RLHFuse: Efficient RLHF Training for Large Language Models with Inter- and Intra-Stage Fusion

Yinmin Zhong[1] Zili Zhang[1] Bingyang Wu[1] Shengyu Liu[1] Yukun Chen[2]
Changyi Wan[2] Hanpeng Hu[2] Lei Xia[2] Ranchen Ming[2] Yibo Zhu[2] Xin Jin[1]

[1]*Peking University* [2]*StepFun*

Abstract

Reinforcement Learning from Human Feedback (RLHF) enhances the alignment between LLMs and human preference. The workflow of RLHF typically involves several models and tasks in a series of distinct stages. Existing RLHF training systems view each task as the smallest execution unit and thus overlooking the opportunities for subtask-level optimizations. Due to the intrinsic nature of RLHF training, *i.e.*, the data skewness in the generation stage, and the pipeline bubbles in the training stage, existing RLHF systems suffer from low GPU utilization in production deployments.

RLHFuse breaks the traditional view of RLHF workflow as a composition of individual tasks, splitting each task into finer-grained subtasks, and performs *stage fusion* to improve GPU utilization. RLHFuse contains two key ideas. First, for generation and inference tasks, RLHFuse splits them into sample-level subtasks, enabling efficient *inter-stage fusion* to mitigate the original generation bottleneck dominated by long-tailed samples. Second, for training tasks, RLHFuse breaks them into subtasks of micro-batches. By leveraging the intuition that pipeline execution can be essentially complemented by another pipeline, RLHFuse performs *intra-stage fusion* to concurrently execute these subtasks in the training stage with a fused pipeline schedule, resulting in fewer pipeline bubbles. In addition, RLHFuse incorporates a series of system optimizations tailored for each stage of RLHF, making it efficient and scalable for our internal product usage. We evaluate RLHFuse on various popular LLMs and the results show that RLHFuse increases the training throughput by up to 3.7×, compared to existing state-of-the-art systems.

1 Introduction

The rise of Large Language Models (LLMs) marks a revolutionary leap in generative AI. From search engines [1] to personal assistants [2], and from operating systems [3] to embodied AI [4], LLMs are redefining what is possible, opening up new frontiers and possibilities. Despite the impressive capabilities of LLMs, concerns around their safety,
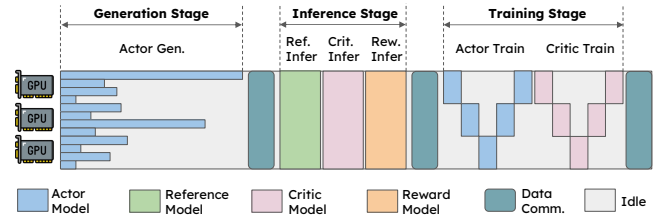

Figure 1: RLHF iteration breakdown of existing frameworks.

robustness, and reliability have been at the forefront since their inception. Many studies [5–7] have shown that LLMs often display unintended behaviors, such as fabricating facts, generating biased or toxic content, or failing to adhere to user instructions. To address these issues, Reinforcement Learning from Human Feedback (RLHF) [8] has been introduced, aiming to align LLMs with human intent after pre-training. With a concrete and well-defined reward target, RLHF enhances the ability of LLMs in multiple domains, including following instructions, suppressing hallucinations, and improving overall safety. It is known to be the key driving force behind the evolution of GPT-3 [9] into ChatGPT [2], and almost all the leading LLMs [10–12] incorporate RLHF to boost their model performance.

Unlike pre-training, RLHF training is significantly more complex in terms of system design, since it involves four different LLMs, three stages, and six tasks in total. As shown in Figure 1, each iteration begins with the `Actor` model, which generates responses to a batch of input prompts (known as the generation stage). These responses are then passed to the `Reference`, `Critic`, and `Reward` model for processing, during the inference stage. Finally, the `Actor` and `Critic` models compute loss and update their parameters, completing the training stage. The data and weight dependencies create synchronization boundaries between stages.

To enable efficient training of RLHF, existing frameworks have proposed a series of optimizations, such as selecting tailored parallel strategies for each task [13] and optimizing the data exchange during stage transitions [14]. However, these techniques *fundamentally treat the RLHF workflow as a sim-*

ple composition of individual tasks, failing to delve into the inherent characteristics and structure inside the tasks, thereby missing significant optimization opportunities. In production deployments, we have observed two issues within current frameworks that severely impact training efficiency, as indicated by the grey area in Figure 1.

The first issue arises from the *data skewness* in the generation stage, where the output length of generated samples follows a long-tailed distribution. This phenomenon is broadly observed during LLM generation (§2.2) and this data diversity is critical to the robustness of RL training. As a result, the generation time is dominated by the longest sample as shown in Figure 1. During the latter part of the generation stage, the GPUs remain idle or are engaged in processing only a small number of long-tailed samples. Due to data dependencies, the inference tasks cannot start in advance, leading to low GPU utilization. Making the problem even worse, as the maximum output length of LLMs continues to grow [15], this long-tail phenomenon will become more significant.

The second issue stems from the *pipeline bubbles* in the training stage. With the explosive growth in the size of LLMs, higher pipeline parallelism (PP) is needed to scale training. When LLMs reach hundreds of billions of parameters, it is quite common for PP size to reach a few dozen [16, 17]. However, the proportion of pipeline bubbles increases with PP size, significantly reducing training efficiency. In RLHF, there are two training tasks, each may evolve a specific LLM with hundreds of billions of parameters, which amplifies this inefficiency. Despite that various approaches have been proposed to reduce pipeline bubbles when training individual models [18–21], the remaining bubbles still pose a significant challenge under synchronous training semantics.

To this end, we propose RLHFuse, the first work to employ subtask-level optimizations for RLHF training. By breaking down the task into finer-grained subtasks, RLHFuse opens up a new design space to *fuse the stage execution*. This is orthogonal to the state-of-the-art RLHF framework ReaLHF [13] that optimizes the parallelism strategy for each task. RLHFuse exploits the optimizations opportunities at the subtask level with two new techniques, i.e., *data-aware inter-stage fusion* and *model-aware intra-stage fusion*. These two techniques leverage the unique system characteristics of RLHF training to address data skewness and pipeline bubbles, respectively.

For data skewness, we split the generation and inference tasks into sample-level subtasks and then the dependency granularity between the generation and inference stage can be refined from stage-level to sample-level. This is due to the fact that the computation of the two stages is essentially independent for different samples. To leverage this insight, we design a lightweight and efficient migration algorithm to automatically detect when the generation stage enters the inefficient long-tail processing phase and migrates the remaining long-tailed samples to a limited number of GPUs. The freed-up resources are used to launch the inference tasks. This

approach enables inter-stage fusion and dynamically adjusts the migration timing in each iteration according to the workload, thereby maximizing overall efficiency.

For pipeline bubbles, we break the training task into subtasks of micro-batches. Pipeline bubbles are essentially caused by dependencies between subtasks of the same model. Fortunately, in the RLHF training stage, there are two independent models, and their subtasks are naturally independent as well. Based on this observation, we employ a fused pipeline schedule to enable intra-stage fusion, which executes the training tasks on the same set of GPUs, effectively filling each other's pipeline bubbles. We design a lightweight pipeline schedule generation algorithm that can produce near-optimal schedules for `Actor` and `Critic` models of any size and parallel configurations.

In addition, we apply a series of system optimizations covering each RLHF stage, transforming RLHFuse into a production-ready framework to support the RLHF training of our internal products. We evaluate RLHFuse on various LLMs and real-world datasets. Compared to state-of-the-art solutions, RLHFuse improves the throughput up to $3.7\times$.

In summary, we make the following contributions:

- We identify the key issues within current RLHF training frameworks and propose to view the RLHF workflow from a finer-grained subtask-level perspective.
- We present RLHFuse, a highly efficient RLHF training framework that utilizes inter- and intra-stage fusion to improve the training throughput.
- We conduct a comprehensive evaluation of RLHFuse and compare it with state-of-the-art RLHF training solutions.

2 Motivation

In this section, we first introduce the basics of RLHF training (§2.1) and then point out the problems in existing RLHF training systems (§2.2).

