PipelineRL: Faster On-policy Reinforcement Learning for Long Sequence Generation

Anonymous authors
Paper under double-blind review

Abstract

Reinforcement Learning (RL) is increasingly utilized to enhance the reasoning capabilities of Large Language Models (LLMs). However, effectively scaling these RL methods presents significant challenges, primarily due to the difficulty in maintaining high AI accelerator utilization without generating stale, off-policy data that harms common RL algorithms. This paper introduces PipelineRL, an approach designed to achieve a superior trade-off between hardware efficiency and data on-policyness for LLM training. PipelineRL employs concurrent asynchronous data generation and model training, distinguished by the novel *in-flight weight updates*. This mechanism allows the LLM generation engine to receive updated model weights with minimal interruption during the generation of token sequences, thereby maximizing both the accelerator utilization and the freshness of training data. Experiments conducted on long-form reasoning tasks using 128 H100 GPUs demonstrate that PipelineRL achieves approximately $\sim 2x$ faster learning compared to conventional RL baselines while maintaining highly on-policy training data. A scalable and modular open-source implementation of PipelineRL is also released as a key contribution.

1 Introduction

Reinforcement Learning (RL) has recently become a popular tool to enhance the reasoning and agentic capabilities of Large Language Models (LLMs) (Guo et al., 2025; Wei et al., 2025). While RL expands the range of training signals one can use to enhance LLMs, this advanced learning paradigm comes with extra challenges, including being particularly hard to effectively scale to more compute. The scaling difficulty arises from the fact that AI accelerators (like GPUs and TPUs) deliver high throughput only when generating sequences at a large batch size. Hence, naively adding more accelerators to an on-policy RL setup brings increasingly diminishing learning speed improvements because the per-accelerator throughput decreases, while the overall generation latency reaches a plateau. The common workaround of generating training data for multiple optimizer steps results in a lag between the currently trained policy and the behavior policy that generates the training data. The lagging off-policy data is known to harm the commonly used effective RL algorithms (Noukhovitch et al., 2024), including, REINFORCE (Williams, 1992), PPO (Schulman et al., 2017) and GRPO (Shao et al., 2024; Guo et al., 2025), because these algorithms were designed to be trained with on-policy or near on-policy data, with the behavior and current policy being very close.

In this paper, we present the PipelineRL approach to RL for LLMs that achieves a better trade-off between hardware utilization and on-policy learning. Like prior work on efficient RL (Espeholt et al., 2018; 2019), PipelineRL features concurrent asynchronous data generation and training. PipelineRL adapts prior asychronous RL ideas to long-sequence generation with LLMs by introducing *in-flight weight updates*. As shown in Figure 1, during an in-flight weight update the LLM generation engine only briefly pauses to receive the model weights via a high-speed inter-accelerator network, and then proceeds to continue the generation of in-progress token sequences. In-flight updates eliminate the wasteful waits for the last sequence to finish, ensure high accelerator utilization at a constant generation batch size, and maximize the policy adherence of the recently generated tokens.

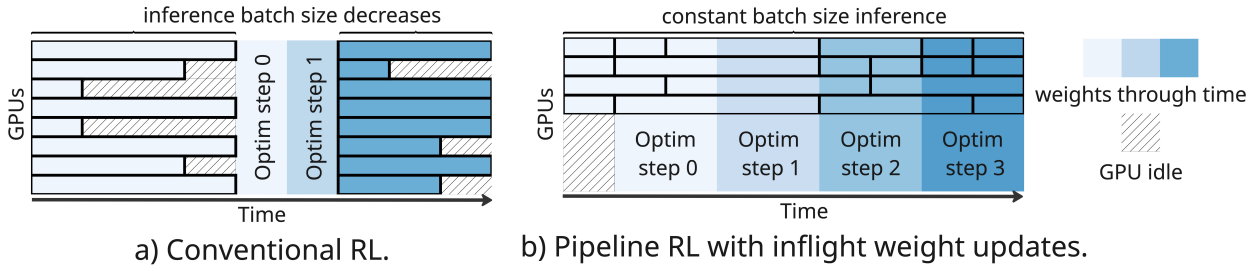


Figure 1: **a)** Conventional RL alternates between using all the GPUs for generation and then training. **b)** PipelineRL runs generation and training concurrently, always using the freshest model weights for generations thanks to the in-flight weight updates.

Our experiments on RL training for long-form reasoning show that on 4 DGX-H100 nodes, PipelineRL learns $\sim 2x$ faster than the comparable conventional RL baseline. We also observe that PipelineRL training data stays highly on-policy, and that models trained by PipelineRL perform comparably to similarly trained models from the literature. Lastly, a key contribution of this work is a scalable and modular PipelineRL implementation that we release as open-source software.

2 Background

2.1 Reinforcement Learning for Large Language Models

Reinforcement learning (RL) is commonly used to train Large Language Models (LLM) to respect human preferences (Ouyang et al., 2022) for the LLM's outputs or to perform long-form reasoning to solve problems (Guo et al., 2025). One can view LLM's weights as parameterizing a multi-step policy that assigns probabilities to the next token y_i given the prompt x and the previously generated tokens $y_{<i}$:

$$\pi(y|x) = \prod_{i=1}^{n} \pi(y_i|x, y_{<i}). \tag{1}$$

Recent works have shown that variations of basic policy gradient algorithms such as REINFORCE (Williams, 1992) are as effective for training LLMs as more sophisticated alternatives (Ahmadian et al., 2024; Roux et al., 2025). Given a set of prompts $x_1, \ldots, x_m$, REINFORCE maximizes the expected return $J(\pi)$ of the policy π by following an estimate $\tilde{\nabla} J(\pi)$ of the policy gradient $\nabla J(\pi)$:

$$J(\pi) = \frac{1}{m} \sum_{j=1}^{m} \left[\mathbb{E}_{y \sim \pi(\cdot|x_j)} R(x_j, y) \right] \tag{2}$$

$$\nabla J(\pi) = \frac{1}{m} \sum_{j=1}^{m} \left[\mathbb{E}_{y \sim \pi(\cdot|x_j)} \nabla \log \pi(y \mid x_j) R(x_j, y) \right] \tag{3}$$

$$\tilde{\nabla} J(\pi) = \frac{1}{m} \sum_{j=1}^{m} \sum_{t=1}^{T_j} \left(R(x_j, y_j) - v_\phi(x_j, y_{j,\leq t}) \right) \nabla \log \pi(y_{j,t} \mid x_j, y_{j,<t}), \tag{4}$$

where $R(x_j, y)$ is the reward and $v_\phi(x_j, y_{j,\leq t})$ is a value function learned by minimizing $\big(R(x_j, y_j) - v_\phi(x_j, y_{j,\leq t})\big)^2$.

