Accelerating Direct Preference Optimization with Prefix Sharing

Franklin Wang[*]
MIT CSAIL
fxwang@mit.edu

Sumanth Hegde[*]
Anyscale
sumanthrh@anyscale.com

Abstract

Offline paired preference optimization algorithms have become a popular approach for fine-tuning on preference data, outperforming traditional supervised fine-tuning in various tasks. However, traditional implementations often involve redundant computations, especially for tasks with long shared prompts. We introduce prefix sharing for preference tuning, a novel technique that processes chosen and rejected responses as one sequence with a shared prefix. To prevent cross-response contamination, we use a custom block-sparse attention mask. Our method achieves 1.1-1.5$\times$ improvement in training throughput on popular DPO datasets, without any effect on convergence. When combined with sequence packing, we observe consistent 1.3-1.6$\times$ speedups, benefiting even datasets with smaller sequence lengths. While we focus on Direct Preference Optimization (DPO), our approach is applicable to other paired preference tuning methods. By enhancing computational efficiency, our work contributes to making preference-based fine-tuning more accessible for a wider range of applications and model sizes. We open-source our code at https://github.com/frankxwang/dpo-prefix-sharing.

1 Introduction

Offline paired preference optimization algorithms such as DPO [23], ORPO [9], and SimPO [18] have emerged as popular approaches for fine-tuning large language models (LLMs) on preference data, attaining higher performance than traditional supervised fine-tuning (SFT) in areas such as instruction-following [11], multi-step reasoning [21, 17, 15], agentic planning [22], and coding [36, 32]. These algorithms leverage paired preference data, where each training sample comprises a shared prompt and two responses, with a label indicating which response is preferred. The LLM is then optimized to increase the likelihood of the chosen (preferred) response while minimizing the likelihood of the rejected (not preferred) one. This enables the LLM to learn from contrasts between the chosen and rejected samples rather than only mimicking the chosen samples through SFT.

Traditional paired preference optimization implementations batch the chosen and rejected sequences together during training. However, this results in redundant computations since each shared prompt is processed by the model twice—once for each response. This is particularly inefficient for tasks such as summarization [28] and mathematics [20, 35] which have disproportionately long prompts compared to responses, along with multi-turn conversational datasets where the prompt consists of multiple previous interactions between the user and assistant [5, 2].

To eliminate this redundancy, we propose prefix sharing for preference tuning, where chosen and rejected responses are processed as one sequence with a shared prefix, as illustrated in Figure 1. We employ a custom attention mask to delineate the two responses during the model's forward pass. Our approach accelerates training by significantly reducing total training tokens, performing best when

[*]Equal contribution.

38th Workshop on Fine-Tuning in Machine Learning (NeurIPS 2024).

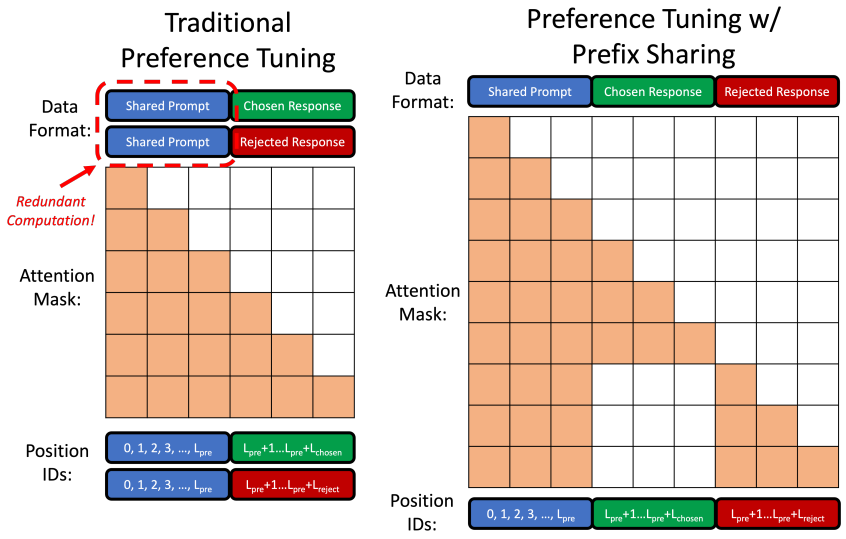

Figure 1: **Method overview.** Prefix sharing removes redundant computation of the shared prompt prefix by combining the responses into a single sequence and modifying the attention mask to prevent cross-response contamination.

training time scales roughly linearly with token count. For popular preference tuning datasets, we find that prefix sharing can enable roughly 1.1-1.5$\times$ improvement to training throughput, without any effect on convergence and with a small reduction in memory consumption. We observe speedups of about 1.5$\times$ for datasets that have long prefixes and high prefix-to-completion length ratios.

