WHEN IS LOW-RANK PRETRAINING ACTUALLY EFFICIENT? A RESOURCE-AWARE BENCHMARK

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-rank methods offer a broadly applicable way to improve the efficiency of LLM pretraining by reducing model parameters, optimizer memory, or FLOPs. Yet their practical efficiency remains difficult to compare. Existing studies use different hardware, model scales, implementations, and training budgets, and often report intermediate savings without showing how they translate into end-to-end memory and training time. Therefore, a meaningful comparison requires a shared training setup, explicit control of the training budget, and direct measurement of both model quality and realized system costs. To provide such a comparison, we introduce the Low-rank Efficient Pretraining Benchmark (LR-EPB), which evaluates ten low-rank pretraining methods that place low-rank structure in model weights, optimizer states, or weight updates, on LLaMA-style models from 60M to 1B parameters. Across 130 runs and approximately 13,000 equivalent A100-40G GPU-hours, we compare model quality under matched-token and matched-FLOPs budgets and measure training FLOPs, peak memory, throughput, and serving latency. Our results show that low-rank efficiency is better characterized as a resource profile than by a single metric. Under matched FLOPs, low-rank weight methods can trade lower per-token cost for more training data and close much of their quality gap to full-rank training. Memory savings depend on which components are reduced, while FLOP reductions translate into wall-clock speed more effectively as model scale grows. Optimizer-state methods show a different profile, with modest memory savings and additional execution overhead despite nearly unchanged FLOPs. During serving, direct execution and V/O absorption favor different workloads. V/O absorption reuses the trained low-rank factors to reduce decoding latency and memory at large batch sizes without retraining. LR-EPB provides resource-specific guidance for selecting low-rank pretraining methods under different training and serving constraints.

1 INTRODUCTION

Pretraining large language models (LLMs) is constrained by memory and compute (Kaplan et al., 2020; Hoffmann et al., 2022). Among approaches to efficient pretraining, low-rank methods (Lialin et al., 2024; Zhao et al., 2024; Han et al., 2024; Zhu et al., 2025; Liu et al., 2025; Mo et al., 2025; Chen et al., 2026) are particularly attractive because they are largely *hardware-agnostic*, *architecture-agnostic*, and relatively *easy to integrate* into existing training pipelines. They rely only on dense matrix multiplications and modify how linear layers are parameterized or updated without requiring specialized hardware or a redesign of the overall model architecture, and several can be used as drop-in optimizers. This makes low-rank methods broadly applicable across model families and training systems. Yet how much efficiency they deliver in practice remains unclear. Existing work evaluates different methods on different hardware, model scales, and implementations, and often reports intermediate quantities such as parameter counts or optimizer-state sizes without measuring how these savings translate into peak memory or wall-clock time.

The comparison depends on both the method and budget. Low-rank methods reduce different resources depending on where low-rank structure enters training (Fig. 1). Low-rank weight methods such as CoLA (Liu et al., 2025) and LORO (Mo et al., 2025) reduce weights, gradients, optimizer states, and FLOPs per token. Optimizer-state methods such as GaLore (Zhao et al., 2024), Fira (Chen et al., 2026), and APOLLO (Zhu et al., 2025) keep dense weights while compressing optimizer states.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Where low rank enters	Methods	Reduced by construction: Weights	Grads	Opt. state	Activ.	FLOPs / token	Deployed checkpoint
Model parameterization	**Vanilla LR**, **SLTrain**, **CoLA**, **LORO**, LPA, LOST, Spectron	↓	↓	↓	$=^{a}$	$\downarrow^{b}$	factorized
Weight updates	**ReLoRA**, SwitchLoRA, LoQT	≈	↓	↓	=	≈	dense
Optimizer state	**GaLore**, **GoLore**, **Fira**, **Alice**, **APOLLO**, GRASS, Flora, OSD, LDAdam, FRUGAL, SAR, COAP, SubTrack++, H-Fac	=	=	↓	=	≈	dense
References (not low-rank)	**Full-size** (full-rank AdamW)	=	=	=	=	=	dense
	SinkGD (stateless optimizer)	=	=	∅	=	≈	dense

↓ reduced = unchanged ≈ small overhead ∅ removed a: ↓ with structure-aware activation checkpointing b: SLTrain ↑ FLOPs

Figure 1: Where low-rank structure enters determines which resources a method can reduce. Methods in bold are evaluated in LR-EPB, and Full-size and SinkGD serve as full-size reference methods.

Low-rank update methods such as ReLoRA (Lialin et al., 2024) train low-rank increments that are periodically merged into dense weights. However, these savings are not interchangeable. Reducing optimizer-state memory has limited effect when activations dominate the peak, while reducing FLOPs per token allows a method to process more data under a fixed compute budget. More generally, the choice of cost metric can change an efficiency comparison (Dehghani et al., 2022). For training algorithms, the budget also changes how much training each method receives. A method using half the FLOPs per token can process twice as many tokens under matched FLOPs, but receives no such advantage under matched tokens. We therefore evaluate both settings.

LORO at 350M illustrates how strongly the conclusion can depend on the resource being compared. After the same 7.86B tokens as standard training, denoted by Full-size, LORO reaches a validation perplexity of 17.03 versus 13.40, a gap of 27%. Because LORO uses 2.1× fewer FLOPs per token, the same FLOP budget allows it to process 16.6B tokens and reach 13.71 perplexity, reducing the gap to 2%. But the FLOP reduction does not translate proportionally into wall-clock time at this scale. LORO reaches only 1.04× the throughput of Full-size, so its matched-FLOPs run takes roughly twice the GPU time. The same method can therefore look substantially worse under matched tokens, nearly comparable under matched FLOPs, and more expensive in GPU time for the longer run. Its efficiency depends on which resource defines the budget.

To measure these trade-offs systematically, we introduce the Low-rank Efficient Pretraining Benchmark (LR-EPB). LR-EPB evaluates ten low-rank methods from the three branches in Fig. 1, together with the stateless optimizer SinkGD (Scetbon et al., 2026) and Full-size as references. We train LLaMA-style models from 60M to 1B parameters under matched-token and matched-FLOPs budgets. Using a common hardware platform and codebase, we measure training FLOPs, peak memory, and throughput across methods and scales, and additionally measure prefill and decode latency for low-rank weight checkpoints. In total, the benchmark contains 130 runs and approximately 13,000 equivalent A100-40G GPU-hours.

Across these measurements, low-rank efficiency is better characterized as a resource profile than by a single metric. The training budget changes the perplexity comparison most strongly for methods that reduce FLOPs per token, since a fixed compute budget allows them to process more data (Sec. 4). Memory savings depend on which component is reduced. Removing optimizer states alone yields limited end-to-end savings when activations dominate, while low-rank weight methods can reduce substantially more memory with activation checkpointing (Sec. 5.1). FLOP savings also translate into wall-clock speed differently across methods and scales. Low-rank weight methods realize little or no throughput gain at small scales but substantially larger gains at 1B, whereas optimizer-state methods often run slower despite nearly unchanged FLOPs (Sec. 5.2). Training and serving further favor different representations. Directly executing low-rank weight checkpoints can slow decoding, while V/O absorption turns the same factorized structure into a speedup at large batch sizes without retraining (Sec. 6). Fig. 2 provides a 350M snapshot of these trade-offs, and Tab. 1 summarizes method choices under different resource constraints.

Our contributions are as follows.

- **A resource-aware benchmark of low-rank pretraining.** LR-EPB evaluates twelve methods at four model scales under matched-token and matched-FLOPs budgets, together with measured FLOPs, peak training memory, throughput, and serving latency.

		Perplexity		Training cost				Decode latency (ms)		
		Matched tokens ↓	Matched FLOPs ↓	GFLOPs / token ↓	TPS (k tok/s) ↑	Mem. (GiB) ↓	+AC mem. (GiB) ↓	b=16 direct ↓	b=128 direct ↓	b=128 V/O abs. ↓
References	Full-size	13.40	13.40	2.09	21.7	17.2	–	16.0	47.8	–
	SinkGD	12.54	12.62	2.09	16.7	15.9	–	=	=	–
Optimizer state	GaLore	14.72	14.95	2.11	18.2	16.4	–	=	=	–
	GoLore	15.05	15.53	2.10	18.2	16.2	–	=	=	–
	Fira	12.72	12.98	2.12	16.4	16.6	–	=	=	–
	Alice	12.98	13.11	2.11	18.6	16.2	–	=	=	–
	APOLLO	12.96	13.03	2.09	20.0	16.4	–	=	=	–
Low-rank updates	ReLoRA	15.96	16.08	2.20	21.2	17.0	–	=	=	–
Low-rank weights	Vanilla LR	21.02	17.99	0.990	25.0	16.3	11.8	n/m	48.3	30.0
	SLTrain	15.26	18.69	3.34	17.1	15.9	n/m	n/m	n/m	n/m
	CoLA	14.63	13.71	0.990	24.7	16.9	12.1	19.4	48.1	29.7
	LORO	17.03	13.71	0.991	22.4	16.3	11.9	25.2	48.5	30.1

2× worse — Full-size — 2× better

Figure 2: Quality and resource comparison at 350M. Each cell reports the absolute value for perplexity, training FLOPs per token, throughput, peak memory, or decode latency, while its background color shows the ratio to Full-size. Blue indicates an advantage and red a disadvantage, with colors saturated at 2×. Arrows indicate the preferred direction. "=" denotes methods that use the same dense checkpoint for inference; "–" denotes not applicable, n/m not measured.

Table 1: Low-rank pretraining method choices under different binding resources, summarized from Secs. 4–6.

Q1	**Under a matched-token budget, which method reaches the lowest perplexity?** 130M–350M → **Fira** 1B → **LORO** 60M → *prefer Full-size*	[F1]
Q2	**Under a matched-FLOPs budget, which method reaches the lowest perplexity?** 130M–350M → **Fira** (*low-rank weight methods close most of their gap*) 1B → **LORO** 60M → *prefer Full-size*	[F1]
Q3	**When wall-clock time is the main cost, which method trains fastest?** 1B → **CoLA, LORO** (1.7×) 350M → **CoLA** (1.14×) ≤130M → *no clear advantage*	[F3]
Q4	**Under a device-memory limit, which method should I use?** → **Low-rank weight method + structure-aware checkpointing** (−36% peak memory at 1B)	[F2]
Q5	**Which checkpoint serves fastest?** Large batch → **LORO, CoLA** *with V/O absorption* (1.6×) Small batch → **dense checkpoints**	[F4]

- **A cross-resource characterization of low-rank efficiency.** We quantify how reductions in parameters, optimizer states, and FLOPs translate into end-to-end memory and training time, and identify where these algorithmic savings do not translate into corresponding system-level gains.
- **Resource-specific guidance for training and serving.** We summarize method choices under different training constraints and show how the factorized structure of low-rank weight models affects serving, including how V/O absorption can turn their decode overhead into a speedup at large batch sizes without retraining.

2 Overview of Low-Rank Pretraining Methods

Existing low-rank pretraining methods differ mainly in where they introduce low-rank structure into training. As shown in Fig. 1, some factorize model weights into low-rank factors, some train low-rank updates that are periodically merged into dense weights, and others use low-dimensional subspaces to reduce optimizer states. This choice largely determines which resources are reduced by design, which remain unchanged, and whether the resulting checkpoint is factorized or dense. LR-EPB evaluates 10 low-rank methods across these three branches: 4 low-rank weight methods, 1 low-rank update method, and 5 low-rank optimizer-state methods. We additionally include Full-size and the stateless optimizer SinkGD (Scetbon et al., 2026) as reference methods.

Low-rank weight methods. These methods parameterize a weight matrix $\boldsymbol{W} \in \mathbb{R}^{m\times n}$ as $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{m\times r}$ and $\boldsymbol{V} \in \mathbb{R}^{n\times r}$. Vanilla low-rank pretraining (Vanilla LR) trains the two factors directly. SLTrain (Han et al., 2024) augments the factorization with a sparse matrix,

CoLA (Liu et al., 2025) inserts a nonlinearity between the factors, and LORO (Mo et al., 2025) jointly updates the factors on the manifold of rank-r matrices. Operating on the factors reduces model weights, gradients, optimizer states, and generally, FLOPs per token. SLTrain is an exception in our benchmark, where the sparse component introduces additional computation. Activation memory is unchanged by default because the backward pass still requires full-dimensional layer inputs. Activation memory can be further reduced with structure-aware activation checkpointing (denoted by +AC), introduced by CoLA for low-rank weight methods, which retains the rank-r bottleneck and recomputes the full-dimensional intermediates during the backward pass. These methods also retain the factorized parameterization after training, making this the only branch with a non-dense deployed checkpoint (Sec. 6).