In most practical RL setups, the *current policy* π will often differ from the *behavior policy* μ that generates y_k, due to the weights lagging, quantization or implementation difference between the inference and training softwares. This difference is usually handled by either a trust region constraint (Schulman et al., 2017) or using Importance Sampling (IS). In practice, the importance sampling weights are truncated to reduce the

Algorithm 1 Conventional RL

Require: Current policy π.
Require: Optimizer state opt_state.
Require: Number of optimizer steps per RL step G.
Require: Training batch size B.
  while True **do**
    // **generation** ▷ RL step starts
    $\mu \leftarrow \pi$ ▷ Initialize behavior policy μ
    sequences $\leftarrow$ generate BG sequences from μ
    batches $\leftarrow$ split sequences in G batches of size B
    // **training**
    lag $\leftarrow 0$ ▷ lag between μ and π
    for batch **in** batches **do**
      π, opt_state $\leftarrow$ optimizer_step(π, opt_state, batch)
      lag $\leftarrow$ lag + 1
    end for ▷ RL step ends
  end while

variance of the estimator (Munos et al., 2016; Espeholt et al., 2018):

$$\tilde{\nabla} J_{IS}(\pi) = \frac{1}{m} \sum_{j=1}^{m} \sum_{t=1}^{T_j} \min\left(c, \frac{\pi(y_k \mid x_j)}{\mu(y_k \mid x_j)}\right) \left(R(x_j, y_j) - v_\phi(x_j, y_{j,\leq t})\right) \nabla \log \pi(y_{j,t} \mid x_j, y_{j,<t}) \tag{5}$$

The Effective Sample Size (ESS) (Kong, 1992) is commonly used to quantify the quality of importance sampling estimators in RL (Schlegel et al., 2019; Fakoor et al., 2020). When using off-policy RL, ESS measures how many samples from the current policy π would yield equivalent performance to weighted samples from the behavior policy μ. The (normalized) ESS is defined as:

$$\text{ESS} = \left(\sum_{i=1}^{N} w_i\right)^2 \Big/ N \sum_{i=1}^{N} w_i^2 \tag{6}$$

where w_i are importance weights for a sample of size N. This metric effectively ranges between 0 and 1 when normalized, with values closer to 1 indicating more efficient sampling, e.g. the ESS of on-policy data is exactly 1. Small ESS will result in a high variance REINFORCE gradient estimate and might destabilize the learning process.

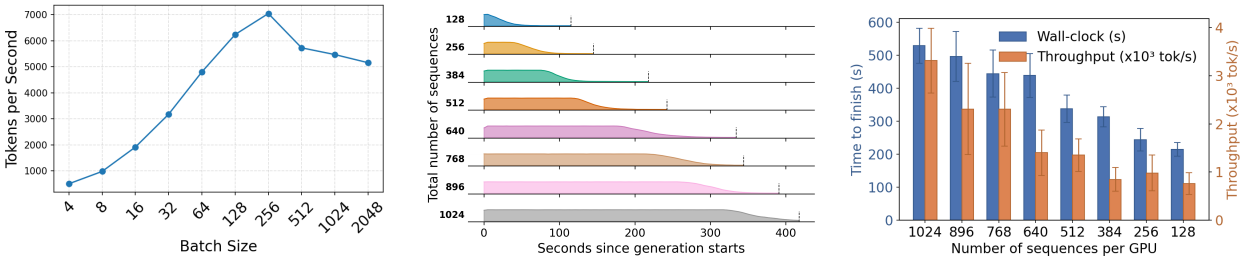
2.2 Conventional RL

Most RL implementations alternate between generating sequences and training the policy on the generated data. We refer to this approach as Conventional RL and describe it in detail in Algorithm 1. When training involves doing $G > 1$ optimizer steps, the current policy π gets ahead of the behavior policy μ that was used to generate the data. We adopt the term *lag* g to refer to the number of optimizer steps between μ and π.

2.3 Efficient Sequence Generation with LLMs

Transformer models generate sequences one token at a time, left-to-right. To make this process efficient, advanced generation (inference) engines such as vLLM and SGLang process a batch of sequences at a time, while carefully managing their past keys and values in a paged structure called KV cache (Kwon et al., 2023b). All modern generation engines support adding new generation requests *in-flight* to the ones in progress without stopping the generation process. Based on accelerator specifications, generation engines should achieve the maximum generation throughput at very large batch sizes of several thousand sequences.[1]

[1] `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`



(a) Throughput vs batch size. (b) Inference batch size vs time. (c) Time vs Throughput.

Figure 2: **Analysis of generation times and throughput.** We perform all measurements using a vLLM engine serving a Qwen 2.5 7B model on a H100 GPU. **(a)** Short prompt generation throughput increases up to batch size 256. **(b)** Generation batch size gradually decreases to suboptimal values as the engine finishes sequences **(c)** Generation time reaches a plateau and throughput decreases as the number of sequences per GPU goes down. We report the average of 5 runs and 95% CI.

Algorithm 2 PipelineRL: Actor and Trainer Processes

Require: Current policy weights π.
Require: Generation batch size H.
Require: Training sequence queue Q_{train}.

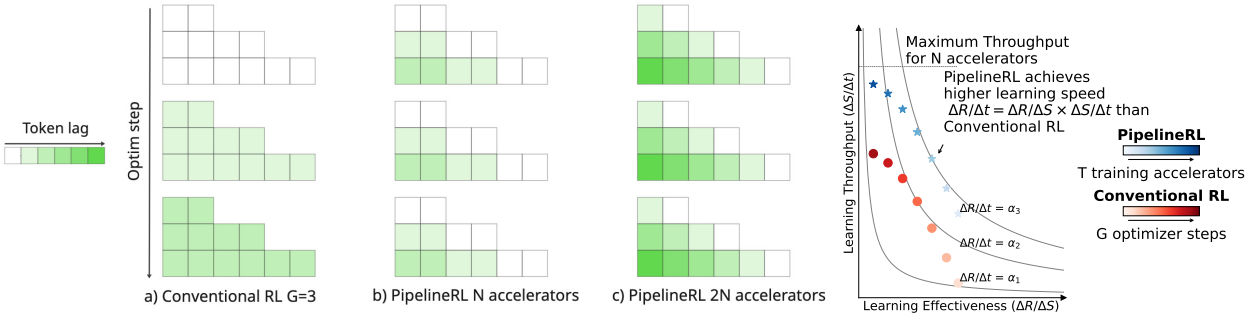
Actor Process:

function ACTOR
  sequences in progress $S_{prog} \leftarrow []$
  while True **do**
    S_{fin}, $S_{prog} \leftarrow$ pop finished sequences from S_{prog}
    $Q_{train}.put(S_{fin})$ ▷ Send finished seqs to the trainer
    if $len(S_{prog} < H)$ **then**
      add $H - len(S_{prog})$ prompts to S_{prog}
    end if
    if Trainer requests weight update **then** ▷ In-flight check for new weights
      $\mu \leftarrow$ receive_weight_update()
    end if
    $S_{prog} \leftarrow$ generate next tokens with μ
  end while
end function

Trainer Process:

function TRAINER(π, opt_state)
  batch $\leftarrow []$
  while True **do**
    request_actor_weight_update(π) ▷ In-flight weight update
    batch $\leftarrow$ get B sequences from Q_{train}
    π, opt_state $\leftarrow$ optimizer_step(π, opt_state, batch)
  end while
end function

In practice, at very large batch sizes, the per-sequence latency can become prohibitively high, KV cache may grow too large to fit in accelerator memory, or the request queue management overheads can dominate.



