T1M2: Recomputation Ablation for Memory–Throughput Trade-offs
Per-layer Timing and Memory Traces on a Qwen Decoder Stack

LAI HU

December 25, 2025

Abstract

Recomputation (a.k.a. activation checkpointing / gradient checkpointing) is a widely used technique to reduce GPU memory footprint by trading extra computation in backward for fewer saved activations. In this short report (T1M2), we run a controlled recompute ablation on a 28-layer Qwen-style decoder stack under an identical microbenchmark regime (`seq`=256, `batch`=1, BF16). We instrument step-level wall time, per-layer forward/backward/total time with confidence intervals, and CPU/GPU memory traces (CPU RSS, GPU allocated/reserved). On an RTX 3080-class GPU, enabling recomputation reduces peak GPU reserved memory from 9.28 GiB to 8.32 GiB (-10.4%), and peak GPU allocated from 7.94 GiB to 7.66 GiB. This memory reduction comes with a step-time increase from 336.7 ms to 475.8 ms ($\times 1.41$), and a modest CPU RSS increase ($2.33 \to 2.70$ GiB). Per-layer deltas reveal recompute overhead is not uniform; a small subset of mid-stack layers exhibits the largest total-time increases.

1 Introduction

Training and fine-tuning transformer LMs is frequently constrained by GPU memory capacity rather than pure compute throughput. To mitigate this, activation checkpointing (recomputation) discards selected intermediate activations in forward and recomputes them during backward, reducing memory at the cost of extra compute [1]. In heterogeneous training systems such as SlideFormer, recomputation interacts with other memory-movement decisions (CPU/GPU weight placement, overlap, prefetch), so it is useful to quantify the standalone recompute trade-off in a controlled setting [2].

T1M2 is the recompute-only ablation phase in our Sideformers replication effort [3]. The goal is not to optimize absolute throughput but to produce paper-ready evidence (plots + CSVs) that clearly shows (i) the memory reduction achieved by recomputation, and (ii) where the extra time is paid (per-layer).

Contributions.

- A reproducible PyTorch microbenchmark for recomputation ON/OFF with step-level and per-layer instrumentation.
- Paper-ready figures: per-layer mean (with 95% CI) for forward/backward/total, per-layer total delta, step wall time curve and delta, and CPU/GPU memory timelines (allocated/reserved).
- A compact summary table with peak memory and throughput deltas, suitable for inclusion in a larger SlideFormer/Sideformers write-up.

2 Background

2.1 Recomputation and checkpointing

In checkpointing, forward activations for selected modules are not stored; instead, the module forward is rerun during backward to recover needed activations. This reduces activation memory but increases compute and can change kernel scheduling characteristics [1]. Huawei Ascend ecosystems commonly expose a similar concept ("recompute") at graph/module level to trade compute for memory, e.g. in MindSpore documentation and tooling [4].

2.2 PyTorch profiling and synchronization

PyTorch launches GPU work asynchronously, so step wall time should be measured with synchronization at iteration boundaries for accurate accounting [5]. We also optionally enable `torch.profiler` for operator traces, but the main evidence in this report comes from our own synchronized timers and memory sampling [6].

3 Method

3.1 Compared configurations

We compare two modes under identical inputs and optimizer settings:

1. **Recompute OFF**: standard forward/backward.
2. **Recompute ON**: gradient checkpointing enabled on the decoder stack (selective layer checkpointing is supported via configuration flags).

3.2 Instrumentation

At each training step, we record:

- **Step times**: wall time and coarse components (forward, backward, optimizer step), with CUDA synchronization for correctness.
- **Per-layer times**: for each decoder layer $l \in \{0, \dots, 27\}$, mean forward/backward/total time and 95% confidence intervals over steady-state steps.
- **Memory**: CPU RSS via `psutil`; GPU allocated/reserved via `torch.cuda.memory_allocated` and `torch.cuda.memory_reserved`.

We exclude the first step as warm-up in per-layer statistics and summary means.