To further improve training throughput we implement sequence packing [14] for prefix-shared inputs. Compared to sequence packing for the baseline paired input format, we find that with prefix sharing, packing can add additional efficiency gains, especially for datasets with smaller overall sequence lengths, enabling our method to provide more consistent 1.3-1.6$\times$ speedups.

In our experiments, we focus mainly on Direct Preference Optimization (DPO) [23] as it is currently the most popular offline paired preference optimization algorithm. However, our insights into prefix sharing for DPO can be easily transferred to other similar paired preference tuning methods.

2 Background

2.1 Preference Optimization

Preference optimization methods enable LLMs to be optimized for arbitrary goals defined by binary preference data, where pairs of responses to a shared prompt are ranked. Traditional on-policy RL-based approaches use preference data to train a reward model that assigns scalar scores to LLM responses [19, 28]. The LLM (π_θ) is then trained using an online RL algorithm such as PPO [25] to maximize the reward model while also being regularized to prevent significant deviation from the original reference LLM (π_{ref}).

Direct Preference Optimization (DPO) simplifies this multi-stage process by reparameterizing the reward model in terms of π_θ and π_{ref} directly. For a single paired preference data sample, where x is the prompt and y_{chosen}, y_{rejected} are the chosen and rejected completions, the DPO loss function is:

$$\mathcal{L}(x, y_{\text{chosen}}, y_{\text{rejected}}; \pi_\theta, \pi_{\text{ref}}) = -\log \sigma \left(\beta \frac{\log \pi_\theta(y_{\text{chosen}}|x)}{\log \pi_{\text{ref}}(y_{\text{chosen}}|x)} - \beta \frac{\log \pi_\theta(y_{\text{rejected}}|x)}{\log \pi_{\text{ref}}(y_{\text{rejected}}|x)} \right) \quad (1)$$

To compute this loss function, we need to calculate the total log probabilities of each of the completions y_{chosen} and y_{rejected}. Typical implementations of DPO accomplish this by formatting each training sample as two full input sequences which are batched together: the shared prompt followed by the chosen response, and the shared prompt followed by the rejected response [30, 10, 23, 7].

The log probabilities are then calculated and extracted for each response independently, and so the computation for the shared prompt is repeated twice for every training sample. Since the shared prompt's activations are the exact same for both sequences due to causal masking, it is unnecessary to compute the shared prompt twice. In this work, we leverage prefix sharing to remove this redundancy while ensuring that the log probabilities are computed identically.

2.2 Prefix Sharing for Inference

While there has not been much exploration surrounding prefix sharing for training, there have been some works that explore prefix sharing for LLM decoding at inference time [13, 1, 34], where a prompt prefix is shared across multiple sequences that are being decoded. Since decoding is often memory-bound [27], these works focus on reducing memory reads of the precomputed prefix KV-cache using custom attention kernel implementations.

On the other hand, LLM training at sufficient batch sizes is mostly compute-bound [33], and thus most of the gains from prefix sharing for DPO training are from directly reducing the number of total training tokens, rather than speedups from improving self-attention. Because of this, we use a simpler approach for the self-attention operation, employing a custom attention mask rather than designing a custom kernel, making our method very flexible and easily extensible to optimizations such as sequence packing. Our approach is most similar to the Medusa speculative decoding method [3], where a custom attention mask is used to simultaneously decode multiple sequences in a tree structure.

3 Methodology

3.1 Prefix Sharing

Instead of separating the paired responses into two different sequences, we place the prompt, chosen response, and rejected response into a single sequence, as shown in Figure 1. In the attention mask, we mask out the region where the rejected response attends to the chosen response, ensuring that the log probabilities of both responses are computed independently of each other. Therefore, this attention computation is identical to the normal paired data format since the chosen and rejected responses can independently attend to the shared prompt. We also set the position IDs so that the sequences are the same as the normal paired format.

