# Systems Opportunities for LLM Fine-Tuning using Reinforcement Learning

Pedro Silvestre
p.silvestre21@imperial.ac.uk
Imperial College London
United Kingdom

Peter Pietzuch
prp@imperial.ac.uk
Imperial College London
United Kingdom

## Abstract

Reinforcement learning-based fine-tuning (RLFT) has emerged as a crucial workload for enhancing large language models (LLMs). RLFT workflows are challenging, involving nested loops, multiple models, dynamically shaped tensors and interleaving sequential generation and parallel inference tasks. Despite these complexities, current RLFT engines rely on coarse-grained algorithm representations, treating each component as an independently optimized black-box. As a result, RLFT engines suffer from redundant computations, scheduling overhead, inefficient memory management, and missed opportunities for parallelism.

We argue that a fine-grained representation is needed to enable holistic optimization for RLFT workloads. Additionally, we demonstrate that existing declarative deep learning engines fail to optimize RLFT workloads end-to-end due to their need for static tensor shapes and loop bounds, leading to excessive peak memory usage and unnecessary computations. Through micro-benchmarks, we quantify these inefficiencies and show that addressing them could enable more efficient and flexible execution. We propose an RLFT system design based on a fine-granularity representation, opening the door to generalizable optimizations, and paving the way for more scalable and efficient RLFT systems.

**ACM Reference Format:**
Pedro Silvestre and Peter Pietzuch. 2025. Systems Opportunities for LLM Fine-Tuning using Reinforcement Learning. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3721146.3721944

## 1 Introduction

Large Language Models [64] (LLMs) have quickly become the key technology powering diverse applications such as translation [64], chatbot assistants [11] and code-generation [12].

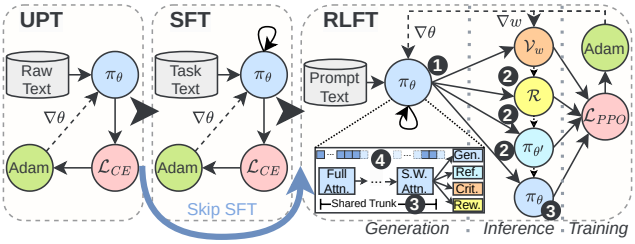



**Fig. 1: LLM Training Pipeline.** RLFT systems represent algorithms at the granularity shown, leading to inefficiencies: ❶ sequential computation-communication; ❷ missed parallelism, ❸ redundant work; and ❹ suboptimal KV cache management.

The traditional LLM training pipeline, depicted in Fig. 1, involves all major deep learning (DL) paradigms: (i) unsupervised pre-training (UPT) on a large corpus of text data teaches the model the structure of language and embues it with general knowledge; (ii) supervised fine-tuning (SFT) on a smaller high-quality dataset adapts the model to a specific task (e.g. chat); finally (iii) reinforcement learning (RL) from human feedback (RLHF) aligns the model with human values and goals. However, SFT has been shown to degrade LLM generalization [67, 75], and underperform RLHF on helpfulness and safety [35], leading researchers to skip SFT [76] and focus further on direct RL-based fine-tuning (RLFT) methods.

RLFT is an increasingly popular generalization of RLHF, and has been recently applied to endow models with novel capabilities, such as structured output generation [43], greatly improved programming ability [20], and mathematical [54], visual [61] and chain-of-thought reasoning [24]. While UPT can only be undertaken by few organizations due to its cost [49] (~$100M), RLFT is a more accessible way to provide new capabilities to and customize the behavior of pre-trained LLMs (~$10K). As long as the task can be framed as a reward function [58], RLFT can be applied.

However, RLFT is a complex and challenging workload to optimize (§2), due to the diversity of algorithms and model architectures (§2.1), the scale and number of models involved (§2.2), and the nested computations and dynamic shapes of the dataflows between them (§2.3). Furthermore, RLFT can be roughly split into three stages (Fig. 1) with diverse demands: (i) sequential *generation* of a response from a prompt, (ii) parallel *inference* of the response by several models and (iii) *training* of the models. Such challenges demand efficient, careful coordination of distributed execution, that can


*EuroMLSys '25, Rotterdam, Netherlands*

ACM ISBN 979-8-4007-1538-9/25/03
https://doi.org/10.1145/3721146.3721944

only be achieved through end-to-end *whole-graph* algorithm representation, optimization and scheduling.

Despite this, RLFT systems today use a coarse-grained graph [44, 56] or have no optimizable graph representation [26, 65, 71]. Instead each component (models, PPO, optimizer) is treated as a black-box, preventing holistic compiler optimization and planning that crosses these barriers. This not only makes RLFT systems difficult to adapt to novel workloads, but results in diverse system inefficiencies (represented by the numbers in Fig. 1) such as:

**(❶) Lack of Computation-Communication Overlap.** During response generation, responses are only transferred once generation finishes, but could be pipelined;

**(❷) Missed Parallelism.** Pipelining transfers (❶) would allow downstream inference models (e.g., critic, reference) to incrementally process responses in parallel with generation. Instead, RLFT engines today will first finish generation, before executing each inference model sequentially;

**(❸) Redundant Work.** Opportunities for reuse are missed and instead recomputed, such as when generator and critic share a trunk [22], or when the logits computed at generation-time are recomputed during inference;

**(❹) Suboptimal Memory Management.** Many models use varied and often interleaved attention mechanisms [47, 60, 74] to drastically reduce memory requirements, requiring fine-grained memory management to be effective. However, current systems use a one-size-fits-all KV-cache policy, leading to suboptimal memory usage and performance.