4 Experimental setup

Model and inputs. We use a Qwen-style decoder-only model with $L = 28$ decoder layers (detected programmatically). Inputs are fixed-length token sequences with `seq`=256 and micro-batch size `bs`=1, using BF16 on GPU.

Hardware/software. Single NVIDIA GPU (RTX 3080 Laptop GPU) with CUDA 12.8 and PyTorch 2.7.0+cu128. The run is performed on Windows. The codebase is part of our Sideformers replication repo [3].

Table 1: T1M2 summary (means exclude the first warm-up step).

Mode	Mean step wall (ms)	Peak GPU alloc (GiB)	Peak GPU resv (GiB)	Peak CPU RSS (Gi
Recompute OFF	336.7	7.94	9.28	2.33
Recompute ON	475.8	7.66	8.32	2.70

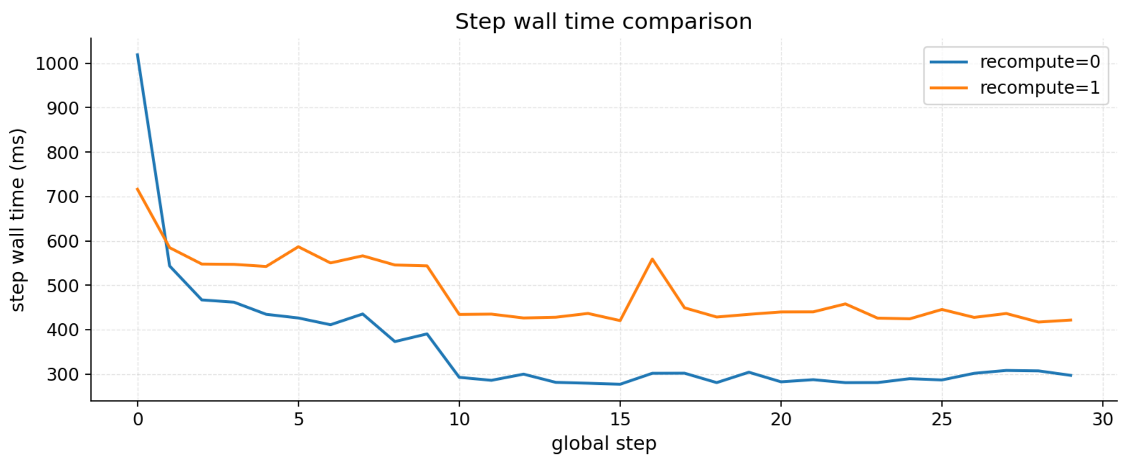

Figure 1: Step wall time comparison (recompute OFF vs ON).

Protocol. For each mode we run a single epoch consisting of 30 steps, and compute steady-state statistics over the last 29 steps (skip step 0). While the original plan targets multi-epoch distributions, step-level sampling already provides enough samples to estimate per-layer means and confidence intervals.

5 Results

5.1 Summary: throughput vs. peak memory

Table 1 summarizes the primary trade-off. Recomputation reduces peak GPU reserved memory by 0.96 GiB (10.4%), but increases mean step wall time by 139.1 ms (41.3%). Peak CPU RSS increases by 0.37 GiB, consistent with extra bookkeeping and different runtime behavior.

5.2 Step wall time curves

Figure 1 plots step wall time for both modes, and Figure 2 shows the per-step delta (ON−OFF). We observe a consistent slowdown under recomputation after warm-up, with occasional spikes attributable to runtime scheduling and cache effects.

5.3 Memory timeline

Recomputation reduces peak GPU memory but does not necessarily change the qualitative "flatness" of reserved memory under PyTorch's caching allocator. We report both allocated and reserved timelines, plus CPU RSS, in Figures 3 and 4. These traces are essential for reasoning about real VRAM headroom under allocator behavior.