We use PyTorch's FlexAttention [8] to implement the attention layers. Using FlexAttention's `mask_mod`, we can leverage block sparsity to skip fully masked-out blocks of our custom attention mask. This enables us to skip computation of the masked region where the rejected response attends to the chosen response. At the beginning of every training step (i.e for every mini-batch), we compute the sparse block mask for our custom attention scheme and pass this block mask into each FlexAttention layer.

FlexAttention's `mask_mod` constructs the custom attention mask using a user-defined function that outputs a boolean for a given batch sample index, query index, and key-value index. We implement this function by keeping track of the start indices of the chosen and rejected completions for each sample (see Algorithm 1).

Algorithm 1 Prefix Sharing Mask

Given: sequence length L, batch size B
Input: batch sample index b; query index q; key-value index kv; per-token chosen and rejected completion start indices $I_{\text{chosen}}, I_{\text{rejected}} \in \{0 \dots L-1\}^{B \times L}$

▷ *apply standard causal mask, masked regions are set to False*
causal_mask $= (kv \leq q)$

▷ *prevent queries in rejected from attending to keys in chosen*
chosen_ind $= I_{\text{chosen}}[b]$
rejected_ind $= I_{\text{rejected}}[b]$
dpo_mask $=!((\text{rejected\_ind} \leq q)$ && $(\text{chosen\_ind} \leq kv < \text{rejected\_ind}))$

return causal_mask && **dpo_mask**

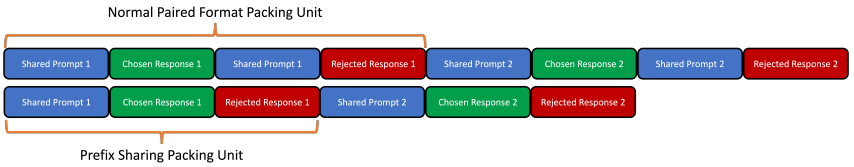

Figure 2: Sequence packing with and without prefix-sharing for paired preference inputs, illustrated for two training samples. Without prefix-sharing, a sequence packing implementation will have to treat the chosen and rejected responses, each prefixed by the common prompt, as a single unit and then pack these units together. With prefix sharing, the unit for sequence packing is now the shared prompt with the chosen and rejected response.

3.2 Sequence Packing

Sequence packing is a popular technique for improving training efficiency in NLP [16, 14]. Typically, an approximate solution to the bin-packing algorithm is used to sample sequences to pack in a mini-batch. While sequence packing for the regular causal language modeling task is well studied, packing for paired-preference data is more involved. Since we have two sequences (chosen and rejected) per training sample, each with varying length, one cannot pack these independently. Thus, a simple strategy could be to just combine the chosen and rejected sequences (i.e. the shared prompt along with the response) for a sample and to treat this as a single unit for packing. As shown in Algorithm 2, one would need to also keep track of the boundaries of the different responses in the batch to prevent cross-contamination. The chosen and rejected responses for each sample are also recovered at loss calculation with these boundaries.

Sequence packing for our prefix sharing method is similarly straightforward (see Figure 2). Without a redundant shared prefix, the maximum sequence length for each packing unit is much shorter. In Section 4.3.2, we show that packing with prefix sharing can lead to much better efficiency gains than prefix sharing alone. We further demonstrate that prefix sharing with packing has no negative impact on model quality in Appendix A.

To implement sequence packing for prefix sharing without cross-sequence contamination, we again leverage FlexAttention and further customize the block mask. As shown in Algorithm 3, we use a `mask_mod` function that is similar to non-packing prefix sharing, except we also keep track of unique document IDs for each sample and store the chosen/rejected start indices as a dense $B \times L$ array.