Furthermore, we demonstrate that existing graph-based DL systems (JAX or TensorFlow) are not a suitable basis for whole-graph RLFT optimization (§3.1), due to their requirement for compile-time *static shapes*. This requirement forces system developers into a choice between suboptimal memory management and redundant work caused by padding and masking or, alternatively, breaking the RLFT graph into smaller pieces, once again preventing holistic optimization. We discuss how this affects the design of RLFT systems today (§3.2), causing them to be imperative and built by ad-hoc composition of specialized engines (e.g., vLLM for generation and DeepSpeed for inference and training), leading to brittleness and impedance mismatch overhead (§3.3). For example, we find that while the use of vLLM speeds up generation, it spends ~40% of the overall iteration time in different scheduling overheads.

To address these challenges, we instead argue that novel *declarative* approaches are needed (§4), where entire RLFT algorithms can be represented as a single, *fine-granularity* computation graph. Such a representation must capture dynamic accesses patterns and shapes using symbolic expressions, allowing for holistic optimization of computation, memory management, and communication by a compiler. We conclude with a discussion of open research directions and challenges in building a declarative RLFT system (§5).

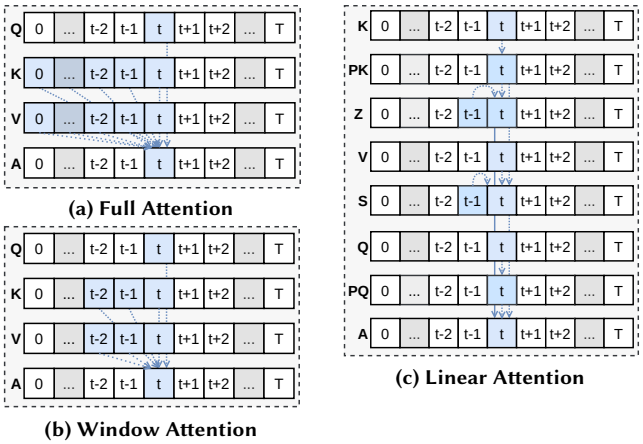



**Fig. 2: Dynamic Access Patterns of Attention Mechanisms.** Different attention mechanisms depend on different past tokens, requiring unique memory management strategies.

## 2 Challenges in RLFT

Generically, RLFT algorithms (Fig. 1) begin with *generation*, by selecting a batch of prompt strings from a dataset, embedding them, and generating a *response* using the generator model $\pi_\theta$. The generation process is itself split into two phases: *prefill* and *decoding*. The prefill phase is highly parallel, and involves processing the prompt tokens to populate a key-value (KV) cache (§2.1). Sequential decoding then generates the LLM response autoregressively, meaning that each further token output by the generator is fed back as input to generate the next token, until an end-of-sequence (EOS) token is generated. During the *inference* stage, the generated sequence is processed by downstream models. A reference model $\pi_{\theta'}$ may be used to prevent divergence between the current generator and the pre-training token distributions [76]. A critic model $\mathcal{V}_w$ may be used to subtract a baseline from the reward signal [68]. The reward function $\mathcal{R}$, which may be a model or a set of *mechanical rules*, is used to generate a reward signal to learn from. Mechanical rules may use compilers, unit tests or math checkers, to verify the correctness of the generated response. Finally, during the *training* stage, an RL algorithm like PPO [53] or GRPO [54] is used together with an optimizer such as Adam [31] to update the generator and critic parameters.

### 2.1 Algorithm Variety

The first challenge in designing systems for RLFT is the growing number of RLFT workloads.

**Attention.** Attention is the core operation behind LLMs. It allows certain tokens to attend to others, influencing their meaning. The full attention formula computes a weighted sum of all past values based on the similarity of keys and queries, and is given by (Eq. 1), with the attention pattern $S(t) = \{0 : t + 1\}$, as depicted in Fig. 2a. However, this scales quadratically with the sequence length, making it infeasible

for long sequences. Variations, such as sliding window attention [10] (Fig. 2b) or block attention [73], can be achieved by changing the attention pattern to $S(t) = \{t-w:t+1\}$ or $S(t) = \{b \cdot \lfloor t/b \rfloor : t+1\}$, dramatically reducing the size needed for storing prior KVs (KV cache). However, sliding window attention loses the long-range dependencies of full attention. Instead, linear attention [30] (Eq. 2) uses a radically different approach based on kernel approximation $\phi$. Each of these mechanisms requires unique memory management strategies for efficiency.

$$A[t] = \sigma\left(\frac{Q[t]K[S(t)]^\intercal}{\sqrt{d_k}}\right)V[S(t)], S(t) = \begin{cases} 0:t+1 \\ t-w:t+1 \end{cases} \quad (1)$$

$$\begin{aligned} PK[t], PQ[t] &= \phi(K[t]), \phi(Q[t]) \\ Z[t] &= Z[t-1] + PK[t], & Z[0] &= 0 \\ S[t] &= PK[t]V[t]^\intercal + S[t-1], & S[0] &= 0 \\ A[t] &= PQ[t]^\intercal S[t] / PQ[t]^\intercal Z[t] \end{aligned} \quad (2)$$

**Model Architectures.** Attention is typically computed in parallel by multiple heads and concatenated before undergoing a linear map. GQA [3] modifies this by having fewer value and key heads than query heads. On the other hand, mixture-of-experts architectures [28] use a dynamic gating mechanism to select which experts to use for each token. In general, each model is independent, but some works [22] explore using a shared frozen trunk, with different heads for each model ($\pi_\theta$, $\pi_{\theta'}$, $\mathcal{V}_w$, $\mathcal{R}$). Others have explored mixing attention types, using full attention only in early layers [74] or interleaving full and window attention [47, 60].

