Liger-Kernel: Efficient Triton Kernels for LLM Training

Pin-Lun Hsu [1 2] Yun Dai [1] Vignesh Kothapalli [1] Qingquan Song [1] Shao Tang [1] Siyu Zhu [1] Steven Shimizu [1] Shivam Sahni [1] Haowen Ning [1] Yanning Chen [1] Zhipeng Wang [1]

Abstract

Training large language models (LLMs) efficiently at scale remains challenging due to rising compute and memory demands. We present `Liger-Kernel`, an open-source Triton kernel suite for core LLM primitives and diverse loss functions (pre-training, SFT, distillation, alignment, RLHF). Each kernel uses aggressive operator fusion, in-place gradient computation, and, where advantageous, input chunking to curb memory traffic and kernel-launch overhead. On widely used LLMs, these optimizations boost throughput by $\approx$20% and cut GPU memory consumption by $\approx$60% versus Hugging Face baselines. The code is available under a permissive license at https://github.com/linkedin/Liger-Kernel.

1. Introduction

Training Large Language Models (LLMs) at scale (Vaswani, 2017; Wei et al., 2022; Brown et al., 2020; Team et al., 2023; Touvron et al., 2023; Dubey et al., 2024; Liu et al., 2024; Bai et al., 2023) hinges on efficient compute infrastructure, yet host/device memory constraints and latency–bandwidth trade-offs often throttle performance. While algorithmic scaling helps, the true potential for optimization lies in kernel-level operation fusion that curtails memory traffic and capitalizes on GPU parallelism. Because such last-mile optimizations are amplified by inherent parallelism of GPUs, even the modest speed-ups propagate into large wall-time and cost reductions. Achieving these gains, however, demands deep expertise in LLM architectures and hardware.

To render such expert-level optimizations broadly accessible, we introduce `Liger-Kernel`—an open source, plug-and-play suite of Triton kernels (Tillet et al., 2019) that delivers state-of-the-art performance gains to any LLM training pipeline with only a handful of lines of code. `Liger-Kernel` enhances the efficiency and scalability of LLM training through a highly flexible and user-friendly interface. It streamlines complex tensor operations, minimizes computational overheads with kernel fusions, and seamlessly integrates with diverse computing environments. Novice users can improve LLM training efficiency with a few lines of code, while advanced users can customize their models with modular components and adaptive layer configurations to suit their needs. `Liger-Kernel` requires minimal dependencies, namely, PyTorch and Triton, while supporting multiple distributed frameworks, such as PyTorch FSDP (Zhao et al., 2023), DeepSpeed ZeRO (Rasley et al., 2020), and ZeRO++(Wang et al., 2023; Dai et al., 2024). Thus ensuring broad compatibility across platforms.

[1]LinkedIn Corporation, CA, USA [2]Now at xAI, Palo Alto, CA, USA. Correspondence to: Pin-Lun Hsu <byronhsu1230@gmail.com>, Yun Dai <yundai424@gmail.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25). Copyright 2025 by the author(s).

2. Related Works

PyTorch eager execution (Paszke et al., 2019) simplifies model authoring experience but incurs function-call, dispatch, and kernel-launch overhead. In addition, materializing activations op-by-op also inflates GPU memory use. A majority of the efforts to address this issue have focused on model compilation and algorithmic operation fusion, with Triton (Tillet et al., 2019) becoming the de-facto route to replace native PyTorch execution.

2.1. Model Compiler

Model compilers lower high-level computation graphs (e.g. `torch.nn.Module`) to hardware-tuned code. `torch.compile` (Ansel et al., 2024) captures the graph just-in-time (JIT), optimizes its intermediate representation (IR), and translates it into Triton (GPU) or OpenMP C++ (CPU). TVM (Chen et al., 2018) offers a unified IR across back-ends; XLA (Sabne, 2020) fuses and schedules TensorFlow/JAX graphs; nvFuser generates CUDA specialized for NVIDIA GPUs. Each of these systems streamlines operation fusion, layout selection, and kernel generation, complementing algorithm-specific kernels.

2.2. Algorithmic Operation Fusion

Operation fusion reduces HBM–SRAM traffic by co-locating successive computations within a single kernel, eliminating per-op launch overhead and materializing intermediate activations between operations. For example, FlashAttention (Dao et al., 2022; Dao, 2023) partitions attention into SRAM-sized tiles, shrinking memory complexity from $O(L^2)$ to $O(L)$ and boosting speed via higher register and thread utilization. Such algorithm-aware kernels often surpass generic compiler fusion by exploiting domain structures (e.g., attention head parallelism or register allocation patterns).

2.3. Custom Operation Fusion with Triton

Triton provides a Pythonic DSL and JIT compiler for writing these kernels without low-level CUDA boilerplate, enabling portable, self-contained libraries. Meta's `xFormers` (Lefaudeux et al., 2022), Dao's FlashAttention repo[1], and Unsloth[2] exemplify Triton-based LLM optimizations, while EfficientCrossEntropy[3] fuses projection and loss to avoid full-logit materialization. `Liger-Kernel` builds on these ideas (Section 3.2) to provide a unified, extensible suite of fused kernels for state-of-the-art LLM training.

3. `Liger-Kernel`

3.1. API Design and Integrations

Ease of use is vital for the wider adoption of any open-source library. To this end, `Liger-Kernel`'s API minimizes disruption to existing code while offering varying levels of customization. Depending on the granularity of control, users can integrate the Liger kernels into their training pipelines in several ways (see also Figure 1):

1. **Using `AutoLigerKernelForCausalLM`:** The simplest way to leverage Liger kernels. It automatically patches supported causal-LM code with a single import—no manual model changes required.
2. **Model-Specific Patching APIs:** For finer control, users can leverage `Liger-Kernel`'s model-specific patching APIs. These APIs can be applied to various architectures such as sequence classifiers.
3. **Customizing Model Architectures:** users can import individual Liger kernels (e.g. `LigerGEGLUMLP` and `LigerCrossEntropyLoss`) and design efficient model architectures (as also shown by the `LigerTransformer` example in Figure 1b).

`Liger-Kernel` has also been successfully integrated with several popular training frameworks within the machine learning community, including Hugging Face transformers' `Trainer` class[4], Hugging Face TRL's (von Werra et al., 2020) `SFTTrainer` class[5], `Axolotl`[6], and `LLaMA-Factory` (Zheng et al., 2024). Thus presenting a flexible option for developers to integrate Liger kernels into their workflows and democratize its usage.

3.2. Kernels

`Liger-Kernel` is mainly composed of two types of kernels: (1) kernels that compute fundamental building blocks in LLM, and (2) memory efficient kernels that fuse vocabulary space projection layer with downstream losses, chunked along token dimension.

3.2.1. LLM Building Blocks

LLMs share a common set of auxiliary building blocks, so providing fast, drop-in implementations for these modules benefits most architectures. `Liger-Kernel` supplies highly optimized kernels for RMSNorm, GeGLU/SwiGLU, RoPE, and other essential components. We deliberately target operations outside the attention and MLP paths—where specialist libraries such as FlashAttention and CUTLASS already excel—to deliver the greatest performance.

Most kernels leverage three key optimization strategies:

Recomputation. This approach trades inexpensive arithmetic for lower peak memory. For example, RMSNorm kernel only caches a single reciprocal RMS value per row in the forward pass, and then reloads it along with the raw inputs to recompute normalized activations on the backward pass, saving $O(n_{\text{col}})$ storage per row at the cost of a few extra FLOPs.

In-place execution. We eliminate extra allocations and copies by reusing buffers whenever an intermediate value has exactly one producer and one consumer. For example, in our RoPE kernel we overwrite each token's Q and K head vectors in-place with their rotated outputs—avoiding a separate copy of the full tensor.

Coarsening. We fuse fine-grained operations into larger compute blocks to boost arithmetic intensity and cut down on kernel launches. For instance, in the RoPE implementation, both the Q-head and K-head rotations for each token are handled within a single Triton program, rather than dispatching two back-to-back kernels, which increases per-program register usage and reduces scheduling overhead.

3.2.2. Fused Kernels with Input-Chunking

The rapid expansion of vocabulary enhances token granularity and achieve more compact prompt representations.