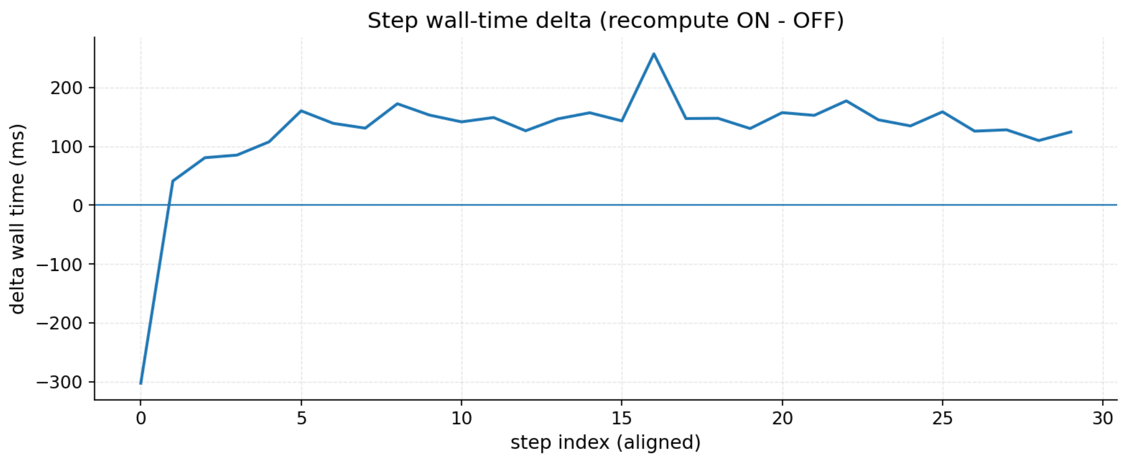

Figure 2: Per-step wall time delta (recompute ON − OFF).

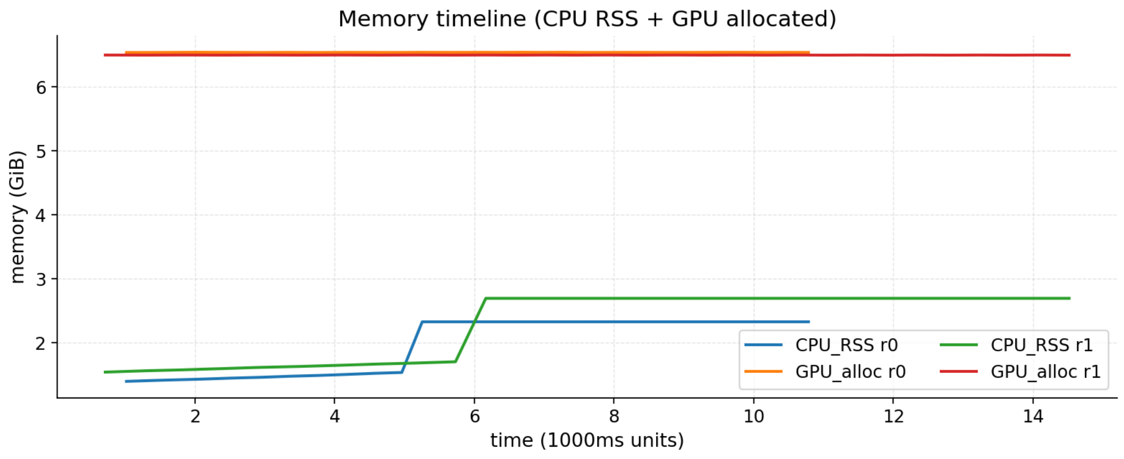

Figure 3: Memory timeline: CPU RSS and GPU allocated memory (recompute OFF vs ON).

5.4 Per-layer timing (mean $\pm$ 95% CI)

Per-layer timing provides the most informative diagnosis for where recomputation overhead is paid. We report forward, backward, and total per-layer time with 95% confidence intervals across steady-state steps in Figures 5, 6, and 7. Figure 8 visualizes the per-layer total delta (ON−OFF).

Notably, the largest total-time deltas occur around layer indices 12, 18, 23 (top-3 by mean delta). This suggests recomputation overhead is not uniform; certain modules (e.g., attention/MLP patterns, tensor shapes, or kernel selection) can amplify the recompute cost.