Low-rank update methods. ReLoRA (Lialin et al., 2024) trains low-rank increments on top of frozen dense weights and periodically merges them back into the model. Between merges, gradients and optimizer states are maintained only for the low-rank increments, reducing both. The underlying weights, FLOPs per token, and deployed checkpoint remain dense.

Low-rank optimizer-state methods. GaLore (Zhao et al., 2024), GoLore (He et al., 2024), Fira (Chen et al., 2026), Alice (Gong et al., 2025), and APOLLO (Zhu et al., 2025) retain dense model weights and dense gradients. They project each gradient, or quantities derived from it, onto a low-dimensional subspace that is refreshed periodically, and maintain the corresponding optimizer states in that subspace. Their primary saving is therefore optimizer-state memory, while the trained checkpoint remains dense. The projections and subspace refreshes introduce additional computation, whose effect on end-to-end training time is measured in Sec. 5.2.

Reference Methods. We include two reference methods that bracket the optimizer-state design space. Full-size denotes standard full-rank Transformer training with AdamW and no low-rank structure, and serves as the reference for all resource ratios in this paper. SinkGD (Scetbon et al., 2026) maintains no optimizer state, updating each weight matrix with a normalized gradient without tracking moment estimates. Although SinkGD is not a low-rank method, it represents the maximum memory saving obtainable by removing optimizer states under the same model configuration, and therefore provides a useful reference for the optimizer-state methods.

3 Benchmark Design

To compare the efficiency–quality trade-offs of low-rank pretraining methods in a fair and controlled manner, LR-EPB evaluates all methods under a shared protocol across four model scales. All methods use the same models, data, codebase, and training setup, allowing their behavior and scaling trends to be examined under a common setting. Because low-rank methods differ in FLOPs per token, the training budget determines how much data each method can process. We therefore evaluate every method under both matched-token and matched-FLOPs budgets. We also measure model quality together with peak memory, throughput, and serving latency on a single hardware platform, since reductions in parameters or FLOPs do not necessarily translate directly into end-to-end memory or wall-clock time. Finally, we use near-equivalent paired runs to define an empirical tolerance for small differences in model quality.

Experiment settings. We train LLaMA-style decoder-only models (Touvron et al., 2023) with 60M, 130M, 350M, and 1B parameters on FineWeb-Edu (Penedo et al., 2024). All models use SwiGLU, RoPE, and RMSNorm, with a sequence length of 256 and a global batch size of 512 sequences, corresponding to 131,072 tokens per update. Training uses bfloat16. Low-rank weight methods use ranks 128, 256, 256, and 512 at the four model scales. For each low-rank method, we follow the hyperparameters recommended by its authors. For Full-size, we select the learning rate at each scale from a grid of six values. More implementation details are provided in App. A.3. All twelve methods are evaluated up to 350M parameters. At 1B, we evaluate four representative methods spanning the low-rank weight and optimizer-state branches together with both references.

Matched-token and matched-FLOPs budgets. Under the matched-token budget, every method processes the same number of tokens as Full-size: 1.31B, 2.62B, 7.86B, and 13.11B at the four scales.

Table 2: Validation perplexity on FineWeb-Edu under matched-token and matched-FLOPs budgets. Gray multipliers report each method's FLOPs relative to Full-size under matched tokens, and the number of tokens processed relative to Full-size under matched FLOPs; values within 2% are omitted. Bold marks the best completed run in each column. ‡: matched-token run reused because its FLOPs are within 0.1% of the matched-FLOPs budget. †: diverged.

	60M		130M		350M		1B	
	Matched tokens	Matched FLOPs	Matched tokens	Matched FLOPs	Matched tokens	Matched FLOPs	Matched tokens	Matched FLOPs
Method	PPL · FLOPs ×	PPL · tokens ×	PPL · FLOPs ×	PPL · tokens ×	PPL · FLOPs ×	PPL · tokens ×	PPL · FLOPs ×	PPL · tokens ×
Full-size	**21.70**	**21.70**	17.43	17.43	13.40	13.40	12.49	12.49
SinkGD	22.62	23.25	17.14	16.80	**12.54**	**12.62**	**10.08**	**10.08**‡
GaLore	26.90	28.01	18.56	18.32	14.72	14.95	11.94 1.02	11.84 0.98
GoLore	26.01	25.42	18.63	18.21	15.05	15.53	–	–
Fira	22.03	22.63	**16.98**	**16.64**	12.72	12.98	12.43 1.03	12.12 0.97
Alice	23.66	24.05	17.53	17.41	12.98	13.11	–	–
APOLLO	23.44	24.37	17.59	17.16	12.96	13.03	–	–
ReLoRA	24.46 1.03	24.52 0.97	19.58 1.14	19.44 0.87	15.96 1.05	16.08 0.95	–	–
Vanilla LR	31.09 0.65	29.18 1.54	24.75 0.65	26.48 1.54	21.02 0.47	17.99 2.11	234.4† 0.44	† 2.28
SLTrain	24.40 1.39	30.17 0.72	20.41 1.50	19.75 0.66	15.26 1.60	18.69 0.63	–	–
CoLA	25.81 0.65	24.17 1.54	18.88 0.65	17.46 1.54	14.63 0.47	13.71 2.11	–	–
LORO	24.50 0.65	24.56 1.54	22.81 0.65	18.09 1.54	17.03 0.47	13.71 2.11	11.54 0.44	10.33 2.28

Under the matched-FLOPs budget, every method receives the same training-FLOP budget as the matched-token Full-size run, and its token count is determined by its own FLOPs per token. We measure FLOPs using the PyTorch FLOP counter throughout training, including optimizer steps and the SVD, QR, and linear-solve operations used by projection- and normalization-based methods (App. B.1). A matched-FLOPs run stops when its accumulated FLOPs reach the budget. In both settings, the learning-rate schedule is parameterized by the fraction of the corresponding budget consumed. We use FLOPs as the compute budget because they are independent of a particular hardware platform and implementation. Sec. 5.2 separately measures how FLOP reductions translate into wall-clock time.

Quality and system measurements. We measure model quality by validation perplexity on a held-out FineWeb-Edu shard at the end of each training budget. We report only the final checkpoint for each budget, since intermediate rankings can depend on the stage of the learning-rate schedule (Wen et al., 2026). Zero-shot accuracy on HellaSwag (Zellers et al., 2019), LAMBADA (Paperno et al., 2016), and ARC-Challenge (Clark et al., 2018) is reported in App. C.4.

System measurements are collected on a single NVIDIA A800 GPU. Peak memory is measured as the maximum memory reserved by the PyTorch caching allocator, and throughput as the steady-state number of tokens processed per second. Both are measured at sequence length 256 and batch size 32 for every method, with additional +AC measurements for low-rank weight methods (App. B). For low-rank weight checkpoints, we further measure time to first token (TTFT) and time per output token (TPOT) using a 512-token prompt, 128 generated tokens, and batch sizes from 1 to 256. We evaluate both the original factorized checkpoint and the checkpoint after V/O absorption (Sec. 6).

Comparison tolerance. Full-size, SinkGD, and the optimizer-state methods generally remain within 1% of Full-size in FLOPs per token, so their matched-token and matched-FLOPs runs process nearly the same number of tokens. Across these paired runs, validation perplexity differs by at most 0.5 at 130M and 350M and by 0.4–1.1 at 60M. We use the observed difference at each model scale as an empirical tolerance for method comparison and report smaller differences as ties.

4 Perplexity Under Matched-Token and Matched-FLOPs Budgets

Validation perplexity measures how well a pretrained model fits the data distribution and is the primary quality metric used to compare low-rank pretraining methods. We report it relative to Full-size. The comparison depends on how the training budget is defined. Under a matched-token budget, every method processes the same amount of data regardless of its per-token cost. Because the three branches in Sec. 2 differ substantially in FLOPs per token, the choice of budget can markedly change their perplexity gaps to Full-size. Tab. 2 reports validation perplexity under both budgets.

Methods that reduce FLOPs per token. For low-rank weight methods with lower FLOPs per token, the training budget can substantially change the perplexity gap to Full-size. At 350M, LORO's perplexity is 27% higher than Full-size under matched tokens, but under matched FLOPs it processes 2.11× more tokens and the gap shrinks to 2%. CoLA follows the same pattern, with its gap decreasing from 9% to 2% after processing 2.11× more tokens. Similar changes appear at 130M, and the differences between the two budgets exceed the comparison tolerance at both scales.

The additional tokens, however, do not fully compensate for restricting the weights to low rank. Vanilla LR, which directly trains a low-rank factorization, still has 34% higher perplexity than Full-size at 350M after processing 2.11× more tokens, and its 1B run diverges. LORO and CoLA close substantially more of this gap with the same FLOPs per token as Vanilla LR at 350M. LORO instead pays in time and runs 10% slower than Vanilla LR (Sec. 5.2).

Methods that keep or increase FLOPs per token. Optimizer-state methods and SinkGD show little change between the two budgets because their FLOPs per token remain close to Full-size. Projection and subspace-update operations add only 0.1–2.9% to the FLOPs per token of the optimizer-state methods, so matched FLOPs changes their token counts by at most 3%. Their perplexity differences between the two budgets remain within the comparison tolerance.

Methods with higher FLOPs per token instead receive fewer tokens under matched FLOPs. At 350M, SLTrain uses 1.60× the FLOPs per token of Full-size and therefore processes only 0.63× as many tokens, widening its perplexity gap from 14% to 39%. Despite these budget-dependent changes, the best-performing low-rank method at each scale is unchanged between the two budgets: Fira at 130M and 350M, and LORO at 1B.

Trends across model sizes. The relative FLOP advantage of low-rank weight methods increases with model size. For LORO, CoLA, and Vanilla LR, FLOPs per token decrease from 0.65× of Full-size at 60M and 130M to 0.47× at 350M and 0.44× at 1B. A matched-FLOPs budget therefore gives these methods progressively more tokens as the model grows. At 1B, LORO processes 29.9B tokens under matched FLOPs and reaches 10.33 perplexity, compared with 12.49 for Full-size and 10.08 for SinkGD. The relative performance of methods also changes with scale. Among the six methods evaluated at both 350M and 1B, LORO moves from fifth to second under matched tokens, while Fira moves from second to fourth. Because each configuration is trained once and Full-size improves less from 350M to 1B than the other methods, the 1B gaps should be interpreted relative to the Full-size baseline obtained under our protocol. Zero-shot downstream results show a similar overall pattern, with smaller differences between methods (App. C.4).

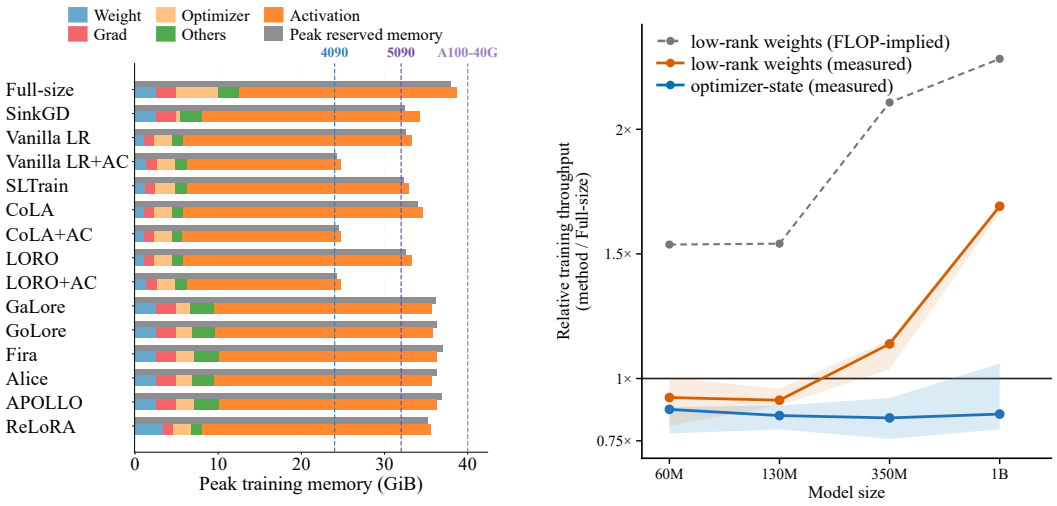
> **Finding 1: The training budget changes the perplexity comparison.** Under a fixed FLOP budget, low-rank weight methods can reduce perplexity by trading lower per-token cost for more training data. Optimizer-state methods have slightly higher per-token cost, though still close to Full-size, so they process slightly fewer tokens. Any improvement in this branch must therefore rely on more effective use of gradients and optimizer states during optimization.