[1] github.com/dao-ailab/flash-attention
[2] github.com/unslothai/unsloth
[3] github.com/mgmalek/efficient_cross_entropy
[4] https://huggingface.co/docs/transformers/en/main_classes/trainer
[5] https://huggingface.co/docs/trl/main/en/sft_trainer
[6] https://axolotl-ai-cloud.github.io/axolotl/#liger-kernel

```python
# Model Agnostic Patching API
from liger_kernel.transformers import \
AutoLigerKernelForCausalLM

path="path/to/some/model"
model=AutoLigerKernelForCausalLM.from_pretrained(path)

# Model Specific Patching API
from liger_kernel.transformers import \
apply_liger_kernel_to_llama()
model=AutoModelForSequenceClassification.from_pretrained(path)
```

(a) API usage via patching.

```python
# Composable Custom Model API
from liger_kernel.transformers import LigerGEGLUMLP
from liger_kernel.transformers import LigerCrossEntropyLoss

class LigerTransformer(torch.nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
        # use Triton-optimized LigerGEGLUMLP
        self.geglu_mlp = LigerGEGLUMLP(...)

# use the Triton-optimized LigerCrossEntropyLoss
loss_fn=LigerCrossEntropyLoss()
model=LigerTransformer(...)
```

(b) Integration of custom kernels into model/training workflows.

Figure 1. Approaches to use `Liger-Kernel` APIs which are compatible with the widely used transformers library.

However, this has revealed a significant challenge: *the materialization of logit tensors during loss computation consumes excessive memory*. This issue has become a major bottleneck in LLM training, limiting the max batch size and context length. For example, training Gemma (with 256K vocab size) with batch size of 8 and sequence length of 4096 results in a 16.8 GB logit tensor of precision bfloat16, causing a huge spike in memory footprint[7].

This motivates us to explore the chunked logit and gradient computation approaches to amortize the memory consumption[8]. In this section, we present the *Fused Linear Chunked Loss*, a flexible and extensible interface that supports chunked optimization across different loss functions. This includes both causal loss such as cross-entropy loss, and post-training objectives such as ORPO, JSD, and GRPO.

We designed the `FusedLinearBase` class to capture the essential chunking logic, and implemented it as a custom `torch.autograd.Function`. This class orchestrates the chunking strategy, performs the forward pass of the language modeling head on the transformer's last hidden states, and calls an abstract loss function, which must be overridden by any downstream subclass implementing a desired loss. Crucially, `FusedLinearBase` is also responsible for computing the gradients of loss computed by each chunk with respect to the inputs during the forward pass itself. To avoid the overhead of coding the gradient computation logic for each specific loss, we make use of `torch.func.grad_and_value`, which provides a convenient way to compute both the loss and its gradients within a single functional interface.

By calculating gradients immediately after each chunk's forward computation, the corresponding chunk's logits can be safely discarded. This strategy ensures that only a single chunk of logits resides in GPU HBM at any point in time, substantially reducing peak memory usage and enabling scalability to larger batch sizes or model variants.

[7]The memory usually peaks at the end of each forward pass right before the release of the activations in the backward pass.

[8]This is inspired from the GitHub discussions https://huggingface.co/docs/transformers/en/main_classes/trainer and the solution from https://github.com/mgmalek/efficient_cross_entropy

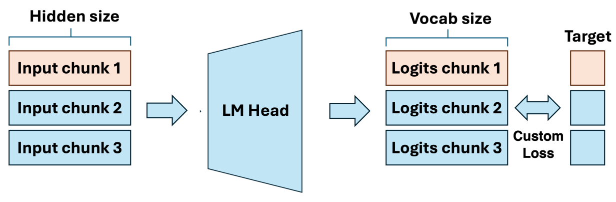


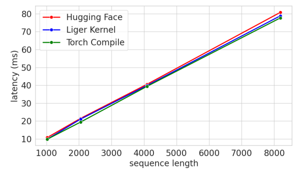
Figure 2. Chunking illustration. The input chunks are fed into LM head sequentially and produce logit chunks, then the loss computation with the target chunks is performed.

Finally, to maximize kernel-level performance, we wrap the forward and backward logic of each chunk in a `torch.compile` context. This allows the PyTorch compiler to fuse operations and eliminate redundant memory transfers, further improving throughput without compromising numerical correctness.

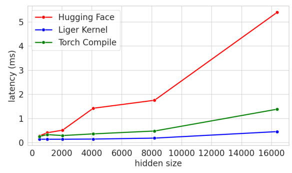
3.3. Testing Best Practices

For every code change, `Liger-Kernel` exercises four stages of validation—correctness, contiguity, convergence, and performance—to ensure that no merge ever degrades precision or throughput. Correctness tests compare each Triton kernel against a pure PyTorch reference (e.g., Hugging Face's implementation) over a range of regular and irregular shapes and dtypes, applying strict tolerances and only relaxing them when convergence-driven tests justify it. Contiguity checks enforce all input tensors are laid out sequentially, catching issues like the RoPE divergence we once saw when derivatives weren't stored contiguously.

Beyond unit tests, we run convergence tests on scaled-down, "real-world" training scenarios to verify that logits, weights, and loss track exactly with the reference across end-to-end epochs. Finally, our performance benchmarks mea-

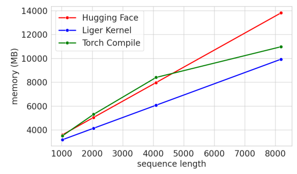


(a) GeGLU

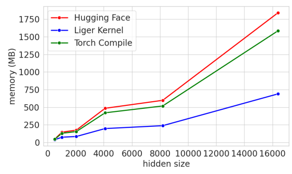


(b) RoPE

Figure 3. Kernel latency benchmarks for GeGLU and RoPE.

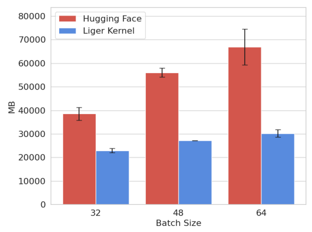


(a) GeGLU

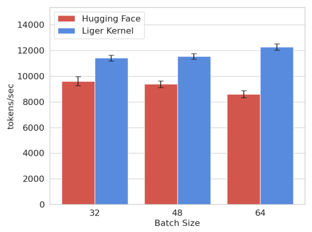


(b) RoPE

Figure 4. Kernel memory benchmarks for GeGLU and RoPE.

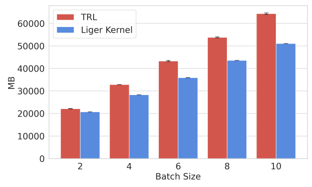


(a) Peak allocated memory

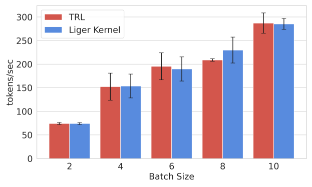


(b) Throughput

Figure 5. Comparison of peak allocated memory and training throughput (tokens/sec) for LLaMA 3-8B.



(a) Peak allocated memory



(b) Throughput

Figure 6. Comparison of peak allocated memory and throughput during GRPO training with TRL on Qwen3-0.6B. Liger's Fused Linear Chunked Loss reduces memory usage by up to 30% and allows training with larger batches (12-16) that would otherwise cause OOM errors – all while maintaining throughput comparable to the standard TRL implementation.

sure speed and memory usage against the baseline using representative hyperparameters, ensuring every Triton re-implementation yields tangible gains before it's merged.

4. Numerical Experiments

This section presents the kernel level and end-end LLM training benchmarks using `Liger-Kernel`.

Kernel Benchmarks Figure 3-4 reports benchmarking result of each LLM building block kernel on a single H100 80GB GPU over 10 trials with $[0.2, 0.8]$ percentile intervals. We benchmark against vanilla implementation in Hugging Face model source code and `torch.compile`'ed version. `Liger-Kernel`'s Triton GeGLU kernel matches `torch.compile` latency while cutting peak memory by $\approx$12–16%, and its RoPE kernel delivers a 2.3× speed-up and $\approx$20% lower memory than the Hugging Face baseline at a 16k hidden size. These double-digit gains achieved with identical numerical accuracy, underscore the impact of swapping even a few ubiquitous primitives for their `Liger-Kernel` counterparts.

End-to-End benchmarks We fine-tune LLMs on the Alpaca dataset (512 token context) using the Hugging Face Trainer across 4 NVIDIA A100 (80 GB) GPUs and varying batch sizes. All runs use bfloat16 precision, AdamW with cosine LR decay, and metrics are sampled after 20 warmup steps, with standard errors computed over 5 repetitions. The results are reported in Figure 5 and Appendix C. At batch size 64, LLaMA 3-8B achieved a 42.8% throughput boost and a 54.8% GPU-memory reduction, enabling larger batches or longer sequences on smaller hardware. At batch size 48, our kernels increased Qwen2 throughput by 25.5% while cutting memory use by 56.8% (Figure 13).

5. Conclusions

We introduce `Liger-Kernel`, a Triton-based library that brings expert-level optimizations to LLM training. Custom RMSNorm, RoPE, and GeGLU/SwiGLU kernels outperform HuggingFace's versions—our RoPE is $8\times$ faster and uses $3\times$ less memory. To alleviate loss-computation bottlenecks on large vocabularies, we implement input-chunked, online-softmax kernels with in-place gradient updates, enabling larger batches and longer contexts. Our new ORPO-loss kernel cuts peak memory $15\times$ and decouples it from batch size, making alignment runs feasible on modest hardware. When fine-tuning LLaMA-3 8B across 4 A100 GPUs, `Liger-Kernel` achieves 42.8% higher throughput and 54.8% lower memory consumption.