(a) Token lags as a function of optimizer steps.

(b) Pareto curves.

Figure 3: **(a)** For Conventional RL, the token lag increases with the number of optimizer steps. In PipelineRL with N accelerators, the token lag varies throughout the sequence, where earlier tokens have higher lag. The lag structure in each batch is the same. Doubling the PipelineRL accelerators, everything else constant, double the lag of early tokens. **(b)** Schematic illustration of PipelineRL's throughput-effectiveness trade-off as a function of training accelerators T and of Conventional RL as a function of lag G. PipelineRL achieves a higher $\frac{\Delta R}{\Delta S}\frac{\Delta S}{\Delta t}$ for the same number N of accelerators.

3 The learning speed ceiling of Conventional RL

Reinforcement learning for LLMs can be slow when the LLM is trained to generate long sequences of tokens, e.g., long-form reasoning to solve mathematical problems, because each generation can take up to several minutes. Here we explain why it is challenging to effectively scale up long sequence RL, i.e. to effectively use a larger number of accelerators N to make average reward $R(t)$ at time t grow faster. As a mathematical function, one can view $R(t)$ as a composition of the functions $R(S)$ and $S(t)$, where S is the number of samples the RL learner will have processed by time t. A faster RL learner will have a higher *learning speed* $\frac{\Delta R}{\Delta t}$ which we can express as the product of *learning effectiveness* and *learning throughput* as follows:

$$\underbrace{\frac{\Delta R}{\Delta t}}_{\text{speed}} = \underbrace{\frac{\Delta R}{\Delta S}}_{\text{effectiveness}} \times \underbrace{\frac{\Delta S}{\Delta t}}_{\text{throughput}} . \tag{7}$$

The Conventional RL algorithm from Algorithm 1 has the highest $\frac{\Delta R}{\Delta S}$ when it is fully on-policy, i.e., when one performs only one optimizer step per each RL step. Yet the throughput $\frac{\Delta S}{\Delta t}$ in the pure on-policy case can be low because the accelerators will be working on at most batch size B samples at a time. Increasing the number of accelerators N will yield diminishing returns in increasing $\frac{\Delta S}{\Delta t}$, because the throughput of each accelerator will decrease when the number of samples per accelerator $\frac{B}{N}$ goes below the optimal range (Figure 2c). For example, see Figure 2a for inference throughput for a 7B Qwen model on a single H100 GPU. One can see that the throughput increases almost linearly up to the generation batch size of 128. Hence, e.g. using $2N$ GPUs to generate 32 samples will not be much faster than using N GPUs to generate 64. Furthermore, as the LLM finishes the shorter generations, there will be fewer longer generations still in progress, see Figure 2b for an illustration. Hence, to make good use of the hardware, one should use each accelerator to generate many times more sequences than the optimal batch size.

Commonly, to increase the throughput, most practitioners perform multiple $G > 1$ optimizer steps per RL step, which entails generating BG rollouts at each generation stage. This way, one can often achieve a higher throughput $\frac{\Delta S}{\Delta t}$ by increasing N up to a point when $\frac{BG}{N}$ becomes too small. It is, however, known from the literature that going too off-policy by using a high value of G will eventually decrease the learning effectiveness $\frac{\Delta R}{\Delta S}$ (Noukhovitch et al., 2024). Clearly, at some points, the rollouts from the old policy become too stale and no longer useful as the source of learning signal for the current policy. Hence, given a fixed optimizer batch size B, one scales up Conventional RL by increasing G and N until the product $\frac{\Delta R}{\Delta S}\frac{\Delta S}{\Delta t}$ no longer improves, and the hard ceiling of $\frac{\Delta R}{\Delta t}$ for the given number of accelerators N is achieved.

4 Pushing the learning speed ceiling with PipelineRL

The Pipeline RL method differs from Conventional RL in two aspects: (1) running training and generation in parallel *asynchronously*, and (2) updating the generation weights after every optimizer step *in-flight*, i.e. without stopping the sequence generation. Algorithm 2 provides an abstracted formal description of PipelineRL in terms of two concurrent Actor and Trainer processes that communicate via a sample queue and a high-bandwidth weight transfer network.

The effectiveness-throughput trade-off for PipelineRL is the opposite of that of Conventional RL. Namely, adding more accelerators to a PipelineRL setup leads to a linear increase of $\frac{\Delta S}{\Delta t}$, but may eventually harm $\frac{\Delta R}{\Delta S}$. In Figure 3a, we illustrate how PipelineRL produces *mixed-policy sequences* in which earlier tokens are more off-policy than the recent ones. Doubling N will double the lag of the earliest tokens as well as the average lag in the PipelineRL batch. Notably, the off-policyness profile is different for PipelineRL and its conventional counterpart. Taking the average token lag as a proxy for off-policyness, in PipelineRL all batches are equally off-policy, whereas for Conventional RL later batches become progressively more off-policy. This difference makes it hard to analytically reason about the $\frac{\Delta R}{\Delta t}$ improvement that PipelineRL can bring over the baseline, because $\frac{\Delta R}{\Delta S}$ can only be estimated empirically by running RL experiments. In supplementary material, we present our simulation of how, for the same maximum lag g_{max} PipelineRL can learn 1.5x faster than Conventional RL. The empirical gains can be even larger, depending on how frequently one can make weight updates without hurting the learning effectiveness $\frac{\Delta R}{\Delta S}$.

Configuring PipelineRL vs Conventional RL For a fixed batch size B and a number of accelerators N, one can configure Conventional RL by choosing the number of optimizer steps G, trading off the learning effectiveness for the throughput. The PipelineRL configuration can likewise be mostly reduced to a single parameter, namely the number of training accelerators T out of N available ones. Setting a higher T will almost linearly decrease the time t_{train} that is needed for the trainer to process B sequences and perform an optimizer step. T effectively determines the optimal generation batch size H to be used at all $N - T$ accelerators. Using a lower H leads to a lower maximum generation latency t_{gen}, which consequently reduces the maximum lag $g_{max} = \lceil t_{gen}/t_{train} \rceil$. Hence, it makes sense to use the smallest H that suffices to produce enough training data. Consequently, the maximum lag g_{max} for PipelineRL grows with the number of training accelerators T, as higher T requires a higher H and leads to a lower t_{train} and a higher t_{gen}. On the contrary, the sample throughput of PipelineRL grows with T up to a point when $N - T$ accelerators cannot generate enough data for the over-powered trainer. We recommend avoiding extreme configurations with T too high (very high lag G) and T too low (bad hardware utilization, one can just as well scale down the compute). Figure 3b visualizes how different configurations of PipelineRL and Conventional RL achieve different learning effectiveness $\frac{\Delta R}{\Delta S}$ and throughput $\frac{\Delta S}{\Delta t}$, with PipelineRL setups reaching higher $\frac{\Delta R}{\Delta t} = \frac{\Delta S}{\Delta t}\frac{\Delta R}{\Delta S}$ isocurves.