5 Training Memory and Throughput

Sec. 2 identifies the resources that each branch reduces by construction. We now examine how much of these structural savings carries over to end-to-end training memory and throughput. The realized benefit depends on how large the targeted component is in the overall workload, which in turn varies with model scale, sequence length, batch size, and hardware. Unless stated otherwise, we measure on a single NVIDIA A800 GPU at sequence length 256 and batch size 32. App. D.4 further examines other sequence lengths and batch sizes. Fig. 3 summarizes the main results.

5.1 Peak Training Memory

At 1B, activations are the largest component of training memory under our measurement setting (Fig. 3a). Full-size reaches a peak of 38.0 GiB, of which 26.2 GiB comes from activations and 5.0 GiB from optimizer states. This composition limits the memory reduction available to optimizer-state methods. SinkGD removes optimizer states entirely but lowers peak memory by only 15%, providing



(a) Memory at 1B

(b) Throughput across scales

Figure 3: How structural savings translate into end-to-end training memory and throughput. (a) Peak training memory at 1B (gray) together with the peak of each memory component. Component peaks can occur at different points in the training step and therefore need not sum to the overall peak. (b) Throughput relative to Full-size across model scales. The lines show the median across methods in the branch, and the shading shows the range.

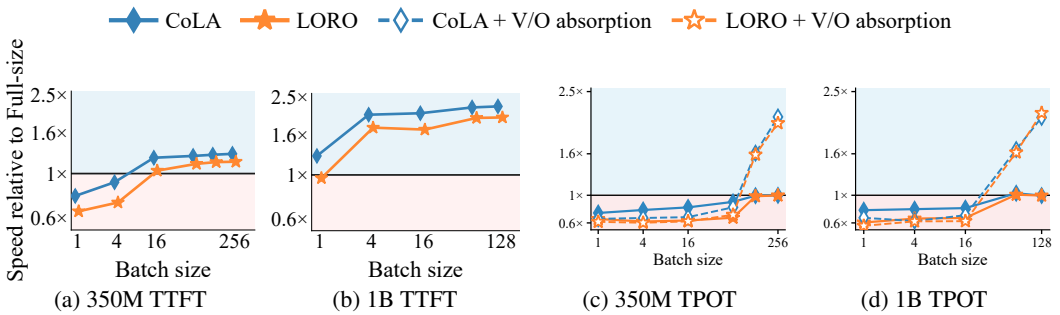
an empirical ceiling on the saving obtainable from optimizer-state removal in this setting. GaLore, GoLore, Fira, Alice, and APOLLO compress rather than remove the optimizer states and reduce peak memory by 3–5%. This ceiling is workload-dependent. As sequence length or batch size increases, activation memory grows while optimizer-state memory remains fixed, so optimizer states account for a smaller fraction of the peak (App. D.4).

Low-rank weight methods can reduce more components, but factorization alone does not reduce activation memory. Without +AC, Vanilla LR, CoLA, and LORO lower peak memory by 11–14% through smaller weights, gradients, and optimizer states, while activation memory remains similar to or slightly higher than Full-size because the rank-r intermediate is also stored. With +AC, only the rank-r bottleneck is retained and the full-dimensional intermediates are recomputed during the backward pass. Peak memory then falls by 36%, to 24.2–24.5 GiB. This reduction comes with recomputation overhead: +AC lowers throughput by 4–6% in the same measurement setting, as shown in Tab. 13.

Finding 2: Peak-memory savings are limited by the memory components a method reduces. At 1B, activations account for about 70% of Full-size peak memory, while optimizer states account for a much smaller share. Removing optimizer states entirely reduces the peak by 15%, whereas low-rank weight methods reach a 36% reduction when activation checkpointing also reduces activation storage. The benefit therefore changes with the memory composition of the workload.

5.2 Training Throughput

FLOP reductions translate into training speed more effectively as model size grows (Fig. 3b). At 60M and 130M, Vanilla LR, CoLA, and LORO use 29–35% fewer FLOPs per token than Full-size, but their measured throughput remains between 0.81× and 1.00× of Full-size. The throughput benefit becomes visible at larger scales, reaching 1.04–1.16× at 350M and 1.66–1.70× at 1B. This increase cannot be explained by a substantially larger FLOP reduction: for LORO and Vanilla LR, the FLOP-implied speedup changes only from 2.11× at 350M to 2.28× at 1B, while the measured speedup rises much more sharply. The larger gap between FLOP-implied and measured speedup at smaller models is likely due to lower GPU efficiency for the reduced matrix multiplications and a larger relative contribution from other training overheads. The realized speedup also decreases at longer sequence lengths, where attention accounts for a larger share of the training step and is not reduced by the low-rank weight factorization (App. D.1).



(a) 350M TTFT (b) 1B TTFT (c) 350M TPOT (d) 1B TPOT

Figure 4: Serving speed of CoLA and LORO relative to Full-size. Values above 1 indicate lower latency than Full-size. The TTFT panels show direct execution. V/O absorption changes TTFT by at most 0.5%. The TPOT panels compare direct execution with V/O absorption.

Optimizer-state methods exhibit a different pattern. Their FLOPs per token generally remain within 1% of Full-size, but their measured throughput is 6–24% lower in 19 of the 20 configurations. GaLore, GoLore, and Fira remain 14–24% slower at 350M and 1B. The projection and subspace-refresh operations therefore incur wall-clock overhead that is only weakly reflected in the aggregate FLOP count. APOLLO at 1B is the only exception, running 6% faster than Full-size. ReLoRA runs 2–12% slower than Full-size up to 350M and 13% faster at 1B.

Memory and throughput can also interact through the micro-batch size. With a fixed global batch, reducing memory can allow a larger micro-batch and fewer gradient-accumulation steps. At 1B and sequence length 1024, CoLA+AC with micro-batch 12 uses 14% less memory than standard CoLA with micro-batch 8 while running 4% faster. Increasing the micro-batch to 16 gives a 12% throughput gain while using 9% more memory (App. D.3).

Finding 3: FLOP savings translate into wall-clock speed differently across methods and scales. For low-rank weight methods, the realized throughput gain grows substantially with model size even when the FLOP reduction changes little, reaching about 1.7× at 1B. Optimizer-state methods have nearly unchanged FLOPs per token but usually run slower because of additional optimization overhead. FLOPs therefore do not translate uniformly into training time.

6 Serving Low-Rank Weight Checkpoints

We finally examine how the trained checkpoints behave during serving. Low-rank weight methods are the only branch whose deployed checkpoints remain factorized (Sec. 2). Optimizer-state and low-rank update methods serve as ordinary dense models. For CoLA and LORO, we measure prefill latency using time to first token (TTFT) and decoding latency using time per output token (TPOT). We consider two execution paths. Direct execution preserves the factorization used during training, while V/O absorption combines adjacent value- and output-projection factors after training. Fig. 4 compares both with Full-size across batch sizes.

Direct execution. Prefill and decoding respond differently to the factorized checkpoint. Prefill processes the prompt tokens in parallel, allowing the lower FLOPs of the factorized projections to translate into lower TTFT once the batch is sufficiently large. At 1B and batch 64, CoLA and LORO prefill 1.8–2.2× faster than Full-size. Decoding shows the opposite behavior at small batch. Each step processes only one new token per sequence, and replacing a dense projection with two thin matrix multiplications introduces additional kernel launches and lower GPU utilization. As a result, directly executed checkpoints have 1.2–1.6× higher TPOT than Full-size at 350M and up to 1.7× higher TPOT at 1B.

As the batch grows, KV-cache traffic accounts for a larger share of decoding time. Direct execution expands each latent value back to the full head dimension before caching it, so its value cache remains the same size as that of a dense model. The decoding speed of the direct path therefore approaches that of Full-size at large batch, as the FLOP advantage becomes less important than KV-cache traffic.

V/O absorption. V/O absorption reduces this cache cost by reusing the factors already present in the checkpoint. Following the weight-absorption idea used in multi-head latent attention (Liu et al., 2024), we combine the adjacent value- and output-projection factors after training so that values remain in the latent space during decoding. For head h, let $\boldsymbol{z}_j \in \mathbb{R}^r$ denote the latent value of token j, $\boldsymbol{U}_h^V \in \mathbb{R}^{r \times d_h}$ the factor that expands it to the head dimension, and $\boldsymbol{D}_h^O \in \mathbb{R}^{d_h \times r}$ the corresponding block of the first output-projection factor. Direct execution computes $\boldsymbol{s}_{i,h} = \Big(\sum_{j \le T} a_h(i,j)\, \boldsymbol{z}_j \boldsymbol{U}_h^V \Big) \boldsymbol{D}_h^O$, where $a_h(i,j)$ is the attention weight and T is the context length. Since $\boldsymbol{U}_h^V$ and $\boldsymbol{D}_h^O$ are adjacent linear maps, they can be multiplied once after training as

$$\boldsymbol{G}_h = \boldsymbol{U}_h^V \boldsymbol{D}_h^O, \qquad \boldsymbol{s}_{i,h} = \Big(\sum_{j \le T} a_h(i,j)\, \boldsymbol{z}_j \Big) \boldsymbol{G}_h. \tag{1}$$

The absorbed execution path therefore caches the rank-r latent value $\boldsymbol{z}_j$ instead of the d-dimensional value. This reduces value-cache traffic, at the cost of additional aggregation in the latent space (App. E.2). The trade-off becomes favorable once KV-cache traffic accounts for a sufficiently large fraction of decoding time.

Results. The measurements follow this trade-off (Fig. 4). At 350M, V/O absorption remains slower than Full-size at batch 64, reaching 0.71–0.76× its speed, but becomes 1.6× faster at batch 128 and 2.0–2.1× faster at batch 256. At 1B, the crossover occurs earlier: absorbed decoding is already 1.5–1.6× faster than Full-size at batch 64. At batch 128, it is 2.1–2.2× faster than Full-size, while direct execution runs at about the speed of Full-size. Across the measured configurations, it also reduces peak generation memory by 30–36% (App. E.3), while changing TTFT by at most 0.5%.

> **Finding 4: Training and serving favor different execution paths for the same low-rank factors.** Direct execution preserves the factorization used during training and benefits prefill, but it increases decoding latency at small batch and loses its advantage as KV-cache traffic grows. V/O absorption reorganizes the same trained factors to reduce value-cache traffic, making decoding faster at sufficiently large batch without retraining.

7 Related Work

Comparisons of training algorithms are sensitive to both how cost is measured and how individual methods are tuned (Dehghani et al., 2022; Dahl et al., 2023; Kaddour et al., 2023; Wen et al., 2026). The closest benchmark to LR-EPB is Glentis et al. (2025), which compares parameter- and memory-efficient pretraining methods on C4 from 60M to 1B parameters and finds that final perplexity is largely determined by training FLOPs. We observe the same trend for LORO and CoLA under matched FLOPs, while Vanilla LR retains a substantial gap even after processing more tokens. LR-EPB extends this comparison beyond model quality by measuring end-to-end memory, throughput, and serving latency under a common implementation and hardware platform. Concurrent work (Shivagunde et al., 2026) studies the solutions reached by different low-rank methods from a complementary perspective. App. F discusses these benchmarks and the individual low-rank methods.

8 Conclusion

LR-EPB shows that low-rank pretraining efficiency is inherently resource-dependent. The training budget can change the quality comparison because methods with lower FLOPs per token can process more data under a fixed compute budget. At the system level, algorithmic savings translate unevenly into end-to-end costs: optimizer-state compression has limited impact when activations dominate memory, while FLOP reductions yield larger throughput gains as model scale grows. The same resource dependence extends to serving, where factorized checkpoints behave differently under direct execution and V/O absorption. Taken together, these results show that no single efficiency metric provides a complete comparison of low-rank pretraining methods. Method selection should therefore follow the resource that constrains the target training or serving workload.

AI USE STATEMENT

In this work, we used generative AI tools to edit and restructure the text of the paper, draft plotting scripts, and cross-check numbers extracted from earlier figure files. We did not use generative AI tools to design the experiments, run training, or produce the reported measurements. All numbers in the paper are generated from the training and profiling logs by scripts that the authors verified. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work benchmarks training methods for language models on a public dataset, FineWeb-Edu, and involves no human subjects or personal data. Lower training cost can reduce the energy needed to train such models, and it also makes them easier to train for any purpose.