6 Discussion

What T1M2 establishes. Under a fixed Qwen decoder stack and fixed input regime, recomputation yields a measurable reduction in peak GPU reserved memory (about 10.4%) at a substantial step-time cost (about 41.3%). This quantifies the baseline memory–throughput slope for the later SlideFormer/Sideformers pipeline work.

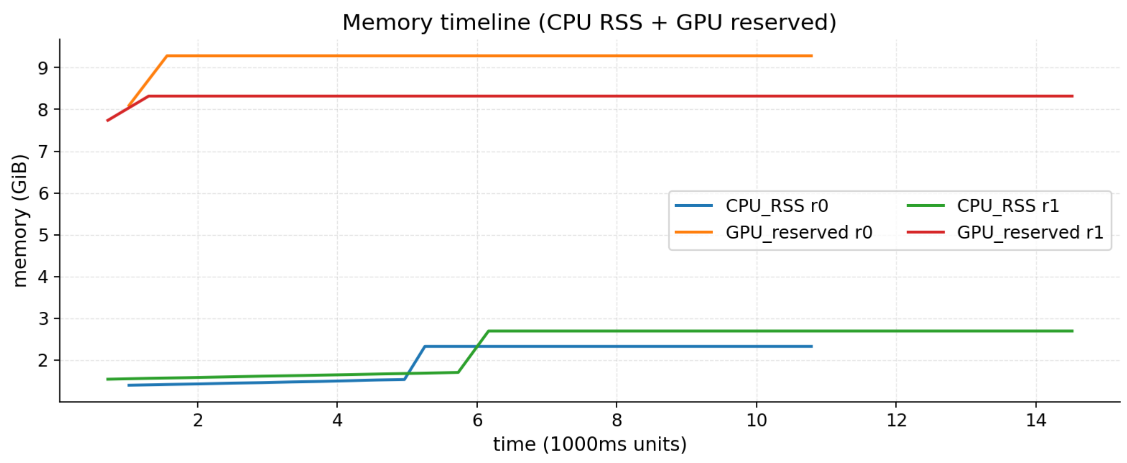

Figure 4: Memory timeline: CPU RSS and GPU reserved memory (recompute OFF vs ON).

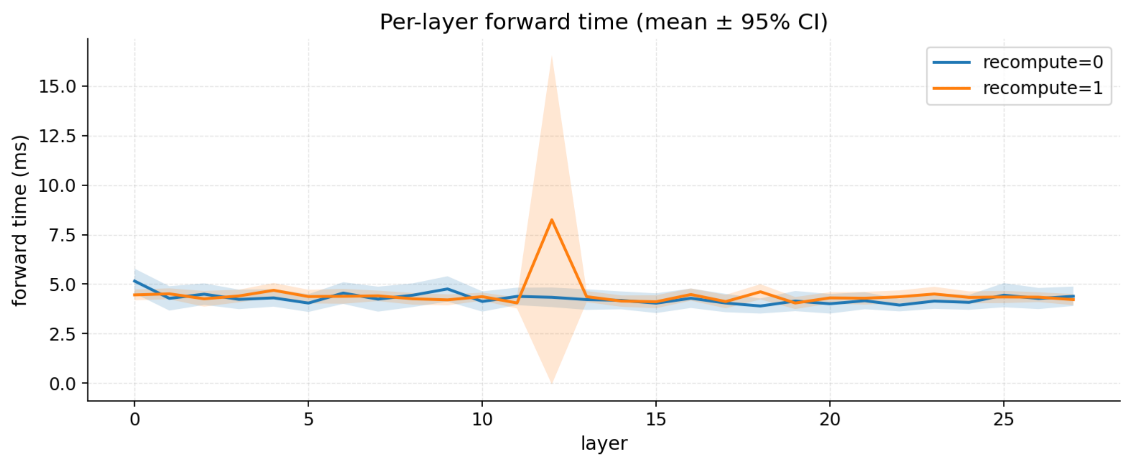

Figure 5: Per-layer forward time (mean with 95% CI).

Why memory reduction may appear "small". With `seq`=256 and `bs`=1, activation memory is a limited fraction of the total footprint; weights and allocator reservation dominate. Larger sequences/batches typically amplify the activation component, making recomputation benefits more pronounced.

