.57× speedup compared to state-of-the-art RLHF systems under various model sizes and cluster scales.

Acknowledgments

We would like to thank our shepherd Y. Charlie Hu and the anonymous reviewers for their constructive feedback. We thank Xin Liu, Yangrui Chen, and Ningxin Zheng for their insightful feedback on this project. This work was supported in part by a ByteDance Research Collaboration Project, and grants from Hong Kong RGC under the contracts HKU 17204423 and C7004-22G (CRF).

References

[1] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN international conference on functional programming*. 1–1.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[4] Riad Akrour, Marc Schoenauer, and Michele Sebag. 2011. Preference-based policy learning. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011. Proceedings, Part I 11*. Springer, 12–27.

[5] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[6] George E Andrews and Kimmo Eriksson. 2004. *Integer partitions*. Cambridge University Press.

[7] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862* (2022).

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[9] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. 2022. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems* 4 (2022), 430–449.

[10] Eric Temple Bell. 1934. Exponential polynomials. *Annals of Mathematics* (1934), 258–277.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[12] I. Caspi. 2017. *Reinforcement learning coach by Intel.* https://github.com/NervanaSystems/coach

[13] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783.

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[15] Collosal-AI Corporation. 2023. *Collosal-Chat.* https://github.com/binmakeswell/ColossalChat

[16] NVIDIA Corporation. 2023. *TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference.* https://github.com/NVIDIA/TensorRT-LLM

[17] NVIDIA Corporation. 2024. *NeMo-Aligner: Scalable toolkit for efficient model alignment.* https://github.com/NVIDIA/NeMo-Aligner

[18] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. {DVABatch}: Diversity-aware {Multi-Entry}{Multi-Exit} batching for efficient processing of {DNN} services on {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.

[19] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2024. Safe RLHF: Safe Reinforcement Learning from Human Feedback. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=TyFrPOKYXw

[20] Frederica Darema. 2001. The spmd model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23–26, 2001 Proceedings 8*. Springer, 1–1.

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[22] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[23] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[24] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 503–521.

[25] Danijar Hafner, James Davidson, and Vincent Vanhoucke. 2017. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878* (2017).

[26] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[27] Alexander Havrilla, Maksym Zhuravinskyi, Duy Phung, Aman Tiwari, Jonathan Tow, Stella Biderman, Quentin Anthony, and Louis Castricato. 2023. trlX: A framework for large scale reinforcement learning from human feedback. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 8578–8595.

[28] C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. 2017. *OpenAI baselines.* https://github.com/openai/baselines

[29] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv preprint arXiv:2401.08671* (2024).

[30] Jian Hu, Xibin Wu, Xianyu, Chen Su, Leon Qiu, Daoning Jiang, Qing Wang, and Weixun Wang. 2023. OpenRLHF: A Ray-based High-performance RLHF framework. https://github.com/OpenLLMAI/OpenRLHF.

[31] Shengyi Huang, Michael Noukhovitch, Arian Hosseini, Kashif Rasul, Weixun Wang, and Lewis Tunstall. 2024. The N+ Implementation Details of RLHF with PPO: A Case Study on TL; DR Summarization. *arXiv preprint arXiv:2403.17031* (2024).

[32] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems 2007*. 59–72.

[34] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 382–395.

[35] Sylvain Jeaugey. 2017. Nccl 2.0. In *GPU Technology Conference (GTC)*, Vol. 2. 23.

[36] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. *arXiv preprint arXiv:2402.15627* (2024).

[37] Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. 2023. A survey of reinforcement learning from human feedback. *arXiv preprint arXiv:2312.14925* (2023).

[38] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[39] I. Kostrikov. 2017. *PyTorch implementation of advantage actor critic (A2C), proximal policy optimization (PPO) and scalable trust-region method for deep reinforcement learning*. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr

[40] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[41] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267* (2023).

[42] Cheng Li. 2023. *LLM-Analysis: Latency and Memory Analysis of Transformer Models for Training and Inference*. https://github.com/cli99/llm-analysis

[43] Ziniu Li, Tian Xu, Yushun Zhang, Zhihang Lin, Yang Yu, Ruoyu Sun, and Zhi-Quan Luo. 2023. ReMax: A Simple, Effective, and Efficient Reinforcement Learning Method for Aligning Large Language Models. *arXiv preprint arXiv: 2310.10505* (2023).

[44] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[45] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International conference on machine learning*. PMLR, 3053–3062.

[46] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, Joseph E Gonzalez, and Ion Stoica. 2021. RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem. *Advances in Neural Information Processing Systems* 34 (2021), 5506–5517.

[47] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. 2014. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2014), 748–760.

[48] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564.

[49] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[50] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[51] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[52] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021).

[53] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM symposium on operating systems principles*. 1–15.

[54] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[55] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[56] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 527–540.

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[58] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Huntsville Ontario Canada, 16–29. https://doi.org/10.1145/3341301.3359642

[59] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[60] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[61] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[62] Gian-Carlo Rota. 1964. The number of partitions of a set. *The American Mathematical Monthly* 71, 5 (1964), 498–504.

