# T1M1: Slide-Style Weight Streaming for Memory-Efficient Transformer Training
# A PyTorch Microbenchmark with Per-Layer Timing and Memory Traces

Lai Hu

December 25, 2025

**Abstract**

GPU memory is a first-order bottleneck for training transformer models, especially under constrained VRAM. We report T1M1, a reproducible microbenchmark in PyTorch that compares three training modes on the same transformer stack: (i) standard GPU training (`gpu`), (ii) gradient checkpointing / activation recomputation (`ckpt`), and (iii) a Slideformers-inspired "sliding" strategy that streams layer weights between CPU and GPU with prefetching and CPU-side optimizer updates (`slide`). Using identical batch/sequence settings, `slide` reduces peak GPU reserved memory from 6.31 GiB to 0.93 GiB ($\sim 6.8\times$) at the cost of higher iteration time (about $4.2\times$ versus `gpu`), shifting the memory footprint to CPU. We further present per-layer forward/backward/recompute/optimizer timing to localize overhead sources and discuss where overlap and kernel/optimizer fusion can close the gap.

## 1 Introduction

Training large transformers is often limited not by compute throughput but by memory capacity and memory movement. A common mitigation is gradient checkpointing, trading recomputation for lower activation memory. More recently, "sliding" or streaming approaches (e.g., Slideformers) propose keeping most parameters off-GPU and moving only the working set to the accelerator, potentially enabling larger models or longer sequences on commodity GPUs.

This report focuses on a system-level question: *what are the concrete time/memory trade-offs among standard GPU training, checkpointing-based recomputation, and a Slideformers-style weight streaming pipeline implemented in PyTorch?* We answer this with a controlled microbenchmark (T1M1), providing both summary metrics and per-layer breakdowns.

**Contributions.**

- A PyTorch microbenchmark comparing `gpu`, `ckpt`, and `slide` training modes under the same model and input regime.
- Paper-ready plots for (i) iteration time breakdown, (ii) peak memory, (iii) intra-iteration memory timeline, and (iv) per-layer time components.
- A concise diagnosis of where `slide` pays overhead (CPU optimizer/update, host-device transfers, and scheduling) and what optimizations are likely to matter next.

# 2 Background

## 2.1 PyTorch execution model

PyTorch provides an imperative programming model with asynchronous GPU execution; accurate wall-time accounting typically requires synchronization at iteration boundaries [1]. This matters when interpreting component timers that can partially overlap (e.g., CPU update vs. GPU kernels).

## 2.2 Checkpointing vs. sliding

Activation checkpointing saves memory by discarding intermediate activations and recomputing them in backward pass. In contrast, Slideformers-style approaches aim to keep model weights (and sometimes optimizer state) largely on CPU, streaming per-layer weights to GPU just-in-time (often with prefetch), and optionally overlapping transfer with compute. **TODO:** cite the exact Slideformers paper once arXiv ID/BibTeX is provided [2].

# 3 Method

## 3.1 Compared modes

We evaluate three modes:

1. `gpu`: Standard training with model parameters and optimizer state on GPU.
2. `ckpt`: Gradient checkpointing / recomputation enabled (baseline for memory-vs-compute trade-off).
3. `slide`: Slideformers-inspired weight streaming:
   - Weights reside on CPU; each layer is transferred to GPU before use.
   - A prefetch mechanism moves weights for upcoming layers ahead of time.
   - Optimizer/update is performed on CPU, increasing CPU time and CPU RSS.

## 3.2 Instrumentation and metrics

We record: (i) iteration wall time (with boundary synchronization), (ii) component times (forward, backward, optimizer/update, and residual overhead), (iii) peak GPU memory (allocated/reserved) and peak CPU RSS, and (iv) per-layer means for forward/backward/recompute/optimizer (where applicable).

# 4 Experimental setup

**Model and inputs.** We use a transformer causal LM stack with $L = 28$ layers (as reflected by the per-layer plots). Inputs are fixed-length token sequences (`seq` $= 256$) with micro-batch size `bs` $= 1$, and BF16 where supported.

**Protocol.** We run 10 iterations and report statistics over the last 9 iterations (excluding the first warm-up iteration).