**RLFT Algorithms.** Many RLFT algorithms have emerged, tweaking the original RLHF [8] for different purposes. Safe-RLHF [15] adds a cost model to explicitly penalize unsafe generations, while ReMax [40] replaces the critic with another reward model. RLAIF [34] uses an LLM as a reward model itself. GRPO [54] and RLOO [2] perform multiple generations per prompt to replace the critic, but the former uses mechanical evaluators, while the later uses a large model. Additionally, RLOO skips certain PPO steps, such as clipping, GAE computation and doing multiple optimization steps over the generation. DPO [50] does away with both reward and critic models, using the generator token probabilities to directly align to preferred responses. Finally, all of these can be modified by parameter efficient fine-tuning techniques [17], such as Lora [25], which can be used to train only a low-rank approximation of the models.

### 2.2 Scale

The scale of RLFT presents another set of challenges on effectively mapping work to available resources. Large models necessitate sharding across multiple accelerators in order to both reduce memory pressure and parallelize computation. We now discuss the different approaches used to scale LLM training.

**3D parallelism** includes data parallelism (DP), pipeline parallelism (PP) and tensor parallelism (TP). DP [32] replicates the model across multiple accelerators, with each accelerator processing a different minibatch of data, synchronizing gradients at the end of each iteration with collective communication. PP [27] has each accelerator processing a different set of layers, and overlaps microbatches to reduce pipeline stalls. TP [57] splits across model layers, with mny accelerators processing the same layer, and synchronizing through collective communication. While DP and PP are typically applied over different hosts, TP is typically applied within a single host [77], as the communication overhead is lower.

**LLM-specific parallelisms** include context parallelism (CP) and sequence parallelism (SP), which optimize the full attention computation. SP [39] is similar to DP but splits data over the sequence dimension, allowing each accelerator to compute local Qs, Ks and Vs, before circulating Ks and Vs in a ring so that attention can be computed. CP [42] further improves on SP by using a softmax decomposition [46] to implement blockwise transformers [41], which enable compute and communication overlap.

**Other techniques** employed to scale LLM training further include mixed precision training [45] which quantizes weights, Zero [51] which shards training states across DP workers and swapping [52], which moves tensors to and from host memory.

Importantly, unlike the traditional supervised paradigm in which these techniques are applied to a single model, RLFT involves multiple models with data dependencies between them, necessitating careful coordination for performance.

> **Key Takeaway 1:** The combinatorial space of RLFT workloads and parallelism strategies is vast, preventing effective manual optimization.

### 2.3 Dynamic Computation

A final challenge in RLFT is the dynamic nature of the computation, which expresses itself in multiple ways. First, the size of the prompts fetched from the dataset can vary. Second, the size of the generated responses can vary, and is determined by when an end-of-sequence (EOS) token is generated. Furthermore, during training, the average response length of the model can change. For example, during the training of DeepSeek-R1 [24], the average response length increases by over an order of magnitude, leading to a significant increase in memory usage and computation. Finally, attention itself is dynamic, with its input shapes changing with the current sequence length (Fig. 2). This dynamism makes it difficult to statically optimize RLFT workloads (§3).

## 3 Issues with Existing Systems

In this section, we show that despite the fine-grained dataflow graphs used by existing DL frameworks being powerful, they

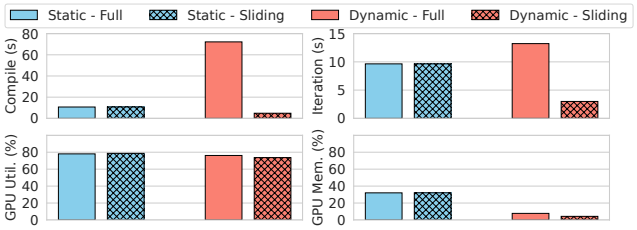



**Fig. 3: JAX/XLA Attention Microbenchmark.** Existing graph representations require static pre-allocation of buffers and masking, preventing memory-savings and causing redundant work for sliding window attention, while dynamic execution is expensive.

are not flexible enough for RLFT algorithms (§3.1). We then show how this forces RLFT systems to be imperative, and thus lack flexibility and whole-graph optimization (§3.2).

### 3.1 Dataflow Graph Compilers

Graph-based DL systems (e.g., TensorFlow [1] or JAX/XLA [19]) today are essentially *compilers* [37]. DL compilers first build a representation of the computation as a graph, then leverage information statically available at compile-time (e.g., operation flops, tensor shapes, hardware characteristics) to generate efficient specialized code [13], plan memory usage [19] and distribute computation across devices [77]. Since (un)supervised learning is static, repeating the same forward and backward pass on static tensor shapes, DL compilers have been designed and optimized for this use-case.