[63] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal

Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv: 2308.12950* (2023).

[64] Michael Santacroce, Yadong Lu, Han Yu, Yuanzhi Li, and Yelong Shen. 2023. Efficient RLHF: Reducing the Memory Usage of PPO. *arXiv preprint arXiv: 2309.00754* (2023).

[65] Hill Kohli Saxton, Grefenstette. 2019. Analysing Mathematical Reasoning Abilities of Neural Models. *arXiv:1904.01557* (2019).

[66] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. PMLR, 1889–1897.

[67] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2018. High-Dimensional Continuous Control Using Generalized Advantage Estimation. arXiv:1506.02438 [cs.LG]

[68] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[69] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[70] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[71] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[72] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2023. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. arXiv:2312.12456 [cs.LG]

[73] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[74] Hanjing Wang, Man-Kit Sit, Congjie He, Ying Wen, Weinan Zhang, Jun Wang, Yaodong Yang, and Luo Mai. 2023. GEAR: a GPU-centric experience replay system for large reinforcement learning models. In *International Conference on Machine Learning*. PMLR, 36380–36390.

[75] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[76] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[77] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 358–369.

[78] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8 (1992), 229–256.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint arXiv: 1910.03771* (2019).

[80] Youshao Xiao, Weichang Wu, Zhenglei Zhou, Fagui Mao, Shangchun Zhao, Lin Ju, Lei Liang, Xiaolu Zhang, and Jun Zhou. 2023. An Adaptive Placement and Parallelism Framework for Accelerating RLHF Training. *arXiv preprint arXiv: 2312.11819* (2023).

[81] Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is dpo superior to ppo for llm alignment? a comprehensive study. *arXiv preprint arXiv:2404.10719* (2024).

[82] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. 2023. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales. *arXiv preprint arXiv:2308.01320* (2023).

[83] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[84] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. *arXiv preprint arXiv: 2402.16363* (2024).

[85] Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, Songfang Huang, and Fei Huang. 2023. Rrhf: Rank responses to align language models with human feedback without tears. *arXiv preprint arXiv:2304.05302* (2023).

[86] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[87] Chi Zhang, Guangming Sheng, Siyao Liu, Jiahao Li, Ziyuan Feng, Zherui Liu, Xin Liu, Xiaoying Jia, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. A Framework for Training Large Language Models for Code Generation via Proximal Policy Optimization. *NL2Code Workshop of ACM KDD 2024* (2024).

[88] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. 2024. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. *arXiv preprint arXiv:2401.05965* (2024).

[89] Shiwei Zhang, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. 2022. Accelerating large-scale distributed neural network training with SPMD parallelism. In *Proceedings of the 13th Symposium on Cloud Computing*. 403–418.

[90] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[91] Rui Zheng, Wei Shen, Yuan Hua, Wenbin Lai, Shihan Dou, Yuhao Zhou, Zhiheng Xi, Xiao Wang, Haoran Huang, Tao Gui, et al. 2023. Improving generalization of alignment with human preferences through group invariant learning. *arXiv preprint arXiv:2310.11971* (2023).

[92] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. arXiv:2401.09670 [cs]

[93] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. 2023. Swift: Expedited Failure Recovery for Large-Scale DNN Training. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 447–449. https://doi.org/10.1145/3572848.3577510

Table 3. The transfer protocols in HybridFlow.

Transfer Protocols	Distribute function	Collect function	Use case
ONE_TO_ALL	Broadcast the data to all ranks.	Gather the data from all ranks.	All the worker methods have the same input and run the ssme codes, e.g. model initialization.
3D_PROTO	Split the data, scatter across all DP ranks and broadcast within the group.	Gather and concatenate the data from the *p=-1, t=0* worker in all DP groups.	The model is sharded among multiple workers within each data-parallel group. The output of the model only exists in the last pipeline stage and is duplicated across the data-parallel groups. This is a typical scenario in 3D parallel training in Megatron-LM, Deepspeed, etc.
3D_ALL_MICRO_DP	Split the data by micro DP size, scatter across all micro DP groups and broadcast among all ranks within the group.	Gather and concatenate the data from the local_rank=0 worker in all micro DP groups.	Used with HybridEngine. It is used to handle the 3D-parallel scheme of the policy model, when switching between training and inference.
3D_PP_ONLY	Broadcast the data to all ranks.	Gather and concatenate the data from the *t=0, d=0* worker in all PP groups.	Used to examine weight names as they are identical in TP and DP groups.
DP_PROTO	Split the data into batches and scatter across all DP ranks.	Gather and concatenate the data from all DP ranks.	Training model in data-parallel mode.
ALL_TO_ALL	No operation.	Gather the data from all ranks.	Used when debugging. Users can manually define the inputs of each worker and examine their outputs respectively.