2.1 Background

One complete RLHF process consists of three key steps: supervised fine-tuning, reward model training, and model alignment using the PPO algorithm [22]. The first two stages both train a single LLM, which has been studied by previous work like Megatron-LM [16] and MegaScale [17]. The third stage, which this paper focuses on, is unique and complex in terms that it involves multiple models and stages. Next, we provide a detailed overview of the RLHF models and workflow.

RLHF models. Typically, RLHF training involves four LLMs in total: `Actor`, `Ref`, `Critic`, and `RW`. The `Actor` model, which is the primary model to be trained and the final product after RLHF, serves as the *agent* in the RL semantics. Given a prompt, each token it generates autoregressively is considered an *action*, and the goal of the PPO algorithm is to guide it to

produce actions that align with the reward target. The Reference model (`Ref`) is initialized from the original `Actor` model, but is frozen (*i.e.*, weights do not update) during training. It provides Kullback-Leibler (KL) divergence regularization to ensure the `Actor` model does not deviate excessively from its original version during training or generate nonsensical outputs. The Reward model (`RW`) is trained on human-labeled preference data to score each sample generated by the `Actor` model, which guides the `Actor` model to generate responses that align with human preferences. It is also frozen during the RLHF training. The `Critic` model, initialized from the `RW` model, serves as the value model to evaluate the actions taken by the `Actor`. It provides finer-grained feedback on action level to guide `Actor` towards better decisions.

RLHF workflow. Figure 1 illustrates the workflow of one training iteration in RLHF, which contains three main stages: *Generation*, *Inference*, and *Training*.

Generation stage. The `Actor` model generates responses for the prompts in the current batch. This process involves a prefill phase and a decoding phase. The prefill phase processes the prompt, which is composed of a sequence of tokens, to generate the first output token. Following this, the decoding phase sequentially generates subsequent tokens in multiple steps autoregressively (i.e., each decoding step generates a new token based on tokens generated in previous steps until reaching a termination token). Ultimately, each prompt and its corresponding response form one training sample for this iteration, referred to as a *rollout* or *trajectory* in the context of RL. To increase sample efficiency, it is common to generate hundreds or even thousands of samples in each iteration.

Inference stage. The `Ref`, `RW`, and `Critic` models each perform a forward pass on the generated samples. It is similar to the prefill phase in the generation stage but does not generate the next token. Instead, the output logits are used to calculate the training loss in the *Training* stage. Note that the execution time of each inference task depends on the model size and is generally not the same.

Training stage. In the PPO algorithm, unlike traditional LLM training, all the samples are divided into several mini-batches and the model weights need to update after training on each mini-batch. Concretely, for each mini-batch, the `Actor` and `Critic` models first perform a forward pass, then calculate the loss using the results from the inference stage and perform a backward pass to update the parameters. After the `Actor` completes training on all mini-batches, it uses the *up-to-date* parameters for the generation in the next iteration.

LLM parallelization. Guided by the scaling law [23], the size of LLMs has increased exponentially. Various parallelism methods are proposed to scale training, including data parallelism, tensor parallelism, and pipeline parallelism.

Data Parallelism (DP). DP replicates the model weights and distribute the data among the replicas to execute in parallel. At the end of each iteration, all DP groups perform all-reduce communication on the gradients before updating parameters.

Tensor Parallelism (TP). TP partitions computationally intensive operators, mainly matrix multiplications, across multiple GPUs to accelerate computation. It requires significant communication, so it is typically used within a node to leverage the high-bandwidth NVLINK connections.

Pipeline Parallelism (PP). PP organizes LLM layers into stages, each running on a separate device or node. It partitions the input batch into multiple "micro-batches" to form pipeline execution and accumulates the gradients of the entire input batch. PP features point-to-point communication and incurs much less communication overhead than TP.

Since the above parallelism methods have their own advantages, all three parallel strategies are used together to scale training in practice, such as Megatron-LM [16] and MegaScale [17] in pre-training [16] and ReaLHF [13] in RLHF training. We use an integer triple (dp, pp, tp) to denote the degrees of DP, PP, and TP in 3D-parallelism.

2.2 Problems in RLHF Training

During our practical deployment of RLHF training, we identified two key issues in the *Generation* and *Training* stage within existing RLHF training systems that contribute to substantial GPU under-utilization.

Generation Stage: data skewness. One major problem in the generation stage is that the response length of generated samples exhibits a long-tailed distribution, meaning that each batch will contain a few samples that are significantly longer than the others. Due to data dependencies, the inference tasks cannot start until the generation task finishes. Consequently, even only a few long-tailed samples force the inference tasks to wait. This leads to significantly low GPU utilization in the generation stage because the decoding phase of LLM generation is memory-bound, requiring a large batch size (typically a few hundred) to maintain high GPU utilization.

This long-tail phenomenon is prevalent and pronounced in the LLM generation. Figure 2 (left) shows the CDF of output length distribution in LMSYS-Chat-1M dataset [24], which collects one million user requests and corresponding responses from various models on the popular LLM evaluation platform Chatbot Arena. We observe this long-tail distribution across models of varying sizes, both open-source and proprietary. The vertical dotted lines mark the 99.9-th percentile length corresponding to each model, which is more than ten times the median length. In practice, it is common to generate over 1,000 samples in each iteration, meaning that almost every iteration will encounter long-tailed samples.

Internally, we also observe this pattern during the RLHF training of a proprietary model with hundreds of billions of parameters. Figure 2 (right) presents the iteration time breakdown under different maximum output length settings. The
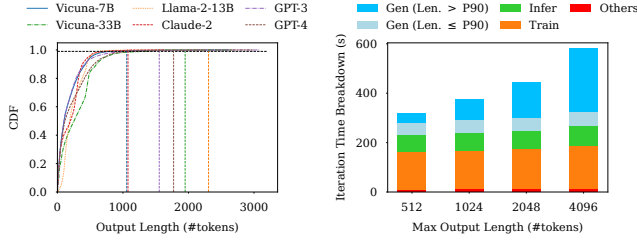
Figure 2: Left: The output length CDF of models in the LMSYS-Chat-1M dataset. The vertical dotted line indicates the P99.9 output length. Right: The RLHF training iteration breakdown on the internal model and datasets under different maximum output lengths.

time spent generating long-tailed samples (dark blue bar) accounts for more than half of the total generation time. The situation gets worse as the maximum generation length increases, which leads to a substantial rise in iteration time. It is worth noting that this change only affects the long-tailed samples ($< 1\%$), but the results indicate that this very small portion of long-tail samples can have a significant impact on the overall system performance. As user applications increasingly demand lengthy outputs [15], the inclusion of samples with long response lengths in RLHF training becomes more common and important, making the long-tail phenomenon an urgent problem in real deployment.

Training Stage: pipeline bubbles. With the exponential growth in the size of LLMs, higher parallelism is required to scale training. While TP size is generally limited to the number of GPUs within a single node (typically 8) due to the significant communication overhead, and DP size increases the memory consumption of model weights linearly. Consequently, PP has become a critical method for scale training.

Scaling the PP size is not a free lunch, as pipeline bubbles can significantly impact training efficiency. Concretely, in the most commonly used 1F1B pipeline schedule [25] as shown in Figure 3 (upper), the *pipeline bubble percentage* is $\frac{N-1}{N-1+M}$ with N PP stages and M micro-batches. One simple way to reduce the bubble percentage is to increase M. However, the global batch size is constrained by the converging conditions and cannot be increased indefinitely. Additionally, in RLHF, the global batch is first divided into several mini-batches, then distributed among the DP groups, and finally split into micro-batches, further limiting M. As N scales with the model size to approach M, which is quite common when scaling LLMs to hundreds of billions of parameters, the bubble percentage is about 50%. This means that about half of the GPUs are idle during training, leading to a significant waste of resources.

