T1M3: Practical Output-Layer Memory Mitigation via Chunked Cross-Entropy
A PyTorch Microbenchmark for Long-Sequence Training

Lai Hu

December 26, 2025

Abstract

For long-sequence transformer training, the *output layer* (the final projection to vocabulary logits plus cross-entropy loss) often becomes a dominant GPU memory spike, because naïvely materializing logits has shape (B, S, V). We report T1M3, a controlled microbenchmark in PyTorch that compares a `full` output-layer computation against a `chunked` variant that computes the cross-entropy in smaller sequence blocks, reducing peak activation pressure. On a single RTX 3080-class GPU with BF16, sequence length $S = 1024$ and batch size $B = 2$, `chunked` reduces peak GPU reserved memory from 13.83 GiB to 10.70 GiB (∼22.6 %), at the cost of a modest wall-time increase from 1282.5 ms to 1370.3 ms (∼6.8 %). These results complement T1M1 (Slideformers-style weight streaming) by addressing a remaining bottleneck: even when weights are streamed/offloaded, the output-layer logits can still exceed VRAM for large S or large vocabulary V.

1 Introduction

Transformer training on commodity GPUs is constrained by memory capacity, not only for parameters and optimizer state, but also for activation tensors produced during forward/backward. While T1M1 demonstrated that Slideformers-inspired *weight streaming* can dramatically reduce *parameter-side* GPU memory, many practical runs still fail at longer sequences because the *output layer* briefly needs large tensors: the final linear projection produces logits over the vocabulary, and a naive cross-entropy often materializes (B, S, V) logits (and sometimes additional softmax buffers) before reduction.

T1M3 targets a simple but high-impact mitigation: compute output-layer loss in *chunks* to avoid keeping all logits alive at once, thereby reducing peak GPU memory at the cost of extra kernel launches and reduced fusion/parallelism.

Contributions.

- A minimal PyTorch benchmark that isolates output-layer memory behavior at long sequence length.
- A `chunked` cross-entropy implementation that reduces peak GPU memory by ∼3.13 GiB (reserved) and 3.18 GiB (allocated) in our setting.
- Paper-ready figures for (i) GPU memory timeline, (ii) peak memory summary, and (iii) step-time breakdown.

- A system-level interpretation for how this mitigation composes with Slideformers-style streaming pipelines.

2 Background

2.1 Why the output layer is memory-heavy

For causal language modeling, the final projection maps hidden states $H \in \mathbb{R}^{B\times S\times d}$ to logits $Z \in \mathbb{R}^{B\times S\times V}$ via $Z = HW^\top$ where $W \in \mathbb{R}^{V\times d}$. Even in BF16, storing Z costs $2 \cdot B \cdot S \cdot V$ bytes, which can be several gigabytes for large vocabularies and long sequences. Moreover, backward pass may require additional intermediates (e.g., softmax outputs, gradients), and the CUDA caching allocator may increase *reserved* memory above *allocated* memory.

2.2 Chunking vs. kernel fusion

A fully fused CUDA kernel can compute cross-entropy and gradients without storing logits explicitly (e.g., via online softmax), but implementing such fusion requires custom CUDA/Triton code. Chunking is a practical, framework-level compromise: it reduces peak live tensor size by slicing along sequence (or vocab) dimension, while still using standard GEMM and loss primitives provided by PyTorch/CUDA.

3 Method

3.1 Compared variants

We compare two output-layer variants under the same base transformer forward/backward:

1. `full`: compute full logits for all tokens, then cross-entropy over the entire sequence.
2. `chunked`: split the sequence dimension into blocks; for each block, compute logits and cross-entropy, accumulate loss (and gradients) incrementally, and discard per-block logits immediately.

3.2 Instrumentation

We record per-step: (i) wall time with synchronization at step boundaries, (ii) time breakdown into base-forward, head-forward, backward, optimizer, (iii) GPU peak memory (allocated/reserved) and memory timeline samples.

4 Experimental setup

Configuration. Sequence length $S = 1024$, batch size $B = 2$, dtype BF16, no activation checkpointing (`ckpt`=0). We report steady-state means, skipping warm-up where applicable.

5 Results

5.1 Summary

Table 1 reports the key trade-off. The `chunked` output layer reduces both allocated and reserved peak GPU memory by about 3 GiB, while increasing wall-time per step by about 7 %.

Table 1: T1M3 summary (time in ms; memory in GiB).