Algorithm 2 Attention Mask for Packing w/o Prefix Sharing

Given: sequence length L, batch size B
Input: batch sample index b; query index q; key-value index kv; per-token response IDs $R \in \mathbb{Z}^{B \times L}$

▷ *apply standard causal mask, masked regions are set to False*
causal_mask $= (kv \leq q)$

▷ *apply masking to prevent tokens from separate samples and different responses from attending to each other*
response_mask $= (R[b][q] == R[b][kv])$

return causal_mask && **response_mask**

4 Experiments

4.1 Baselines

We compare our prefix sharing approach against two baselines: FlexAttention with the normal paired data format and FlashAttention-3 [26] with the normal paired data format. Despite FlexAttention's advantages, its base performance is worse compared to less general attention kernel implementations such as FlashAttention-3, which does not easily support arbitrary attention masks (and thus can not be used with prefix sharing) but is more optimized than FlexAttention. Since the normal paired format is compatible with FlashAttention-3 (as it only requires causal masks), while prefix sharing is

Algorithm 3 Prefix Sharing Mask with Packing

Given: sequence length L, batch size B
Input: batch sample index b; query index q; key-value index kv; per-token chosen and rejected completion start indices $I_{\text{chosen}}, I_{\text{rejected}} \in \{0 \dots L-1\}^{B \times L}$; document IDs $D \in \mathbb{Z}^{B \times L}$

▷ *apply standard causal mask, masked regions are set to False*
causal_mask $= (kv \leq q)$

▷ *apply document masking to prevent tokens in separate samples from attending to each other*
document_mask $= (D[b][q] == D[b][kv])$

▷ *prevent queries in rejected from attending to keys in chosen*
chosen_ind $= I_{\text{chosen}}[b][q]$
rejected_ind $= I_{\text{rejected}}[b][q]$
dpo_mask $=!((\text{rejected\_ind} \leq q)$ && $(\text{chosen\_ind} \leq kv < \text{rejected\_ind}))$

return causal_mask && **document_mask** && **dpo_mask**

only compatible with FlexAttention, we include FlashAttention-3 with the normal paired format as a baseline in our benchmarking experiments.

4.2 Individual Layer Micro-benchmarking

We perform microbenchmarks in three settings: the MLP layer, the self-attention computation, and the full Attention layer (including linear projections) of Mistral-7B [12]. For each experiment we report the sum of the forward and backward pass time, sweeping across different prefix and completion lengths. All of the following microbenchmarking experiments were conducted on a single NVIDIA H100 GPU.

4.2.1 MLP Micro-benchmarking

For the MLP layer experiments, we compare the prefix sharing data format to the normal paired format. As shown in Figure 3, we find that for longer prefix lengths, we get nearly ideal linear speedups[1] from the reduction in total tokens due to the operation being mostly compute-bound. For the shorter prefix lengths of 128 and 256, the improvements are still valuable but are smaller due to constant time overheads (such as memory reads of the weights) taking up a larger proportion of the total time.

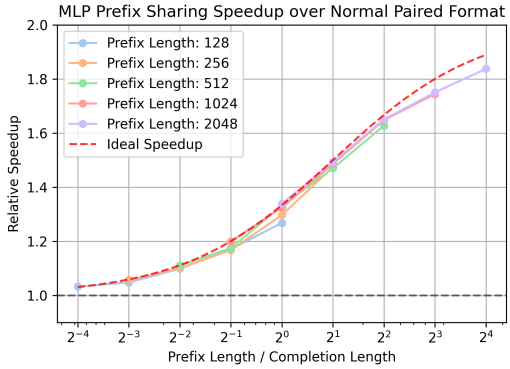

Figure 3: Microbenchmarking results of the MLP layer for Mistral 7B. Relative speedups of prefix sharing over normal paired data are shown in comparison to the ideal speedup (assuming linear runtime). We see that the MLP layer scales very closely to the ideal speedup and that increasing the prefix length helps push the speedup closer to the ideal for a given prefix to completion ratio.

[1]We compute the ideal linear speedup for a given prefix length p and completion length c as $\frac{2(p+c)}{p+2c}$.

4.2.2 Attention Micro-benchmarking

For the attention experiments, we compare FlexAttention with the prefix sharing format to both FlexAttention and FlashAttention-3 with the normal paired format.

In Figure 4, we can see that FlexAttention with prefix sharing shows clear trends towards higher performance boosts at greater prefix to completion length ratios, attaining close to ideal speedups[2] over FlexAttention with normal paired data. However, when compared to FlashAttention-3, FlexAttention tends to be much slower, only matching or exceeding performance at ratios that are $\geq$8.