Since its release, `Liger-Kernel` has attracted contributions from the open-source community, extending support to new models (e.g., Qwen2.5-VL (Bai et al., 2025), LlaVA (Liu et al., 2023)) and kernels (e.g., GRPO, SparseMax). We will extend support to alignment, RLHF, and more open-source models to further empower research and deployment.

Acknowledgements

We thank AMD and Intel for funding GPUs for our AMD and Intel CI. We also thank Modal for funding credits from GPU MODE IRL for our NVIDIA CI.

We thank Triton[9], flash-attention[10], and Unsloth[11] for the reference of Triton kernels for LLM training, tiny shakespeare dataset[12] and llm.c[13] for convergence testing design, Efficient Cross Entropy[14] for fused linear cross entropy reference, AutoAWQ[15] and Robert Shaw for Automodel API design, as well as Hugging Face, PyTorch Lightning, Axolotl, and Llama-Factory for the collaboration on framework integration.

Finally, we extend our sincere appreciation to the `Liger-Kernel` open-source community. The meticulous bug reports, well-crafted pull requests, and exhaustive maintenance efforts have been indispensable to the maturity and robustness of this project.

Impact Statement

`Liger-Kernel` is an open-source Triton library that accelerates key LLM operations and significantly reduces peak GPU memory. Our work facilitates smaller research labs and companies to train state-of-the-art models, minimize energy consumption, and contributes to a more sustainable and accessible AI development life-cycle.

References

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, 2024.

Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Bai, S., Chen, K., Liu, X., Wang, J., Ge, W., Song, S., Dang, K., Wang, P., Wang, S., Tang, J., et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pp. 1877–1901, 2020.

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Dai, Y., Dharamsi, T., Hsu, P.-L., Song, T., and Firooz, H. Enhancing stability for large models training in constrained bandwidth networks. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Hong, J., Lee, N., and Thorne, J. Reference-free monolithic preference optimization with odds ratio. *arXiv preprint arXiv:2403.07691*, 2024.

Lefaudeux, B., Massa, F., Liskovich, D., Xiong, W., Caggiano, V., Naren, S., Xu, M., Hu, J., Tintore, M., Zhang, S., Labatut, P., Haziza, D., Wehrstedt, L., Reizenstein, J., and Sizov, G. xFormers: A modular and hackable transformer modelling library. `https://github.com/facebookresearch/xformers`, 2022.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Liu, H., Li, C., Wu, Q., and Lee, Y. J. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023.

[9] https://triton-lang.org/main/getting-started/tutorials/index.html
[10] https://github.com/dao-ailab/flash-attention
[11] https://github.com/unslothai/unsloth
[12] https://huggingface.co/datasets/karpathy/tiny_shakespeare
[13] https://github.com/karpathy/llm.c
[14] https://github.com/mgmalek/efficient_cross_entropy
[15] https://github.com/casper-hansen/AutoAWQ

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. 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*, pp. 3505–3506, 2020.

Sabne, A. XLA : Compiling machine learning for peak performance, 2020.

Shazeer, N. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Roformer, Y. L. Enhanced transformer with rotary position embedding., 2021. *DOI: https://doi. org/10.1016/j. neucom*, 2023.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

von Werra, L., Belkada, Y., Tunstall, L., Beeching, E., Thrush, T., Lambert, N., Huang, S., Rasul, K., and Gallouédec, Q. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

Wang, G., Qin, H., Jacobs, S. A., Holmes, C., Rajbhandari, S., Ruwase, O., Yan, F., Yang, L., and He, Y. Zero++: Extremely efficient collective communication for giant model training. *arXiv preprint arXiv:2306.10209*, 2023.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

Zhang, B. and Sennrich, R. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12): 3848–3860, 2023.

Zheng, Y., Zhang, R., Zhang, J., Ye, Y., Luo, Z., Feng, Z., and Ma, Y. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

A. Kernels and Benchmarks

Throughout the discussion, vectors and matrices are represented by bolded lowercase and uppercase letters, e.g., $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{W} \in \mathbb{R}^{m\times n}$. Vectors are assumed to be column vectors unless otherwise specified. The all-ones vector is denoted as $\mathbf{1}_n \in \mathbb{R}^n$. Functions are applied to the variable element-wise, i.e., $f(\boldsymbol{x})_i = f(x_i)$. We use $\odot$ to denote the element-wise product between tensors, and the super-script $^\top$ to denote the matrix transpose. Unless otherwise specified in our kernel implementations, both input and output tensors are reshaped into two-dimensional matrices with the shape $(B \times T, H)$, where B is the batch size, T is the sequence length and H is the hidden dimension. In each kernel, Triton parallelizes operations on each row of input[16]. Therefore, we focus on the mathematical operations given a row of input $\boldsymbol{x} \in \mathbb{R}^H$ and the corresponding output $\boldsymbol{y} \in \mathbb{R}^H$ across all kernels discussed below. Finally, considering a loss $\mathcal{L} \in \mathbb{R}$ during training, we use $\nabla_{\boldsymbol{y}}\mathcal{L}$ to denote the gradient back-propagated to $\boldsymbol{y}$ during the backward-pass.

A.1. Rotary Position Embedding (RoPE)

We fuse the query and key rotation embedding computation into a single kernel to reduce overheads. For each rotary position embedding computation, given the input $\boldsymbol{x} \in \mathbb{R}^d$, the token position m and the rotation matrix $\boldsymbol{R}^H_{\Theta,m} \in \mathbb{R}^{H\times H}$, the output $\boldsymbol{y} \in \mathbb{R}^H$ is

$$\boldsymbol{y} = \boldsymbol{R}^H_{\Theta,m}\boldsymbol{x}. \tag{1}$$

Here the $\boldsymbol{R}^H_{\Theta,m}$ matrix is given by:

$$\begin{pmatrix}
\cos m\theta_1 & \dots & 0 & -\sin m\theta_1 & \dots & 0 \\
0 & \dots & 0 & 0 & \dots & 0 \\
0 & \dots & 0 & 0 & \dots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & \cos m\theta_{H/2} & 0 & \dots & -\sin m\theta_{H/2} \\
\sin m\theta_1 & \dots & 0 & \cos m\theta_1 & \dots & 0 \\
0 & \dots & 0 & 0 & \dots & 0 \\
0 & \dots & 0 & 0 & \dots & 0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \dots & \sin m\theta_{H/2} & 0 & \dots & \cos m\theta_{H/2}
\end{pmatrix}. \tag{2}$$

Our implementation of RoPE assumes a rotation matrix in the form of HuggingFace model instead of the rotation matrix described in Su et al. (2023). The parameters Θ are model specific. In the backward pass, we have:

$$\nabla_{\boldsymbol{x}}\mathcal{L} = (\boldsymbol{R}^H_{\Theta,m})^\top \nabla_{\boldsymbol{y}}\mathcal{L}. \tag{3}$$

In the implementation, due to the sparsity of $\boldsymbol{R}^H_{\Theta,m}$, we adopt the efficient sparse computation in Su et al. (2023).

A.2. GeGLU

Given the input $\boldsymbol{x} \in \mathbb{R}^H$ and learnable parameters $\boldsymbol{W} \in \mathbb{R}^{H\times H}, \boldsymbol{V} \in \mathbb{R}^{H\times H}, \boldsymbol{b} \in \mathbb{R}^H$ and $\boldsymbol{c} \in \mathbb{R}^H$, the output $\boldsymbol{y} \in \mathbb{R}^H$ is defined as (Shazeer, 2020):

$$\boldsymbol{y} = \text{GELU}(\boldsymbol{W}x + \boldsymbol{b}) \odot (\boldsymbol{V}x + \boldsymbol{c}), \tag{4}$$

where we use the `tanh` approximation of GELU (Hendrycks & Gimpel, 2016). Let $c_1 = 0.044715, c_2 = 0.134145$. The $\text{GELU}(z)$ is formulated as:

$$\text{GELU}(z) \approx 0.5z\left(1 + \tanh\left[\sqrt{2/\pi}\left(z + c_1 z^3\right)\right]\right). \tag{5}$$

Let $\boldsymbol{x}^{(1)} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^H$ and $\boldsymbol{x}^{(2)} = \boldsymbol{V}\boldsymbol{x} + \boldsymbol{c} \in \mathbb{R}^H$. The forward pass can be computed as:

$$\boldsymbol{y}(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) = \text{GELU}(\boldsymbol{x}^{(1)}) \odot \boldsymbol{x}^{(2)}. \tag{6}$$

[16]We compute the number of warps based on the block size, which is dependent upon the size of each row. We reuse the `calculate_settings` function from https://github.com/unslothai/unsloth/blob/main/unsloth/kernels/utils.py.