Variant	Wall	Peak alloc	Peak reserved	Head fwd / Bwd (ms)
`full`	1282.5	13.08	13.83	61.5 / 872.7
`chunked`	1370.3	9.90	10.70	88.2 / 926.2

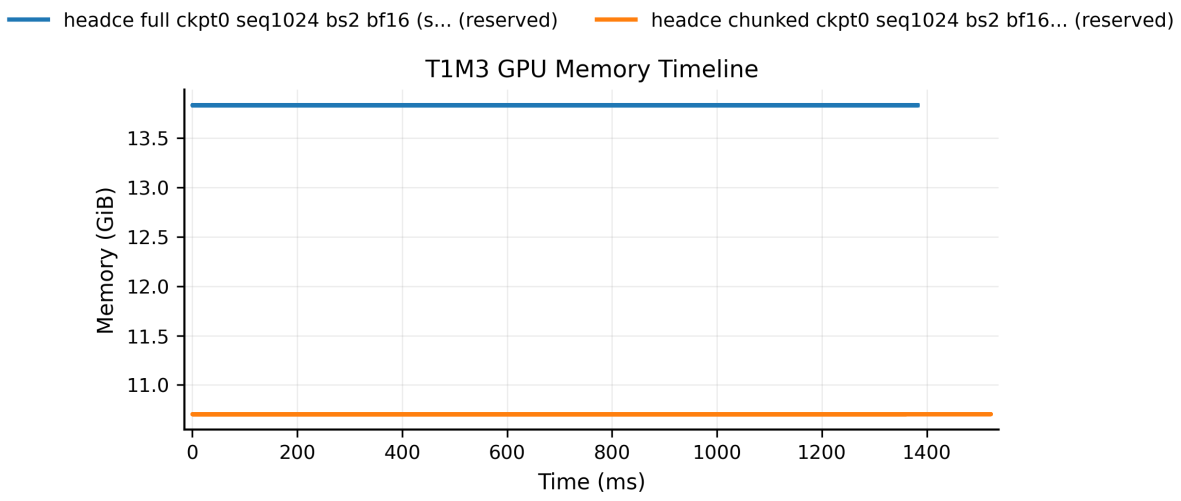

Figure 1: T1M3 GPU reserved memory timeline: `full 13.83G` vs. `chunked 10.70G`.

5.2 Memory timeline

Figure 1 shows the GPU reserved memory timeline within a step. The `chunked` curve is consistently lower, indicating reduced peak pressure from output logits.

5.3 Peak memory chart

5.4 Step time breakdown

6 Discussion

What T1M3 demonstrates. A practical output-layer mitigation (sequence-chunked cross-entropy) reduces peak GPU reserved memory by $\sim$22.6 % without requiring any custom CUDA kernels. This addresses an important bottleneck that remains even after parameter offloading: the (B, S, V) logits can dominate VRAM at long context lengths.

Where the overhead comes from. Compared to `full`, `chunked` increases head-forward and backward time because it executes more head/loss calls and loses some fusion opportunities. The base-forward portion remains nearly unchanged, confirming that the slowdown is localized to the output head.

Next step: fused CUDA head. Chunking is an effective baseline. A next optimization is to implement a fused CUDA/Triton head that avoids explicit logits materialization (online softmax + fused backward), reducing both memory and time further—a natural continuation for M3.

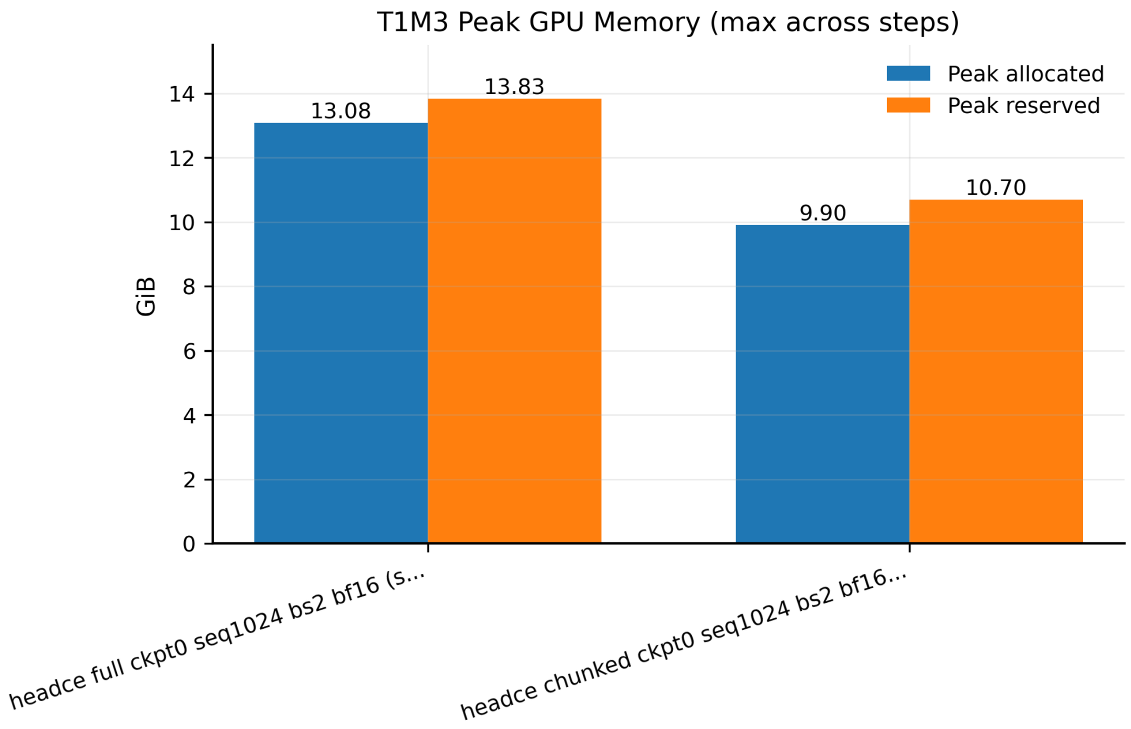

Figure 2: Peak GPU memory across steps (allocated vs. reserved).