REPRODUCIBILITY STATEMENT

App. A describes the evaluated methods, hyperparameters, and training budgets, and App. B describes the FLOP, memory, throughput, and serving protocols.

REFERENCES

Xi Chen, Kaituo Feng, Changsheng Li, Xunhao Lai, Xiangyu Yue, Ye Yuan, and Guoren Wang. Fira: Can we achieve full-rank training of llms under low-rank constraint? *Advances in Neural Information Processing Systems*, 38:120680–120712, 2026.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

George E Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, et al. Benchmarking neural network training algorithms. *arXiv preprint arXiv:2306.07179*, 2023.

Mostafa Dehghani, Anurag Arnab, Lucas Beyer, Ashish Vaswani, and Yi Tay. The efficiency misnomer. In *International Conference on Learning Representations*, 2022.

Athanasios Glentis, Jiaxiang Li, Qiulin Shang, Andi Han, Ioannis Tsaknakis, Quan Wei, and Mingyi Hong. Scalable parameter and memory efficient pretraining for llm: Recent algorithmic advances and benchmarking. *arXiv preprint arXiv:2505.22922*, 2025.

Wenbo Gong, Meyer Scetbon, Chao Ma, and Edward Meeds. Towards efficient optimizer design for llm via structured fisher approximation with a low-rank extension. *arXiv preprint arXiv:2502.07752*, 2025.

Andi Han, Jiaxiang Li, Wei Huang, Mingyi Hong, Akiko Takeda, Pratik Jawanpuria, and Bamdev Mishra. Sltrain: a sparse plus low rank approach for parameter and memory efficient pretraining. *Advances in Neural Information Processing Systems*, 37:118267–118295, 2024.

Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 17554–17571. PMLR / OpenReview.net, 2024.

Yutong He, Pengrui Li, Yipeng Hu, Chuyan Chen, and Kun Yuan. Subspace optimization for large language models with convergence guarantees. *arXiv preprint arXiv:2410.11289*, 2024.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022.

Paul Janson, Edouard Oyallon, and Eugene Belilovsky. Stabilizing native low-rank llm pretraining. *arXiv preprint arXiv:2602.12429*, 2026.

Tao Ji, Bin Guo, Yuanbin Wu, Qipeng Guo, Lixing Shen, Zhan Chen, Xipeng Qiu, Qi Zhang, and Tao Gui. Towards economical inference: Enabling deepseek's multi-head latent attention in any transformer-based llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33313–33328, 2025.

Jean Kaddour, Oscar Key, Piotr Nawrot, Pasquale Minervini, and Matt J Kusner. No train no gain: Revisiting efficient training algorithms for transformer-based language models. *Advances in Neural Information Processing Systems*, 36:25793–25818, 2023.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Jiaxi Li, Lu Yin, Li Shen, Jinjin Xu, Liwu Xu, Tianjin Huang, Wenwu Wang, Shiwei Liu, and Xilu Wang. Lost: Low-rank and sparse pre-training for large language models. *arXiv preprint arXiv:2508.02668*, 2025.

Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. Relora: High-rank training through low-rank updates. In *International Conference on Learning Representations*, 2024.

Kaizhao Liang, Bo Liu, Lizhang Chen, and Qiang Liu. Memory-efficient llm training with online subspace descent. *Advances in Neural Information Processing Systems*, 37:64412–64432, 2024.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

Ziyue Liu, Ruijie Zhang, Zhengyang Wang, Mingsong Yan, Zi Yang, Paul D Hovland, Bogdan Nicolae, Franck Cappello, Sui Tang, and Zheng Zhang. Cola: Compute-efficient pre-training of llms via low-rank activation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 4627–4645, 2025.

Sebastian Loeschcke, Mads Toftrup, Michael J Kastoryano, Serge Belongie, and Vésteinn Snæbjarnarson. Loqt: Low-rank adapters for quantized pretraining. *Advances in Neural Information Processing Systems*, 37:115282–115308, 2024.

Xingtai Lv, Ning Ding, Kaiyan Zhang, Ermo Hua, Ganqu Cui, and Bowen Zhou. Scalable efficient training of large language models with low-dimensional projected attention. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14588–14599, 2024.

Fanxu Meng, Pingzhi Tang, Xiaojuan Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. Transmla: Multi-head latent attention is all you need. *arXiv preprint arXiv:2502.07864*, 2025.

Zhanfeng Mo, Long-Kai Huang, and Sinno Jialin Pan. Parameter and memory efficient pretraining via low-rank riemannian optimization. In *The Thirteenth International Conference on Learning Representations*, 2025.

Aashiq Muhamed, Oscar Li, David Woodruff, Mona Diab, and Virginia Smith. Grass: Compute efficient low-memory llm training with structured sparse gradients. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14978–15003, 2024.

Son Nguyen, Lizhang Chen, Bo Liu, and Qiang Liu. Memory-efficient optimization with factorized hamiltonian descent. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2025, Mai Khao, Thailand, 3-5 May 2025*, volume 258 of *Proceedings of Machine Learning Research*, pp. 2863–2871. PMLR, 2025.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc-Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 1525–1534, 2016.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Sahar Rajabi, Nayeema Nonta, and Sirisha Rambhatla. Subtrack++: Gradient subspace tracking for scalable llm training. *Advances in Neural Information Processing Systems*, 38:31584–31613, 2026.

Thomas Robert, Mher Safaryan, Ionut-Vlad Modoranu, and Dan Alistarh. Ldadam: Adaptive optimization from low-dimensional gradient statistics. In *International Conference on Learning Representations*, volume 2025, pp. 4778–4814, 2025.

Meyer Scetbon, Chao Ma, Wenbo Gong, and Ted Meeds. Gradient multi-normalization for efficient llm training. *Advances in Neural Information Processing Systems*, 38:43001–43030, 2026.

Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pp. 9367–9376. PMLR, 2021.

Namrata Shivagunde, Vijeta Deshpande, Sherin Muckatira, and Anna Rumshisky. Beyond perplexity: A geometric and spectral study of low-rank pre-training. *arXiv preprint arXiv:2605.13652*, 2026.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. In *International Conference on Learning Representations*, volume 2026, pp. 144731–144838, 2026.

Jinqi Xiao, Shen Sang, Tiancheng Zhi, Jing Liu, Qing Yan, Linjie Luo, and Bo Yuan. Coap: Memory-efficient training with correlation-aware gradient projection. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 30116–30126, 2025.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

Haochen Zhang, Junze Yin, Guanchu Wang, Zirui Liu, Lin Yang, Tianyi Zhang, Anshumali Shrivastava, and Vladimir Braverman. Breaking the frozen subspace: Importance sampling for low-rank optimization in llm pretraining. *Advances in Neural Information Processing Systems*, 38: 10122–10174, 2026.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61121–61143. PMLR / OpenReview.net, 2024.

Kaiye Zhou, Shucheng Wang, and Jun Xu. Switchlora: Switched low-rank adaptation can learn full-rank information. *arXiv preprint arXiv:2406.06564*, 2024.

Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, Vikas Chandra, Bo Long, David Z Pan, Zhangyang Wang, and Jinwon Lee. Apollo: Sgd-like memory, adamw-level performance. *Proceedings of Machine Learning and Systems*, 7, 2025.

Philip Zmushko, Aleksandr Beznosikov, Martin Takác, and Samuel Horváth. FRUGAL: memory-efficient optimization by reducing state overhead for scalable training. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025.

APPENDIX CONTENTS

Appendix Contents **14**

A Methods and Implementation **14**

A.1 Evaluated Methods 14

A.2 Shared Implementation and Budget Accounting 16

A.3 Model Configurations, Budgets, and Hyperparameters 16

A.4 Tuning Protocol 16

B Measurement Protocols **17**

B.1 FLOP Accounting 17

B.2 Quality Evaluation 17

B.3 Training Memory 18

B.4 Training Throughput 18

B.5 Serving 18

C Additional Quality Results **18**

C.1 Absolute Costs 18

C.2 Comparison tolerance 18

C.3 C4 results 19

C.4 Downstream Evaluation 20

C.5 Divergence and Anomalies 21

D Additional System Results **22**

D.1 Throughput across configurations 22

D.2 Parameters and peak memory 22

D.3 Structure-aware Activation Checkpointing 24

D.4 Rank and Workload Sensitivity 25

E Serving Details **25**

E.1 Latency under direct execution 25

E.2 V/O absorption 25

E.3 Complete absorption scan 27

F Extended Related Work **27**

A METHODS AND IMPLEMENTATION

A.1 EVALUATED METHODS

We evaluate ten low-rank pretraining methods from the three branches in Fig. 1, together with SinkGD and Full-size as reference methods. For low-rank weight methods, we factorize all linear layers in the

attention and MLP blocks, including the Q, K, V, O, gate, up, and down projections, while keeping the embedding layer and output head dense. Unless stated otherwise, these methods use ranks 128, 256, 256, and 512 at 60M, 130M, 350M, and 1B, respectively. We write a factorized matrix as $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{m\times r}$ and $\boldsymbol{V} \in \mathbb{R}^{n\times r}$. This subsection introduces these methods with their specific implementation choices. Complete learning rates and numerical hyperparameters are collected in Tab. 4.

Vanilla LR. Vanilla LR directly trains both factors of $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}^\top$ with AdamW. It serves as the basic factorized baseline without an additional mechanism for addressing the rank constraint. It reduces weights, gradients, optimizer states, and FLOPs per token.

SLTrain (Han et al., 2024) augments the low-rank product with a sparse matrix of fixed random support, $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}^\top + \boldsymbol{S}$, and jointly trains the factors and the nonzero entries of $\boldsymbol{S}$. We use a sparsity ratio of 3%. The sparse component introduces additional degrees of freedom outside the rank-r subspace, but it also increases FLOPs per token above Full-size (Tab. 2). SLTrain reduces weights, gradients, and optimizer states.

CoLA (Liu et al., 2025) inserts a nonlinearity between the two factors, $f(\boldsymbol{x}) = \boldsymbol{U}\sigma(\boldsymbol{V}^\top\boldsymbol{x})$, changing the model function in addition to its parameterization. At 60M, we use weight decay 0.1 and gradient clipping at 0.5. We also evaluate its memory-efficient execution with structure-aware activation checkpointing as CoLA+AC (App. D.3). CoLA reduces weights, gradients, optimizer states, and FLOPs per token.

LORO (Mo et al., 2025) uses the same factorized parameterization as Vanilla LR but couples the updates of the two factors through Riemannian optimization on the manifold of rank-r matrices, with an exact Riemannian step every 500 updates. Because the parameterization is unchanged, LORO and Vanilla LR have nearly the same FLOPs per token. LORO reduces weights, gradients, optimizer states, and FLOPs per token.

ReLoRA (Lialin et al., 2024) trains low-rank additive updates on top of dense weights, merges them into the weights every 2000 updates, and reinitializes the low-rank factors after each merge. Successive intervals can therefore update different subspaces, while the resulting checkpoint remains dense. Following the authors' recipe, we use rank 128 with a short full-rank warm start before the first merge (2000 / 4000 / 10000 updates at 60M / 130M / 350M and 1B) and learning rate 3×10^{-3} at 60M and 130M, or 10^{-3} at 350M and 1B. ReLoRA reduces gradients and optimizer states within each interval.

GaLore (Zhao et al., 2024) retains dense model weights and projects each weight gradient onto a rank-r subspace obtained by SVD. Adam moments are maintained in the projected space. We refresh the projection every 200 updates and use projection scale of 0.25. GaLore reduces optimizer states.

GoLore (He et al., 2024) constructs and refreshes the gradient subspace using randomized projections instead of the dominant singular vectors used by GaLore. We refresh the subspace every 200 updates and use projection scale 1.0. GoLore reduces optimizer states.

Fira (Chen et al., 2026) builds on the GaLore projection and adds a scaled correction from the gradient component outside the projected subspace, yielding a full-rank update while keeping the optimizer states low-dimensional. We use the same projection rank, refresh interval, and projection settings as GaLore. Fira reduces optimizer states.

Alice (Gong et al., 2025) maintains a low-dimensional basis from gradient information and constructs memory-efficient adaptive updates in this basis. We refresh the basis every 200 updates. Alice reduces optimizer states.

APOLLO (Zhu et al., 2025) projects gradients into a low-dimensional space to estimate compact adaptive scaling factors, avoiding full-size moment estimates. We refresh the projection every 200 updates and use the same scale-dependent rank schedule as GaLore. APOLLO reduces optimizer states.