In the backward pass, we have:

$$\begin{aligned}\nabla_{\boldsymbol{x}^{(1)}}\mathcal{L} &= \nabla_{\boldsymbol{y}}\mathcal{L} \odot \nabla_{\boldsymbol{x_1}}\mathrm{GELU}(\boldsymbol{x}^{(1)}) \odot \boldsymbol{x}^{(2)},\\ \nabla_{\boldsymbol{x}^{(2)}}\mathcal{L} &= \nabla_{\boldsymbol{y}}\mathcal{L} \odot \mathrm{GELU}(\boldsymbol{x}^{(1)}),\end{aligned} \tag{7}$$

where:

$$\begin{aligned}\nabla_{\boldsymbol{x}^{(1)}}\mathrm{GELU}(\boldsymbol{x}^{(1)}) \approx\ & 0.5 \odot (1 + u(\boldsymbol{x_1})) +\\ & \sqrt{1/(2\pi)}\boldsymbol{x}^{(1)} \odot \left(1 - u(\boldsymbol{x}^{(1)})^2\right) \odot \left(1 + c_2(\boldsymbol{x}^{(1)})^2\right),\\ u(\boldsymbol{x}^{(1)}) = \tanh & \left[\sqrt{2/\pi}\left(\boldsymbol{x}^{(1)} + c_1(\boldsymbol{x}^{(1)})^3\right)\right].\end{aligned} \tag{8}$$

A.3. Cross-Entropy (CE)

We move the gradient computation to the forward function along with an inplace replacement of the logit tensor to avoid them being materialized simultaneously. We also adopt online softmax computation to compute the gradient on the fly. Given the input logits $\boldsymbol{x} \in \mathbb{R}^V$, where V is the vocabulary size, and target one-hot encoded label $\boldsymbol{t}$, the output probabilities are given as:

$$\boldsymbol{y} = \mathrm{softmax}(\boldsymbol{x}), \tag{9}$$

and the cross-entopy loss is defined as $\mathcal{L} = -\sum_i t_i \log(y_i)$. The gradient back-propagated to $\boldsymbol{x}$ is given by:

$$\nabla_{\boldsymbol{x}}\mathcal{L} = \boldsymbol{y} - \boldsymbol{t}. \tag{10}$$

Additionally, we also employ the safe $\log$ operation to avoid numerical instabilities.

A.4. RMSNorm.

We fuse the normalization and scaling steps of the RMSNorm computation into a single Triton kernel[17]. Specifically, given the input $\boldsymbol{x} \in \mathbb{R}^H$ and the learnable parameters $\boldsymbol{\gamma} \in \mathbb{R}^H$, the output $\boldsymbol{y} \in \mathbb{R}^H$ is defined as (Zhang & Sennrich, 2019):

$$\boldsymbol{y} = \hat{\boldsymbol{x}} \odot \boldsymbol{\gamma}, \qquad \hat{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\mathrm{RMS}(\boldsymbol{x})}, \tag{11}$$

where $\hat{\boldsymbol{x}} \in \mathbb{R}^H$ is the normalized input, $\mathrm{RMS}(\boldsymbol{x}) = \sqrt{\sum_i x_i^2/H + \epsilon}$ and ϵ is a small constant for numerical stability. In the backward pass, we have the gradient back-propagated to $\boldsymbol{x}$ and $\boldsymbol{\gamma}$ as

$$\begin{aligned}\nabla_{\boldsymbol{x}}\mathcal{L} &= \frac{1}{\mathrm{RMS}(\boldsymbol{x})}\left(\nabla_{\boldsymbol{y}}\mathcal{L} \odot \boldsymbol{\gamma} - \underbrace{\left[\hat{\boldsymbol{x}}^\top(\nabla_{\boldsymbol{y}}\mathcal{L} \odot \boldsymbol{\gamma})/H\right]}_{\text{a numerical value}}\hat{\boldsymbol{x}}\right),\\ \nabla_{\boldsymbol{\gamma}}\mathcal{L} &= \nabla_{\boldsymbol{y}}\mathcal{L} \odot \hat{\boldsymbol{x}}.\end{aligned} \tag{12}$$

Since the same $\boldsymbol{\gamma}$ is applied to all input vectors $\boldsymbol{x}$ in the same batch, the gradients need to be summed up.

A.5. SwiGLU.

We fuse the element-wise operations in the SwiGLU computation into a single kernel. Given the input $\boldsymbol{x} \in \mathbb{R}^H$ and learnable parameters $\boldsymbol{W} \in \mathbb{R}^{H\times H}$, $\boldsymbol{V} \in \mathbb{R}^{H\times H}$, $\boldsymbol{b} \in \mathbb{R}^H$ and $\boldsymbol{c} \in \mathbb{R}^H$, the output $\boldsymbol{y} \in \mathbb{R}^H$ is defined as (Shazeer, 2020):

$$\begin{aligned}\boldsymbol{y} &= \mathrm{Swish}_{\beta=1}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \odot (\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c})\\ &= \mathrm{SiLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \odot (\boldsymbol{V}\boldsymbol{x} + \boldsymbol{c}),\end{aligned} \tag{13}$$

[17]The implementation is referenced the code from https://github.com/unslothai/unsloth/blob/main/unsloth/kernels/rms_layernorm.py and https://triton-lang.org/main/getting-started/tutorials/05-layer-norm.html.

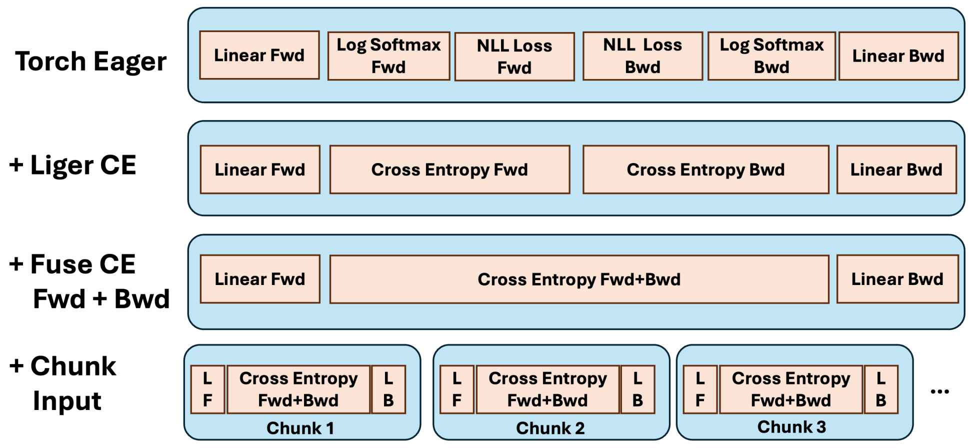


Figure 7. Profiling trace view of FLCE optimization. Here '+' denotes the optimization added to the previous step.

where $\text{SiLU}(z) = z\sigma(z)$ and $\sigma(z) = (1 + \exp(-z))^{-1}$ is the sigmoid function. We only consider the $\beta = 1$ case here where Swish degenerates to SiLU, which aligns with the implementation of existing supported HuggingFace LLMs. Denote the values $\boldsymbol{x_1} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^H$ and $\boldsymbol{x_2} = \boldsymbol{V}\boldsymbol{x} + \boldsymbol{c} \in \mathbb{R}^H$, we implement the kernel for the forward pass as:

$$\boldsymbol{y}(\boldsymbol{x_1}, \boldsymbol{x_2}) = \text{SiLU}(\boldsymbol{x_1}) \odot \boldsymbol{x_2}. \tag{14}$$

Recall $\nabla_{\boldsymbol{y}}\mathcal{L}$ as the gradient back-propagated from $\mathcal{L}$ to $\boldsymbol{y}$. In the backward pass, we have

$$\begin{aligned} &\nabla_{\boldsymbol{x_1}}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L} \odot [\sigma(\boldsymbol{x_1}) + \text{SiLU}(\boldsymbol{x_1}) \odot (1 - \sigma(\boldsymbol{x_1}))] \odot \boldsymbol{x_2}, \\ &\nabla_{\boldsymbol{x_2}}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L} \odot \text{SiLU}(\boldsymbol{x_1}). \end{aligned} \tag{15}$$

A.6. FusedLinearCrossEntropy (FLCE)

The main idea of FLCE is to avoid the materialization of output logits of the language model head. We achieved so by conducting three levels of optimization compared with torch eager mode. Firstly, in eager pytorch, cross entropy consists of LogSoftmax + NLL Loss. We leverage Liger CE to fuse Log softmax and NLL loss into one kernel call. Secondly, we fuse CE forward and backward, so the gradient is computed right after we get the activations in a single kernel call. Finally, we chunk the input based on batch size and seq length dimension and compute the input sequentially. We show an illustration of chunking performed in FLCE in Figure 2.