Actionable next steps. Two extensions are most valuable for the project roadmap: (i) **Selective recompute** (checkpoint only a subset of layers) to obtain a clear VRAM–throughput curve, and (ii) **Optimizer sensitivity** (e.g., Adam) which increases optimizer state memory and can change the net benefit under constrained VRAM. Both are directly supported by our scripting interface and align with the SlideFormer co-design motivation.

Limitations. This report uses single-epoch runs; per-epoch time distributions become meaningful when repeating for multiple epochs/runs. However, step-level samples already support stable per-layer means and confidence intervals.

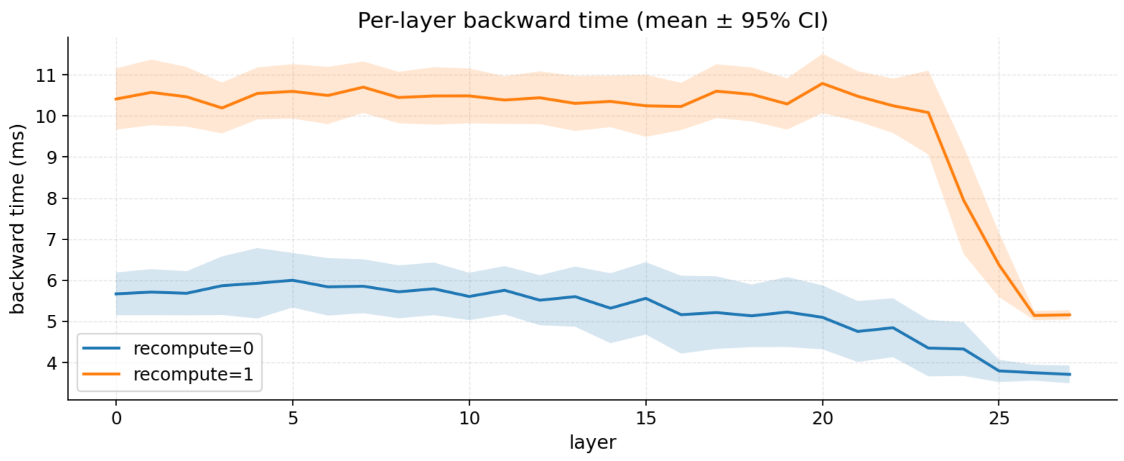

Figure 6: Per-layer backward time (mean with 95% CI).

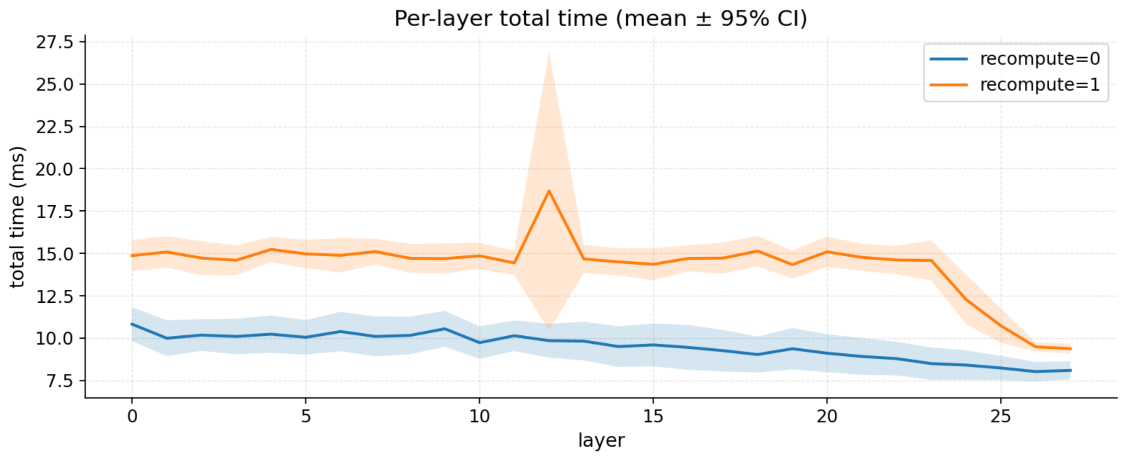

Figure 7: Per-layer total time (mean with 95% CI).