Architecture and Implementation Details Our PipelineRL implementation concurrently runs many distributed vLLM generation engines and DeepSpeed training workers in a three stage pipeline that we describe in Figure 4. The middle Preprocessor stage that we omitted from Algorithm 2 for simplicity, computes reference model log-probabilities often used in Reinforcement Learning from Human Feedback (Ouyang et al., 2022). The PipelineRL architecture is highly modular — any generation software that supports the three HTTP API endpoints that PipelineRL requires can be easily integrated in the future. The three APIs are the popular `/v1/chat/completions` for generation, `/init_process_group` for creating the weight transfer process group, and `/request_weight_update` for initiating the in-flight weight update. Key optimizations in PipelineRL include online sequence packing for fast training and using ring buffers to minimize the lag when earlier pipeline stages run faster than the later ones, e.g. when the trainer makes a checkpoint.

5 Experiments

For the experimental validation of PipelineRL's high learning effectiveness $\frac{\Delta R}{\Delta S}$ and throughput $\frac{\Delta S}{\Delta t}$, we have chosen the challenging task of training a base (i.e. not instruction-tuned) model to perform long-form reasoning to solve mathematical problems. We find this task to be a great testbed for PipelineRL because the

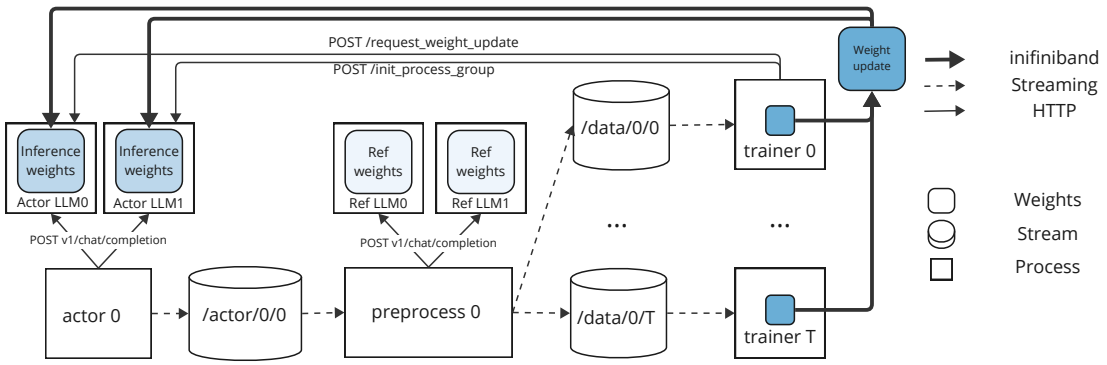
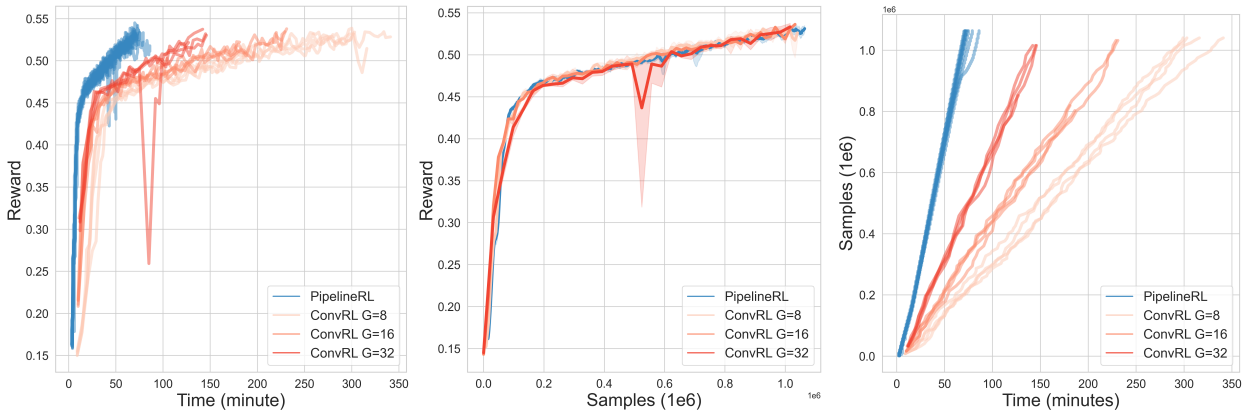


Figure 4: The three pipeline stages of PipelineRL implementation: actor, preprocessor and trainer. Earlier stages stream the data to the latter ones using Redis as the streaming broker.

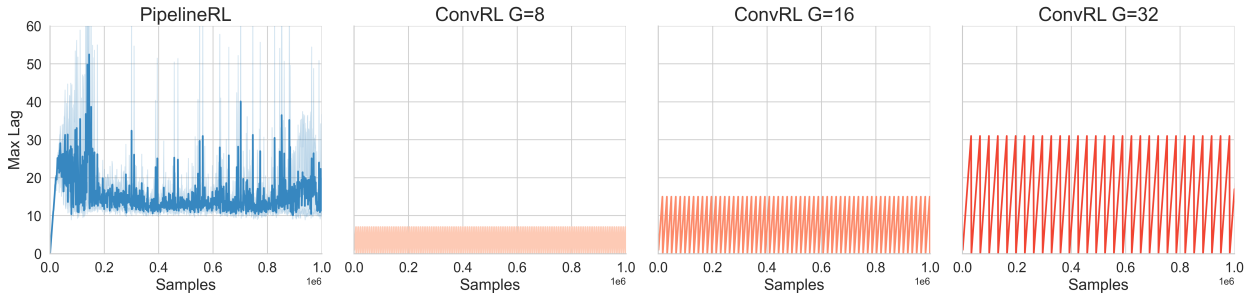


(a) Reward vs time. (b) Reward vs samples. (c) Samples vs time.

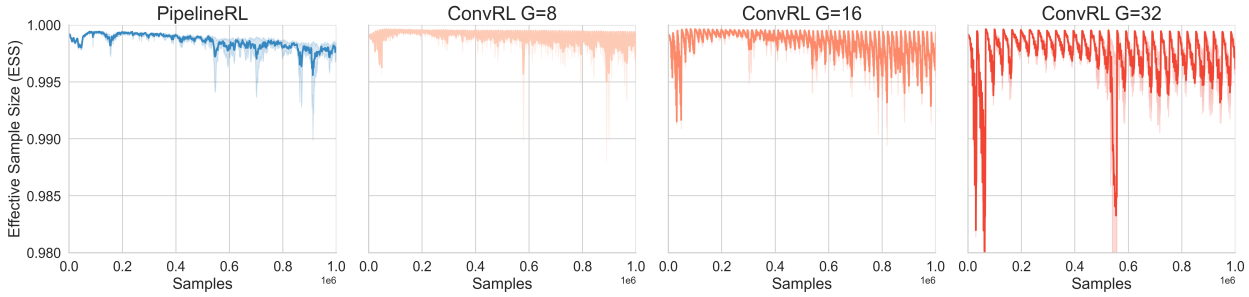
Figure 5: **(a)** PipelineRL attains the same average reward faster than the conventional RL baselines. **(b)** PipelineRL achieves the same sample efficiency as $G = 8$ and $G = 16$. **(c)** PipelineRL generates samples $\tilde{2}$x faster than the conventional RL $G = 32$ baseline.

policy undergoes rapid changes over the course of training. In particular, the length of generated sequences grows dramatically (Guo et al., 2025), making it essential to stay on-policy for effective learning.