Table 1: T1M1 summary (mean ± std over 9 steady-state iterations; warm-up excluded).

| Mode | Wall time (s) | Peak GPU reserved (GiB) | Peak CPU RSS (GiB) |
|---|---|---|---|
| `gpu` | 0.375 ± 0.007 | 6.31 | 1.28 |
| `ckpt` | 0.479 ± 0.039 | 6.00 | 1.28 |
| `slide` | 1.571 ± 0.012 | 0.93 | 4.93 |

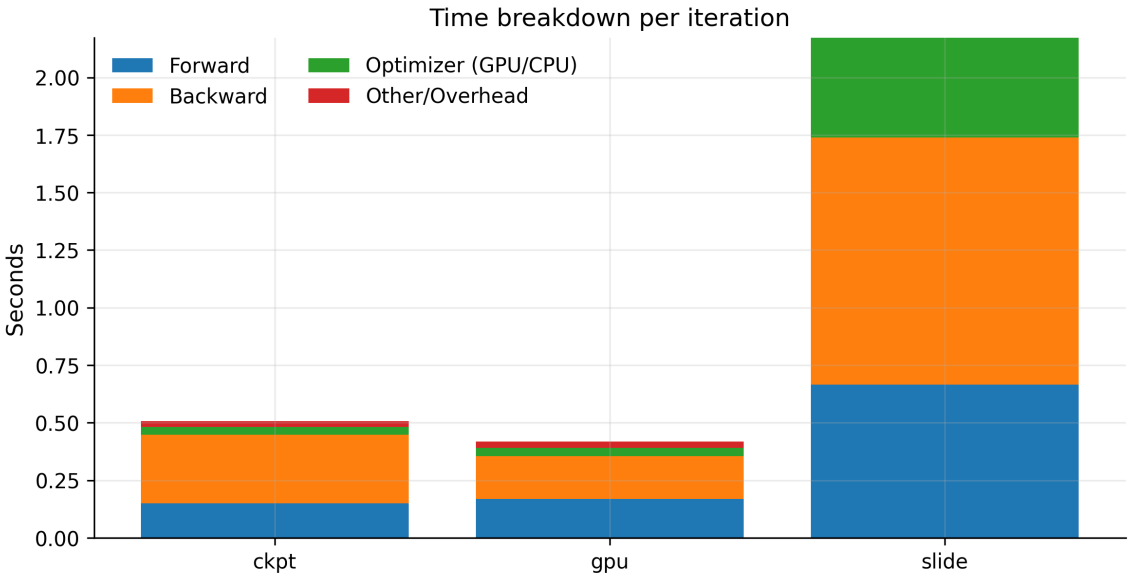


Figure 1: Time breakdown per iteration for `gpu`, `ckpt`, and `slide`.

**Hardware/software.** Single NVIDIA GPU (RTX 3080-class, 16GB VRAM) and PyTorch for training/runtime [1]. (You can paste exact driver/CUDA/PyTorch versions here if you want the report fully self-contained.)

# 5 Results

## 5.1 End-to-end time and peak memory

Table 1 summarizes the main trade-off. The `slide` mode reduces peak GPU reserved memory from 6.31 GiB to 0.93 GiB ($\sim 6.8\times$), but iteration time increases from 0.375 s to 1.571 s on average. Checkpointing (`ckpt`) increases runtime relative to `gpu` while providing negligible peak GPU memory reduction in this configuration.

## 5.2 Iteration time breakdown

Figure 1 shows the per-iteration time decomposition. A key observation is that `slide` shifts significant time into optimizer/update and transfer-related overhead, consistent with the design choice of CPU-side optimizer state and streamed weights.