Given the linear head matrix $\boldsymbol{W} \in \mathbb{R}^{H \times V}$, with a vocabulary size V, and $\boldsymbol{Z_i} \in \mathbb{R}^{S_{\text{chunk}} \times H}$, denoting a chunk of the flattened hidden state matrix $\boldsymbol{Z} \in \mathbb{R}^{BT \times H}$ where B is the batch size, T is the sequence length, H is the hidden size, and S_{chunk} is the chunk size, the forward pass for a chunk through the linear layer is computed as:

$$\boldsymbol{O_i} = \boldsymbol{Z_i}\boldsymbol{W} \quad \text{for } i = 1, 2, \ldots, n_{\text{chunk}}. \tag{16}$$

Here, $\boldsymbol{O_i}$ represents the logits projected from $\boldsymbol{Z_i}$, for which, gradients can be derived based on (10). Since the same weight $\boldsymbol{W}$ is used for projecting all chunks, its final gradient needs to be summed up. The gradient computation in the backward pass can be represented as:

$$\nabla_{\boldsymbol{Z_i}}\mathcal{L} = \nabla_{\boldsymbol{O_i}}\mathcal{L} \cdot \boldsymbol{W}^\top, \ \ \nabla_{\boldsymbol{W}}\mathcal{L} = \sum_{\boldsymbol{i}}^{\boldsymbol{n}_{\text{chunk}}} \boldsymbol{Z_i}^\top \nabla_{\boldsymbol{O_i}}\mathcal{L} \tag{17}$$

While it is intuitive to assume that chunking might degrade performance, our findings indicate that careful chunking of the input tensor allows us to retain the compute-bound behavior. This is attributable to the sufficiently large size of $\boldsymbol{W}$, which allows us to effectively leverage chunking while maintaining computational efficiency. In practice, we set the chunk size to be $2^{\lceil \log_2 \lceil \frac{BT}{\lceil V/H \rceil} \rceil \rceil}$ with an intuition on picking the chunk size to be closer to the hidden dimension size to balance the trade-off between memory allocation and processing speed. Figure 7 shows the optimizations from a profiling trace perspective.

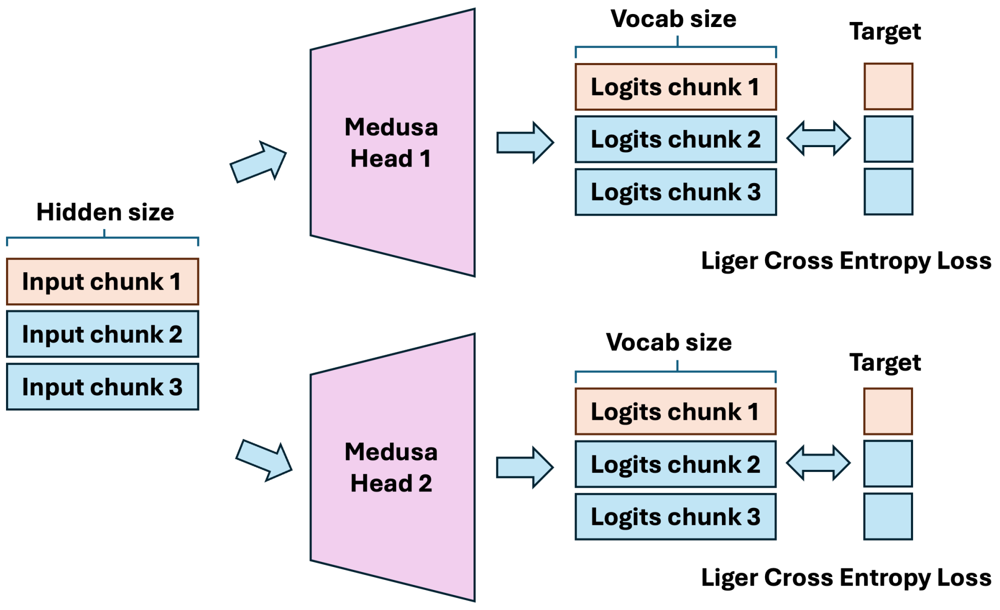


Figure 8. FLCE chunking on medusa. The input chunks are fed into multiple LM heads sequentially.

A.7. Medusa

Medusa (Cai et al., 2024) is a simple framework that accelerates token decoding in auto-regressive LLMs by using multiple decoding heads to predict several subsequent tokens in parallel. Especially, the linear head matrix $\boldsymbol{W} \in \mathbb{R}^{H \times V}$ along with K Medusa heads $\{\widetilde{\boldsymbol{W}}_i\}_{i=1}^{K} \in \mathbb{R}^{H \times V}$ are employed to predict the next $(K+1)$ tokens in parallel. Medusa training has two flavors. The first, called *Stage-1*, involves training only the additional Medusa heads $\{\widetilde{\boldsymbol{W}}_i\}_{i=1}^{K}$ while keeping the backbone LLM frozen. The second approach, called *Stage-2*, tunes the backbone model, the original LM head and the Medusa heads $\{\widetilde{\boldsymbol{W}}_i\}_{i=1}^{K} \cup \boldsymbol{W}$ simultaneously. Due to the parallel decoding nature of the K Medusa heads, the peak memory required to train the model scales almost linearly with K, especially when the vocabulary size is significantly larger than the hidden feature dimension.

The Liger FLCE kernel is particularly effective in this context, as it eliminates the need to materialize logits for each decoding head (See Figure 8). This is critical in scenarios with large vocabulary sizes, such as LLaMA-3's 128K tokens, where materializing logits can lead to significant memory consumption. In particular, by leveraging the Liger FLCE kernel, we ensure that the computes gradients are stored 'in-place', without materializing the full logits tensor. Thus enabling the users to scale hyper-parameters pertaining to sequence lengths/batch size, hidden dimension for exploration and development in multi-token prediction.

A.8. Odds Ratio Loss

The Liger optimized CE/FLCE kernels are generic in nature and can be widely employed in LLM pre-training and supervised fine-tuning (SFT) workflows. Additionally, to cater to use-cases which require preference tuning of LLMs, we develop an optimized *Odds Ratio Loss* kernel as part of the *Odds Ratio Preference Optimization (ORPO)* training framework (Hong et al., 2024). Formally, for an input sequence $\boldsymbol{s}$ to the LLM, the probability of predicting a specific output sequence $\boldsymbol{o}$ of length M is given by:

$$\log P(\boldsymbol{o}|\boldsymbol{s}) = \frac{1}{M}\sum_{i=1}^{M} \log P(o_i|\boldsymbol{s}, \boldsymbol{o}_{<i}). \tag{18}$$

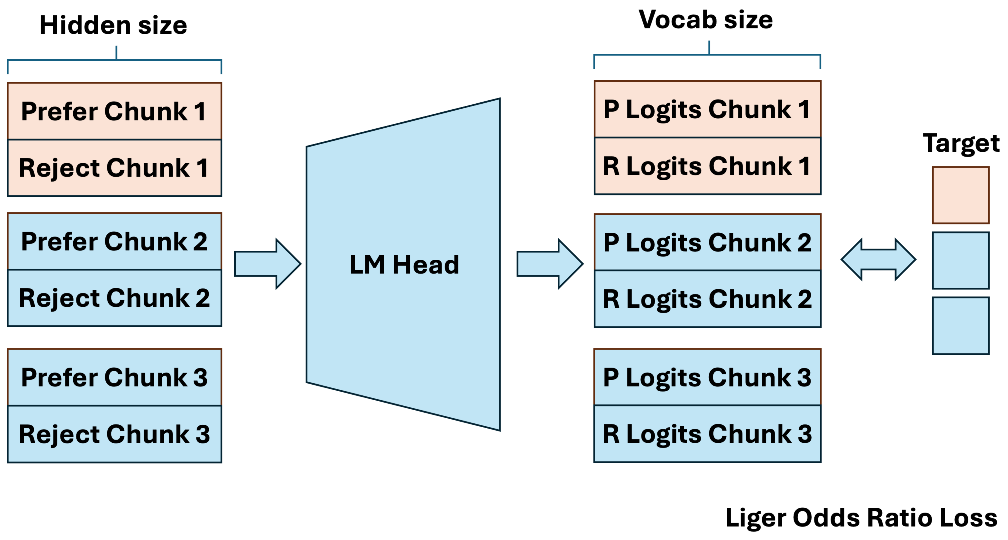


Figure 9. Odds Ratio Loss with chunked batches of preferred and rejected responses, that are sequentially fed into the LM head to reduce peak memory consumption.