Interleaved 1F1B scheduling [16] is proposed to mitigate the pipeline bubbles. As shown in Figure 3 (bottom), it divides the LLM into more fine-grained model chunks, with each stage hosting $K (K > 1)$ model chunks, reducing the pipeline bubble percentage to $\frac{N-1}{N-1+KM}$. However, it not only introduces a K-fold communication overhead but also exacerbates the imbalances across PP stages. As a result, K is typically set to 2 in practice, leaving room for further optimizations.
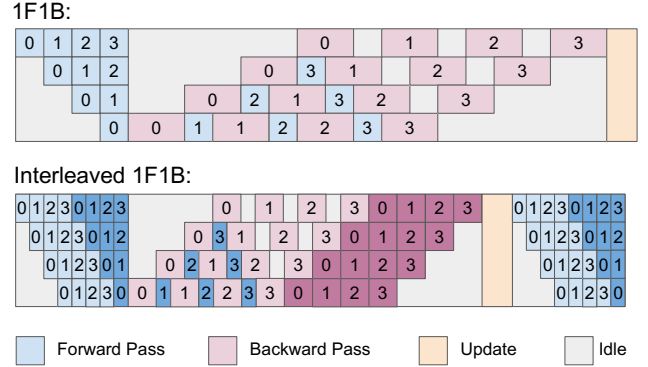

Figure 3: The timeline of 1F1B and interleaved 1F1B pipeline schedule with 4 pipeline stages and 4 micro-batches. Interleaved 1F1B deploys two model chunks on each pipeline stage in this example.

There are other works that aim to completely eliminate pipeline bubbles; however, they either violate the synchronous training semantics [25] or rely on specific, often unpractical, assumptions [21]. As a result, pipeline bubbles still pose a significant problem in LLM training, especially when the model size keeps scaling.

Summary. Essentially, the above issues are caused by the inherent characteristics of RLHF, which are difficult to eliminate through task-level optimizations. Existing systems treat the task as the smallest execution unit and overlook its internal structure, thereby missing huge optimization opportunities.

3 RLHFuse Overview

To this end, we present RLHFuse to address data skewness in the generation stage through data-aware inter-stage fusion (§4) and mitigates pipeline bubbles in the training stage with model-aware intra-stage fusion (§5). Additionally, RLHFuse adopts a series of system optimizations (§6) tailored for RLHF training. Here we provide a brief overview of RLHFuse as shown in Figure 4.

Workflow. With specific algorithm, model, and cluster configurations as input, RLHFuse first configures efficient parallel and deployment strategies following the approach in ReaLHF [13]. For each task within the RLHF workflow, RLHFuse will assign it a tailored parallel strategy $P_i = (dp, pp, tp)$ to maximize the GPU utilization. To deploy the task onto the physical cluster, we introduce the concept of *device mesh*, which is the unit for executing an individual task. A device mesh $D = (N, M)$ is defined as a two-dimensional grid of GPUs, covering N nodes where each is equipped with M devices. RLHFuse will designate a device mesh D_i to each task which can be executed with parallel strategy P_i. At runtime, RLHFuse will launch each task with its corresponding parallel strategy on the designated device mesh following the workflow dependency and handle the weight redistribution and data transmission between different tasks to ensure syn-
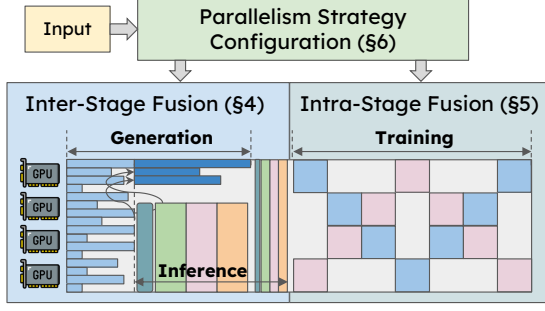
Figure 4: RLHFuse architecture.

Figure 5: The timeline of serial (top) and fused (bottom) execution of generation and inference stages.

chronous training semantics. Meanwhile, RLHFuse utilizes data-aware inter-stage fusion to fuse the generation and inference tasks and model-aware intra-stage fusion to fuse the training tasks.

Inter-stage fusion. As depicted in the left part of Figure 4, during the generation stage, there are multiple model instances, each managing complete model weights and part of the samples. RLHFuse actively monitors each generation instance and triggers migration when the number of remaining samples is below a particular threshold. The remaining samples will be migrated to some dedicated generation instances which are designated to handle long-tailed samples. Subsequently, the resources of the original generation instances are repurposed to launch inference tasks, thereby allowing for the overlap of inference tasks with the generation of long-tailed samples and fusing the execution of the two stages.

Intra-stage fusion. As shown in the right part of Figure 4, in the training stage, RLHFuse leverages the insight that pipeline execution can be complemented by another pipeline and generates a fused pipeline schedule based on the model size and parallel strategies of `Actor` and `Critic` model. It utilizes a light-weight schedule generation algorithm to minimize the pipeline bubbles and activation memory usage, thus improving the training efficiency.

4 Data-aware Inter-Stage Fusion

In this section, we first analyze the opportunities of fusing the generation and inference stages (§4.1). We then introduce the fused execution plan to maximize the overlap between the two stages while preserving the data dependency (§4.2).

4.1 Opportunities and Challenges

Opportunities. The dependency between the two stages is predicated on the sample level, implying that once a sample completes its generation stage, it can seamlessly advance to the inference stage. This key observation motivates us to break the task into sample-level subtasks without violating original synchronous training semantics. Consequently, we can initiate inference tasks as soon as there exist completed samples in the generation stage.
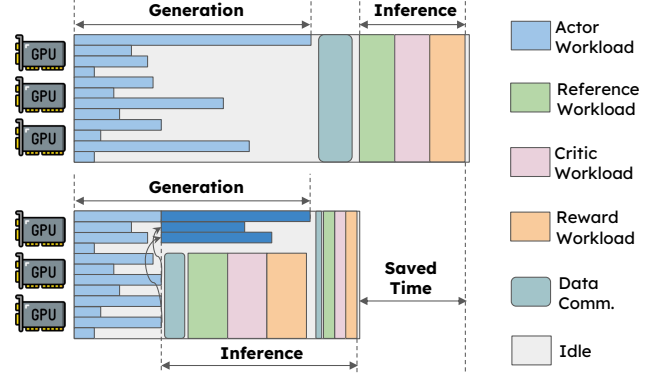
Based on this, we design a fused execution plan that overlaps the generation and inference stages. Specifically, the fused plan first detects the point when the generation instance enters the long-tail decoding phase. At this juncture, most of the samples have finished their generation stage but are waiting for the inference stage to start. Each instance only leaves a few long-tailed samples to process, which is memory-bound as discussed in §2.2. This condition facilitates the migration of samples from all instances to a few designated instances. This migration not only consolidates the GPU utilization of the receiving instances but also releases resources from the sending instances. The released resources are then promptly utilized to launch inference tasks in advance. One example of a fused execution plan is shown in Figure 5. This approach greatly improves the GPU utilization and optimizes the overall execution time of the two stages.

Challenges. With the inter-stage fusion, the problem now lies in generating an efficient fused execution plan that decides the migration timing, destination, and mechanism. These three factors significantly affect the overall execution time of the two stages, and therefore must be chosen carefully.

Given that inter-stage fusion appears so promising, a natural question arises: can we break the synchronization between the inference and training stages similarly? Unfortunately, training tasks require each mini-batch to maintain the same data distribution, which necessitates random sampling from all the generated samples. To satisfy this requirement, it implicitly introduces an unavoidable synchronization boundary.

4.2 Fused Execution Plan

The objective of the inter-stage fusion is to overlap the execution of the two stages as much as possible without affecting the original generation task's execution time, thereby minimizing the overall execution time of the two stages. Following this objective, we detail the migration triggering, destination, and mechanism below.

Migration triggering. Triggering the migration significantly influences the efficiency of overlapping the two stages. If

migration is triggered too early, there remain excessive generation samples. The migration results in only a few model instances remaining available to continue the generation task, as most are redirected to handle inference tasks. This scenario places an excessive load on the few generation instances, leading to prolonged execution times for the original generation task and ultimately extending the overall process. Conversely, if the migration is triggered too late, it loses most of the overlapping opportunity. Based on the above analysis, we propose a migration threshold, denoted as R_t. When *the number of remaining samples in the generation stages falls below R_t*, the migration is triggered. To optimize system performance and minimize total execution time, carefully determining R_t is essential to find the best trade-off.

To determine the optimal R_t, we initially conducted offline generation trials to analyze the length distribution of the generated samples. This analysis enables us to estimate the computational load for both the generation and inference stages accurately. We then systematically test R_t values ranging from 5% to 95% of the global batch size, simulating the execution time for each value. The optimal R_t is identified as the value that yields the minimum simulated execution time. Additionally, during runtime, we refine the distribution by incorporating new generation samples, which allows us to update R_t as needed to continually optimize performance.