Nevertheless, in Figure 5 we illustrate that when benchmarking the full attention layer (QKV projection + self-attention), the discrepancy between FlexAttention and FlashAttention-3 is diluted significantly since the self-attention operation is a small proportion of the overall speed. For the full attention layer, FlexAttention with prefix sharing is faster than FlashAttention-3 for most sequences with ratios $\geq$1, thus making prefix sharing valuable for many datasets despite the worse base performance of FlexAttention.

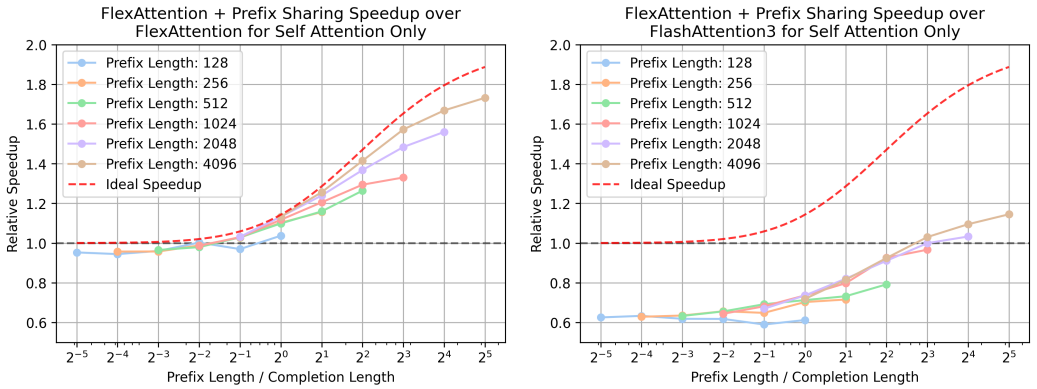

Figure 4: Microbenchmarking results of the self-attention operation only for Mistral 7B. Relative speedups of FlexAttention with prefix sharing over FlashAttention-3 and FlexAttention are shown, along with the ideal speedup (assuming perfect quadratic scaling). We see that for high prefix lengths, FlexAttention with prefix sharing attains nearly ideal speedups over FlexAttention without prefix sharing, but overall it is still slower or similar in speed to FlashAttention-3. Nevertheless, we find in practice that self-attention contributes little to overall training time and thus has minimal impacts.

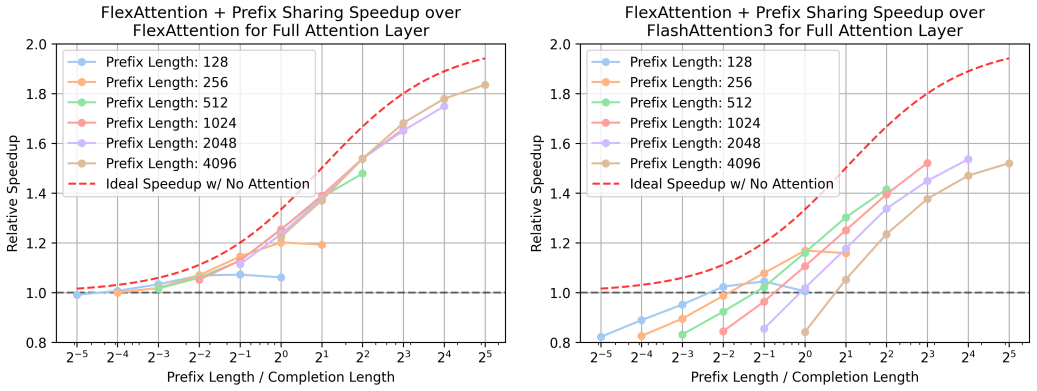

Figure 5: Microbenchmarking results of the full self-attention layer (QKV projection + self-attention) for Mistral 7B. Relative speedups of FlexAttention with prefix sharing over FlashAttention-3 and FlexAttention are shown, along with the ideal speedup (assuming linear runtime). We see that although FlexAttention is slower than FlashAttention-3 for lower ratios between the prefix and completion length, as the ratio grows, FlexAttention with prefix sharing become faster.

[2]We compute the ideal quadratic attention speedup for prefix length p and completion length c as $\frac{2(p+c)^2}{2(p+c)^2-p^2}$.

Overall, these results suggest that our approach is most promising for longer sequences where performance is mostly compute-bound, and for higher prefix length to response ratios where the total token reduction is greatest. Although FlashAttention-3 is inherently faster than FlexAttention, the effects are minimal as self-attention is a small proportion of the overall time, and in the following full model training experiments in Section 4.3 we further validate this.