However, as we showed in §2.3, the assumption of static computation is broken in RLFT. In order to model dynamic computations in a single static dataflow graph, users are thus forced into one of two undesirable choices. The first option, available in JIT-compiled systems like JAX, is for users to give up on whole-graph optimization, and instead recompile the inner attention computation for each new shape. This causes large compilation times and prevents whole-graph optimization, but achieves low memory usage. Alternatively, to enable whole-graph compilation, users must make the computation static by setting all dynamic shapes to their maximum possible sizes (which may be unknown, requiring overestimation). Users must then pre-allocate buffers for that size and apply padding and masking to the computation in the correct locations. While this allows for whole-graph optimization, it loses the fine-grained semantics of the dynamic attention patterns, involves significant user effort, and leads to wasted memory and computation.

This is visible in Fig. 3, where we benchmark autoregressive decoding in JAX, using both full and sliding window attention. The memory usage and iteration time of the static version is the same for both full and windowed attention, as the windowed attention is achieved by applying a mask. Furthermore, the dynamic version achieves 4.1× and 7.5× lower memory usage than the static version for full and windowed attention, respectively. Discounting compilation time, while the static version is faster for full attention, it is 3.2× slower for windowed attention than the dynamic version, due to needing to compute all masked positions.

**Tab. 1: A comparison of existing RLFT systems**

| System | DL Eng. | Graph Based | Multi Algo. | Unif. Eng. | Par. Space |
|---|---|---|---|---|---|
| NeMo-Aligner[55] | PT | ✗ | ✓ | ✓ | 3D§ |
| OpenRLHF[26] | PT | ✗ | ~ | ✗* | 3D§ |
| DS-Chat[71] | PT | ✗ | ~ | ✓ | 3D§ |
| TRL[65] | PT | ✗ | ✓ | ✗* | 3D§ |
| FlexRLHF[69] | PT | ✗ | ✗ | ✓ | 3D§ |
| Puzzle[36] | PT | ✗ | ✗ | ✗‡ | 3D§ |
| ReaLHF[44] | PT | ✓◊ | ✓ | ✓ | 3D |
| RLHFuse[79] | PT | ✓◊ | ✓ | ✗† | 3D |
| HybridFlow[56] | PT | ✓◊ | ✓ | ✗* | 3D |

* vLLM for generation. † In-house vLLM-like engine for generation.
‡ DeepSpeed-Inference for generation. § Fixed strategy. ◊ Coarse-grained.

> **Key Takeaway 2:** Existing graph-based DL systems are not optimized for the dynamic computations that emerge from sequence processing and RL workloads.

### 3.2 Existing RLFT Systems

In Tab. 1 we survey the key properties of existing RLFT systems work. Due to the need for dynamic shapes, most RLFT systems today instead build on top of PyTorch (PT) [48], which provides an imperative programming model. As a consequence, no RLFT systems today optimize the computation end-to-end, due to the lack of a graph representation. Instead, it is common to leverage tracing-based JIT compilers, such as torch.compile [5] to provide some optimization at the level of individual models. Additionally, no work goes beyond 3D parallelism, as they build on 3D parallel engines [51, 57].

**Library Systems.** TRL [65] and NeMo-Aligner [55] support a large number of algorithms without a representation, through the implementation effort of large teams of contributors. DS-Chat [71] and OpenRLHF [26] support considerably fewer algorithms, which are simple variations of RLHF. While DS-Chat collocates all models on all devices, OpenRLHF seperates them completely, while NeMo-Aligner collocates the generator and reference models on one set of devices and the critic and reward on another. These systems thus offer little flexibility, as the same distribution strategy is used for all algorithm and model variations.

**RLHF-coupled Distribution Optimization.** Most systems work focuses on high-level distribution strategies for RLHF, which are highly coupled to the algorithm. DS-Chat [71] first proposes using a hybrid engine to switch the generator from data parallelism for training to TP for generation. OpenRLHF [26] first proposes utilizing vLLM to handle the generation stage, as well as, offloading optimizer states for larger batch sizes. FlexRLHF[69] notes that trainable models (generator and critic) require much more memory, informing

two model placement strategies: interleaved and disaggregated. Interleaved improves inference times by leveraging intra-node TP for reward and reference models, while the disaggregated strategy improves training times for heterogeneous clusters by allowing uneven shardings. Puzzle [36] also optimizes the communication in RLHF with a new schedule for inference, while also using similar 3D parallelism strategies for the generator model for both generation and inference stages, minimizing the redistribution overhead. What these projects have in common is that they all attempt to manually design efficient distribution strategies for RLHF, remaining highly coupled to it.