Note that it is possible to achieve finer-grained overlapping by triggering migration more than once. However, we find that in practice, the generation time of long-tailed samples often exceeds the execution time of inference tasks, so one migration is usually sufficient to fully overlap the entire inference stage, as shown in §7.2.

Migration destination. Once the migration timing is determined, we need to select m generation instances to handle the remaining samples. Assume there are n generation instances in total, then there will be $n-m$ instances to be repurposed for inference tasks. To determine m, we revisit the computation characteristics of the generation task. During the memory-bound decoding phase of generation, as long as the batch size does not exceed a certain threshold that saturates the GPU (denoted as BS_{max}), the latency of each decoding iteration remains almost constant [26, 27]. The value of BS_{max} depends on the specific GPU hardware and can be determined through prior profiling. Based on this, we set the first constraint as $m \geq \frac{R_t}{BS_{max}}$, which ensures that the generation time of long-tailed samples remain unchanged as before the migration.

Another factor to consider is the memory constraint. During LLM generation, the key-value cache is maintained for each token position for autoregressive generation. When the sequence length is long, this memory consumption becomes non-negligible [28]. Therefore, we need to ensure that the target instances have sufficient memory to accommodate the remaining long-tailed samples, avoiding out-of-memory issue or blocking the sample processing. Thus, we set the second constraint as $m \geq \frac{R_t * M}{C}$, where M denotes the key-value cache

consumption of the sample with the maximum output length and C is the available GPU memory of the target instance allocated for key-value cache. In summary, the final m is determined by the maximum of the two constraints.

Next, we need to determine which m out of the n generation instances to process the remaining long-tailed samples. To minimize migration overhead, we select the top m instances that have the most samples remaining. This strategy minimizes the total number of samples requiring migration.

Migration mechanism. To migrate an unfinished sample to the target instance, we have two choices. The first approach is to transfer the generated key-value cache to the target instance over the network, allowing it to continue the subsequent generation immediately upon receipt. The overhead in this case primarily comes from the network transmission. The second approach is to discard the key-value cache and only transmit the generated tokens of the sample. Compared to the first approach, this method incurs minimal network latency but comes at the cost of rerunning the prefill phase to generate the key-value cache. The choice of migration mechanism depends on the GPU hardware and network bandwidth. In our experiments, thanks to the high-bandwidth RDMA connections, we choose the first approach and the migration overhead is negligible compared to the iteration time, as shown in §7.2.

After completing the migration, we can release the GPU resources of the remaining $n-m$ generation instances to launch inference tasks. This process involves weights redistribution and data transmission to prepare the context of inference tasks, which we discuss in §6. Once the inference tasks are launched, the remaining long-tailed samples can be streamed to the inference instances as soon as their generation stage is completed, allowing for seamless processing. Additionally, if the long-tail generation task finishes first, we release its resources for the inference tasks.

5 Model-aware Intra-Stage Fusion

In this section, we first analyze the opportunities of fusing the two tasks in the training stage (§5.1). Then we propose a lightweight algorithm to generate the fused pipeline schedule to minimize the pipeline bubbles and memory usage (§5.2).

5.1 Opportunities and Challenges

Opportunities. During the RLHF training stage, the `Actor` and `Critic` models are trained independently. Inspired by the bi-directional pipeline schedule [20] from Chimera, we can break the two training tasks into subtasks of micro-batches and co-locate these subtasks to mutually fill the pipeline bubbles of each other. Originally, Chimera enhances single-model training by replicating the model and trains the replicated model with 1F1B schedule in opposite pipeline directions, as depicted in Figure 6(a). This approach not only mitigates pipeline bubbles but also brings a balanced distribution of
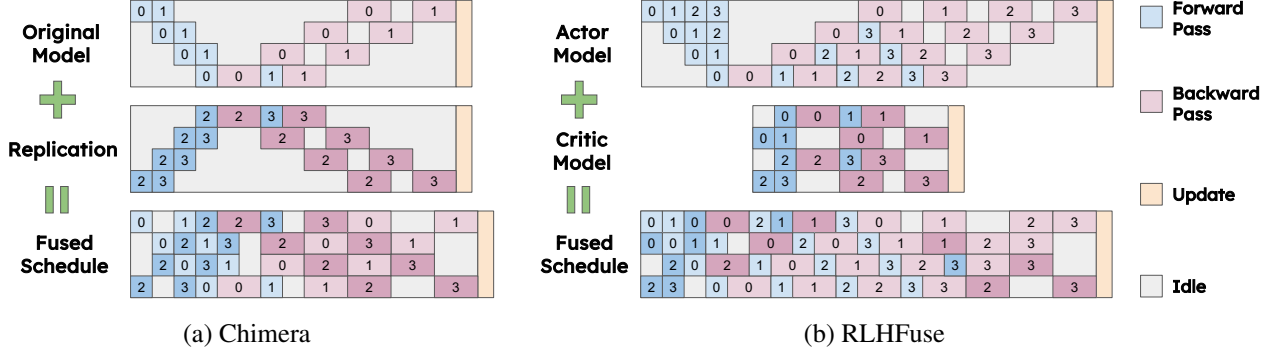
(a) Chimera   (b) RLHFuse

Figure 6: (a) An example of the symmetric bi-directional pipeline schedule used in Chimera [20] with four pipeline stages and four micro-batchs. (b) An example to show our method can fuse two different models with four and two pipeline stages, respectively.

activation memory. RLHF training stage inherently involves two distinct models, enabling us to apply the fused pipeline schedule without extra model replication.

Challenges. In Chimera, the bi-directional pipeline schedule utilizes one identical model replica with uniform size and parallel strategy, resulting in a symmetric execution flow. In contrast, RLHF involves training heterogeneous `Actor` and `Critic` models, which differ not only in size but also in their optimal parallel strategies. Consequently, the symmetric schedule from Chimera is no longer applicable in the face of model heterogeneity in terms of size and parallel configuration. In the following, we introduce a lightweight and efficient algorithm to automatically generate the fused pipeline schedule under this more generalized setting.

5.2 Fused Pipeline Schedule

Problem transformation. Assume the parallel strategies for models A and B are (dp_1, pp_1, tp_1) and (dp_2, pp_2, tp_2). Each will utilize all the GPUs in the cluster. We require that tp is the powers of two, which is commonly adopted in practice. For the case where $tp_1 \neq tp_2$, the challenge for a fused pipeline schedule lies in the fact that the pipeline stage of the two models contains a different number of GPUs. Without loss of generality, we assume $tp_1 = s \times tp_2$. In this situation, we merge every s consecutive pipeline stages of model B into one stage, redividing model B into $\frac{pp_2}{s}$ pipeline stages. This ensures that each stage of both models uses the same number of GPU resources. Note that here we require pp_2 to be divisible by s, which is easy to realize in practice. After that, we can transform the problem into fusing K_1 disjoint pipeline groups of model A with K_2 pipeline groups of model B, where K denotes the fusion factor and K_1 and K_2 are coprime. Figure 6(b) illustrates an example of $(K_1, K_2) = (1, 2)$. Next, we precisely define the fused pipeline schedule and formulate the problem. Table 1 summarizes the key notations.

Problem formulation. We assume the two models to be fused each has N_1 and N_2 pipeline stages ($K_1 \times N_1 = K2 \times N_2 = N$), and each pipeline needs to process M_1 and M_2 micro-batches, respectively. Since the global batch size is fixed,

Symbol	Description
S	The fused pipeline schedule.
l_{ij}	The latency of subtask S_{ij}.
C	The activation memory capacity of each stage.
K_1	The fusion factor of model A.
K_2	The fusion factor of model B.
N	The total number of pipeline stages to be fused.
N_1	The number of pipeline stages of model A.
N_2	The number of pipeline stages of model B.
M_1	The number of micro-batches for model A.
M_2	The number of micro-batches for model B.

Table 1: Key notations in problem formulation.

we have $K_1 \times M_1 = K2 \times M_2$. The fused pipeline schedule is represented as a matrix S, where S_{ij} represents the j-th subtask (micro-batch) to be scheduled in the i-th stage. Since there are N stages in total and each stage needs to process the forward and backward computation for all micro-batches of the two models exactly once, the shape of S is $N \times 2(M_1 + M_2)$. The latency of subtask S_{ij} can be profiled in advance and is denoted as l_{ij}. Note that not all S can be scheduled. A valid schedule must satisfy the following constraints:

1. Data dependency: the forward and backward of the same micro-batch across different stages must be executed following the original data dependency.
2. Deadlock avoidance: there exists no cycles in the overall dependency graph to avoid deadlocks.
3. Memory constraint: the peak activation memory usage of each stage must be less than C to avoid out-of-memory.