With this notation in place, the odds of predicting a specific preferred response $\boldsymbol{o}_w$ is given by:

$$\text{odds}(\boldsymbol{o}_w|\boldsymbol{s}) = \frac{P(\boldsymbol{o}_w|\boldsymbol{o})}{1 - P(\boldsymbol{o}_w|\boldsymbol{s})}. \tag{19}$$

In the preference tuning paradigm, we are generally provided with a dis-preferred (rejected) response $\boldsymbol{o}_l$ along with a preferred response $\boldsymbol{o}_w$ for a given input $\boldsymbol{s}$. The odds ratio loss $\mathcal{L}_{OR}$ aims to maximize the odds of predicting $\boldsymbol{o}_w$ over $\boldsymbol{o}_l$, and can be formulated as:

$$\mathcal{L}_{OR} = -\log \sigma \left(\log \frac{\text{odds}(\boldsymbol{o}_w|\boldsymbol{s})}{\text{odds}(\boldsymbol{o}_l|\boldsymbol{s})} \right), \tag{20}$$

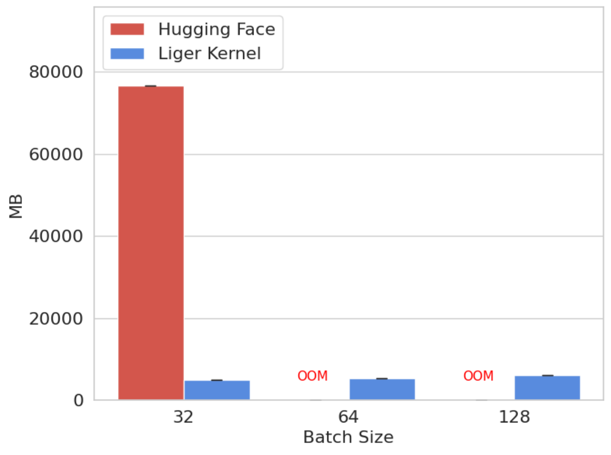
where σ denotes the softmax function. The ORPO training strategy combines $\mathcal{L}_{OR}$ with the standard CE loss (denoted by $\mathcal{L}_{SFT}$) over the tokens of the preferred response $\boldsymbol{o}_w$. Thus, $\mathcal{L}_{ORPO}$ can be given as (Hong et al., 2024):

$$\mathcal{L}_{ORPO} = \mathbb{E}_{(\boldsymbol{s},\boldsymbol{o}_w,\boldsymbol{o}_l)} \left[\mathcal{L}_{SFT} + \lambda \mathcal{L}_{OR} \right] \tag{21}$$

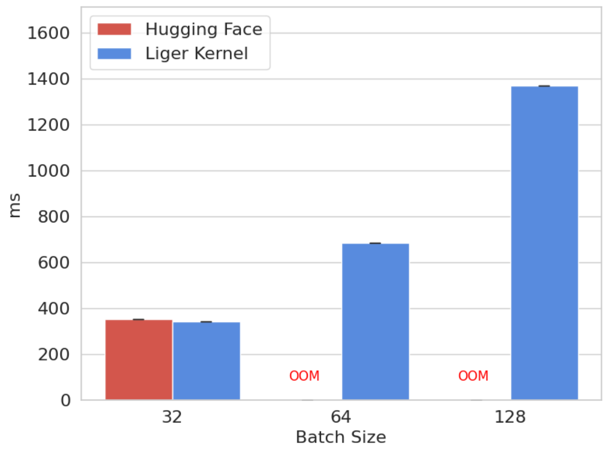
Observe that the logits corresponding to both $\boldsymbol{o}_w, \boldsymbol{o}_l$ are required for computing $\mathcal{L}_{OR}$ over a single sequence $\boldsymbol{x}$. In essence, the memory overheads posed by $\mathcal{L}_{OR}$ are relatively higher than $\mathcal{L}_{SFT}$ and hinder the user from scaling the batch size/sequence length during training.

To addresses these issues for a given batch of sequences, we create multiple *chunked* batches with pairs of preferred and dis-preferred (rejected) responses to calculate $\mathcal{L}_{OR}, \mathcal{L}_{SFT}$ and accumulate the gradients of the materialized logits 'in-place' (see Figure 9). Furthermore, the softmax operation is computed in an online fashion, resulting in significant memory savings (similar to FLCE).

Remark on compiled variants. As discussed in Section 2.1, employing `torch.compile` to capture and optimize the IR computational graph during model training is an approach that is being gradually embraced by the community. However, we emphasize that the compiler optimizations are currently not rich enough to automatically implement *input-chunking* based forward pass operations and in-place gradient computations by themselves. On the other hand, one can implement torch native *input-chunking* operations, and defer the atomic operations on each of the chunk to the compiler for obtaining an optimized triton kernel. We leverage a similar approach to develop our *Odds Ratio Loss* kernel and improve it further using online softmax computations.



(a) Peak allocated memory



(b) Throughput

Figure 10. Comparison of peak allocated memory and throughput for ORPO loss computation.

Remark on scaling gradients with FLCE variants. We additionally scale the gradients of the chunked inputs and the projection layer weights with the ratio of $\frac{\text{chunk size}}{B \times T}$. Formally, when a mean reduction is employed during the CE loss calculation, the gradients are calculated for a particular input chunk and are not normalized over the entire input sequence. This additional scaling factor addresses such approximation issues.

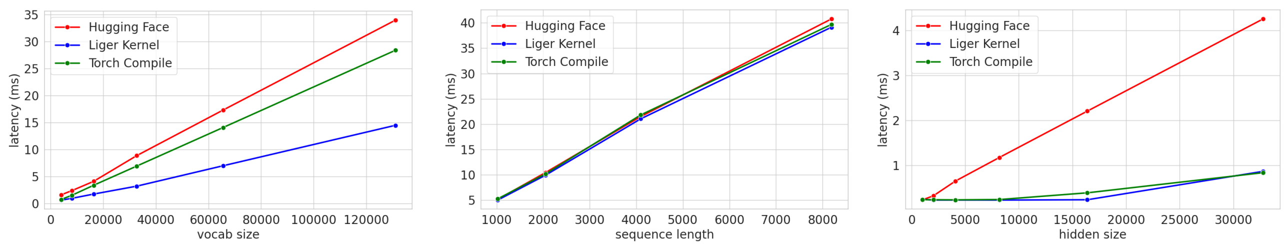
Numerical Experiments. Recall from equation 21 that $\mathcal{L}_{ORPO}$ is the weighted sum of $\mathcal{L}_{SFT}$ and $\mathcal{L}_{OR}$ and the chunking operation is performed on the batch of pairs of preferred and rejected responses. We consider a setting with vocab size 128256 and final hidden states with a sequence length 1024 and dimension 768. By varying the batch size from $\{32, 64, 128\}$, we plot the peak memory usage and the throughput (number of tokens processed/sec) in Figure 10. At a batch size of 32, the default HF implementation requires a peak memory of 76 GB, whereas the Liger variant requires just 5 GB, i.e a reduction factor of $\approx 15\times$. Furthermore, since the Liger variant is limited by the peak memory required for a chunked-batch, our approach can be scaled to batch sizes of 128 with just 6.1 GB peak memory. Additionally, the throughput scales linearly with batch size as the chunked-batch operations pose minimal latency overheads.

A.9. Kernel Benchmark

Setup. All benchmarks are run on a single NVIDIA H100 GPU (80 GB) with library versions `torch==2.4.0+cu118`, `triton==3.0.0`, `transformers==4.51.3`. The CrossEntropy kernel is benchmarked on vocab sizes in the set $\{40960, 81920, 122880, 163840\}$. The GeGLU and SwiGLU kernels are benchmarked on varying sequence lengths, whereas the RMSNorm and RoPE kernels are benchmarked on varying hidden dimensions. The sequence lengths and hidden dimension sizes are chosen from $\{4096, 8192, 12288, 16384\}$. All benchmarks are repeated 10 times to plot the median speed and memory along with $[0.2, 0.8]$ quantile values as the lower and upper bounds.