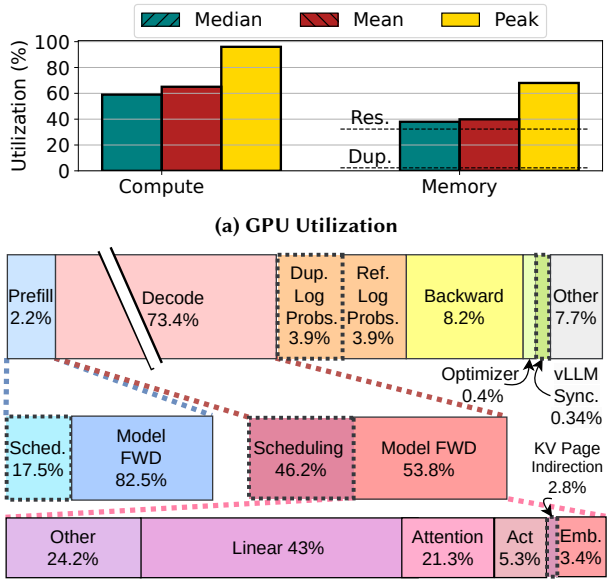
**Coarse-grained Dataflow Graphs.** A number of systems build a coarse-grained dataflow graph of model invocations, allowing them to plan high-level distributions strategies, and more easily support new algorithms. RealHF uses profiling and MCMC to search for fast end-to-end 3D parallelism strategies for each model and task. RLHFuse [79] builds on RealHF and shows that long-tailed generations delay the start of the inference stage, and proposes resolving this by dynamically migrating long-tail sample generation to a specific set of devices, allowing inference to start for finished samples. They also propose using chimera [38] pipeline parallelism during inference between generator and critic, reducing pipeline stalls. HybridFlow [56] instead exhaustively searches the space and provides users more control over low-level communication. However, coarse dataflow graphs cannot support the holistic optimizations needed for the best performance, as each node is still a black-box.

> **Key Takeaway 3:** Existing RLFT systems are coupled to RLHF, focus on 3D parallelism optimization and use coarse-grained dataflows, limiting their flexibility.

### 3.3 Impedance Mismatch

Optimizing components in isolation prevents holistic optimizations, introduces brittleness and can cause overheads. For example, the FlashAttention [16] kernel, while fast, prevents fusion with other kernels due to being implemented in low-level CUDA or triton. On the other hand, different parallelism strategies and sub-engines are often employed for different stages of the computation. For example, RLFT systems often use fast serving engines such as vLLM [33], SGLang [78] or DeepSpeed-Inference [4] for the generation stage, while the inference and training stages often use large-scale training engines such as DeepSpeed [51] or Megatron-LM [57]. While this specialization of the generation engine speed-up generation, it still introduces overheads due to the ad-hoc composition of the systems and optimizations designed for serving unpredictable request load.

To demonstrate this, we experiment with TRL [65], the most popular[1] RLFT system found, applying the GRPO [54] algorithm to Qwen2.5-0.5B-Instruct [70], using the gsm8k dataset [14], on a single A100 GPU (40GB), with VLLM for generation. We observe that using vLLM impacts memory management and introduces overheads, as shown in Fig. 4a. First, it requires maintaining a separate copy of the model, consuming 2.3% of the total GPU memory. It also requires a memory reservation, which by default is 30% of the total GPU memory, preventing flexible memory sharing throughout execution. This prevents us from increasing the batch size beyond 1, despite the GPU being underutilized during generation, peaking only during training.




**(a) GPU Utilization**

**(b) Execution Profile.** Dashed boxes indicate overheads.

**Fig. 4: GRPO on TRL using vLLM.** The use of vLLM for generation causes significant overheads, limits flexible memory management and underutilizes GPUs due to sequential execution.

We analyze a profile of the execution (Fig. 4b), and find that even in this single GPU setting, ~40% of the total execution time is spent on different overheads caused by impedance mismatch with vLLM. At the highest level, 3.9% of time is spent recomputing logits which were just computed and discarded by vLLM during generation, while a further 0.34% is spent maintaining the vLLM model in sync with the training engine. Looking deeper, we find that prefill and decoding (which account for 75.6% of iteration time) spend 17.5% and 46.2% of time performing unnecessary scheduling work respectively. This is because vLLM is designed for serving an unpredictable request load, and thus applies techniques like *iteration-level scheduling* and *selective batching* [72] (depicted in Fig. 6a), which help dynamically balance service across requests, but are unnecessary for RLFT, where the load pattern is known ahead of time. Finally, within the useful decoding work, a further 2.6% of time is spent on page-table indirections caused by the vLLM's *paged attention* [33], which virtualizes the KV cache to reduce serving fragmentation.

[1] Measured in GitHub stars.

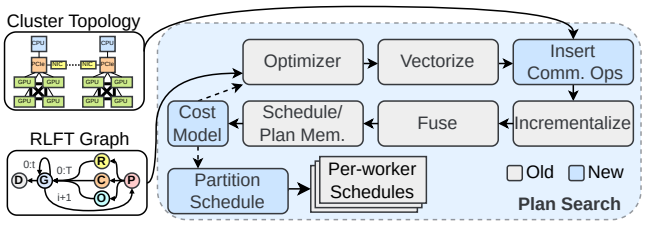



**Fig. 5: TimeRL Compilation Pipeline for RLFT.** By simply inserting communication operators, we can leverage the existing passes to achieve pipelined communication and parallelism.

> **Key Takeaway 4:** There are significant opportunities for accelerating RLFT by co-designing the generation and training engines.

## 4 A Vision for Whole-Graph Optimization

To support our vision of whole-graph RLFT optimization, we seek a graph representation that is simultaneously *expressive* enough to capture the variety of RLFT algorithms (§2.1) in a single graph, as well as, *fine-grained* enough to enable optimizations across components (§3.2). The representation should be amenable to *parallelization* and *distribution* (§2.2), to accommodate the large-scale nature of RLFT workloads. The key challenge is that most existing DL compilers are designed for static graphs, which are cannot effectively capture the dynamic dependencies of RLFT (§2.3).