The main optimization target is to find a valid schedule S that has the minimum latency. Additionally, among schedules with the same latency, we prefer the ones with lower peak activation memory to optimize resource usage.

Algorithm overview: The problem has many constraints and variables. It can be reduced to a special case of flow job scheduling problem which proves to be NP-hard [29]. A naive solution is to greedily fuse the 1F1B schedules of two models with the bi-directional pipeline, which always schedules feasible micro-batches. If the micro-batches of two models

Algorithm 1 Generate Fused Pipeline Schedule.

1: **function** GENERATEFUSEDPIPELINESCHEDULE(S_0)
2:     $s_{current} \leftarrow S_0$
3:     $e_{current} \leftarrow ComputeEnergy(s)$
4:     $T \leftarrow e_{current}$
5:     $s^* \leftarrow s_{current}$
6:     $e^* \leftarrow e_{current}$
7:     **while** $T > \varepsilon$ **do**
8:         $s_{neighbor} \leftarrow ComputeNeighbor(s_{current})$
9:         $e_{neighbor} \leftarrow ComputeEnergy(s_{neighbor})$
10:         **if** $e_{neighbor} < e^*$ **then**
11:             $s^* \leftarrow s_{neighbor}$
12:             $e^* \leftarrow e_{neighbor}$
13:         **if** $P(e_{current}, e_{neighbor}, T) > Rand(0,1)$ **then**
14:             $s_{current} \leftarrow s_{neighbor}$
15:             $e_{current} \leftarrow e_{neighbor}$
16:         $T \leftarrow T \times \alpha$
17:     **return** s^*

Algorithm 2 Generate A Random Neighbor Schedule.

1: **function** COMPUTENEIGHBOR(S)
2:     **while** $True$ **do**
3:         $i \leftarrow RandInt(1, N)$
4:         $j \leftarrow RandInt(1, 2(M_1 + M_2) - 1)$
5:         $S' = Swap(S_{ij}, S_{i(j+1)})$
6:         **if** $CheckValid(S')$ **then**
7:             **return** S'

Algorithm 3 Compute Energy

1: **function** COMPUTEENERGY(S)
2:     **return** $max_{1 \leq i \leq N}$ComputeFinishTime($S_{i,M}$)
3: **function** COMPUTEFINISHTIME(S_{ij})
4:     **if** $S_{ij} \in Memo$ **then return** $Memo.get(S_{ij})$
5:     $S^{intra} \leftarrow GetIntraDependency(S_{ij})$
6:     $S^{inter} \leftarrow GetInterDependency(S_{ij})$
7:     $IntraTime \leftarrow ComputeFinishTime(S^{intra})$
8:     $InterTime \leftarrow ComputeFinishTime(S^{inter})$
9:     $FinishTime = max(IntraTime, InterTime) + l_{ij}$
10:     $Memo.put(S_{ij}, FinishTime)$
11:     **return** $FinishTime$

---

are both ready, it favors the larger model, with the expectation that the smaller one can flexibly fill in the bubbles later. However, the greedy method lacks a global perspective. For example, certain micro-batches could be deliberately delayed to fill the bubbles later. Moreover, it provides no optimization for memory usage. Thus, the greedy approach does not yield optimal performance, as we will demonstrate in §7.3.

Instead, we use simulated annealing [30] to search for a better solution. The motivation for using simulated annealing is that we have a huge search space with many variables. Simulated annealing is effective in finding acceptable local optimums in a reasonable amount of time while finding the global optimum is computationally expensive. Furthermore, we will show that our method can achieve the theoretical lower bound most of the time as shown in §7.3.

At a high level, we use the fused pipeline schedule S as the state in simulated annealing. We use the schedule found by the greedy algorithm mentioned above as the initial state and probabilistically jump to a neighbor state in each iteration, aiming to find a schedule with the lowest latency. Our approach has two benefits. First, it is easy to scale out. We can perform searching using different random seeds across hundreds of CPU cores and select the best result among them. Second, it can easily extend to more models, which arises in multimodal [31] and multi-agent [32] training scenarios. Now we describe the algorithm in more detail below.

Simulated Annealing (Algorithm 1): The algorithm uses the solution found by the greedy algorithm as the initial state S_0, and its execution time as the initial temperature (line 2-3). s^* is used to store the schedule with the lowest latency and e^* is the energy (latency) of s^*. The algorithm searches until temperature T is less than an epsilon value (line 7-16). T is decreased by a factor of α in every iteration. At each iteration,

it uses *ComputeNeighbor* subroutine to find a neighbor state of the current one and uses *ComputeEnergy* to compute the energy of the neighbor state. If the neighbor state has a lower energy than s^*, it updates s^* (line 10-12). The algorithm uses a probabilistic function P to decide whether to transition from the current state to the neighbor state. The probabilistic function P is defined as follows: if the neighbor state has a lower energy than the current state, the probability is 1; otherwise, the probability is $e^{(e_{current} - e_{neighbor})/T}$.

ComputeNeighbor (Algorithm 2): This routine finds a neighbor state of the current state. It randomly swaps two adjacent subtasks from a random stage. If the neighbor state is invalid, it will undo the change and repeat the random swapping until finding a valid schedule. We select adjacent subtasks rather than any two subtasks in one stage primarily to control the degree of disturbance, which is found to be more effective in practice.

ComputeEnergy (Algorithm 3): This function computes the execution time for a given valid schedule S. We use a memoized recursion to calculate the finish time of the last subtask in each stage and pick the maximum as the execution time. For subtask S_{ij}, its start time is decided by two dependencies: inter-stage data dependency and intra-stage data dependency. The inter-stage dependency is the completion of subtask in the upstream pipeline stage corresponding to the same micro-batch. The intra-stage dependency is the completion of the preceding subtask in the same pipeline stage. Consequently, the end time of S_{ij} is determined by the maximum of these two dependencies plus its own computation time, l_{ij}. The detailed

algorithm is shown in Algorithm 3. It has time complexity of $O(N \times (M_1 + M_2))$.

Optimizing memory usage. When we obtain a latency-optimized solution S^*, we use it as the initial state to run another round of simulated annealing similar to Algorithm 1. However, this time we replace the *ComputeEnergy* function with one that calculates the peak activation memory for the given schedule. Additionally, we only allow state transitions when the neighbor's latency does not degrade. In this way, we can achieve a solution that not only has promising latency but also optimizes the activation memory usage.

6 System Implementation and Optimizations

We implement RLHFuse based on Megatron-LM [16] with 7K lines of code in Python, C++, and CUDA. Megatron-LM applies 3D-parallelism for single-model training. We extend it to support multiple device meshes to launch different tasks asynchronously with tailored parallelism and deployment strategy.

Parallelism strategy configuration. Optimizing the parallel strategy for LLM training is a well-studied problem [13,16,33,34]. Due to the deterministic nature of LLM computation, the execution time and memory cost can be accurately modeled through minimal profiling. An optimal solution can then be found through optimizations [13,25,33,34]. The generation and inference task involves only the forward pass, which is a subproblem of the LLM training. ReaLHF [13] is a state-of-the-art solution that optimizes the parallel strategy for RLHF training and RLHFuse adopts a similar model-then-optimize approach. We build a simulator to accurately estimate the runtime statistics of the tasks under a specific parallel strategy and workload. Then we follow the guidelines in the Megatron-LM [16] paper to prune the design space and brute-force search the optimal strategy.

Stage fusion. For inter-stage fusion, we use the simulator mentioned above to choose the best migration threshold R_t to minimize the fused execution time under a specific workload pattern. For intra-stage fusion, we use MPI [35] to parallelize the simulated annealing with different random seeds. After finding the optimal solution, it will generate the sequences of NCCL [36] operations on each device following the schedule. The runtime will follow the instruction flow to execute the fused pipeline schedule.

System optimizations. Additionally, we adopt a series of system optimizations tailored for RLHF training. These optimizations are implemented in our own system as well as open-source RLHF frameworks like ReaLHF [13]. To separate the differences in low-level system implementations from the core techniques (inter-stage fusion in §4 and intra-stage fusion in §5), our evaluation in §7 uses RLHFuse-Base that includes these optimizations but without inter- and intra-stage fusion as an additional baseline.