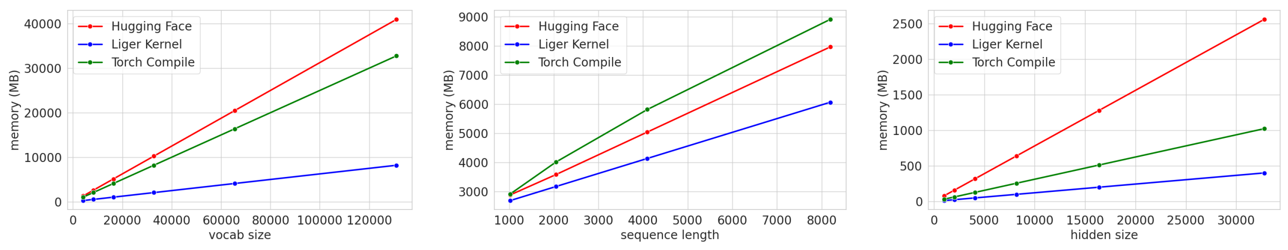
Results. The kernel speed and memory benchmarks are illustrated in Figure 11, 12 respectively. Observe that all the `Liger-Kernel` implementations either execute faster, consume less memory or provide both of these benefits when compared to the baseline implementations. In the case of the CrossEntropy kernel, the online softmax computation along with in-place replacement of the kernel inputs with their gradients leads to approximately $3\times$ faster execution (Figure 11a) and consumes approximately $5\times$ less memory (Figure 12a) for a vocab size of 163840. For GeGLU, we maintain parity with the baseline in terms of speed (Figure 3a) and reduce the peak memory consumption by roughly $1.6\times$ (when sequence length is 16384) by recomputing the SiLU$(\cdot)$ and GELU$(\cdot)$ outputs during the backward pass (Figure 4a).

The RMSNorm implementation fuses the normalization and scaling operations into a single triton kernel and caches the root mean square values for usage in the backward pass. This avoids repetitive data transfers and floating point operations with minimal memory overheads. Figure 11c illustrates approximately $7\times$ reduction in execution time and roughly $3\times$ reduction in peak memory consumption for a hidden dimension of 16384 respectively. Finally, for the RoPE kernel, we employ a



(a) CrossEntropy (b) SwiGLU (c) RMSNorm

Figure 11. Kernel execution speed benchmarks.



(a) CrossEntropy (b) SwiGLU (c) RMSNorm

Figure 12. Kernel peak allocated memory benchmarks.

flattened 1D tensor to represent the rotation matrix and leverage the repeated blocks in $\boldsymbol{R}^d_{\Theta,m}$ to significantly reduce the growth in latency with an increase in hidden dimension size. In particular, we achieve approximately $8\times$ speedup with approximately $3\times$ lower memory consumption for a hidden size of 16384.

B. Testing Best Practices

B.1. Correctness

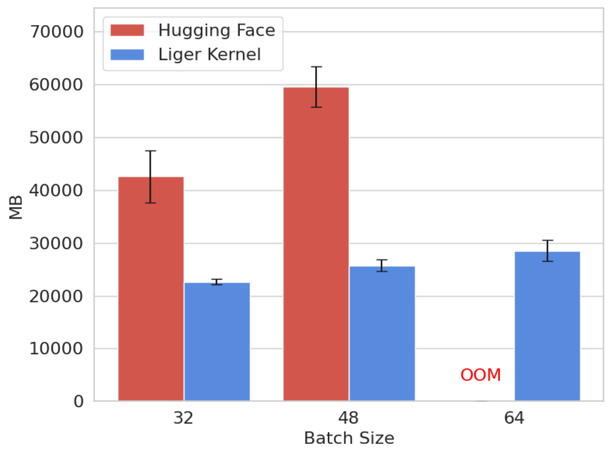
Ensuring kernel precision is crucial, as any deviation from the original implementation could impact model convergence or cause critical errors. To achieve this, we prepare a pure PyTorch implementation (e.g., one provided by HuggingFace) for comparison and test the implementation with various input shapes and data types. We include regular shapes (e.g., powers of 2) and test irregular shapes to ensure proper handling of edge cases. We set appropriate absolute and relative tolerance levels: for fp32, use atol = 10^{-7} and rtol = 10^{-5}; for bf16, use atol = 10^{-3} and rtol = 10^{-2} [18].

Furthermore, large tensor dimensions can lead to inadvertent memory access issues. By default, the `program_id` in the kernels are stored as `int32`. If `program_id * Y_stride > 2,147,483,647`, the value becomes negative, resulting in illegal memory access. Such overflows and incorrect memory addressing errors can be avoided by converting it to `int64` when dealing with large dimensions.

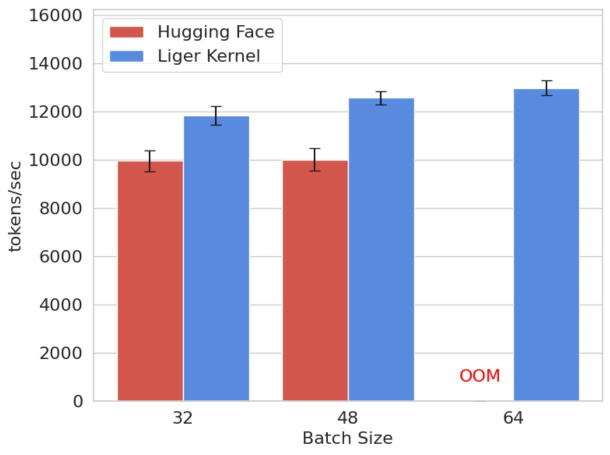
B.2. Contiguity

Since Triton operates directly on physical memory, non-contiguous tensors (where elements are not arranged sequentially) can lead to illegal memory access or incorrect outputs. For example, when deploying our RoPE kernel for production training, we encountered loss divergence issues because the derivative from the `scaled_dot_product_attention` function was not stored contiguously. To prevent such issues, it's best practice to ensure contiguity before passing tensors to the kernel.

[18]Note that in practice, the tolerance may need further relaxation by one or two orders of magnitude, even for exact kernels. We use convergence tests to ensure exactness in cases where the tolerance for correctness needs to be loose.

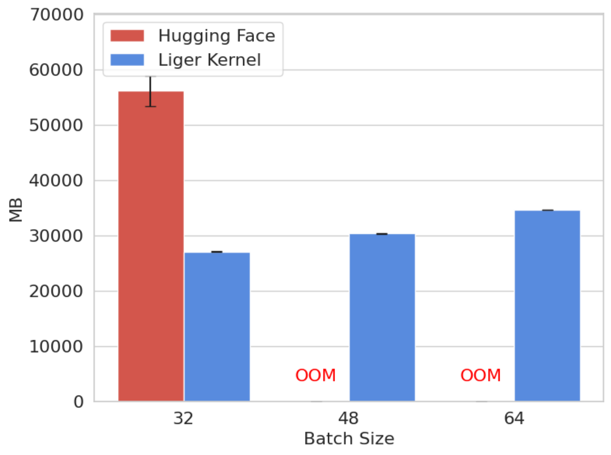


(a) Peak allocated memory

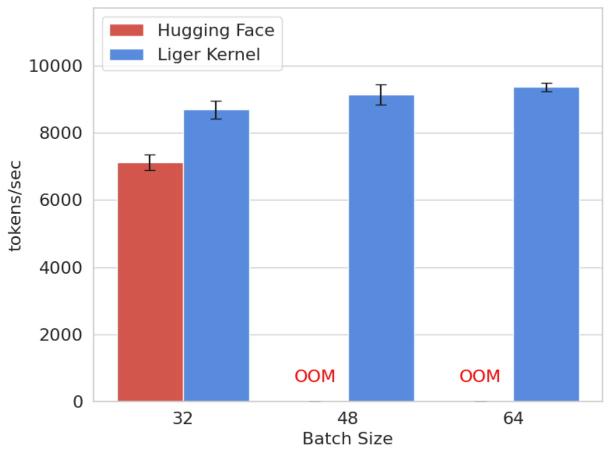


(b) Throughput

Figure 13. Comparison of peak allocated memory and training throughput (tokens/sec) for Qwen2.



(a) Peak allocated memory



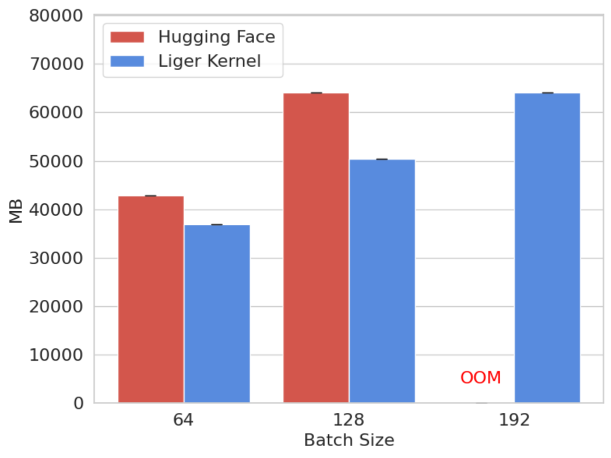
(b) Throughput

Figure 14. Comparison of peak allocated memory and training throughput (tokens/sec) for Gemma 7b.