4.3 Full Model Training Benchmarking

To highlight the usefulness of our approach in real-world settings, we evaluate our approach on popular preference tuning datasets. We include five datasets with high prefix to completion ratios spanning from single-turn chat (Tulu-Helpsteer [7, 31]) and multi-turn chat (Capybara [5], HH-RLHF [2]) to specific domains like math (MetaMath-DPO [20, 35]) and summarization (TLDR [28]). We also include Ultrafeedback [4], a single-turn chat dataset with low prefix to completion ratios, to showcase how our approach still leads to small speedups for low ratios.

We base our DPO training implementation on the open-source TRL library [30] and use DeepSpeed ZeRO-3 [24] to accelerate training. All our experiments are conducted on 1 8xH100 instance with CUDA 12.3 and PyTorch 2.5.0.

4.3.1 Prefix Sharing Benchmarking

We use a per-device batch size of 4 and fix the maximum prompt and total sequence length based on dataset statistics to minimize thresholding. For the FlexAttention experiments, we use our own attention layer implementation that leverages FlexAttention and constructs a block-sparse attention mask at the beginning of each training step.

Dataset	Median Overall Len	Prefix / Completion	FA3	Flex Attn	Flex + Prefix Sharing
Capybara [5]	1160	1.59	8.38	7.75	11.90 (1.42×, 1.54×)
HH-RLHF [2]	186	2.15	33.71	30.25	36.11 (1.07×, 1.19×)
MetaMath-DPO [20, 35]	872	3.91	13.86	13.02	19.13 (1.38×, 1.47×)
TLDR [28]	416	11.14	31.43	29.53	35.36 (1.12×, 1.20×)
Tulu-Helpsteer [7, 31]	775	6.34	14.83	13.93	21.75 (1.47×, 1.56×)
Ultrafeedback [4]	409	0.42	18.40	17.31	20.46 (1.11×, 1.18×)

Table 1: Comparison of training samples per second for different attention implementations. Relative speedups over FlashAttention-3 and FlexAttention, respectively, are shown for the FlexAttention with Prefix Sharing column. FlexAttention with prefix sharing consistently outperforms the baselines, with speedups ranging from 1.1-1.5×, with FlexAttention alone being slower than FA3. For the Prefix / Completion column, we report the median ratio. For high median overall lengths (> 500), the gains from prefix sharing are $> 35\%$, with better gains for high prefix to completion ratios.

Training throughput in samples per second[3] is shown in Table 1. We find that FlexAttention with prefix sharing consistently speeds up training time, even though FlexAttention alone is generally slower than FlashAttention-3. We further observe a small reduction in memory consumption with the paired data format, as is expected given the overall reduction in input (and thus activation) size. Our approach is especially effective for sequences that have high ratios and long overall lengths, enabling us to reach gains of up to 1.4-$1.5\times$ over FlashAttention-3. Even for Ultrafeedback, which has a very low ratio, we see some speedups, showing that our method is helpful even for low prefix to completion ratio settings.

However, we can see that for datasets like HH-RLHF and TLDR which have high ratios but low overall lengths, our speedups are smaller. To address this, we show in the following Section 4.3.2 that sequence packing helps remedy this, pushing training closer towards linear compute bound scaling by increasing the effective overall sequence length.

[3]We report samples per second instead of tokens per second, since "tokens processed" can be confusing across methods

Dataset Name	Median Overall Len	Prefix / Completion	FA3 + Packing	Flex Attn + Packing	Flex Attn + Prefix Sharing + Packing
Capybara [5]	1160	1.59	17.89	17.63	23.89 (1.34×, 1.36×)
HH-RLHF [2]	186	2.15	109.77	104.99	155.04 (1.41×, 1.48×)
MetaMath-DPO [20, 35]	872	3.91	24.21	23.83	38.07 (1.57×, 1.60×)
TLDR [28]	416	11.14	44.11	43.22	59.76 (1.35×, 1.38×)
Tulu-Helpsteer [7, 31]	775	6.34	29.85	28.98	44.10 (1.48×, 1.52×)
Ultrafeedback [4]	409	0.42	45.46	44.13	53.21 (1.17×, 1.21×)

Table 2: Comparison of training samples per second with sequence packing. For Flex Attn + Prefix Sharing + Packing, relative speedups over FA3 + Packing and Flex Attn + Packing are shown in parentheses, respectively. For the Prefix / Completion column, we report the median ratio. Our method (Prefix sharing + Packing) demonstrates at least a 30% increase in training throughput for most datasets. The impact of sequence packing is especially prominent for datasets like HH-RLHF and TLDR with shorter overall sequence lengths. Only Ultrafeedback, which has a extremely low prefix-to-completion ratio (0.3), shows a modest improvement of 21% over the FlexAttention baseline.