Generation stage. We use an in-house inference engine with optimized CUDA kernels. It supports continuous batching [37] to allow the short samples to leave the batch early. For samples that share the same prompt, it supports prefix sharing [38] to avoid key-value cache duplication.

Inference stage. During the post-processing of the inference stage, one metric called Generalized Advantage Estimation [39] is needed which requires recursive computation along the output length dimension for each sample. When the output length is long, it incurs numerous kernel launches which takes even longer than the inference computation itself. To address this, we unroll the recursive formula along the output length dimension and precompute the coefficient matrix. This trick transforms the original recursive computation into a single matrix multiplication, successfully reducing the latency from tens of seconds to just tens of milliseconds.

Training stage. In RLHF training, DP groups need to synchronize after processing each mini-batch. Due to data skewness, different DP groups may have varying workloads, leading to stragglers and wasted GPU time. To mitigate this, we evenly distribute each mini-batch across the DP groups based on the sequence length of the samples, ensuring that the workloads are roughly balanced across the DP groups. Note that this will not violate the original training semantics.

Task switching. The primary overhead during task switching lies in model weights reloading. For the `Actor` and `Critic` models, the latest weights after the training stage need to be redistributed to the appropriate devices according to the new parallel strategy for generation or inference. This communication pattern is well-studied [13, 14, 40] and we adopt a similar all-to-all approach to minimize cross-node communication. For the `Ref` and `RW` models whose weights remain unchanged, we keep them in CPU memory and swap them into GPU memory as needed. The swapping can overlap with the computation of previous tasks. With these optimizations, the task switching overhead is negligible compared to the total iteration time, as shown in §7.2.

7 Evaluation

In this section, we evaluate RLHFuse under different sizes of LLMs ranging from 13B to 65B on real-world datasets. First, we compare the end-to-end performance of RLHFuse to other RLHF training frameworks (§7.1). Then we present the latency breakdown to delve into the performance improvement of RLHFuse (§7.2) and conduct ablation studies of our proposed techniques (§7.3). Finally, we provide a case study to show one execution plan generated by RLHFuse (§7.4).

Cluster testbed. We deploy RLHFuse on a production cluster for RLHF training with 32 nodes and 256 GPUs. Each node has 2TB of memory and 8 NVIDIA SXM H100-80GB GPUs connected with 900GB/s NVLINK. Nodes are connected by
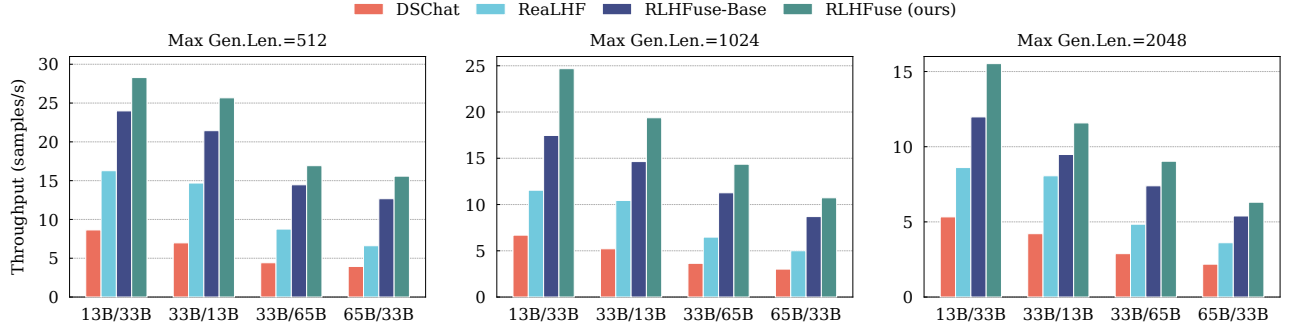
Figure 7: End-to-end throughput of RLHF training systems under different generation and model size settings.

Models	# of Layers	# of Heads	Hidden Size	Intermediate Size
LLaMA-13B	40	40	5120	20480
LLaMA-33B	60	52	6656	26624
LLaMA-65B	80	64	8192	32768

Table 2: LLM specifications.

$8 * 200$ Gbps RDMA network based on RoCEv2 with rail-optimized topology. The simulated annealing algorithm in §5 is conducted on four CPU nodes, each with 2 AMD 9654 CPUs and there are 768 physical cores in total.

Models and datasets. Following previous work [14], we choose the LLaMA models [41] ranging from 13B to 65B, which is a popular LLM family used in academia and industry. The detailed specifications are listed in Table 2. We use the HH-RLHF [42] dataset, which is open-sourced by Anthropic to train a helpful and harmless assistant with RLHF.

Settings. Since the `Actor` and `Ref` models are of the same size, as are the `Critic` and `RW` models, we choose four different model size settings for `Actor`/`Critic` pair : 13B/33B, 33B/13B, 33B/65B, and 65B/33B. We also vary the maximum output length to see the performance under different generation settings. In each training iteration, we use a global batch size of 512, a mini-batch size of 64, and take one gradient step per mini-batch following the LLaMA technical report [43].

Metrics. For the end-to-end experiment, we measure the *sample throughput* following previous work [14]. Sample throughput is defined as the average number of samples processed per second. Under each setting, we record the sample throughput over 20 consecutive training iterations after warm-up.

7.1 End-to-End Results

We compare the end-to-end performance of RLHFuse against the following RLHF training frameworks. For baselines, we explore all feasible configurable options and report the best performance achieved without out-of-memory errors.

- **DeepSpeed-Chat [44]** (DSChat) only supports ZeRO-3 data parallelism [45] during training and utilizes a *HybridEngine* to switch from ZeRO-3 DP to TP in the generation stage. Since each GPU requires at least one sample during training and it only supports ZeRO-3 DP, we increase

its mini-batch size to 256 while maintaining the original global batch size in order to run it successfully on our cluster. Note that with a larger batch size, this adjustment is more favorable for its throughput performance.
- **ReaLHF [13]** proposes parameter reallocation to flexibly redistribute parameters between tasks, enabling tailored 3D-parallel strategy for each task. However, as discussed in §2.2, it suffers from the issues of data skewness and pipeline bubbles without employing the subtask-level optimizations.
- **RLHFuse-Base** is RLHFuse without inter- and intra-stage fusion but with all the system optimizations enabled. We include this baseline to demonstrate the performance improvements brought by our core techniques in §4 and §5, eliminating any unfair comparisons caused by the differences in underlying framework implementations.

Figure 7 shows the end-to-end performance of the RLHF training systems under three generation settings. We set the maximum generation length to 512, 1024, and 2048, respectively. Under a specific generation setting, the configuration X/Y denotes parameter counts in actor and critic models, mirrored in the reference and reward models, respectively. Compared to DSChat, RLHFuse achieves $2.5\times$–$3.7\times$ higher throughput due to our flexible execution plan generator, which selects the most efficient 3D-parallel strategy for each task. Compared to ReaLHF, which also supports tailored parallel strategies, RLHFuse achieves $1.4\times$–$2.4\times$ higher throughput due to more fine-grained system optimizations. Specifically, we observe that the primary speedup comes from the customized kernels in the inference engine and the mini-batch load balancing during the training stage. Compared to RLHFuse-Base, our system achieves a relative improvement of $1.2\times$–$1.4\times$ on the throughput. This improvement is entirely from stage fusion, which significantly alleviates the issues of data skewness in the generation stage and pipeline bubbles in the training stage. In the following, we conduct a detailed breakdown analysis to show the performance improvement.