We are thus building a prototype RLFT system on top of TimeRL [59], a DL compiler with support for the dynamic dependencies of RLFT through a polyhedral dependence graph (PDG) representation. In PDGs, similar to dataflow graphs, nodes are low-level linear algebra operations, while edges represent dependency. Unlike dataflow graphs however, each node executes many times and the graph can be cyclic, allowing nodes to depend on past and future iterations of themselves and other tensors, which is essential for RLFT. To capture dynamic access patterns within the representation, TimeRL adds *symbolic expressions*, e.g., $0 : t$ where $t$ is a *symbolic dimension*, to each edge, which are evaluated at runtime at each loop iteration, yielding a concrete dependence with dynamic shape (e.g., at $t = 10$, the dependence is on $0, 1, \ldots, 8, 9$).

We also inherit the same declarative recurrent tensor API, which allows RLFT algorithms and attention mechanisms to be expressed naturally as recurrence equations such as those in Eq. 1 and Eq. 2. Instead of hard-coding execution strategies, we declaratively express the computation as a set of recurrent relationships. This results in a single fine-grained graph for the entire RLFT algorithm, which can be optimized as a whole.

The advantage of this approach is that the existing compiler passes in TimeRL (Fig. 5) become directly applicable to RLFT since they operate on the same fine-grained representation. For example, the vectorization pass will automatically find parallelism across the batch and sequence dimensions, while the scheduling pass can find parallelism between operations such as generation and inference by analysing dependencies (❷). Since the whole computation is represented in a single graph, the optimizer pass can eliminate redundant work by finding duplicate nodes with the same inputs (❸). Finally, bespoke KV-cache management policies can be derived by analyzing the symbolic access patterns (❹). Additionally, through operator fusion, TimeRL can *progressively coarsen* the graph to reduce the search space for optimization, which we plan to leverage to reduce the complexity of our optimization problem. However, TimeRL does not currently support 3D parallelism and distribution, which is essential for RLFT.

### 4.1 RLFT Distribution with PDGs

To model 3D parallelism and distribution, we extend TimeRL with a few novel compiler passes, depicted in Fig. 5. We first extend TimeRL's primitive operation set with communication operators, including peer-to-peer (P2P) send and receive, as well as, collective communication (CC) operators (e.g., all-reduce), which allow us to model both inter- (PP) and intra-operator (DP, TP) parallelism [77] respectively. At compilation-time, we insert communication operators (both P2P and CC) into the PDG at key points to represent different distribution strategies.

The key challenge lies in choosing where to insert which communication operators. Our approach decomposes this problem into two steps. We first choose where to insert P2P operators by balancing the flops and memory usage of each subgraph created using integer linear programming. In this way, we define balanced PP stages. Then, for each subgraph created, we shard the computation over a 2D mesh of devices and promote reductions, gathers, and scatters to their CC equivalents (e.g., all-reduce, all-gather, all-to-all) if they require acceleration. We use an analytical cost model to guide these decisions, which we plan to refine in future work. Finally, since TimeRL creates a single whole-program schedule, we partition the resulting monolithic schedule into per-worker schedules by filtering out operations that belong to each stage. This is done iteratively, yielding the best found strategy once a time budget is met.

The advantage of this approach is that it composes with the existing TimeRL compiler passes, yielding advantages. For example, the incrementalization pass of TimeRL will tile both computation and communication operators, allowing us to overlap tile $k$ of computation with tile $k+1$ of communication transparently (❶). Beyond parallelism within a single iteration, a whole-graph representation enables finding parallelism across iterations [21]. For example, it is possible to overlap the generation at iteration $i + 1$ with critic training at iteration $i$, since the critic is not part of the critical path. Finally, this also allows us to support SP transparently, viewing it as simply intra-operator parallelism applied to the sequence dimension.

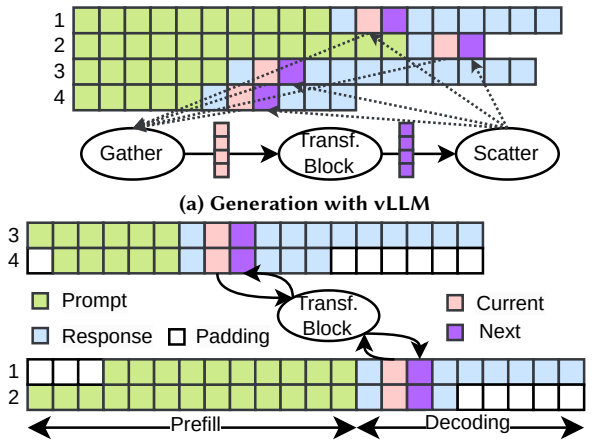



**Fig. 6: Generation Methods.** Aligned microbatches avoid the scheduling and data movement overhead of vLLM by aligning the decoding timesteps.

| Model | DeepSeek-R1 | GPT-4 Turbo | Llama 3 | Mean |
|---|---|---|---|---|
| **Correlation** | 0.22 | 0.24 | 0.44 | 0.30 |

**Tab. 2: Correlation of prompt and response length.** Estimated from 0.5M samples (~180K each model).

### 4.2 Efficient Generation without vLLM

While serving engines such as vLLM accelerate generation, we have shown that they also introduce serving-related overheads (§3.2), such as gathering and scattering inputs to create a batch each step (Fig. 6a). To avoid this overhead, we propose emphaligned microbatches (Fig. 6b).