7 Conclusion

T1M2 provides a controlled recompute ablation with reproducible, paper-ready evidence. Recomputation reduces peak GPU memory (allocated/reserved) but increases step wall time by roughly ×1.41. Per-layer analysis shows recompute overhead is concentrated in a subset of layers, motivating selective checkpointing as the next optimization lever.

Reproducibility checklist

- GPU: NVIDIA GeForce RTX 3080 Laptop GPU
- CUDA / PyTorch: CUDA 12.8, PyTorch 2.7.0+cu128
- Model: Qwen-style decoder stack (28 layers), `seq`=256, `bs`=1, BF16
- Scripts: `t1m2_train_profiler.py` and `t1m2_time_report.py` in [3]

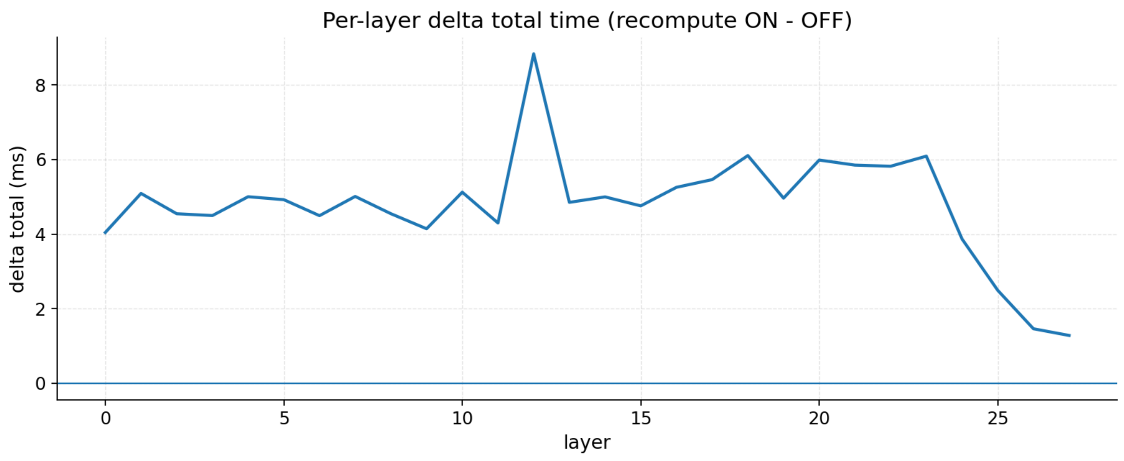

Figure 8: Per-layer total time delta (recompute ON − OFF).

- Figures used in this report: `layer_bwd_mean_ci.png`, `layer_delta_total.png`, `layer_fwd_mean_ci.png`, `layer_total_mean_ci.png`, `mem_timeline_cpu_gpualloc_compare.png`, `mem_timeline_cpu_gpureservec` `step_wall_ms_compare.png`, `step_wall_ms_delta.png`

References

[1] PyTorch Team. Pytorch activation checkpointing (torch.utils.checkpoint). `https://pytorch.org/docs/stable/checkpoint.html`, 2025.

[2] Zeyi et al. An efficient heterogeneous co-design for fine-tuning on a single gpu (slideformer), 2025. Manuscript PDF provided by the authors (anonymized formatting in the copy).

[3] Lai Hu and contributors. Sideformer rework of slideformer: T1m2 recomputation ablation codebase. `https://github.com/LaiHOLA/SideFormer-ReworkOfSlideformers`, 2025.

[4] MindSpore Team. Mindspore recomputation (recompute) documentation. `https://www.mindspore.cn/docs/en/master/api_python/mindspore/mindspore.recompute.html`, 2025.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Luojiang Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

[6] PyTorch Team. Pytorch profiler documentation. `https://pytorch.org/docs/stable/profiler.html`, 2025.