4.3.2 Packing Benchmarking

We use a First-Fit-Decreasing (FFD) based sampler for efficient sequence packing[4]. For both the FlexAttention-based baseline and our prefix sharing algorithm, we add a document ID-based attention mask to prevent cross-contamination. For FlashAttention-3, we use the packed position IDs (which is internally translated to an array of cumulative sequence lengths) to track the boundaries of different responses (and different documents) in a packed mini-batch. We set the packing length to be the desired batch size, `bsz` (set to be the same as before), multiplied by the maximum sequence length (`seq_len`) of the packing unit in the dataset. This provides higher packing efficiency than packing until `seq_len` and sampling `bsz` number of packed sequences. We pad all batches to the fixed packing length for better performance.

We show our results in Table 2. Overall, packing with prefix sharing consistently outperforms the baselines, with most datasets achieving about 1.3-1.6× improvement in training throughput compared to FlashAttention-3 with packing. Packing provides a significant boost in training throughput for datasets with low median overall lengths (such as HH-RLHF, TLDR) compared to the non-packing results. Since prefix sharing decreases the number of tokens in each packing unit (see Figure 2), sequence packing is more effective here than the baselines. When comparing the speedups with and without packing, we observe that the relative improvement over the FlashAttention-3 baseline increases from 1.07× to 1.41× for HH-RLHF (a 32% increase), and from 1.12× to 1.35× for TLDR (a 21% increase). UltraFeedback also still sees performance boosts (1.11× to 1.17×) with packing despite having a low prefix-to-completion ratio.

5 Conclusion

In this work, we present prefix sharing, a simple but effective technique for improving training throughput in paired preference optimization. By processing chosen and rejected responses as one sequence with a shared prefix, our method achieves up to 1.5× speedups on Direct Preference Optimization (DPO) datasets with high overall lengths and high prefix to completion length ratios. We further demonstrate the advantages of sequence packing with prefix sharing, enabling our method to benefit even datasets with smaller sequence lengths. While our experiments focus on DPO, our approach is applicable to other paired preference tuning methods, which we wish to explore in future work.

Acknowledgements

This research was supported with compute resources from Anyscale. We thank Will Lin and Sarah Schwettmann for their helpful feedback throughout this project.

[4] `https://github.com/imoneoi/multipack_sampler`

References

[1] Ben Athiwaratkun, Sujan Kumar Gonugondla, Sanjay Krishna Gouda, Haifeng Qian, Hantian Ding, Qing Sun, Jun Wang, Jiacheng Guo, Liangfu Chen, Parminder Bhatia, Ramesh Nallapati, Sudipta Sengupta, and Bing Xiang. Bifurcated attention: Accelerating massively parallel decoding with shared prefixes in llms, 2024.

[2] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. Training a helpful and harmless assistant with reinforcement learning from human feedback, 2022.

[3] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024.

[4] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Ultrafeedback: Boosting language models with scaled ai feedback, 2024.

[5] Luigi Daniele and Suphavadeeprasit. Amplify-instruct: Synthetically generated diverse multi-turn conversations for efficient llm training. *arXiv preprint arXiv:(coming soon)*, 2023.

[6] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

[7] Hamish Ivison and Yizhong Wang and Jiacheng Liu and Ellen Wu and Valentina Pyatkin and Nathan Lambert and Yejin Choi and Noah A. Smith and Hannaneh Hajishirzi. Unpacking DPO and PPO: Disentangling Best Practices for Learning from Preference Feedback, 2024.

[8] Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. Flexattention: The flexibility of pytorch with the performance of flashattention, 8 2024. PyTorch Blog.

[9] Jiwoo Hong, Noah Lee, and James Thorne. Orpo: Monolithic preference optimization without reference model, 2024.