SinkGD (Scetbon et al., 2026) applies five iterations of alternating matrix normalization to each gradient and maintains no first- or second-moment optimizer states. We use a learning-rate scale of 0.05. SinkGD is not a low-rank method. By removing optimizer states entirely, it provides the reference for the maximum memory reduction obtainable from optimizer-state removal under the same model configuration.

Full-size denotes standard full-rank Transformer training with AdamW and the settings in Tab. 4. It contains no low-rank structure and serves as the reference for all resource ratios reported in the paper.

A.2 Shared Implementation and Budget Accounting

All methods use the same data pipeline, training loop, logging system, evaluation code, and resource-accounting framework. Method-specific implementations are confined to the model parameterization, the handling of gradients and optimizer states, and the update rule. We train on FineWeb-Edu-100BT (Penedo et al., 2024) using the SentencePiece tokenizer from T5-base and reserve the lexicographically last shard for validation.

All training uses bfloat16 with random seed 42 on four GPUs and a global batch size of 512 sequences of length 256. At 60M, 130M, and 350M, the micro-batch sizes are 128, 64, and 32 sequences, with 1, 2, and 4 gradient-accumulation steps, respectively. At 1B, Full-size uses a micro-batch size of 16 with 8 accumulation steps, while the other methods use a micro-batch size of 32 with 4 accumulation steps.

A matched-token run stops after the number of updates listed in Tab. 3. A matched-FLOPs run stops when its accumulated measured FLOPs reach the corresponding budget in the same table (App. B.1). In both settings, the learning-rate schedule is parameterized by the fraction of the corresponding budget consumed, with the first 10% used for warmup.

A.3 Model Configurations, Budgets, and Hyperparameters

Tab. 3 specifies the four model scales and their training budgets. The matched-FLOPs budget at each scale is defined by the measured computation of the corresponding Full-size matched-token run.

Table 3: Model configurations and training budgets. The first 10% of each budget is used for warmup. Rank gives the default scale-specific rank used by low-rank weight methods and, where applicable, optimizer-state methods.

Model	Params	d	d_{ffn}	Layers	Heads	Rank	Updates	Warmup	Tokens	Budget (PFLOPs)
60M	58M	512	1376	8	8	128	10,000	1,000	1.31B	344.4
130M	134M	768	2048	12	12	256	20,000	2,000	2.62B	1,796.8
350M	368M	1024	2736	24	16	256	60,000	6,000	7.86B	16,408.1
1B	1339M	2048	5461	24	32	512	100,000	10,000	13.11B	102,127.0

Tab. 4 collects the numerical hyperparameters used for each method. Method-specific choices that affect the implementation are described in App. A.1; the table serves as the complete configuration record.

A.4 Tuning Protocol

For Full-size, we select the learning rate independently at each model scale from $\{10^{-4}, 3\times10^{-4}, 10^{-3}, 3\times10^{-3}, 10^{-2}, 3\times10^{-2}\}$. The selected values are reported in Tab. 4. Full-size uses cosine decay with warm restarts.

For the remaining methods, we start from the hyperparameters and learning-rate schedules recommended by their authors. When a recommended configuration diverges on FineWeb-Edu, we make the smallest adjustment needed to restore stable training and record the final setting in Tab. 4.

This protocol gives Full-size a scale-specific learning-rate search, while the other methods primarily follow their published configurations. Since optimizer comparisons can be sensitive to tuning choices (Wen et al., 2026), we report the exact settings used for every method and document all deviations from the recommended configurations.

Table 4: Hyperparameters used in LR-EPB. Low-rank methods and SinkGD start from the settings recommended by their authors, with any adjustments documented in App. A.4. Full-size selects its learning rate independently at each model scale from the grid in App. A.4.

Method	Learning rate	Method-specific settings
Full-size	3×10^{-3} at 60M, 130M, and 350M; 3×10^{-4} at 1B	AdamW $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-6}$; cosine decay with warm restarts (cycle length 500, or 200 at 1B, with 5 restart warmup steps)
SinkGD	10^{-2}	five normalization iterations; learning-rate scale 0.05
GaLore	10^{-2}	projection refresh every 200 updates; projection scale 0.25; rank from Tab. 3
GoLore	10^{-2}	projection refresh every 200 updates; projection scale 1.0; rank from Tab. 3
Fira	10^{-2}, or 3×10^{-3} at 1B	projection settings and rank as GaLore
Alice	2×10^{-2}	basis refresh every 200 updates; rank from Tab. 3
APOLLO	10^{-2}	projection refresh every 200 updates; rank from Tab. 3
ReLoRA	3×10^{-3} at 60M and 130M, 10^{-3} at 350M and 1B	merge every 2000 updates, rank 128
Vanilla LR	10^{-3}	rank from Tab. 3
SLTrain	3×10^{-3} at 60M and 130M; 10^{-3} at 350M and 1B	sparsity 3%; rank from Tab. 3
CoLA	6×10^{-3} at 60M; 3×10^{-3} at 130M and 350M	weight decay 0.1 and gradient clipping 0.5 at 60M; rank from Tab. 3
LORO	10^{-2} at 60M; 3×10^{-3} at 130M, 350M, and 1B	exact Riemannian step every 500 updates; rank from Tab. 3

B Measurement Protocols

B.1 FLOP Accounting

We profile actual optimizer updates near the beginning of each run using PyTorch's `FlopCounterMode` and use the average FLOPs per update to accumulate training computation over the full run. The profiled updates include the forward and backward passes, gradient clipping, method-specific end-of-step operations, and the optimizer update. PyTorch's default registry covers the main matrix operations; we additionally implement FLOP formulas for `svd`, `qr`, and `solve` in `torch.linalg`, which are used by the projection, subspace-refresh, and matrix-normalization operations of several methods. These measured FLOPs define the matched-FLOPs budget. Wall-clock execution time is measured separately through the throughput protocol in App. B.4.

B.2 Quality Evaluation

All methods are evaluated on the same held-out FineWeb-Edu shard using sequences of length 256. We compute perplexity as

$$\mathrm{PPL} = \exp\Big(-\frac{1}{N}\sum_{i=1}^{N}\log p(x_i \mid x_{<i})\Big), \tag{2}$$

where N denotes the number of evaluated target tokens. For matched-token runs, we evaluate every 1,000 updates; for matched-FLOPs runs, we evaluate every 5×10^{16} accumulated FLOPs. The main results use the checkpoint at the end of each training budget.

We additionally report zero-shot accuracy on HellaSwag (Zellers et al., 2019), LAMBADA (Paperno et al., 2016), and ARC-Challenge (Clark et al., 2018), with a maximum sequence length of 512. For HellaSwag and ARC-Challenge, we select the candidate with the highest length-normalized log-probability. For LAMBADA, we report exact top-1 accuracy. Accuracy is reported together with its binomial standard error.

B.3 Training Memory

We profile training memory on a single NVIDIA A800 GPU using locally generated tokens at sequence length 256 and batch size 32. Our primary metric is the maximum memory reserved by the PyTorch caching allocator during the measurement window. This includes both allocated tensors and unused memory blocks retained by the allocator. It corresponds to the gray bar in Fig. 3a and the peak-memory value in Tab. 12. All memory values are reported in GiB (2^{30} bytes).

To analyze the source of memory use, we separately record the peak allocated memory associated with model parameters, gradient buffers, optimizer states, activations, and other allocations. Activation memory is measured as the peak increase in allocated memory during the monitored forward pass relative to the allocation immediately before that pass, and includes temporary attention and matrix-multiplication workspaces. The "other" category contains residual allocated memory after activations are released. Component peaks are measured at the stages of the training step relevant to each component and can therefore occur at different times. They are not expected to sum to the overall reserved-memory peak; Fig. 3a reports the two quantities separately for this reason.

B.4 Training Throughput

We measure training throughput on a single NVIDIA A800 GPU using one training process and locally generated tokens. To examine workload dependence while keeping the number of tokens per update fixed, we use four (sequence length, batch size) configurations: $(256, 32)$, $(512, 16)$, $(1024, 8)$, and $(2048, 4)$, each corresponding to 8,192 tokens per update. The main-text results use $(256, 32)$.

For each configuration, we run 100 warmup steps followed by 500 measured steps. Throughput is computed as the number of tokens processed during the measurement window divided by its wall-clock duration. This metric characterizes execution speed at a fixed workload; differences in the number of training tokens required by each method are accounted for separately by the training-budget experiments. The micro-batch experiments in App. D.3 vary the batch size at sequence length 1024.

B.5 Serving

Serving experiments use a single NVIDIA A800 GPU with a 512-token prompt and 128 generated tokens. For each low-rank weight model, direct execution and V/O absorption are evaluated under the same workload. Time to first token (TTFT) measures the latency from the start of a request to generation of the first output token and is averaged over 20 measurements after five warmup runs. Time per output token (TPOT) measures decoding latency after the first token and is averaged over 100 decode steps following ten warmup steps. Peak generation memory is measured over the full generation using `torch.cuda.max_memory_allocated()`. It includes model weights, the KV cache, and temporary tensors. This metric uses peak allocated memory and is therefore distinct from the reserved-memory metric used for training.

C Additional Quality Results

C.1 Absolute Costs

Tables 5 and 6 complement Tab. 2 with the corresponding absolute training costs: the measured FLOPs consumed by matched-token runs and the number of tokens processed by matched-FLOPs runs.

C.2 Comparison tolerance

We use near-equivalent matched-token and matched-FLOPs runs to calibrate small differences in perplexity. Full-size, SinkGD, and the optimizer-state methods have FLOPs per token close to Full-size, so the two budgets expose them to nearly the same amount of training data. Tab. 7 reports the difference between their matched-FLOPs and matched-token perplexities.

Table 5: Validation perplexity and absolute training costs at 60M and 130M. Under matched tokens, PFLOPs report the measured training computation of each run. Under matched FLOPs, Tokens report the number of training tokens processed within the common FLOP budget. Bold marks the best completed run in each column.

	60M				130M			
	Same tokens		Same FLOPs		Same tokens		Same FLOPs	
Method	PPL	PFLOPs	PPL	Tokens (B)	PPL	PFLOPs	PPL	Tokens (B)
Full-size	**21.70**	344	**21.70**	1.31	17.43	1,797	17.43	2.62
SinkGD	22.62	344	23.25	1.31	17.14	1,797	16.80	2.62
GaLore	26.90	345	28.01	1.31	18.56	1,804	18.32	2.61
GoLore	26.01	345	25.42	1.31	18.63	1,804	18.21	2.61
Fira	22.03	345	22.63	1.31	**16.98**	1,807	**16.64**	2.61
Alice	23.66	345	24.05	1.31	17.53	1,811	17.41	2.60
APOLLO	23.44	345	24.37	1.31	17.59	1,800	17.16	2.62
ReLoRA	24.46	356	24.52	1.27	19.58	2,056	19.44	2.29
Vanilla LR	31.09	224	29.18	2.02	24.75	1,166	26.48	4.04
SLTrain	24.40	478	30.17	0.94	20.41	2,701	19.75	1.74
CoLA	25.81	224	24.17	2.02	18.88	1,166	17.46	4.04
LORO	24.50	224	24.56	2.02	22.81	1,166	18.09	4.04

Table 6: Validation perplexity and absolute training costs at 350M and 1B. Columns follow Tab. 5. ‡: matched-token run reused because its FLOPs are effectively identical to the matched-FLOPs budget. †: diverged.

	350M				1B			
	Same tokens		Same FLOPs		Same tokens		Same FLOPs	
Method	PPL	PFLOPs	PPL	Tokens (B)	PPL	PFLOPs	PPL	Tokens (B)
Full-size	13.40	16,408	13.40	7.86	12.49	102,127	12.49	13.11
SinkGD	**12.54**	16,411	**12.62**	7.86	**10.08**	102,159	**10.08**‡	13.11
GaLore	14.72	16,559	14.95	7.79	11.94	104,138	11.84	12.86
GoLore	15.05	16,484	15.53	7.83	–	–	–	–
Fira	12.72	16,630	12.98	7.76	12.43	105,128	12.12	12.73
Alice	12.98	16,560	13.11	7.79	–	–	–	–
APOLLO	12.96	16,445	13.03	7.86	–	–	–	–
ReLoRA	15.96	17,297	16.08	7.46	–	–	–	–
Vanilla LR	21.02	7,785	17.99	16.58	234.4†	44,735	†	29.93
SLTrain	15.26	26,216	18.69	4.92	–	–	–	–
CoLA	14.63	7,785	13.71	16.58	–	–	–	–
LORO	17.03	7,786	13.71	16.57	11.54	44,738	10.33	29.93

At 130M and 350M, the absolute difference is at most 0.5 perplexity. At 60M, it ranges from 0.39 to 1.11. We use these observed ranges as a scale-specific empirical tolerance when comparing methods and report smaller differences as ties. This tolerance is intended to distinguish small protocol-level variation from the larger differences discussed in the main results.