The key idea is to divide request minibatches into microbatches, grouping together prompts of similar lengths and right-align them, so that generation always happens on aligned contiguous memory. This means that to perform a single generation step, we can simply slice the input, avoiding the overhead of gathering and scattering. Furthermore, since PP already requires microbatches to fill pipeline bubbles, we can combine the techniques transparently.

However, doing this necessitates padding on both the prompt and response sides. On the prompt side, since we group prompts by length, this padding is minimal. We also hypothesize that it will be small on the response side due to longer prompts leading to longer responses. To verify this hypothesis, we analyzed 0.5M responses generated by popular LLMs, and found a mean weighted correlation of up to 0.44 (mean 0.3) between prompt and response length (Tab. 2). Note that, in the case of DeepSeek-R1, this includes any reasoning traces generated by the model as part of the response. This is a meaningful correlation, which indicates that the padding amount will be significantly reduced over random grouping. Prior work has also shown that response length can be predicted from requests using a smaller language model [29], and while more expensive, such approaches can be used to further reduce padding.

## 5 Discussion and Open Questions

We now discuss open research directions we believe are important to the development of fine-grained DL compilation.

**Efficient Fused Kernels.** Manually implemented CUDA kernels such as FlashAttention [16] have brought significant speed-ups to LLMs through intelligent memory access patterns through tiling. Our end-to-end compilation approach should match or exceed these performance improvements. How to derive these optimizations automatically [9] for dynamic computations remains an open question, but polyhedral code-generation [7, 62] is a promising direction.

**Searching & Pruning.** The finer-grained the representation, the larger the search space for optimizations. Since we aim to simultaneously optimize for algebraic transformations, parallelism, scheduling and memory management, the search space is vast. How to effectively search and prune this space remains an open question, but prior work has leveraged problem decomposition [63] and hierarchical optimization [77],

**Modelling Advanced Distribution Strategies.** While 3D parallelism is powerful, our design does not yet take into account optimizations such as Zero [51], recomputation [66] or CP [41], which are needed for truly large-scale training. We wish to explore how to model these strategies with PDGs in future work.

**Sophisticated Algebraic Optimizations.** We believe there are opportunities for high-level algebraic optimizations that cannot be expressed in typical dataflow systems due to lack of semantic understanding of dynamic dependencies. Given to the connections between sequence and signal processing, we aim to explore the Fourier- and Z-transforms [18], diagonalization [23], chains of recurrences [6] and other signal processing techniques for PDG optimization.

## 6 Conclusion

In this paper, we have shown that existing RLFT systems suffer from diverse inefficiencies due to their coarse-grained representation. However, we also showed that existing fine-grained DL representations, such as JAX/XLA, are not well-suited for RLFT due to their lack of support for dynamic shapes. Thus, we proposed a design for a fine-grained distributed compiler based on TimeRL's [59] PDG representation, allowing for holistic optimization of computation, memory and communication. We believe fine-grained dynamic DL compilers are the key to unlocking the potential of RLFT, and we hope that our work will inspire future research in this direction.

## References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to Basics: Revisiting Reinforce Style Optimization for Learning from Human Feedback in LLMs. *arXiv preprint arXiv:2402.14740* (2024).

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. DeepSpeed-Inference: enabling efficient inference of transformer models at unprecedented scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[6] Olaf Bachmann, Paul S Wang, and Eugene V Zima. 1994. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*.

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *International Symposium on Code Generation and Optimization (CGO)*.

[8] 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).

[9] Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Workshop on Hot Topics in Operating Systems (HotOS)*.

[10] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150* (2020).

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[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] Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2023. Safe RLHF: Safe Reinforcement Learning from Human Feedback. *arXiv preprint arXiv:2310.12773* (2023).

[16] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[17] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence (Nat. Mach. Intell.)* (2023).

[18] Jacob Fein-Ashley. 2025. The FFT Strikes Back: An Efficient Alternative to Self-Attention. *arXiv preprint arXiv:2502.18394* (2025).

[19] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling Machine Learning Programs via High-Level Tracing. In *Conference on Machine Learning and Systems (MLSys)*.

[20] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. *arXiv preprint arXiv:2410.02089* (2024).

[21] Gábor E Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *International Conference on Data Engineering (ICDE)*.

[22] Amelia Glaese, Nat McAleese, Maja Trębacz, John Aslanides, Vlad Firoiu, Timo Ewalds, Maribeth Rauh, Laura Weidinger, Martin Chadwick, Phoebe Thacker, et al. 2022. Improving Alignment of Dialogue Agents via Targeted Human Judgements. *arXiv preprint arXiv:2209.14375* (2022).

[23] Albert Gu, Karan Goel, and Christopher Ré. 2021. Efficiently Modeling Long Sequences with Structured State Spaces. *arXiv preprint arXiv:2111.00396* (2021).

[24] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

[25] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685* (2021).

[26] Jian Hu, Xibin Wu, Weixun Wang, Dehao Zhang, Yu Cao, et al. 2024. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework. *arXiv preprint arXiv:2405.11143* (2024).

[27] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[28] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[29] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[30] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning (ICML)*.