A Primitive APIs in HybridFlow

In HybridFlow, we implemented the primitive of each model in RLHF training by inheriting the `3DParallelWorker`, `FSDP Worker` and `ZeROWorker`. The functions of these model classes are designed to decouple the distributed computation code and provide fundamental operations in RLHF for the users. This primitive design is compatible with the auto-regressive generation, forward pass, backward pass, and model update operations in the existing distributed inference and training frameworks. Users can easily customize the RLHF training dataflow (by adapting the numerical computation in the provided functions) according to the algorithm's design and benefit from reusing the underlying distributed computation implementation. We illustrate the meaning and the actual computations of these APIs in Table 4.

B Transfer Protocols

We implemented transfer protocols that cover all common use cases of data resharding between models in RLHF dataflow. Users can utilize these pre-defined protocols to generate any RLHF dataflow. Moreover, Users can easily define their own transfer protocols by implementing a collect function and a distribute function. Transfer protocols decoupled the complicated data resharding and distributed training. We denote *p, t, d* as the rank of the worker in pipeline-, tensor- and data-parallel group respectively. We illustrate these predefined protocols in Table 3.

C Auto-Parallelism Algorithm

Algorithm 2 outlines the search process of the optimal parallelism strategy of each model. Starting from the minimal model parallelism size of each model (to prevent OOM when colocating with multiple workers), we enumerate all feasible parallel configurations based on the number of GPUs and the number of GPUs per machine U. The default number of U is set to 8. We use `simu` module to estimate the latency of each model based on their workload. This module includes three simulators for training, inference, and generation workload, all are analytical models following previous research [42, 84, 92]. The training and inference workload is compute-bound while the generation workload is memory-bound. For the actor model, we first find the parallelism strategy for training and record the memory usage in the training stage. During actor generation, KVCache requirements are calculated using the batch size and max sequence length. If the model-parallel size for the generation stage cannot accommodate both parameters and KVCache, we increase it. Then, we seek the optimal strategy with corresponding KVCache allocation by comparing the latency estimation. Developing a comprehensive autoregressive generation simulator that accounts for variable KVCache sizes could further enhance the auto-mapping process in RLHF research.

Algorithm 2 Auto Parallelism Algorithm

Input: Device allocation A, minimal device allocation and model parallel size for each model in a set A_{min}, workload W, the number of GPUs per machine U
Output: the parallelism strategy for the model in a set
Procedure auto_parallel(A, A_{min}, l, W):
  $N_l = A[l]$ *// Get device allocation of the model*
  $t_{min} = A_{min}[l].t$ *// Get minimal model parallel size*
  $p_{min} = A_{min}[l].p$
  best_para $\leftarrow \emptyset$
  best_para.cost $\leftarrow \infty$
  for all t $\in \{t_{min}, t_{min}+1..., U\}$ **do**
    for all p $\in \{p_{min}, p_{min}+1..., \frac{N_l}{U}\}$ **do**
      d $\leftarrow \frac{N_l}{\text{p}\times\text{t}}$
      para_plan $\leftarrow (p, t, d)$
      cost $\leftarrow$ simu($para\_plan, l, W[l]$)
      if best_para.$cost$ > cost **then**
        best_para.$cost \leftarrow$ cost
        best_para $\leftarrow$ para_plan
  return best_para

Table 4. Key functions provided in each model class. The users can use these provided functions to construct various RLHF algorithms in a few lines of code.

Model	APIs	Computation	Interpretation
Actor	`generate_sequence`	auto-regressive generation	Based on a batch of prompts, the actor model generates a batch of responses and returns the log probability of each token in the responses.
	`compute_log_prob`	a forward pass	The actor model computes the log probability of each token in the prompts and responses. This log probability is the same as the return log probability when performing generation using the same model precision. (Optional in PPO)
	`compute_loss`	a forward pass	The actor model computes the pretrain loss based on the pertaining dataset [7, 19, 55].
	`update_actor`	a forward, backward pass and model update	Based on the advantages, returns (calculated from `compute_advantage`) and pertaining loss, the actor model calculate the training loss and update its weights. We implement various loss for diverse RLHF algorithms including PPO [55], Safe-RLHF [19], ReMax [43], GRPO [70] and others.
Critic	`compute_values`	a forward pass	The critic model computes the values for each prompt and response.
	`update_critic`	a forward, backward pass and model update	Based on the values and returns, the critic computes a squared-error loss to update its weights. We also implement critic loss for diverse RLHF algorithms including PPO [55], Safe-RLHF [19], ReMax [43], GRPO [70] and others.
Reference Policy	`compute_ref_log_prob`	a forward pass	The reference model computes the reference log probability of each token in the prompts and responses. This log probability is utilized as a benchmark to evaluate the divergence of the actor model and constrain its learning process.
Reward	`compute_reward`	a forward pass	The reward model conducts forward computation to calculate scores for a given set of prompts and responses. The rewards could be token-level or sample-level.
-	`compute_advantage`	numerical computation	Based on the values rewards from the value model and reward model respectively, the function estimates the advantages on the given prompts and the current policy model's responses. This computation involves no model forward passes.