[10] Jian Hu, Xibin Wu, Weixun Wang, Xianyu, Dehao Zhang, and Yu Cao. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework, 2024.

[11] Hamish Ivison, Yizhong Wang, Valentina Pyatkin, Nathan Lambert, Matthew Peters, Pradeep Dasigi, Joel Jang, David Wadden, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. Camels in a changing climate: Enhancing lm adaptation with tulu 2, 2023.

[12] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[13] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes, 2024.

[14] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *arXiv preprint arXiv:2107.02027*, 2021.

[15] Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms, 2024.

[16] Yinhan Liu. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[17] Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. Step-controlled dpo: Leveraging stepwise error for enhanced mathematical reasoning, 2024.

[18] Yu Meng, Mengzhou Xia, and Danqi Chen. Simpo: Simple preference optimization with a reference-free reward, 2024.

[19] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[20] Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddartha Naidu, and Colin White. Smaug: Fixing failure modes of preference optimisation with dpo-positive, 2024.

[21] Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason Weston. Iterative reasoning preference optimization, 2024.

[22] Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents, 2024.

[23] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024.

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

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[26] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[27] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

[28] Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. Learning to summarize from human feedback, 2022.

[29] Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, Nathan Sarrazin, Omar Sanseviero, Alexander M. Rush, and Thomas Wolf. Zephyr: Direct distillation of lm alignment, 2023.

[30] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. Trl: Transformer reinforcement learning. `https://github.com/huggingface/trl`, 2020.

[31] Zhilin Wang, Yi Dong, Jiaqi Zeng, Virginia Adams, Makesh Narsimhan Sreedhar, Daniel Egert, Olivier Delalleau, Jane Polak Scowcroft, Neel Kant, Aidan Swope, and Oleksii Kuchaiev. Helpsteer: Multi-attribute helpfulness dataset for steerlm, 2023.

[32] Martin Weyssow, Aton Kamanda, and Houari Sahraoui. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences, 2024.

[33] Yuchen Xia, Jiho Kim, Yuhan Chen, Haojie Ye, Souvik Kundu, Cong Hao, and Nishil Talati. Understanding the performance and estimating the cost of llm fine-tuning, 2024.

[34] Zihao Ye, Ruihang Lai, Bo-Ru Lu, Chien-Yu Lin, Size Zheng, Lequn Chen, Tianqi Chen, and Luis Ceze. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024.

[35] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models, 2024.

[36] Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, Zhenghao Liu, Bowen Zhou, Hao Peng, Zhiyuan Liu, and Maosong Sun. Advancing llm reasoning generalists with preference trees, 2024.

[37] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

Supplementary Material

A Packing Full Training Results

Naively applying packing with the same settings as non-packing could potentially lead to worse downstream performance, since for the same batch size the model trained with packing takes less gradient steps. Further, performance can be affected by cross-contamination across packed sequences. To verify that packing can match the performance of non-packing DPO, we evaluate the downstream performance of models trained with and without packing. To account for differences in effective batch size due to packing, we also sweep across different batch sizes.

We use the UltraFeedback dataset [4], which is a large synthetically generated preference dataset that has been a popular choice for aligning open-source models such as Zephyr-7B-β [29]. Using the same hyperparameters as Zephyr, we finetune the Zephyr-7B-SFT model[5] on UltraFeedback using DPO, sweeping across different packing and batch size settings. Finally, we evaluate each model using MT-Bench [37], a popular multi-turn benchmark that judges conversational and instruction following abilities.

As shown in Table 3, we find that packing roughly matches or exceeds the normal paired format for MT-Bench scores across all batch sizes, thus showing that packing for DPO does not harm performance.

Method	**Batch Size**	**MT-Bench Score**
Packing + Prefix Sharing	32	7.3
	64	**7.4**
	96	7.3
	128	7.0
Normal Paired Format	32	7.0
	64	7.1
	96	7.2
	128	7.1
Baseline (Zephyr-7B-SFT)	N/A	6.4

Table 3: MT-Bench [37] scores for different packing and non-packing DPO training across different batch sizes. Models were trained with Ultrafeedback using hyperparameters from Zephyr [29].

[5]The Zephyr model is trained by first applying SFT on Mistral-7B-v0.1 using the Ultrachat [6] dataset, and then applying DPO with UltraFeedback.