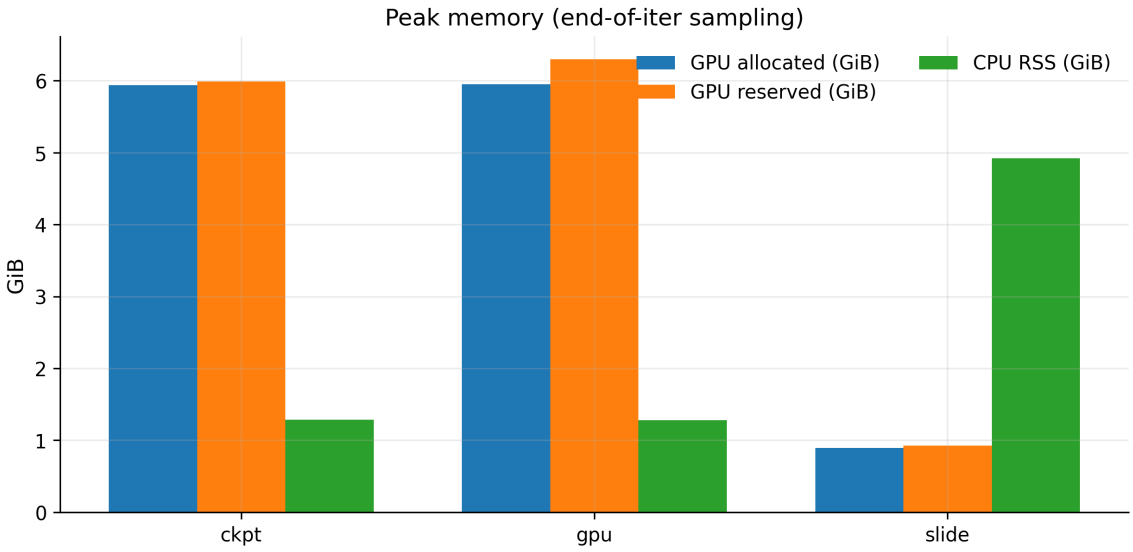


Figure 2: Peak memory (end-of-iteration sampling): GPU allocated/reserved and CPU RSS.

### 5.3 Peak memory and memory timeline

Figure 2 confirms that `slide` achieves a large reduction in both allocated and reserved GPU memory, while increasing CPU RSS by $\sim 3.9\times$ due to CPU-resident weights and optimizer state. Figure 3 shows that `slide` keeps GPU reserved memory nearly flat across the iteration progress, whereas `gpu`/`ckpt` exhibit higher GPU reservation during the iteration.

### 5.4 Per-layer timing: forward, backward/recompute, optimizer/update

Per-layer timing localizes overhead: (i) Figure 4 shows forward time trends across layers, (ii) Figure 5 highlights backward and recomputation behavior (notably spikes in `ckpt` recompute), and (iii) Figure 6 shows `slide` incurs a much larger optimizer/update time per layer, consistent with CPU-based updates and parameter movement.

## 6 Discussion

**What T1M1 supports (T1M1 conclusion).** Under this single-GPU microbenchmark, Slideformers-style weight streaming (`slide`) provides a $\sim 6.8\times$ reduction in peak GPU reserved memory (6.31 GiB $\rightarrow$ 0.93 GiB), enabling significantly larger memory headroom at the expense of iteration throughput. Checkpointing (`ckpt`) increases runtime and shows recomputation hotspots, but does not materially reduce peak GPU memory in this configuration.

**Why `slide` is slower here.** The per-layer optimizer/update curve indicates the dominant cost is CPU-side updates and/or scheduling/transfer overhead. This suggests that practical performance hinges on (i) overlapping transfers with compute, (ii) reducing CPU optimizer overhead (fused kernels, lower precision states), and (iii) careful stream/event design to minimize synchronization.

**Limitations.** We report short-run microbenchmark statistics (10 iterations, small batch/sequence) and focus on system metrics, not final model quality. Absolute timing may change with larger