Experimental setup. For each experiment, we train the Qwen 2.5 base model (Yang et al., 2024) with 7B parameters on 17K math problems from the OpenReasoner Zero dataset (Hu et al., 2025) for 1024 optimizer steps with the batch size $B = 1024$ and maximum sequence length of 32k. We use Adam optimizer (Kingma, 2014) with the learning rate 1e-6. We run the PipelineRL experiments on 16 DGX-H100 nodes, using 48 GPUs for generation at batch size $H = 64$ and 80 GPUs for training. We tweak PipelineRL to simulate Conventional RL by accumulating and shuffling a buffer of BG samples at the Preprocessor stage before the G optimizer steps of each RL step start. To estimate the Conventional RL throughput, we use 2 nodes for generation at batch size $H = 64$ and 2 nodes for training, and then add a correction for training on 8x fewer GPUs than what an efficient Conventional RL implementation with a quick generation-training transition could use. To estimate the inference throughput on 128 GPUS instead of 16 GPUs, we submit $\frac{128}{16}$ batches of $\frac{16\times1024\times G}{128}$ and take the maximum completion time. We give reward 1 to completed sequence with the correct answer and 0 otherwise. We also give a soft penalty to the model when it produces between 30k and 32k tokens. We train every model with importance weighted REINFORCE as described in Section 2



(a) Max lag.



(b) Effective sample size.

Figure 6: In Figure 6a, PipelineRL attains a higher max lag that every conventional RL method, but as observed in Figure 6b, the Effective Sample Size is similar to G=8. This indicates that while the max lag is quite high, PipelineRL stays mostly on-policy as measured by the ESS.

Table 1: Success rate of models trained with PipelineRL compared to results in the literature.

Method	**Math 500**	**AIME24**	**# samples ($\cdot 10^6$)**	**training data**
Qwen 2.5 base 7b	31.6	3.3	-	-
SimpleRL Zero (Zeng et al., 2025)	78.2	20.0	0.82	Math Level 3-5
OpenReasoner Zero (Hu et al., 2025)	$\sim$ 82.0	$\sim$ 20.0	8.2	OpenReasoner
PipelineRL (batch size 1024)	81	17.5	2.0	OpenReasoner
PipelineRL (batch size 4096)	84.6	19.8	6.2	OpenReasoner

and clamp the importance weights to 5. For our experiment we use vLLM (Kwon et al., 2023a) to generate trajectories and use DeepSpeed (Rasley et al., 2020) through accelerate to train the model.

PipelineRL learns faster due to higher throughput. We compare the learning speed of PipelineRL to that of Conventional RL with $G = 32$ optimizer steps, as that was the maximum G for which Conventional RL training was stable. PipelineRL achieves the same reward values approximately $\sim 2x$ faster than this baseline (Figure 5a) due to $\sim 2x$ faster sample throughput (Figure 5c). The main cause of the throughput increase is that GPU utilization for $G = 32$ experiment on 128 GPUs is relatively low for each GPU when it has to generate just $32 \times 1024/128 = 256$ sequences (see Figure 2b). Further increasing G to 64 results in divergence, see Figure 10.

PipelineRL learns effectively. To better measure learning effectiveness $\frac{\Delta R}{\Delta S}$ of PipelineRL, we also run Conventional RL experiments with $G = 8$, $G = 16$, and $G = 32$ optimizer steps. Notably, the $R(S)$ curves are indistinguishable for all compared methods up to a point where $G = 32$ is slower and unstable, likely

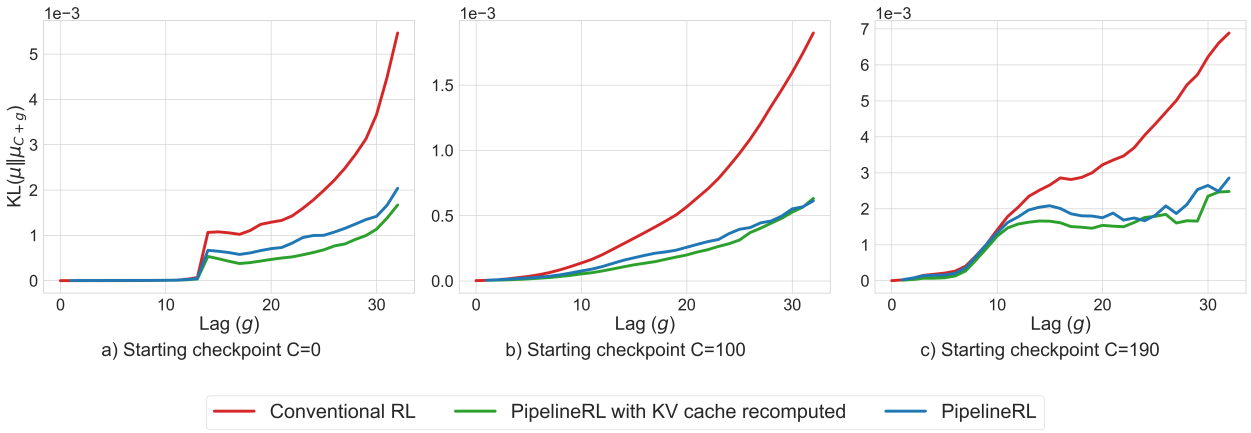


Figure 7: For three different starting checkpoint, PipelineRL with and without KV cache recomputation stay more on-policy than Conventional RL as measured by the KL divergence.

because of going too far off-policy. This result validates that PipelineRL's signature in-flight weight updates do no harm to the sequence generation process.

PipelineRL matches comparable results on reasoning tasks. Table 1 compares the test performance of PipelineRL to similar experiments that start training from the same Qwen 2.5 7B model. In this experiment we used batch size 4096 because we found it leads to a higher performance. On the math reasoning benchmarks MATH500 (Hendrycks et al., 2021) and AIME2024 (Li et al., 2024). PipelineRL matches or exceeds the success rate of Open Reasoner Zero and SimpleRL Zero.

PipelineRL stays more on-policy. To gain a better understanding of which training methods stay more on-policy, we plot the evolution of the max lag and the ESS on-policyness measure throughout the training. Figure 6a shows that PipelineRL obtains a higher max lag than the conventional RL baselines. Notably some tokens have a lag of more than 50k samples. However Figure 6b shows that, in terms of ESS, PipelineRL maintains a similar on-policyness as $G = 8$. We further observe that the ESS of $G = 16$ and in particular $G = 32$ drops throughout training.