C.3 C4 RESULTS

We repeat the 60M and 130M experiments on C4 (Raffel et al., 2020) to examine whether the small-scale trends observed on FineWeb-Edu depend on the pretraining corpus. Tab. 8 reports validation perplexity under both matched-token and matched-FLOPs budgets.

The main patterns remain similar. Under matched FLOPs, CoLA and LORO benefit from the additional training tokens and narrow their gaps to Full-size; at 130M, LORO reaches 23.77 perplexity

Table 7: Difference in validation perplexity between matched-FLOPs and matched-token runs for methods whose FLOPs per token remain close to Full-size. Positive values indicate higher perplexity under matched FLOPs. These paired runs are used to define the empirical comparison tolerance. ‡: matched-token run reused at 1B. Dashes indicate methods not evaluated at 1B.

Method	60M	130M	350M	1B
Full-size	same run	same run	same run	same run
SinkGD	+0.63	−0.34	+0.08	‡
GaLore	+1.11	−0.24	+0.23	−0.10
GoLore	−0.59	−0.42	+0.48	–
Fira	+0.60	−0.34	+0.26	−0.31
Alice	+0.39	−0.12	+0.13	–
APOLLO	+0.93	−0.43	+0.07	–

Table 8: Validation perplexity on C4 at 60M and 130M under matched-token and matched-FLOPs budgets. Bold marks the best completed run in each column.

	Matched tokens		Matched FLOPs	
Method	60M	130M	60M	130M
Full-size	**30.69**	23.69	**30.69**	23.69
SinkGD	31.13	23.21	31.49	22.83
GaLore	34.61	25.30	34.52	24.99
GoLore	34.50	25.96	41.51	25.21
Fira	30.88	**23.08**	30.86	**22.74**
Alice	31.91	23.90	32.94	23.65
APOLLO	32.16	23.84	32.31	23.37
ReLoRA	35.05	29.73	34.18	30.05
Vanilla LR	40.94	34.30	38.03	35.53
SLTrain	33.70	25.64	35.25	27.90
CoLA	35.01	25.96	33.13	24.13
LORO	34.00	25.17	32.65	23.77

compared with 23.69 for Full-size. SLTrain shows the opposite behavior because its higher FLOPs per token reduce the amount of data it can process under the same compute budget. SinkGD and the optimizer-state methods receive nearly the same number of tokens under the two budgets and generally show only small changes in perplexity, with GoLore at 60M as the main exception. Fira gives the lowest perplexity at 130M under both budgets, consistent with the FineWeb-Edu results.

C.4 Downstream Evaluation

Tabs. 9 and 10 report zero-shot accuracy on HellaSwag (HS), LAMBADA (LB), and ARC-Challenge (ARC) at 60M, 130M, and 350M. The downstream results broadly follow the perplexity trends, although the differences between methods are smaller.

At 350M under matched tokens, SinkGD, Fira, and Alice, which also achieve the lowest perplexities, obtain the strongest results on HellaSwag and ARC-Challenge. Vanilla LR and ReLoRA, whose perplexities are substantially higher, also perform worse on HellaSwag and LAMBADA. The effect of the training budget is visible for low-rank weight methods as well. For LORO at 350M, moving from matched tokens to matched FLOPs improves HellaSwag accuracy by 4.2 points and LAMBADA accuracy by 7.7 points, consistent with the large reduction in perplexity reported in Sec. 4.

At 60M and 130M, HellaSwag and ARC-Challenge remain close to their 25% chance level, and many differences between methods are small relative to the reported standard errors. We therefore focus the downstream comparison on the broader trends rather than fine-grained rankings at these scales.

Table 9: Zero-shot accuracy (%) after matched-token training at 60M, 130M, and 350M. Standard errors are 0.4–0.5 points for HellaSwag, 0.2–0.6 for LAMBADA, and 1.2–1.3 for ARC-Challenge. Bold marks the best completed run in each column.

	60M			130M			350M		
Method	HS	LB	ARC	HS	LB	ARC	HS	LB	ARC
Full-size	**26.1**	10.4	23.8	27.4	13.6	**26.8**	32.1	17.5	27.4
SinkGD	25.8	10.3	24.1	27.9	13.7	26.5	34.1	**20.4**	**29.2**
GaLore	25.4	9.9	24.3	27.0	13.8	26.3	30.2	17.0	27.5
GoLore	25.2	8.9	24.2	27.3	13.1	26.3	29.4	16.1	26.1
Fira	25.9	**11.3**	24.9	**28.0**	**13.9**	26.3	33.8	19.4	28.2
Alice	25.5	9.8	24.5	27.9	13.5	26.4	**34.2**	18.0	28.7
APOLLO	25.5	10.0	23.7	27.8	13.8	25.9	32.6	18.5	27.6
ReLoRA	25.3	8.7	23.9	26.4	12.4	25.4	29.5	15.5	27.3
Vanilla LR	24.8	7.3	22.5	25.8	10.2	25.3	26.7	8.9	25.3
SLTrain	25.4	10.2	24.5	26.5	12.3	24.7	29.8	16.8	25.0
CoLA	25.2	9.1	24.6	27.0	13.0	23.8	30.0	17.4	26.2
LORO	25.4	10.1	**25.1**	25.8	8.4	25.3	28.0	11.7	26.0

Table 10: Zero-shot accuracy (%) after matched-FLOPs training at 60M, 130M, and 350M. Standard errors and notation follow Tab. 9.

	60M			130M			350M		
Method	HS	LB	ARC	HS	LB	ARC	HS	LB	ARC
Full-size	**26.1**	10.4	23.8	27.4	13.6	**26.8**	32.1	17.5	27.4
SinkGD	**26.1**	10.3	24.3	27.8	14.0	26.3	**34.0**	19.2	28.6
GaLore	24.9	8.4	23.4	27.2	13.9	25.8	30.2	16.4	26.2
GoLore	24.9	9.6	24.1	27.4	13.6	25.3	29.3	16.1	27.3
Fira	25.6	10.8	24.7	**27.8**	14.4	26.2	33.1	18.5	28.0
Alice	25.6	9.1	24.7	27.4	12.7	26.4	33.9	18.3	**29.1**
APOLLO	25.4	9.8	24.6	27.6	14.2	24.7	33.0	18.3	27.6
ReLoRA	25.4	8.9	24.2	26.6	12.3	26.0	28.5	14.0	26.7
Vanilla LR	25.0	8.4	24.1	25.1	7.3	23.5	27.4	12.5	25.3
SLTrain	25.1	7.7	22.4	26.6	12.1	26.2	27.4	10.6	25.5
CoLA	25.2	10.1	**25.0**	27.4	13.1	24.7	32.4	18.3	26.4
LORO	25.3	**11.0**	24.7	27.5	**14.5**	25.9	32.2	**19.4**	26.9

C.5 Divergence and Anomalies

We summarize runs that deviate noticeably from the overall trends observed for the corresponding methods.

- **Vanilla LR at 1B.** Vanilla LR diverges under both budgets, with final perplexity above 200. We searched the learning rates in our candidate grid but did not find a configuration that avoids divergence.
- **Vanilla LR at 130M.** Under matched FLOPs, Vanilla LR performs worse despite processing 1.54× more tokens. Its perplexity increases from 24.75 to 26.48 on FineWeb-Edu and from 34.30 to 35.53 on C4. The same behavior across both corpora suggests that this result is specific to the matched-FLOPs configuration at this scale rather than to the pretraining corpus.
- **LORO at 60M.** On FineWeb-Edu, processing 1.54× more tokens under matched FLOPs does not improve perplexity (24.50 vs. 24.56). On C4, the corresponding run improves from 34.00 to 32.65.

Table 11: Training throughput relative to Full-size on a single NVIDIA A800 GPU. Each entry reports the ratio at the main setting of sequence length 256 and batch size 32, with the gray range spanning all four workload configurations in App. B.4. The Full-size row gives the corresponding absolute throughput in tokens/s.

	60M	130M	350M	1B
Full-size (tokens/s)	115,702	52,631	21,654	5,587
SinkGD	0.69 0.65–0.70	0.71 0.71–0.79	0.77 0.75–0.80	0.91 0.87–0.91
GaLore	0.88 0.88–0.91	0.87 0.87–0.91	0.84 0.84–0.87	0.86 0.82–0.86
GoLore	0.89 0.89–0.92	0.85 0.85–0.89	0.84 0.84–0.87	0.84 0.80–0.84
Fira	0.78 0.78–0.84	0.80 0.80–0.85	0.76 0.76–0.81	0.80 0.75–0.80
Alice	0.79 0.79–0.84	0.83 0.83–0.89	0.86 0.85–0.89	0.94 0.93–0.94
APOLLO	0.88 0.88–0.91	0.89 0.89–0.94	0.92 0.92–0.94	1.06 0.94–1.06
ReLoRA	0.89 0.89–0.94	0.88 0.88–0.95	0.98 0.97–0.98	1.13 1.06–1.13
Vanilla LR	1.00 0.98–1.03	0.89 0.89–1.03	1.16 1.10–1.16	1.69 1.39–1.69
SLTrain	0.83 0.83–0.87	0.80 0.80–0.87	0.79 0.76–0.84	0.77 0.72–0.77
CoLA	0.81 0.81–0.98	0.96 0.96–1.02	1.14 1.10–1.14	1.70 1.43–1.70
LORO	0.92 0.92–0.95	0.91 0.87–0.93	1.04 1.03–1.06	1.66 1.35–1.66

- **SLTrain at 130M.** On FineWeb-Edu, the matched-FLOPs run processes only 0.66× as many tokens but improves perplexity from 20.41 to 19.75. The same behavior does not appear on C4, where perplexity increases from 25.64 to 27.90.
- **GoLore at 60M.** GoLore shows an unusually large difference between the two budgets at 60M, with matched-FLOPs minus matched-token perplexity of $+7.0$ on C4.

D Additional System Results

D.1 Throughput across configurations

Tab. 11 reports training throughput relative to Full-size at all four model scales under the four workload configurations defined in App. B.4, each with 8,192 tokens per update. The main trends in Sec. 5.2 are consistent across these configurations. Vanilla LR, CoLA, and LORO are generally no faster than Full-size at 60M and 130M, but become consistently faster from 350M onward. SLTrain remains slower at every scale and configuration because its sparse component increases computation. Optimizer-state methods also generally reduce throughput: GaLore, GoLore, and Fira are slower than Full-size in every measured configuration, while Alice and APOLLO follow the same trend except for the small 1B advantage of APOLLO in the sequence-length-256 setting.

For low-rank weight methods, the realized speedup decreases as sequence length increases while the number of tokens per update is held fixed. At 1B, increasing the sequence length from 256 to 2048 reduces throughput relative to Full-size from 1.69× to 1.39× for Vanilla LR, from 1.70× to 1.43× for CoLA, and from 1.66× to 1.35× for LORO. This trend is consistent with attention accounting for a larger fraction of the training step at longer sequences, while the low-rank factorization reduces the cost of the linear layers but not attention.

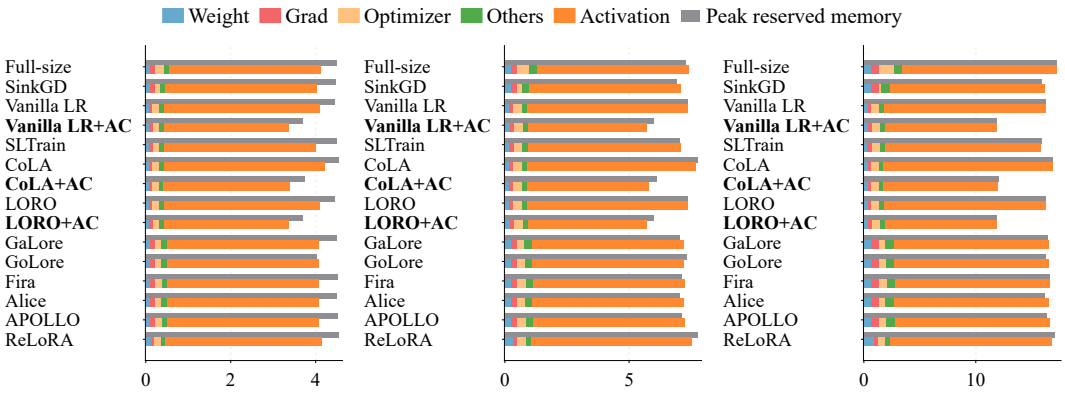
D.2 Parameters and peak memory

Tab. 12 reports parameter counts and peak reserved training memory for all methods at the four model scales, while Fig. 5 provides the corresponding memory breakdown at 60M, 130M, and 350M. Under the fixed workload of sequence length 256 and batch size 32, optimizer-state memory becomes a larger source of savings as model size grows. Removing optimizer states entirely with SinkGD reduces peak memory by less than 1% at 60M, 5% at 130M, 8% at 350M, and 15% at 1B.