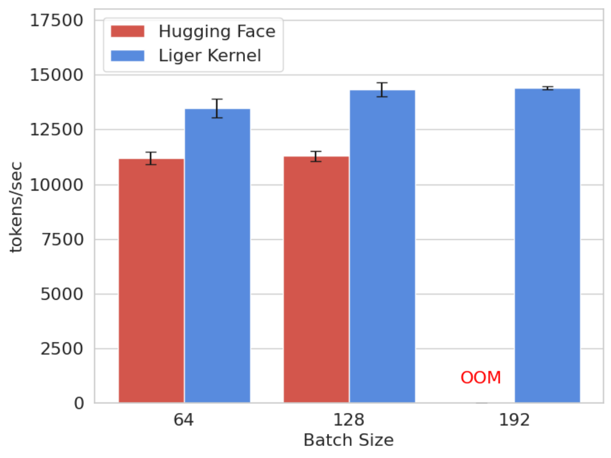
C. Usecase Benchmark

Performance Comparison. At a batch size of 64, LLaMA 3-8B demonstrates a **42.8% increase in throughput**, coupled with a **54.8% reduction in GPU memory usage** (Figure 5). This enables training on smaller GPUs or using larger batch sizes and longer sequence lengths with lower resource consumption. Similarly, at a batch size of 48 our kernels improve the throughput of Qwen2 by **25.5%**, while achieving a **56.8% reduction in GPU memory usage** (Figure 13). For Gemma, throughput improves by **11.9%** with a **51.8% reduction in memory usage** at a batch size of 48 (Figure 14). Mistral, at a batch size of 128, exhibits a **27% increase in throughput**, with a **21% drop in GPU memory usage** (Figure 15). Finally, Phi3, at a batch size of 128, shows a **17% increase in throughput**, while reducing memory usage by **13%** (Figure 16). Overall, the results highlight several notable use cases. LLaMA 3-8B's exceptional improvements make it ideal for resource-constrained environments where GPU memory is a bottleneck. Additionally, Qwen2's strong memory reductions position it well for tasks involving large datasets or extended training durations. Mistral's high throughput gains make it advantageous for workloads requiring large batch sizes.

Medusa. Recall that Medusa training has two flavors. The first approach involves training only the additional Medusa heads while keeping the backbone LLM frozen. The second approach tunes both the backbone and the LLM heads

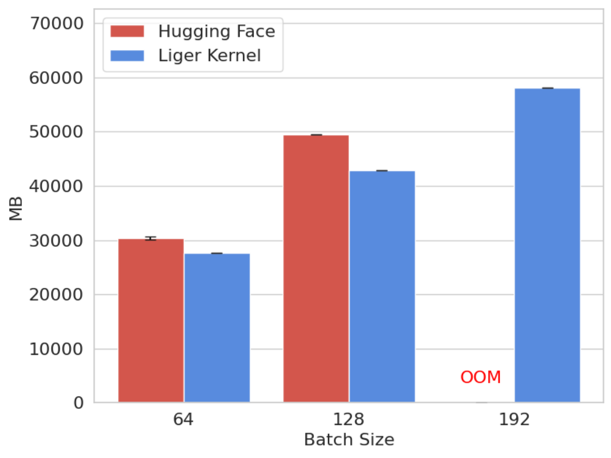


(a) Peak allocated memory

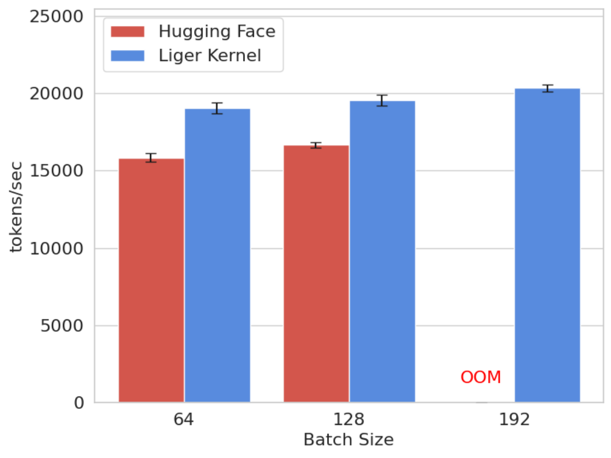


(b) Throughput

Figure 15. Comparison of peak allocated memory and training throughput (tokens/sec) for Mistral 7b.

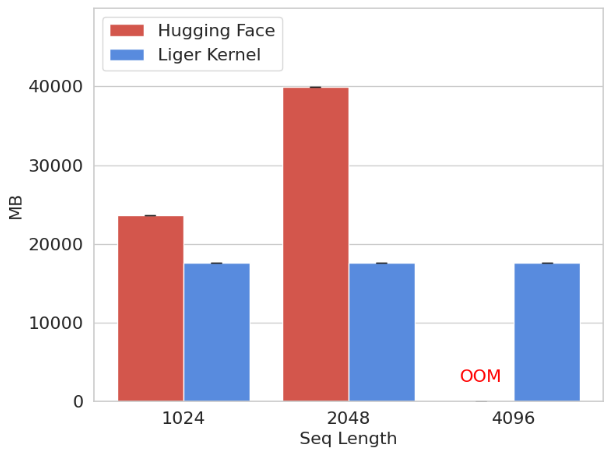


(a) Peak allocated memory

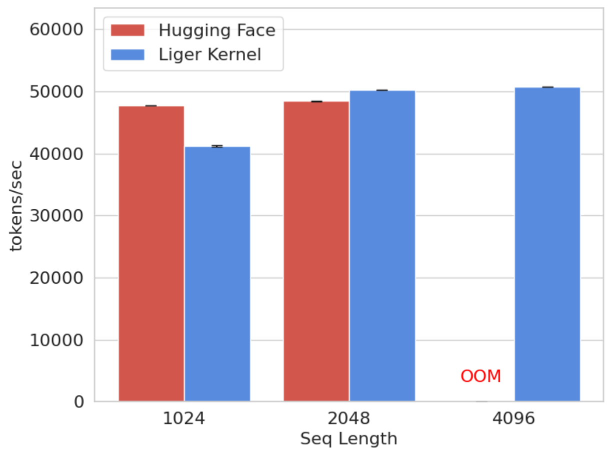


(b) Throughput

Figure 16. Comparison of peak allocated memory and training throughput (tokens/sec) for Phi3.

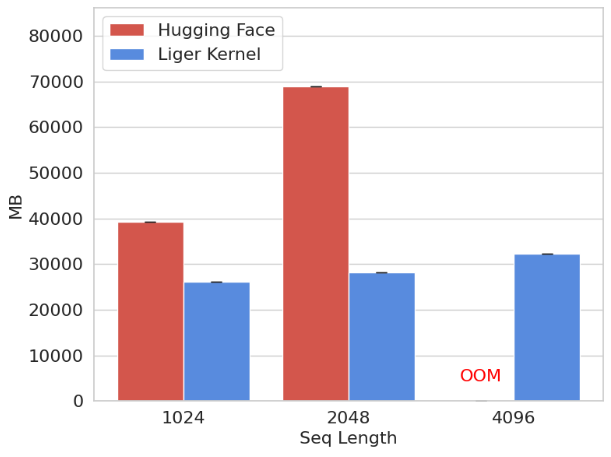


(a) Peak allocated memory

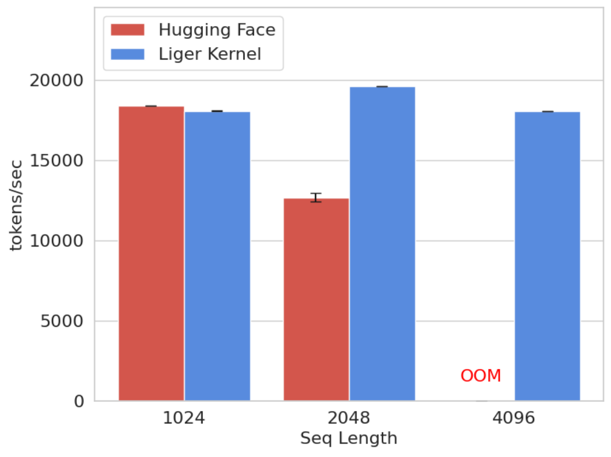


(b) Throughput

Figure 17. Comparison of peak allocated memory and throughput for Stage 1 with 3 Medusa heads.



(a) Peak allocated memory



(b) Throughput

Figure 18. Comparison of peak allocated memory and throughput for Stage 2 with 5 Medusa heads.

simultaneously. We have benchmarked both cases, and the Liger kernel has demonstrated reduced memory usage and improved throughput. Without it, experiments are highly prone to out of memory issues. In Figures 17-18, the standard errors measured from repetitive runs are $< 1\%$.

Note: Our work focuses solely on improving the computational performance. Generating LM heads that can accelerate inference for the LLaMA3-8B model is not within the scope of this paper. Such work requires extra work for training data selection, hyperparameter tuning, and warmup techniques to ensure proper model convergence. Our experiments utilize 8 NVIDIA A100 GPUs (80 GB each) to train the LLaMA 3-8B model with a variable sequence length, a batch size of 4, bfloat16 precision and AdamW optimizer.