7 Conclusion

T1M3 provides evidence that output-layer chunking is an effective, low-complexity technique to mitigate GPU memory spikes from (B, S, V) logits at long sequence lengths, and it composes naturally with Slideformers-style streaming.

References

[1] Adam Paszke and others. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[2] NVIDIA Corporation. CUDA C++ Programming Guide, NVIDIA Developer Documentation.

[3] Zeyi Wen and collaborators. Slideformers (weight streaming / sliding training), original work by the Zeyi Team.

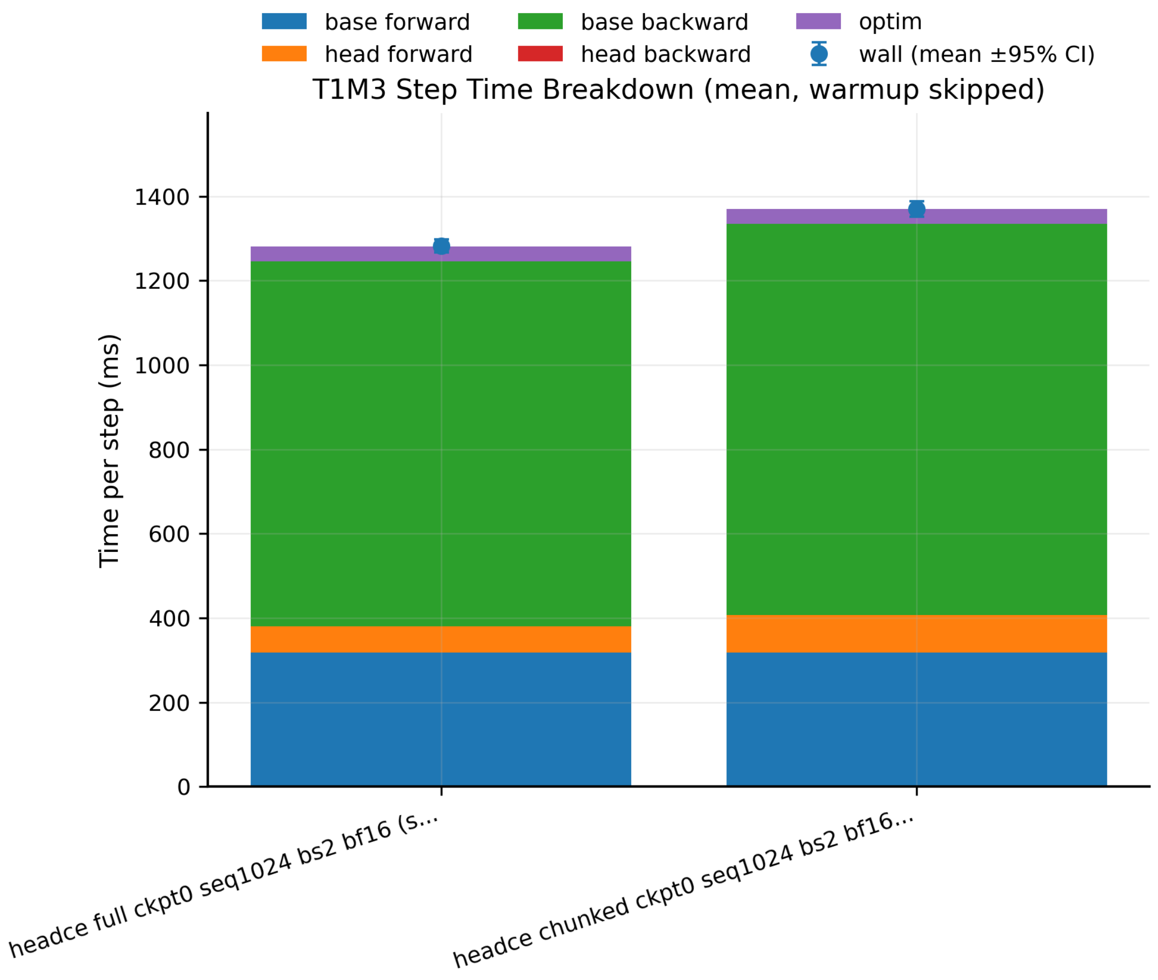

Figure 3: Step time breakdown (mean, warm-up skipped).