Factorization alone provides limited peak-memory savings at smaller scales because activation memory remains largely unchanged and the rank-r intermediate introduces additional activation storage. Without +AC, Vanilla LR, CoLA, and LORO remain close to Full-size at 60M and 130M,

Table 12: Parameter counts and peak reserved training memory (GiB) at sequence length 256 and batch size 32 on a single NVIDIA A800 GPU. Params count the trainable weights in each method's parameterization. Rows marked +AC use structure-aware activation checkpointing.

	60M		130M		350M		1B	
Method	Params	Peak	Params	Peak	Params	Peak	Params	Peak
Full-size	58M	4.50	134M	7.27	368M	17.22	1,339M	37.96
SinkGD	58M	4.47	134M	6.90	368M	15.91	1,339M	32.44
GaLore	58M	4.50	134M	7.04	368M	16.43	1,339M	36.18
GoLore	58M	4.51	134M	7.30	368M	16.23	1,339M	36.23
Fira	58M	4.52	134M	7.10	368M	16.58	1,339M	36.93
Alice	58M	4.50	134M	7.03	368M	16.20	1,339M	36.27
APOLLO	58M	4.52	134M	7.10	368M	16.35	1,339M	36.91
ReLoRA	58M	4.54	134M	7.76	368M	17.02	1,339M	35.12
Vanilla LR	43M	4.45	94M	7.34	185M	16.26	609M	32.48
+AC		3.70		5.97		11.84		24.25
SLTrain	44M	4.50	97M	7.05	194M	15.92	646M	32.32
CoLA	43M	4.55	94M	7.78	185M	16.88	609M	33.94
+AC		3.73		6.09		12.05		24.48
LORO	43M	4.45	94M	7.35	185M	16.28	609M	32.58
+AC		3.70		5.98		11.87		24.18



(a) 60M (b) 130M (c) 350M

Figure 5: Training-memory breakdown at 60M, 130M, and 350M on a single NVIDIA A800 GPU, with sequence length 256 and batch size 32. Gray bars show the overall peak reserved memory. Colored segments show separately measured peaks for individual memory components. Because these components can peak at different points in the training step, they do not form a partition of the gray bar.

with CoLA using up to 7% more memory at 130M. Their savings become clearer at larger scales, reaching 2–6% at 350M and 11–14% at 1B. With +AC, the same methods reduce peak memory by 16–18% at 60M and 130M, 30–31% at 350M, and 36% at 1B.

The parameter reduction of low-rank weight methods also grows with model scale. For Vanilla LR, CoLA, and LORO, the factorized models contain about 74% as many parameters as Full-size at 60M but only 45% at 1B. The difference arises because the embedding and output layers remain dense and account for a progressively smaller fraction of the model as the Transformer blocks grow.

Table 13: Effect of structure-aware activation checkpointing (+AC) at sequence length 256 and batch size 32 on a single NVIDIA A800 GPU. Each change compares +AC with standard execution of the same method at the same micro-batch size.

		Peak memory (GiB)			Activations (GiB)		Throughput (tokens/s)		
Scale	Method	Standard	+AC	Change	Standard	+AC	Standard	+AC	Change
60M	Vanilla LR	4.45	3.70	−17%	3.67	2.92	116,198	101,328	−13%
	CoLA	4.55	3.73	−18%	3.78	2.97	93,671	96,385	+3%
	LORO	4.45	3.70	−17%	3.67	2.92	106,877	94,229	−12%
130M	Vanilla LR	7.34	5.97	−19%	6.44	4.75	46,950	44,636	−5%
	CoLA	7.78	6.09	−22%	6.77	4.90	50,464	48,795	−3%
	LORO	7.35	5.98	−19%	6.44	4.75	48,054	38,805	−19%
350M	Vanilla LR	16.26	11.84	−27%	14.46	9.96	25,018	22,764	−9%
	CoLA	16.88	12.05	−29%	15.12	10.24	24,661	21,723	−12%
	LORO	16.28	11.87	−27%	14.46	9.96	22,445	21,207	−6%
1B	Vanilla LR	32.48	24.25	−25%	27.49	18.46	9,449	8,865	−6%
	CoLA	33.94	24.48	−28%	28.80	19.03	9,485	9,067	−4%
	LORO	32.58	24.18	−26%	27.49	18.46	9,302	8,794	−5%

Table 14: Micro-batch scan at 1B and sequence length 1024 with the global batch held fixed. Standard execution uses micro-batch 8; +AC allows larger micro-batches by reducing activation memory. Full-size at micro-batch 8 uses 43.71 GiB and reaches 5,526 tokens/s.

	Vanilla LR		CoLA		LORO	
Execution, micro-batch	GiB	tokens/s	GiB	tokens/s	GiB	tokens/s
Standard, 8	38.22	8,146	38.22	8,108	39.39	7,841
+AC, 8	23.72	7,137	23.76	7,232	23.71	7,703
+AC, 12	32.45	8,243	32.78	8,429	32.44	8,890
+AC, 16	41.18	8,951	41.78	9,060	41.17	9,330
+AC, 24	58.63	9,752	59.80	9,754	58.62	9,827

D.3 Structure-aware Activation Checkpointing

Structure-aware activation checkpointing (+AC) follows the memory-efficient execution introduced by CoLA (Liu et al., 2025). For a factorized layer with row-vector inputs $\boldsymbol{X}$, we first compute the rank-r bottleneck $\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{V}$ and then $\boldsymbol{Y} = \boldsymbol{Z}\boldsymbol{U}^\top$; for CoLA, the bottleneck is $\boldsymbol{Z} = \sigma(\boldsymbol{X}\boldsymbol{V})$. During the forward pass, +AC retains the compact tensor $\boldsymbol{Z}$ and discards selected full-dimensional intermediates, which are recomputed during the backward pass. This changes neither the model function nor the optimizer, and the complete policy is applied to all factorized Transformer layers.

Tab. 13 reports the effect of +AC under the main-text workload. Relative to standard execution of the same method, +AC reduces peak memory by 17–29% at 60M–350M and by 25–28% at 1B. The reduction comes primarily from activation memory. Its throughput cost varies across the smaller models, from a 3% increase to a 19% decrease, while the three 1B methods show a consistent overhead of 4–6%.

The saved memory can also be exchanged for a larger micro-batch. Tab. 14 varies the micro-batch size at 1B and sequence length 1024 while keeping the global batch fixed. With +AC, increasing the micro-batch reduces the number of gradient-accumulation steps and progressively improves throughput. At micro-batch 12, all three low-rank weight methods remain well below the memory used by their standard micro-batch-8 execution while recovering much of the recomputation overhead. At micro-batch 16, they use 41.2–41.8 GiB, still below the 43.7 GiB used by Full-size at micro-batch 8, and reach 8.95–9.33K tokens/s. This illustrates how activation-memory savings can be converted into additional throughput when device memory is the binding constraint.

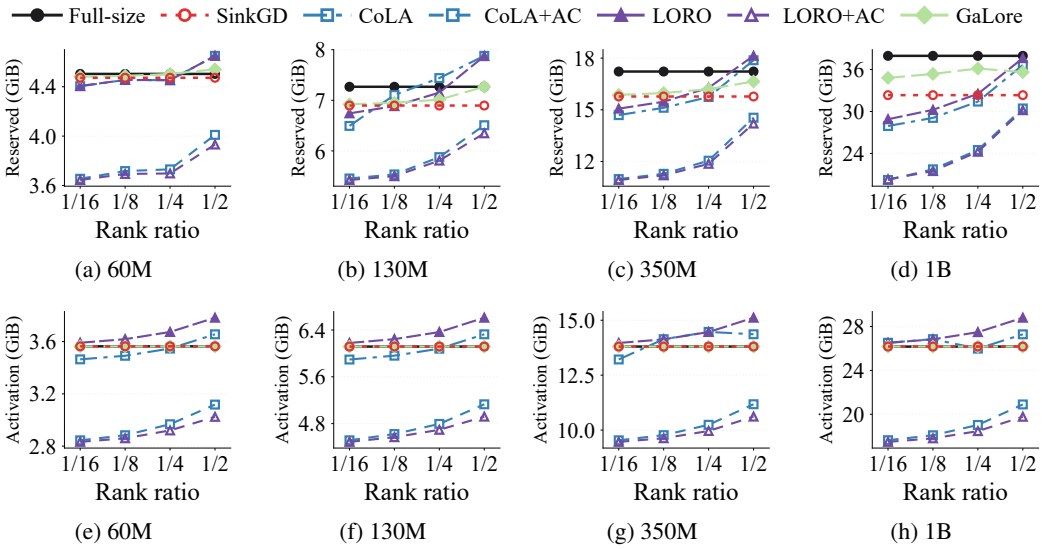


Figure 6: Peak reserved memory (top) and activation memory (bottom) as a function of the rank ratio r/d, measured at sequence length 256 and batch size 32. For CoLA and LORO, increasing the rank enlarges both the factorized parameterization and the rank-r activation bottleneck; for GaLore, only the projected optimizer states depend on rank. Full-size and SinkGD are rank-independent and are shown as horizontal references.

D.4 Rank and Workload Sensitivity

Fig. 6 examines how training memory changes with the rank ratio r/d at sequence length 256 and batch size 32. The effect depends on where the low-rank structure enters. For GaLore, model weights and gradients remain dense, so changing the rank affects only the projected optimizer states and has little effect on peak memory. For CoLA and LORO, the rank also changes the size of the factorized parameters and the intermediate rank-r bottleneck. Their peak memory therefore increases with rank under both standard execution and +AC. The same trend appears in activation memory because a wider bottleneck requires more intermediate storage.

Fig. 7 instead varies the workload at 1B. Model parameters, gradients, and optimizer states remain nearly fixed as batch size or sequence length increases, while activation memory grows substantially. Optimizer states therefore account for a progressively smaller fraction of total memory under larger workloads, reducing the end-to-end benefit available from optimizer-state compression. This explains the workload dependence observed in Sec. 5.1.

E Serving Details

E.1 Latency under direct execution

Tab. 15 reports the absolute TTFT and TPOT measurements underlying the direct-execution curves in Fig. 4. All measurements use the serving protocol in App. B.5: a single NVIDIA A800 GPU, a 512-token prompt, and 128 generated tokens.

E.2 V/O absorption

We give the full derivation of V/O absorption in Eq. (1), allowing the value and output bottlenecks to have different widths, r_V and r_O. We use row-vector notation and bias-free projections. The derivation follows the same associativity used for weight absorption in latent attention (Liu et al., 2024).