5.1 Impact of in-flight weight updates on on-policyness

In this section, we compare the sampling distribution of in-flight weight updates to 1) conventional RL with different max lag and 2) in-flight weight update with KV cache recomputation. For this experiment, we save a set of consecutive checkpoints C_i, one after every optimizer step. To replicate the in-flight weight update, we start from a checkpoint and update the weights of the behavior policy every $\frac{L}{g_{\max}}$ tokens with the subsequent checkpoint, where L is the maximum sequence length and $g_{\max}$ is the maximum lag. Specifically, the PipelineRL behavior policy is defined as:

$$\mu := \mu_C(x_{1:t_1}) \dots \mu_{C+g}(x_{t_g:t_{g+1}} \mid \hat{x}_{1:t_1}, \dots \hat{x}_{t_{g-1}:t_g}) \tag{8}$$

where $t_1 = \frac{2L}{g_{\max}}$ and $t_g = t_{g-1} + \frac{L}{g_{\max}}$ tokens for lag $g > 1$ since the first weight update takes longer than the next updates due to the bubble at the beginning of training, see Figure 1 b). We also use $\hat{x}$ to stress that the KV cache for the previous tokens is stale - as it was computed under previous model weights. We then compute the KL between the mixed behavior policy $\mu_{C:C+g}$ and the on-policy behavior policy μ_{C+g}. We also report the KL with the mixed behavior policy with updated KV cache which we denote as *PipelineRL with KV cache recomputed*. To replicate conventional RL, we sample N sequences from the behavior policy $\mu := \mu_C$ and compute the KL with on-policy behavior policy μ_{C+g} for different lag k.

In this experiment, we fine-tune Qwen 2.5 base 7B on the OpenReasoner Zero (Hu et al., 2025) data for 222 optimizer steps. We consider three stages in training to measure KL-divergence: starting at checkpoint

0, 100, and 190. The maximum lag $g_{\max}$ is set to 32 and the maximum sequence length L is 2048. As presented in Figure 7, the distribution of mixed-policy sequences closely aligns with that of fully on-policy sequences across all three stages in the training. In contrast, off-policy sequences exhibit consistently higher divergences as lag increases. Also, using stale KV-cache for mixed policy sequences introduces only slightly higher divergence compared to recomputing the cache. This supports our design choice in Pipeline-RL to opt for the more efficient approach of retaining the KV cache.

6 Related work

Asynchronous and high-throughput RL has been extensively studied. IMPALA (Espeholt et al., 2018) decoupled acting from learning to maximize GPU utilization. Like PipelineRL, IMPALA used truncated importance weights to estimate the value function from off-policy samples. Furthermore, IMPALA kept the policy weights constant for the length of an episode. SeedRL (Espeholt et al., 2019) proposed to update the model's parameters during an episode, resulting in trajectories where different actions were sampled by different policies. OpenAI Five (OpenAI et al., 2019) was trained using asynchronous PPO to achieve superhuman performance on Dota 2. These previous works were focused on RL for video games. Closer to our work, (Noukhovitch et al., 2024) explores asynchronous RL for LLMs. In their approach, data generation for the next G optimizer steps is synchronized with training on the previous G optimizer steps, leading to higher off-policyness than Conventional RL, unlike PipelineRL. The same study shows that offline methods such as DPO (Rafailov et al., 2023) can better tolerate off-policyness.

There exist several other scalable open-source RL implementations. veRL (Sheng et al., 2024) implements Conventional RL efficiently by using a sophisticated hybrid generation-training engine that supports quick transitions between training and generation on the same GPUs. We believe veRL's throughput would be similar to our Conventional RL baseline. Without the hybrid engine, in OpenRLHF (Hu et al., 2024) training GPUs idle during generation and vice-versa. Concurrently, Magistral(Mistral-AI et al., 2025) also introduced in-flight weight updates.

7 Conclusion and Discussion

We have shown how in-flight weight updates help PipelineRL break the learning speed ceiling of the conventional two-stage RL approach. We believe that for long sequence generation, in particular, this speedup would be very difficult to attain with another asynchronous RL approach, as synchronous waits for generation to finish would hurt the throughput and/or learning effectiveness. The stale KV-cache risk that in-flight updates introduce can be mitigated by recomputing the KV cache after each update, which can be done fast at a high GPU utilization, but will still lower the throughput.

We believe PipelineRL may be particular useful for training LLMs to excel at agentic behaviors that involve multiple LLM generations interspersed with environment interactions. Another promising direction for future work is to study when the recent low lag tokens in PipelineRL are helpful, and on the contrary, where PipelineRL's constantly high lag of early tokens in long sequences hurts.

Limitations PipelineRL will only bring a limited throughput increase over Conventional RL if the LLM is asked to generate the exact same number of tokens for the same prompt. In this unlikely scenario, Conventional RL will be likewise capable of maintaining a constant generation batch size. The PipelineRL's stable average token lag and the low lag of recent tokens in each batch may, however, still affect the learning effectiveness. The PipelineRL throughput advantages will likewise decrease in setups with scarce or extensive compute resources. In the former case, each GPU will get enough generation tasks for the GPU utilization to be high. In the latter, the learning speed will be bounded not by the hardware utilization but by the best possible generation latency and by the environment feedback delay.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting REINFORCE style optimization for learning from

human feedback in LLMs. *arXiv preprint arXiv:2402.14740*, 2024.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and efficient deep-RL with accelerated central inference. *arXiv preprint arXiv:1910.06591*, 2019.

Rasool Fakoor, Pratik Chaudhari, and Alexander J Smola. P3O: Policy-on policy-off policy optimization. In *Uncertainty in artificial intelligence*, pp. 1017–1027. PMLR, 2020.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework, November 2024. URL `http://arxiv.org/abs/2405.11143`. arXiv:2405.11143 [cs].

Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. Open-Reasoner-Zero: An open source approach to scaling up reinforcement learning on the base model. *arXiv preprint arXiv:2503.24290*, 2025.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Augustine Kong. A note on importance sampling using standardized weights. *University of Chicago, Dept. of Statistics, Tech. Rep*, 348:14, 1992.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023a. URL `https://arxiv.org/abs/2309.06180`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023b. URL `http://arxiv.org/abs/2309.06180`. arXiv:2309.06180 [cs].

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. NuminaMath: The largest public dataset in AI4Maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13:9, 2024.

Mistral-AI, :, Abhinav Rastogi, Albert Q. Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmentlo, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, Léonard Blier, Lucile Saulnier, Matthieu Dinot, Maxime Darrin, Neha Gupta, Roman Soletskyi, Sagar Vaze, Teven Le Scao, Yihan Wang, Adam Yang, Alexander H. Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Andy Ehrenberg, Anmol Agarwal, Antoine Roux, Arthur Darcet, Arthur Mensch, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Chris Bamford, Christian Wallenwein, Christophe Renaudin, Clémence Lanfranchi, Darius Dabert, Devon Mizelle, Diego de las Casas, Elliot Chane-Sane, Emilien Fugier, Emma Bou Hanna, Gauthier Delerce, Gauthier Guinet, Georgii Novikov, Guillaume Martin, Himanshu Jaju, Jan Ludziejewski, Jean-Hadrien Chabran, Jean-Malo Delignon, Joachim Studnia, Jonas Amar, Josselin Somerville Roberts, Julien Denize, Karan Saxena, Kush Jain, Lingxiao Zhao, Louis Martin, Luyu Gao, Lélio Renard Lavaud, Marie Pellat, Mathilde Guillaumin, Mathis Felardos, Maximilian Augustin, Mickaël Seznec, Nikhil Raghuraman, Olivier Duchenne, Patricia Wang, Patrick von Platen, Patryk Saffer, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Romain Sauvestre, Rémi Delacourt, Sanchit Gandhi, Sandeep

Subramanian, Shashwat Dalal, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Thibault Schueller, Thibaut Lavril, Thomas Robert, Thomas Wang, Timothée Lacroix, Valeriia Nemychnikova, Victor Paltz, Virgile Richard, Wen-Ding Li, William Marshall, Xuanyu Zhang, and Yunhao Tang. Magistral, 2025. URL https://arxiv.org/abs/2506.10910.

Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous RLHF: Faster and more efficient off-policy RL for language models. *arXiv preprint arXiv:2410.18252*, 2024.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019. URL https://arxiv.org/abs/1912.06680.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*, pp. 3505–3506, 2020. URL https://doi.org/10.1145/3394486.3406703.

Nicolas Le Roux, Marc G Bellemare, Jonathan Lebensold, Arnaud Bergeron, Joshua Greaves, Alex Fréchette, Carolyne Pelletier, Eric Thibodeau-Laufer, Sándor Toth, and Sam Work. Tapered off-policy REINFORCE: Stable and efficient reinforcement learning for LLMs. *arXiv preprint arXiv:2503.14286*, 2025.

Matthew Schlegel, Wesley Chung, Daniel Graves, Jian Qian, and Martha White. Importance resampling for off-policy prediction. *Advances in Neural Information Processing Systems*, 32, 2019.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv:2409.19256*, 2024.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. SimpleRL-Zoo: Investigating and taming zero reinforcement learning for open base models in the wild. *arXiv preprint arXiv:2503.18892*, 2025.

A Analyical estimate of PipelineRL speedup for fixed max lag

In this additional section we estimate how much faster PipelineRL can be compared to Conventional RL for the same value of maximum token lag g_{max}. We will be using the following notation, mostly the same as in the main text:

- N is the number of accelerators
- $S = BG$ is the number of sequences that are processed in each Conventional RL step
- L is the maximum and $\overline{L}$ is the average sequence length for the current policy π
- $K = S\overline{L}$ is total number of tokens that Conventional RL processes in each optimizer step

We will additionally use $U(h)$ to refer to the accelerator's maximum flops utilization when running typical Transformer kernels at batch size h.

A.1 Units

To compare throughputs of different RL approaches it useful to adopt time and throughput units that don't depend on the particular GPU model and the LLM size. To this end we introduce a time unit called *flash*:

$$f = \frac{F_{gen}}{M} \tag{9}$$

where F_{gen} is the number of FLOPs required for one token forward pass for the chosen LLM, and M is the maximum theoretical FLOPs throughput for the given GPU. The meaning of a flash is the theoretically smallest amortized time that a token generation can take. Thus generating K tokens will take at least K flashes, though at a more typical generation utilization of ~ 0.1 rate it will take $10K$ flashes. For very long sequences F_{gen} can vary significantly due to attention FLOPs becoming a large part of total FLOPs, but for simplicity here we will abstract away from this detail.

Having introduced flash f as the unit, we will measure the system throughput in *tokens per flash*.

Let τ be the amortized training time per token. τ will be similar at scale for PipelineRL and Conventional RL, because both approaches can benefit from sequence packing.

A.2 Conventional RL throughput

We can express Conventional RL throughput as follows:

$$r_{conv} = \frac{K}{t^{gen}_{conv} + t^{train}_{conv}}, \tag{10}$$

where t^{gen}_{conv} and t^{train}_{conv} are times that generation and training take respectively. Let's look at these terms closer:

$$t^{gen}_{conv} = \sum_{l=1}^{L} \frac{h(l)/Nf}{U(h(l)/N)} \tag{11}$$

$$t^{train}_{conv} = \frac{K\tau}{N} \tag{12}$$

where $h(l)$ is the number of sequences still in progress after l steps of decoding, and $U(h)$ is the GPU utilization at batch size h. To understand Equation (11), recall that generating k tokens by definitions takes k flashes under perfect GPU utilization and $k/U(k)$ at the utilization $U(k)$.

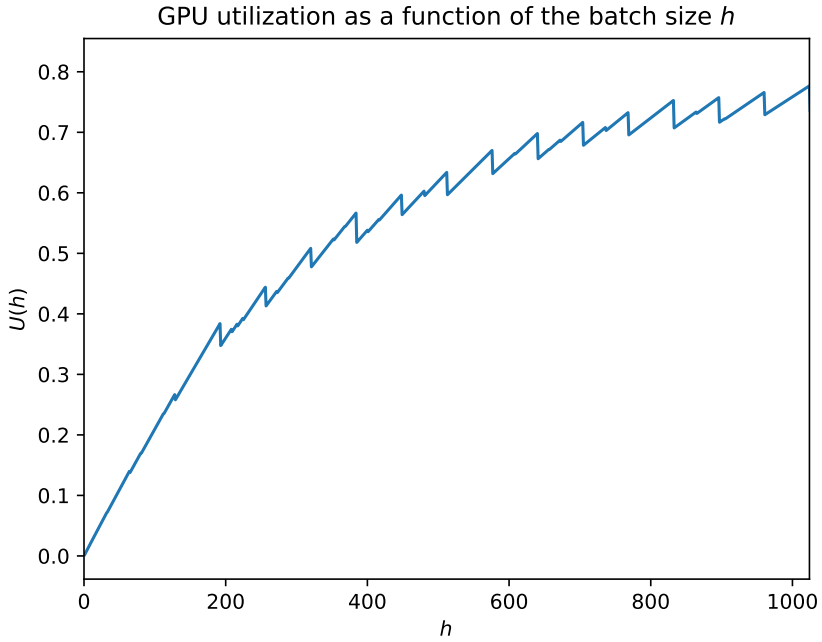


Figure 8: H100 utilization at batch size h as the ratio of maximum theoretical bf16 FLOPS throughput. We use $(4096, h) \cdot (h, 16384)$ matrix multiplications for the measurement. For every h we consider padding up to $h + 64$ to increase the speed, because empirically we observed large utilization bumps when h is divisible by a higher power of 2 (up to 128).

We can rewrite this in terms of tokens / flash throughputs:

$$r_{conv} = \frac{1}{\frac{1}{r^{gen}_{conv}} + \frac{1}{r^{train}_{conv}}} \tag{13}$$

$$r^{gen}_{conv} = \frac{K}{\sum_{l=1}^{L} \frac{h(l)/N}{U(h(l)/N)}} \tag{14}$$

$$r^{train}_{conv} = \frac{N}{\tau} \tag{15}$$

At low batch size per GPU at step l, $h_N(l) = h(l)/N$, the ratio $h_N(l)/U(h_N(l))$ will only decrease very slowly as a function of N, because for modern GPUs $\frac{x}{U(x)}$ is nearly constant for small x. This is the formal explanation for Conventional RL's decreasing efficiency as N grows.

The maximum token lag in the setup we described above is $S - 1$.