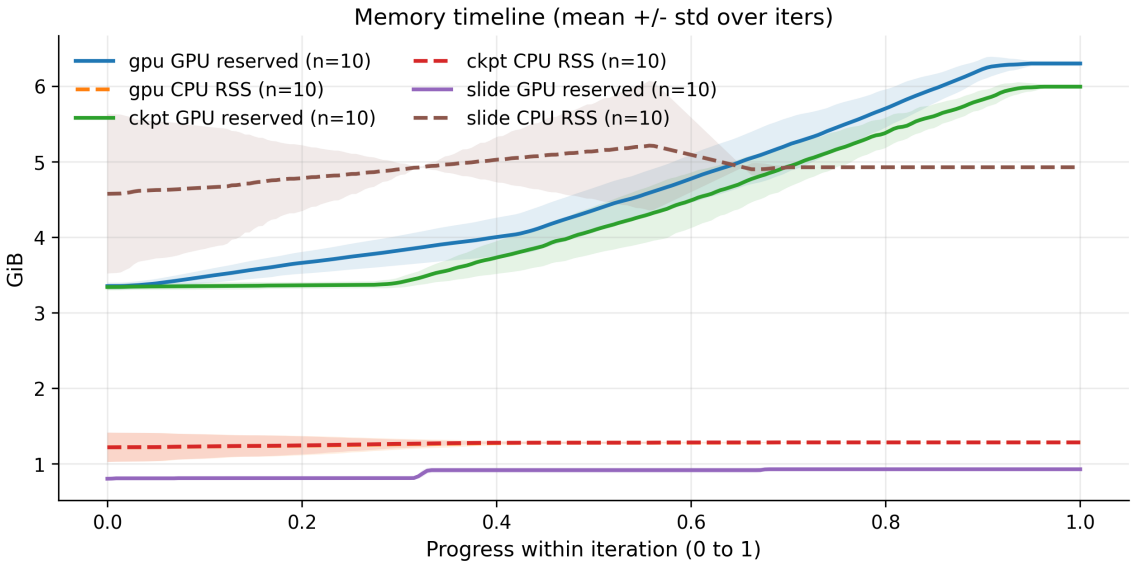


Figure 3: Memory timeline (mean ± std over iterations) as a function of normalized progress within iteration.

batches, longer sequences, different optimizers, and stronger overlap implementation.

## 7 Conclusion

T1M1 demonstrates that a Slideformers-inspired training pipeline implemented in PyTorch can dramatically reduce GPU memory usage on a single commodity GPU, but introduces substantial runtime overhead dominated by CPU update and movement/scheduling costs. Future work should prioritize overlap, optimizer fusion, and a broader sweep over sequence length and model scale to map the full Pareto frontier.

## Reproducibility checklist (fill-in)

- GPU model / driver / CUDA: 12.8 with RTX3080/16G
- PyTorch version: 2.7
- Github Source / scripts: https://github.com/LaiHOLA/SideFormer-ReworkOfSlideformers

## 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] **Zeyi et al..** Slideformers.

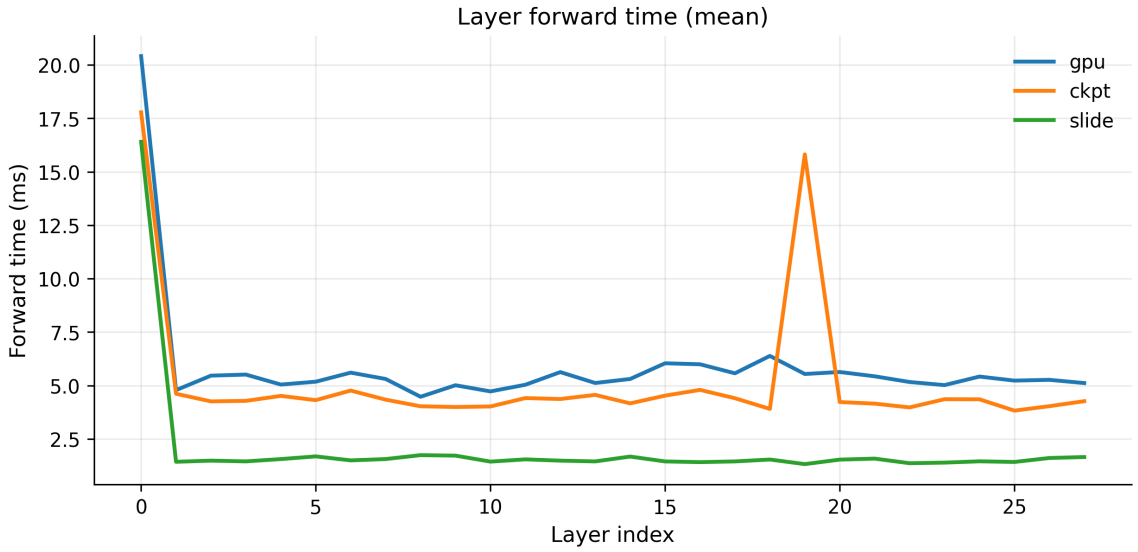


Figure 4: Mean per-layer forward time for the three modes.