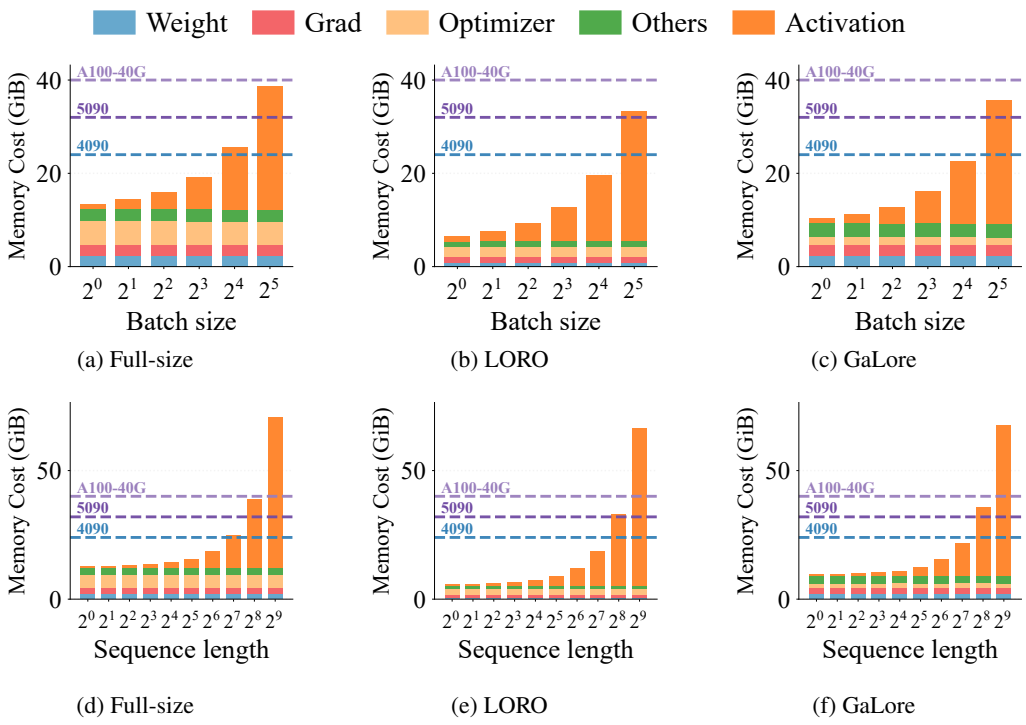


Figure 7: Component-wise memory peaks at 1B for Full-size, LORO, and GaLore. The top row varies batch size at sequence length 256, and the bottom row varies sequence length at batch size 32. Parameter-related components remain nearly fixed as the workload grows, while activation memory increases substantially. Each component is recorded at its own peak and the stacked total therefore does not represent the overall peak reserved memory. Dashed lines mark nominal 24, 32, and 40 GB device capacities for reference.

Table 15: Absolute serving latency under direct execution for CoLA and LORO at 350M and 1B. TTFT measures prefill latency, and TPOT measures per-token decoding latency after the first generated token.

		TTFT (ms)			TPOT (ms)		
Scale	Batch	Full-size	CoLA	LORO	Full-size	CoLA	LORO
350M	1	15.0	19.4	23.2	15.1	20.3	23.0
	4	18.0	19.9	25.1	15.5	19.7	25.0
	16	59.8	49.8	57.8	16.0	19.4	25.2
	64	220.1	177.9	214.8	18.9	20.9	28.0
	128	432.4	345.1	414.5	47.8	48.1	48.5
	256	861.5	682.7	822.8	198.0	198.3	198.6
1B	1	24.7	19.8	25.6	14.9	19.0	24.6
	4	78.9	39.0	45.3	15.9	19.9	24.0
	16	275.8	133.8	162.1	16.2	19.9	24.1
	64	1113.9	498.5	607.5	49.5	48.1	49.1

Projection structure. Let $d = Hd_h$, where H is the number of attention heads and d_h is the dimension of each head. The value projection first computes a latent representation $\boldsymbol{z}_j = \phi_V(\boldsymbol{x}_j \boldsymbol{D}_V) \in \mathbb{R}^{r_V}$ and then expands it as $\boldsymbol{v}_j = \boldsymbol{z}_j \boldsymbol{U}^V \in \mathbb{R}^d$, with $\boldsymbol{D}_V \in \mathbb{R}^{d \times r_V}$ and $\boldsymbol{U}^V \in \mathbb{R}^{r_V \times d}$. For a linear factorization, ϕ_V is the identity; for CoLA, it is the bottleneck nonlinearity. Let $\boldsymbol{U}_h^V \in \mathbb{R}^{r_V \times d_h}$ denote the columns of $\boldsymbol{U}^V$ corresponding to head h, so that $\boldsymbol{v}_{j,h} = \boldsymbol{z}_j \boldsymbol{U}_h^V$.

The output projection is written as $\boldsymbol{y}_i = \phi_O(\boldsymbol{o}_i \boldsymbol{D}_O)\boldsymbol{U}_O$, where $\boldsymbol{D}_O \in \mathbb{R}^{d \times r_O}$, $\boldsymbol{U}_O \in \mathbb{R}^{r_O \times d}$, and $\boldsymbol{o}_i$ is the concatenation of the head outputs $\boldsymbol{o}_{i,h}$. Let $\boldsymbol{D}_h^O \in \mathbb{R}^{d_h \times r_O}$ denote the rows of $\boldsymbol{D}_O$ corresponding to head h. Then $\boldsymbol{o}_i \boldsymbol{D}_O = \sum_h \boldsymbol{o}_{i,h} \boldsymbol{D}_h^O$.

Absorption. For head h, the attention output is $\boldsymbol{o}_{i,h} = \sum_{j \le T} a_h(i,j) \boldsymbol{v}_{j,h}$. Substituting the factorized value projection gives

$$\boldsymbol{o}_i \boldsymbol{D}_O = \sum_{h=1}^{H} \Big(\sum_{j \le T} a_h(i,j)\, \boldsymbol{z}_j \Big) \boldsymbol{G}_h, \qquad \boldsymbol{G}_h = \boldsymbol{U}_h^V \boldsymbol{D}_h^O \in \mathbb{R}^{r_V \times r_O}. \tag{3}$$

The matrices $\boldsymbol{G}_h$ depend only on the trained checkpoint and are computed once before serving. During decoding, the model therefore caches the latent values $\boldsymbol{z}_j$ instead of the expanded values $\boldsymbol{v}_j$, aggregates them using the attention weights of each head, and then applies $\boldsymbol{G}_h$. The attention weights, key path, ϕ_O, and $\boldsymbol{U}_O$ are unchanged. Setting $r_V = r_O = r$ recovers Eq. (1).

Arithmetic and cache storage. Consider one layer, one sequence, and one decoding query with context length T, and omit operations shared by the two execution paths. Direct execution expands the latent value, aggregates values in the d-dimensional space, and applies the first output factor, giving a cost of $O(Td + dr_V + dr_O)$. After absorption, attention aggregation is performed separately for each head in the r_V-dimensional latent space, followed by multiplication with $\boldsymbol{G}_h$, giving $O(HTr_V + Hr_V r_O)$.

The attention-value aggregation therefore requires Hr_V/d times as much arithmetic as direct execution. For the configurations used in our serving experiments, this factor is 4 at 350M ($H = 16$, $r_V = 256$, $d = 1024$) and 8 at 1B ($H = 32$, $r_V = 512$, $d = 2048$). The trade-off is a smaller value cache. For batch size b, direct execution stores bTd value entries per layer, whereas the absorbed path stores bTr_V, reducing the value cache by a factor of $d/r_V = 4$ at both scales. Since the key cache is unchanged, the total KV cache is reduced by a factor of $2d/(d + r_V) = 1.6$ when $r_V = d/4$.

The absorbed matrices add $Hr_V r_O$ parameters per layer, independent of batch size and context length. V/O absorption therefore exchanges additional latent-space arithmetic for lower value-cache storage and memory traffic. The benefit grows as KV-cache traffic becomes a larger part of decoding time, matching the large-batch behavior observed in Sec. 6.

Scope. V/O absorption applies to the value and output projections because the corresponding factors are adjacent linear maps. On the key path, rotary position embeddings (Su et al., 2024) are applied between the key up-projection and the attention dot product, preventing the same rearrangement. We therefore use absorption only for the V/O path. We also keep direct execution for prefill, where absorption provides no measurable latency benefit in our experiments. The latent values produced during factorized prefill are retained for subsequent decoding, so switching to the absorbed decoding path requires no additional conversion of the prompt cache.

E.3 Complete absorption scan

Tab. 16 reports all measured configurations of V/O absorption. At 350M and batch 64, absorption reduces decoding speed to 0.84–0.90× that of direct execution. The trade-off reverses at larger batches: absorption is 1.6× faster at batch 128 and 2.0–2.1× faster at batch 256. At 1B, it is already 1.5–1.6× faster at batch 64 and 2.1–2.2× faster at batch 128. Peak generation memory decreases by 30–36% across all measured configurations, including 350M at batch 64 where absorption increases latency. The memory reduction therefore appears before the latency crossover.

F Extended Related Work

Benchmarking training efficiency and low-rank pretraining. Comparisons of training algorithms can change with the evaluation protocol, tuning budget, and cost metric (Schmidt et al., 2021; Dahl et al., 2023; Kaddour et al., 2023; Dehghani et al., 2022). AlgoPerf (Dahl et al., 2023), for example, fixes the hardware and compares algorithms by the time required to reach a target, while Kaddour et al. (2023) show that apparent gains can change under a fixed compute budget and a fully decayed

Table 16: Decoding latency and generation memory with V/O absorption. "vs. direct" compares absorbed and direct execution of the same model using paired measurements from the same profiling session; "vs. Full-size" compares absorbed decoding with Full-size at the same batch size. Direct TPOT values can therefore differ slightly from the independently profiled values in Fig. 4.

			TPOT				Generation memory (GiB)		
Scale	Batch	Method	Direct	Absorbed	vs. direct	vs. Full-size	Direct	Absorbed	Change
350M	64	Vanilla LR	23.4	26.1	0.90×	0.72×	12.1	8.5	−30%
		CoLA	20.8	24.8	0.84×	0.76×	12.1	8.4	−31%
		LORO	23.6	26.5	0.89×	0.71×	12.1	8.5	−30%
	128	Vanilla LR	48.3	30.0	1.61×	1.59×	23.8	16.5	−31%
		CoLA	48.0	29.7	1.62×	1.61×	23.8	16.5	−31%
		LORO	48.4	30.1	1.61×	1.59×	23.8	16.5	−31%
	256	Vanilla LR	197.9	97.2	2.04×	2.04×	47.2	32.6	−31%
		CoLA	197.9	92.8	2.13×	2.13×	47.2	32.6	−31%
		LORO	198.2	96.9	2.05×	2.04×	47.2	32.6	−31%
1B	64	Vanilla LR	48.4	32.4	1.49×	1.53×	23.7	15.7	−34%
		CoLA	48.2	31.0	1.55×	1.60×	23.7	15.4	−35%
		LORO	48.2	31.5	1.53×	1.57×	23.7	15.7	−34%
	128	Vanilla LR	204.8	94.4	2.17×	2.16×	46.2	29.9	−35%
		CoLA	204.4	95.7	2.14×	2.13×	46.2	29.5	−36%
		LORO	204.4	92.9	2.20×	2.19×	46.2	29.9	−35%

baseline. The Efficiency Misnomer (Dehghani et al., 2022) further shows that parameter count, FLOPs, and throughput need not lead to the same efficiency comparison. The closest benchmark to LR-EPB is Glentis et al. (2025), which evaluates parameter- and memory-efficient pretraining methods on C4 from 60M to 1B parameters and finds that final perplexity is largely explained by training FLOPs. LR-EPB evaluates both matched-token and matched-FLOPs budgets and additionally measures end-to-end memory, throughput, and serving latency under a common implementation and hardware platform. Wen et al. (2026) study pretraining optimizers under careful tuning, while concurrent work (Shivagunde et al., 2026) compares the solutions reached by several low-rank methods through loss landscapes, weight spectra, and activation similarity.

Low-rank pretraining methods. Low-rank weight methods directly factorize model weights, sometimes together with additional sparse components or nonlinear bottlenecks (Han et al., 2024; Liu et al., 2025; Mo et al., 2025; Lv et al., 2024; Li et al., 2025; Janson et al., 2026). Optimizer-state methods retain dense model weights while reducing optimizer states through projected gradients, low-dimensional subspaces, or structured approximations (Zhao et al., 2024; Chen et al., 2026; He et al., 2024; Gong et al., 2025; Zhu et al., 2025; Muhamed et al., 2024; Hao et al., 2024; Liang et al., 2024; Robert et al., 2025; Zmushko et al., 2025; Zhang et al., 2026; Xiao et al., 2025; Rajabi et al., 2026; Nguyen et al., 2025). Low-rank update methods train low-rank increments that are periodically merged into dense weights (Lialin et al., 2024; Zhou et al., 2024; Loeschcke et al., 2024). SinkGD (Scetbon et al., 2026) provides a complementary stateless reference by removing optimizer states without imposing low-rank structure.

Latent attention and weight absorption. Multi-head latent attention (Liu et al., 2024) reduces inference cost by caching compressed latent representations and absorbing adjacent projections. TransMLA (Meng et al., 2025) and MHA2MLA (Ji et al., 2025) convert pretrained dense-attention models into latent-attention forms and use additional fine-tuning to recover performance. Low-rank weight checkpoints already contain factorized value and output projections, enabling the V/O absorption studied in Sec. 6 without modifying the trained model.