[31] Diederik P Kingma. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[33] 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 *Symposium on Operating Systems Principles (SOSP)*.

[34] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. 2023. RLAIF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. *arXiv preprint arXiv:2309.00267* (2023).

[35] Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Ren Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, et al. 2024. RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. In *International Conference on Machine Learning (ICML)*.

[36] Kinman Lei, Yuyang Jin, Mingshu Zhai, Kezhao Huang, Haoxing Ye, and Jidong Zhai. 2024. PUZZLE: Efficiently Aligning Large Language Models through Light-Weight Context Switch. In *2024 USENIX Annual Technical Conference (ATC)*. 127–140.

[37] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *Transactions on Parallel and Distributed Systems (TPDS)* (2020).

[38] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[39] Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. 2021. Sequence Parallelism: Making 4D Parallelism Possible. *arXiv preprint arXiv:2105.13120* (2021).

[40] 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. In *Forty-first International Conference on Machine Learning*.

[41] Hao Liu and Pieter Abbeel. 2024. Blockwise parallel transformers for large context models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[42] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv preprint arXiv:2310.01889* (2023).

[43] Yaxi Lu, Haolun Li, Xin Cong, Zhong Zhang, Yesai Wu, Yankai Lin, Zhiyuan Liu, Fangming Liu, and Maosong Sun. 2025. Learning to Generate Structured Output with Schema Reinforcement Learning. *arXiv preprint arXiv:2502.18878* (2025).

[44] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. 2024. ReaLHF: Optimized RLHF Training for Large Language Models through Parameter Reallocation. *arXiv preprint arXiv:2406.14088* (2024).

[45] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed Precision Training. *arXiv preprint arXiv:1710.03740* (2017).

[46] Maxim Milakov and Natalia Gimelshein. 2018. Online Normalizer Calculation for Softmax. *arXiv preprint arXiv:1805.02867* (2018).

[47] Mistral AI Team. 2024. Un Ministral, des Ministraux. https://mistral.ai/news/ministraux/ Mistral AI News.

[48] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[49] Ray Perrault and Jack Clark. 2024. *Artificial Intelligence Index Report 2024.* Technical Report. Institute for Human-Centered Artificial Intelligence, Stanford University. https://aiindex.stanford.edu/

[50] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[51] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[52] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *International Symposium on Microarchitecture (MICRO)*.

[53] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[54] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300* (2024).

[55] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, et al. 2024. NeMo-Aligner: Scalable Toolkit for Efficient Model Alignment. *arXiv preprint arXiv:2405.01481* (2024).

[56] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv:2409.19256* (2024).

[57] 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).

[58] David Silver, Satinder Singh, Doina Precup, and Richard S Sutton. 2021. Reward is enough. *Journal of Artificial Intelligence (AIJ)* (2021).

[59] Pedro F. Silvestre and Peter Pietzuch. 2025. TimeRL: Efficient Deep Reinforcement Learning with Polyhedral Dependence Graphs. *arXiv preprint arXiv:2501.05408* (2025).

[60] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving Open Language Models at a Practical Size. *arXiv preprint arXiv:2408.00118* (2024).

[61] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi K1.5: Scaling Reinforcement Learning with LLMs. *arXiv preprint arXiv:2501.12599* (2025).

[62] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *International Workshop on Machine Learning and Programming Languages (MAPL)*.

[63] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[64] A Vaswani. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[65] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformer Reinforcement Learning. https://github.com/huggingface/trl.

[66] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[67] Yihan Wang, Si Si, Daliang Li, Michal Lukasik, Felix Yu, Cho-Jui Hsieh, Inderjit S Dhillon, and Sanjiv Kumar. 2022. Two-Stage LLM Fine-Tuning with Less Specialization and More Generalization. *arXiv preprint arXiv:2211.00635* (2022).

[68] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning (Mach. Learn.)* (1992).

[69] Youshao Xiao, Zhenglei Zhou, Fagui Mao, Weichang Wu, 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).

[70] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).

[71] 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).

[72] 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 *Symposium on Operating Systems Design and Implementation (OSDI)*.

[73] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[74] Qingru Zhang, Dhananjay Ram, Cole Hawkins, Sheng Zha, and Tuo Zhao. 2023. Efficient Long-Range Transformers: You Need to Attend More, but Not Necessarily at Every Layer. *arXiv preprint arXiv:2310.12442* (2023).

[75] Zheng Zhang, Chen Zheng, Da Tang, Ke Sun, Yukun Ma, Yingtong Bu, Xun Zhou, and Liang Zhao. 2023. Balancing Specialized and General Skills in LLMs: The Impact of Modern Tuning and Data Strategy. *arXiv preprint arXiv:2310.04945* (2023).

[76] Chen Zheng, Ke Sun, Hang Wu, Chenguang Xi, and Xun Zhou. 2024. Balancing Enhancement, Harmlessness, and General Capabilities: Enhancing Conversational LLMs with Direct RLHF. *arXiv preprint arXiv:2403.02513* (2024).

[77] 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 *Symposium on Operating Systems Design and Implementation (OSDI)*.

[78] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. *arXiv preprint arXiv:2312.07104* (2024).

[79] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. 2024. RLHFuse: Efficient RLHF Training for Large Language Models with Inter-and Intra-Stage Fusion. *arXiv preprint arXiv:2409.13221* (2024).