7.2 Performance Analysis

To further understand the performance improvement of RLHFuse, we show its RLHF iteration breakdown and compare it with RLHFuse-Base. As shown in Figure 8, each row rep-
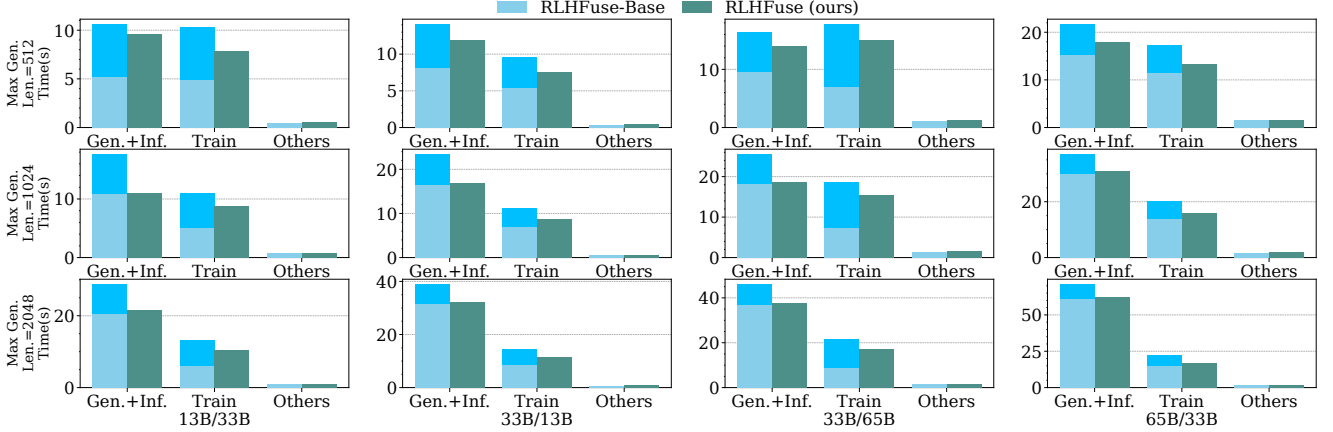
Figure 8: The RLHF iteration breakdown under different generation and model size settings.
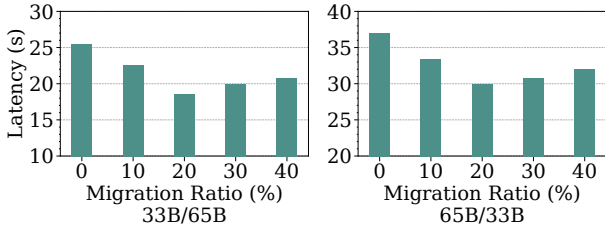

Figure 9: The fused generation and inference stage execution time of RLHFuse under different migration ratio and model settings. The maximum generation length is set to 1024.

Settings				Latency Speedup relative to 1F1B			Memory Usage relative to 1F1B	
Models	PP0	PP1	#MB	greedy	ours	optimal	greedy	ours
33B/13B	8	4	8	1.29	**1.38**	**1.38**	1.51	**1.0**
			16	1.12	**1.30**	**1.30**	2.14	**1.0**
			32	1.10	**1.15**	**1.15**	2.75	1.26
		8	8	**1.4**	**1.4**	**1.4**	1.51	**1.0**
			16	1.15	**1.32**	**1.32**	2.00	1.19
			32	1.08	**1.17**	**1.17**	2.75	1.31
65B/33B	16	8	16	1.28	**1.48**	**1.48**	1.61	**1.0**
			32	1.14	**1.27**	**1.27**	2.23	**1.0**
			64	1.12	**1.15**	**1.15**	2.88	1.26
		16	16	1.27	**1.5**	**1.5**	1.57	**1.0**
			32	1.16	**1.33**	**1.33**	2.00	1.22
			64	1.09	1.16	**1.17**	2.88	1.47

Table 3: The latency speedup and peak activation memory cost of pipeline schedules found by different algorithms under different model, pipeline stage, and micro-batch settings.

resents a generation setting, and each column represents a model setting X/Y. We divide one RLHF iteration into three main parts: the generation plus inference stage (Gen.+Inf.), the training stage (Train), and all other overheads (Others), such as the data transmission and weights redistribution time.

For Gen.+Inf., since RLHFuse-Base does not perform inter-stage fusion and executes the two stages in serial, we use light and dark colors to represent the time spent on the generation and inference stages respectively to better show the speedup of inter-stage fusing. It can be seen that as the maximum generation length increases, the processing time of long-tailed samples is long enough for RLHFuse to fully overlap the inference stage execution, achieving $1.2\times$–$1.6\times$ speedup. For the training stage, similarly, we use light and dark colors to represent the Actor and Critic training time in RLHFuse-Base. Through intra-stage fusion, RLHFuse greatly mitigates pipeline bubbles and reduces the execution time of the training stage by $1.2\times$–$1.3\times$. As for the other overhead, thanks to our optimizations in task switching (§6), it only accounts for less than 3% of the total iteration time, and the migration overhead of inter-stage fusion is also negligible.

7.3 Effectiveness of Stage Fusion

In this section, we show the effectiveness of RLHFuse's key techniques in §4 and §5.

Inter-stage fusion. One key parameter in inter-stage fusion is the migration threshold R_t. As R_t increases from zero, it

shifts from fully serial execution to more aggressive fused execution. However, if R_t is set too large, the long-tail generation instances may be overloaded, extending the original generation time. We measure the execution time of fused generation and inference stage under different migration ratios (R_t/BS) as shown in Figure 9. We can see the optimal latency is achieved when the remaining samples account for about 20% of the batch size. Note that R_t is related to the output length distribution of the model and this experiment is conduct during the early stage of training. As training progresses, R_t requires periodic adjustments to adapt the distribution change. In practice, such changes are usually not too drastic, so the adjustment frequency remains low.

Intra-stage fusion. We compare RLHFuse's simulated annealing algorithm against the greedy approach mentioned in §5.2 and show the latency speedup and peak activation memory usage relative to the serial execution of the two models with 1F1B schedule in Table 3. For the latency, we also include a lower bound calculated as follows: we first compute the earliest possible completion time on each stage, which consists of three parts: the earliest possible arrival time of the
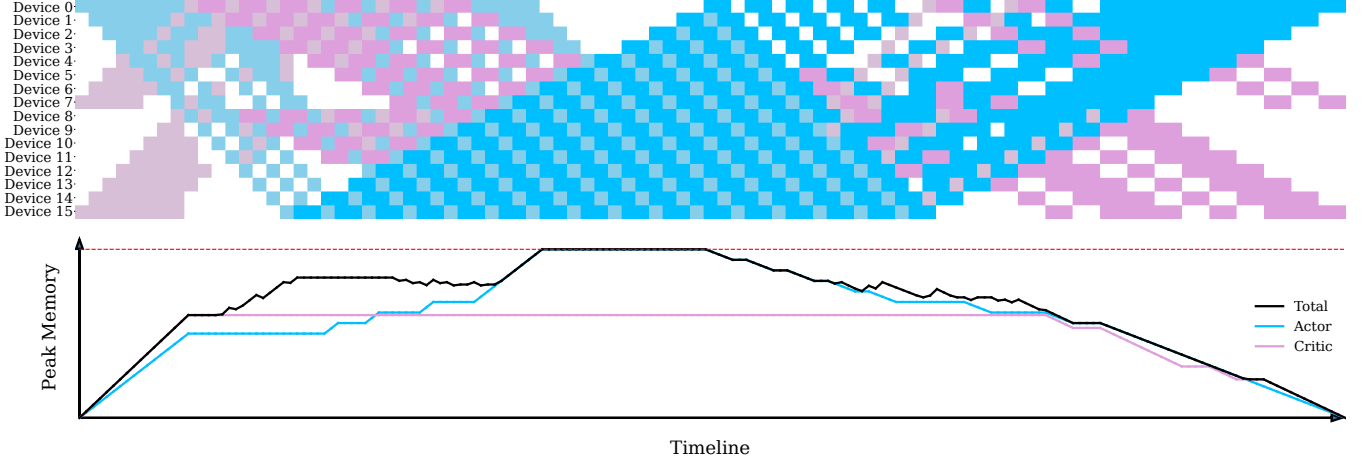
Figure 10: The fused pipeline schedule of 65B `Actor` and 33B `Critic` model generated by RLHFuse. The two models have 16 and 8 PP stages respectively and #micro-batches is set equal to PP. Top: the GPU execution timeline. Bottom: the peak activation memory usage timeline. The dotted horizontal line denotes the memory cost of serial 1F1B execution.

first task, the total time required to process all the tasks, and the remaining time needed for the subsequent pipeline stages of the final task; we then take the maximum of these times across all stages as the lower bound of the fused pipeline schedule. Note that there may not necessarily exist a schedule that reaches this lower bound, but we can use it as a metric to assess how close our algorithm is to the optimal solution. Under all the settings, our approach outperforms the greedy method and achieves the theoretical lower bound except for the last case. As for the memory cost, the serial 1F1B execution serves as the lower bound for the fused schedule. Our method also greatly outperforms the greedy approach and often reaches the lower bound. Even when it does not achieve the lower bound, the extra overhead remains within an acceptable range.