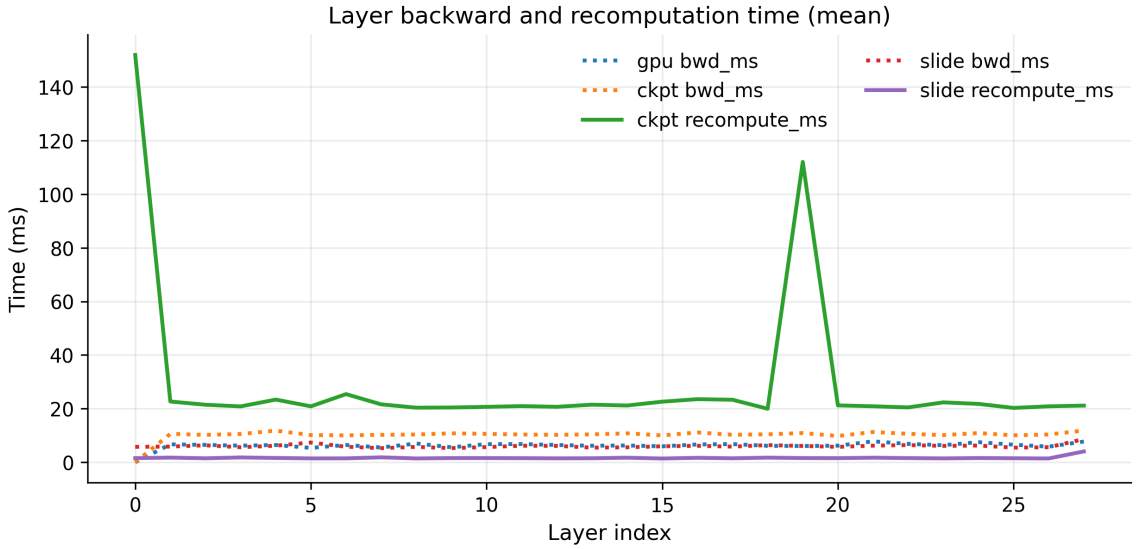


Figure 5: Mean per-layer backward and recomputation time. The `ckpt` mode exhibits noticeable recomputation spikes on specific layers.

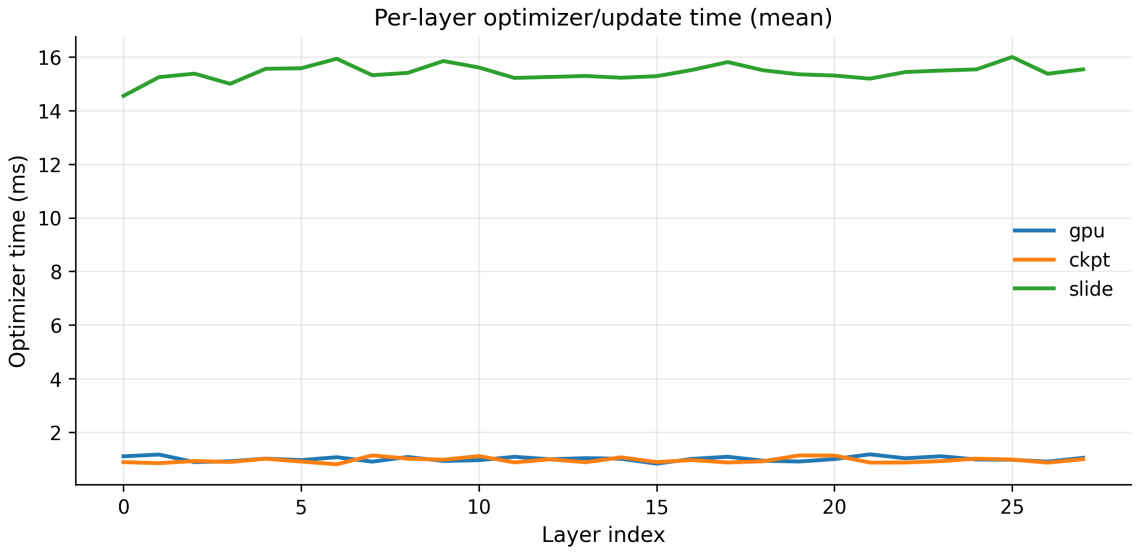


Figure 6: Mean per-layer optimizer/update time. `slide` is dominated by CPU update overhead compared with `gpu`/`ckpt`.