A.3 PipelineRL throughput

For PipelineRL the system throughput is determined by the slowest pipeline stage. Using the concepts introduced above, the throughput of PipelineRL can be estimated as follows:

$$r_{pipeline} = \min(r^{gen}_{pipeline}, r^{train}_{pipeline}) \tag{16}$$

$$r^{gen}_{pipeline} = U(H)I \tag{17}$$

$$r^{train}_{pipeline} = \frac{N - I}{\tau} \tag{18}$$

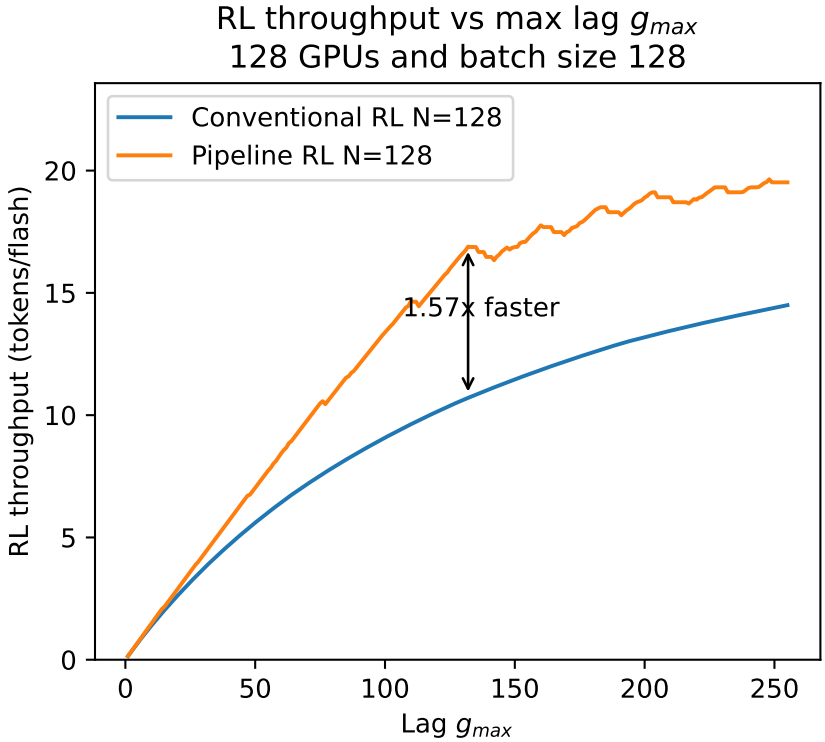


Figure 9: Pipeline RL and Conventional RL throughputs as the function of the maximum lag g_{max} for a setup with $N = 128$ GPUs and batch size $B = N$.

To understand the maximum lag of Pipeline RL consider the fact that the generation GPUs will produce HIL tokens during the time it takes to generate the longest possible sequence of length L. On average there will be $\frac{HIL}{L}$ sequences in these tokens. Thus, in the worst case when an optimizer step happened just before the longest sequence generation started, a long sequence will be used for training $g_{max} = \lceil \frac{HIL}{LB} \rceil$ optimizer steps later than its generation started.

To build a same-lag equivalent for a conventional RL system, one needs to maximize $r_{pipeline}(H, I)$ while keeping $\lceil \frac{HIL}{LB} \rceil \leq S - 1$. We could found this problem difficult to solve analytically, and performed a straight-forward search of all (H, I) configurations for our investigations below.

A.4 A PipelineRL speedup case study

To compute the exact throughput boost that PipelineRL brings it is necessary to make assumptions about the sequence length distribution and the hardware that is used for the experiments. For the case-study below, we assume uniform length distribution from 1 to the max length L and $H100$ as the GPU. We visualize the GPU utilization table $U(h)$ in Figure 8. The reader can see that $U(h)$ grows almost linearly up to $h \sim 200$, which makes it possible to compress the generation on fewer GPUs at a higher utilization. For a setup with $N = 128$ GPUs and training batch $B = 128$ we considered all possible (I, H) configurations of PipelineRL and plotted their throughput as a function of the lag g_{max}. Figure 9 shows that PipelineRL can be up to $1.57x$ faster for $g_{max} \sim 133$. This lag value can to be too high for many practical setups, but with a higher batch size of e.g. $B = 2048$ the same number of sequences to be generated by each GPU will correspond to a practical 16x lower lag $g_{max} \sim 8$.

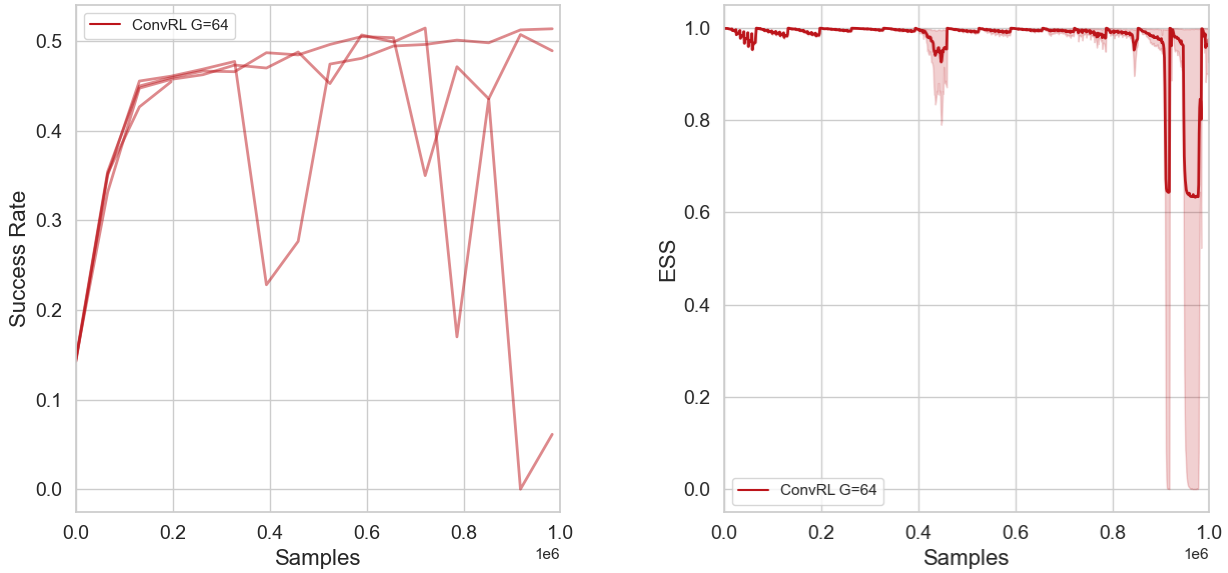
The mechanics of how PipelineRL achieved the improvement are as follows:

- $r^{gen}_{pipeline} = 16.9$, $r^{train}_{pipeline} = 17.08$, $\mathbf{r_{pipeline}} = \mathbf{16.9}$, $H = 192$, $I = 44$

- $r^{gen}_{conv} = 18.3$, $r^{train}_{conv} = 26.02$, $\mathbf{r_{conv}} = \mathbf{10.7}$

Clearly, the root cause of PipelineRL's speedup is that the 44 generation GPUs can produce 16.9 tokens per flash, that is more efficient than having 128 GPUs produce 18.3 tokens per flash in the Conventional RL case.

B Additional Results



(a) G=64 reward.

(b) G=64 Effective Sample Size.

Figure 10: G=64 diverges.