7.4 Deep Dive

In this section, we deep dive into RLHFuse by showing a fused pipeline schedule produced by RLHFuse for the 65B/33B model setting. The upper of Figure 10 shows the GPU execution timeline, with forward and backward tasks in light and dark colors, respectively. With intra-stage fusion, RLHFuse strategically fuses one 65B model of 16 PP stages (blue grids) with two 33B models that each has 8 PP stages (pink grids) to fill the pipeline bubbles of each other in reversed pipeline directions. Ultimately, the execution time of this fused pipeline schedule is the same as individually executing the 65B model using 1F1B schedule, which means we completely overlap the training of the 33B model, achieving the theoretical lower bound. Meanwhile, as shown in the bottom of Figure 10, the peak activation memory usage also achieves the lower bound of executing the two models in serial with 1F1B schedule. This complex schedule, which achieves optimal performance in both time and memory usage, is difficult to design manually, further demonstrating

the effectiveness of our fused pipeline schedule generation algorithm proposed in §5.2.

8 Related Work

RLHF training systems. As RLHF gradually becomes the primary approach for LLM alignment, many distributed frameworks are specifically designed for RLHF training. trlX [46] proposes algorithmic system enhancements to reduce compute and memory requirements. OpenRLHF [47] notices that different RLHF training tasks have different resource requirements and proposes to designate customized parallel strategies for different tasks. ReaLHF [13] further proposes parameter reallocation to dynamically redistribute LLM parameters on shared GPU devices to avoid under-utilization. PUZZLE [14] proposes lightweight context switching to reduce the task switching overhead, which is orthogonal and partly integrated into RLHFuse. However, all the above systems overlook the subtask-level optimization opportunities and suffer from the data skewness and pipeline bubble issues.

LLM generation optimizations. Many systems focus on optimizing the LLM generation performance [37, 38, 48–50], and many of them have been adopted by RLHFuse to accelerate the generation stage. Besides, AlpaServe [51] multiplexes the LLM execution with colocated parallelism to improve the inference throughput under bursty workload. FastServe [27] proposes preemptive scheduling on iteration-level to mitigate the head-of-line blocking caused by long-tailed samples. Split-wise [52] and DistServe [26] split the prefill and decoding phases to avoid interference between them. These works are orthogonal to RLHFuse and can be integrated into RLHFuse.

LLM training optimizations. LLM training has been studied by many research works from various aspects. In particular, GPipe [18] and (Interleaved) 1F1B [53, 54] use pipeline parallelism to improve the training throughput and reduce mem-

ory footprint. Chimera [20] combines bi-directional pipeline parallelism to further reduce pipeline bubbles. Alpa [55] automatically generates the optimal parallel strategy to improve training performance. MegaScale [17] provides a detailed experience in building a production LLM training system at a large scale. DistTrain [56] focuses on addressing the model and data heterogeneity in multimodal LLM training. These works are agnostic to RLHF thus do not utilizes the characteristics to optimize the RLHF training stage.

9 Conclusion

We present RLHFuse, a highly efficient RLHF training framework for LLMs. By carefully revisiting the inherent characteristics and task dependencies, RLHFuse views the RLHF workflow from a finer-grained subtask-level perspective while preserving the original synchronous training semantics. This opens up opportunities for efficient inter- and intra-stage fused execution, greatly mitigating the data skewness and pipeline bubbles in existing systems. The evaluation shows that RLHFuse can significantly increase GPU utilization, boosting training throughput by up to 3.7× compared to current state-of-the-art systems.

References

[1] "Perplexity is a free ai-powered answer engine that provides accurate, trusted, and real-time answers to any question.." https://perplexity.ai, 2024.

[2] "Introducing chatgpt." https://openai.com/blog/chatgpt, 2022.

[3] "Apple intelligence preview." https://www.apple.com/apple-intelligence/, 2024.

[4] "Figure is the first-of-its-kind ai robotics company bringing a general purpose humanoid to life.." https://figure.ai, 2024.

[5] S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith, "Realtoxicityprompts: Evaluating neural toxic degeneration in language models," 2020.

[6] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?," in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, (New York, NY, USA), pp. 610–623, 2021.

[7] S. Lin, J. Hilton, and O. Evans, "Truthfulqa: Measuring how models mimic human falsehoods," 2022.

[8] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama,

A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[10] "Introducing gemini: our largest and most capable ai model." https://blog.google/technology/ai/google-gemini-ai/, 2023.

[11] "Introducing the next generation of claude." https://www.anthropic.com/news/claude-3-family, 2024.

[12] "Qwen2.5: A party of foundation models!." https://qwenlm.github.io/blog/qwen2.5/, 2024.

[13] Z. Mei, W. Fu, K. Li, G. Wang, H. Zhang, and Y. Wu, "Realhf: Optimized rlhf training for large language models through parameter reallocation," *arXiv preprint arXiv:2406.14088*, 2024.

[14] K. Lei, Y. Jin, M. Zhai, K. Huang, H. Ye, and J. Zhai, "{PUZZLE}: Efficiently aligning large language models through {Light-Weight} context switch," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 127–140, 2024.

[15] "Gpt-4o long output." https://openai.com/gpt-4o-long-output/, 2024.

[16] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[17] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, *et al.*, "{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 745–760, 2024.

[18] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism ," in *NeurIPS*, 2019.

[19] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *SOSP*, 2019.

[20] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

[21] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble pipeline parallelism," *arXiv preprint arXiv:2401.10241*, 2023.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[23] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[24] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," 2024.

[25] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *ACM SOSP*, 2019.

[26] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[27] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[28] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[29] H. Emmons and G. Vairaktarakis, *Flow shop scheduling: theoretical results, algorithms, and applications*. Springer Science & Business Media, 2012.

[30] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, pp. 671–680, 1983.

[31] Z. Zhang, Y. Zhong, R. Ming, H. Hu, J. Sun, Z. Ge, Y. Zhu, and X. Jin, "Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models," 2024.

[32] C. Sun, S. Huang, and D. Pompili, "Llm-based multi-agent reinforcement learning: Current and future directions," *arXiv preprint arXiv:2405.11106*, 2024.

[33] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *USENIX OSDI*, 2022.

[34] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui, "Galvatron: Efficient transformer training over multiple gpus using automatic parallelism," *Proceedings of the VLDB Endowment*, pp. 470–479, 2022.

[35] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[36] "Optimized primitives for collective multi-gpu communication." https://github.com/NVIDIA/nccl, 2024.

[37] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *USENIX OSDI*, 2022.

[38] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," 2024.

[39] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[40] Y. Zhuang, H. Zhao, L. Zheng, Z. Li, E. P. Xing, Q. Ho, J. E. Gonzalez, I. Stoica, and H. Zhang, "On optimizing the communication of model parallelism," 2024.

[41] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[42] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan, "Training a helpful and harmless assistant with reinforcement learning from human feedback," 2022.

[43] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

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

[45] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models." ArXiv, May 2020.

[46] A. Havrilla, M. Zhuravinskyi, D. Phung, A. Tiwari, J. Tow, S. Biderman, Q. Anthony, and L. Castricato, "trlx: A framework for large scale reinforcement learning from human feedback," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8578–8595, 2023.

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

[48] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[49] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, H. Dong, and Y. Wang, "Flashdecoding++: Faster large language model inference on gpus," *arXiv preprint arXiv:2311.01282*, 2023.

[50] L. Chang, W. Bao, Q. Hou, C. Jiang, N. Zheng, Y. Zhong, X. Zhang, Z. Song, Z. Jiang, H. Lin, *et al.*, "Flux: Fast software-based communication overlap on gpus through kernel fusion," *arXiv preprint arXiv:2406.06858*, 2024.

[51] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," *arXiv*, 2023.

[52] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024.

[53] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, *et al.*, "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*, pp. 431–445, 2021.

[54] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

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

[56] Z. Zhang, Y. Zhong, R. Ming, H. Hu, J. Sun, Z. Ge, Y. Zhu, and X. Jin, "Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models," *arXiv preprint arXiv:2408.04275*, 2024.