CoLA: Compute-Efficient Pre-Training of LLMs via Low-Rank Activation

Anonymous ACL submission

Abstract

Large language models (LLMs) are revolutionizing many science and engineering fields. However, their huge model sizes impose extremely demanding needs of computational resources in the pre-training stage. Although low-rank factorizations can reduce model parameters, their direct application in LLM pre-training often lead to non-negligible performance loss. To address this fundamental challenge, we introduce **CoLA** and its memory-efficient implementation, **CoLA-M**. We leverage the low-rank structure observed widely in model activations, enforcing non-linear transformations between factorized weight matrices to reduce model size, boost model capacity and training efficiency. Experiments on LLaMA models with 60 million to 7 billion parameters show that CoLA reduces the computing cost by $\mathbf{2\times}$ and improves training throughput by $\mathbf{1.86\times}$ while maintaining full-rank level performance. CoLA-M further squeezes memory cost without sacrificing throughput, offering a pre-training approach with collectively superior parameter, computing, and memory efficiency. The LLMs produced are also $\mathbf{2\times}$ smaller, enabling faster inference with lower memory cost on resource-constrained platforms. [1]

1 Introduction

Large foundation models have revolutionized the landscape of artificial intelligence, achieving unprecedented success in the language, vision, and scientific domains. In a quest to improve accuracy and capability, foundation models have become huge. Several studies (Kaplan et al., 2020; Hoffmann et al., 2022; Krajewski et al., 2024; Kumar et al., 2024) have highlighted a rapid increase in the size of the model and the number of training tokens. Models such as 175B GPT-3 (Brown et al., 2020), 405B LLaMA-3 (Dubey et al., 2024), and

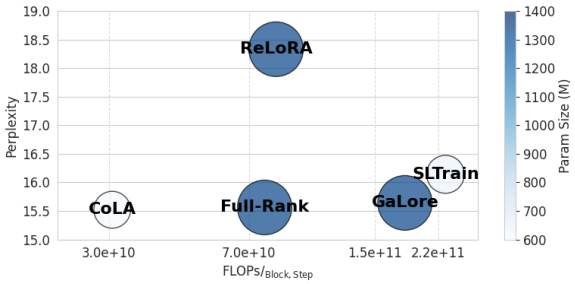

Figure 1: A joint comparison of validation perplexity, estimated compute FLOPs (per block, step) and model size between various pre-training methods on a LLaMA-1B model with a token batch size of 256. Among them, our proposed CoLA is the only one that reduces both compute FLOPs and model size while demonstrating on par validation perplexity with full-rank training.

540B PaLM (Chowdhery et al., 2023) are just a few examples of this trend. Under such circumstances, a large number of GPUs are needed in order to provide the computational and high-bandwidth memory capacity needed to pre-train large fundation models over long periods of time (months). The staggering increase in cost results in an unsustainable trend, prompting the need to develop cost-efficient pre-training techniques that reduce the scale, FLOPs, and GPU memory cost.

Motivation: At the core of increasing resource utilization and cost is the simple practice of scaling up full-size linear layers in decoder-only architectures, which has proven to be a viable and straightforward strategy. Thus, to break free from this unsustainable trend, it is imperative to improve architecture efficiency. This has been widely studied in the deep learning community, involving different levels of factorization of weight matrices: from simple matrix factorizations, i.e., a singular value decomposition (SVD), to higher-order tensor factorizations such as Canonical Polyadic, Tucker, and Tensor-Train (TT) format. Extensive studies have shown that such factorizations can effectively reduce the total number of parameters needed to

[*] Equal contribution
[1] Code available here.

achieve similar performance in numerous domains (Sainath et al., 2013; Jaderberg et al., 2014; Lebedev et al., 2014; Novikov et al., 2015; Tjandra et al., 2017; Dao et al., 2021; Sui et al., 2024; Yang et al., 2024; Zhang et al., 2024), especially when neural networks are overparameterized.

Limitations of state-of-art: The techniques mentioned above have been applied only to a limited degree to pre-training tasks, and their findings suggest that the pure low-rank or sparse structure often downgrades model performance (Khodak et al., 2021; Kamalakara et al., 2022; Chekalina et al., 2023; Zhao et al., 2024; Hu et al., 2024; Mozaffari et al., 2024). This has pivoted most recent work of efficient pre-training into two directions: 1) Accumulating multiple low-rank updates (Huh et al., 2024; Lialin et al., 2023); 2) Enforcing low-rank structures in gradients rather than parameters (Zhao et al., 2024; Chen et al., 2024; Huang et al.; Liao et al., 2024; Hao et al., 2024; Zhu et al., 2024). Both approaches have their limitations. 1) The accumulation of low-rank updates requires instantiating a full-rank matrix and a deeply customized training strategy that periodically merges and restarts the low-rank components. This creates computing overhead in practice and can only achieve (if only) marginal computing and memory reduction. 2) Enforcing low-rank gradients reduces only the optimizer memory and adds additional computation that downgrades training throughput. Furthermore, the memory saving caused by gradient compression becomes negligible as the training batch size increases, as activations dominate the total memory cost. A recent paper called SLTrain (Han et al., 2024) revisited the notion of parameter efficiency in foundation model pre-training, by having both low-rank factors and an unstructured sparse matrix. SLTrain effectively reduces the total number of parameters without significantly hurting model performance. However, it still introduces computing overhead on top of full-rank training due to the necessary reconstruction of low-rank factors. We note that none of the above works has achieved superior efficiency of **parameter**, **computing**, and **memory** simultaneously **without performance drop** in both **training** and **inference** for foundation model pre-training.

Contributions: In this paper, we propose **CoLA**: **Co**mpute-Efficient Pre-Training of LLMs via **L**ow-rank **A**ctivation, and its memory efficient implementation **CoLA-M**, to achieve all the desirable properties mentioned above. We summarize our contributions as follows:

		CoLA(-M)	SLTrain	GaLore	ReLoRA
Parameter ↓		✓	✓	×	×
Compute ↓	Training	✓	×	×	✓
	Inference	✓	×	×	×
Memory ↓	Training	✓	✓	✓	✓
	Inference	✓	✓	×	×
Throughput ↑	Training	✓	×	×	×
	Inference	✓	×	×	×

Table 1: Summary and comparison of different types of efficiency across various pre-training methods.

- We propose **CoLA**, a novel architecture that enforces explicit low-rank activations by injecting non-linear operations between factorized weight matrices. CoLA can greatly reduce the computing FLOPS while maintaining the performance of full-rank pre-training.
- We provide a memory efficient implementation, namely **CoLA-M**, to achieve superior memory reduction without sacrificing throughput.
- We extensively pre-train LLaMA with 60M up to 7B parameters. CoLA reduces model size and computing FLOPs by $\mathbf{2\times}$, while maintaining on-par performance to its full-rank counterpart. At the system level, CoLA improves $\mathbf{1.86\times}$ training and $\mathbf{1.64\times}$ inference throughput. CoLA-M reduces total pre-training memory by $\mathbf{2/3}$, while still manages to improve $\mathbf{1.3\times}$ training throughput over full-rank baselines.

A high-level comparison of CoLA/CoLA-M with main baselines is provided in Table 1.

2 Related Work

Model Compression. Recent research on efficient LLM pre-training primarily focuses on memory savings. To out best knowledge, SLTrain (Han et al., 2024) is the first method that reduces both trainable parameters and total parameters in LLM pre-training, without significantly hurting model performance. This also reduces memory usage for model, gradients, and optimizer states (see its smaller circle in Fig. 1). However, the existence of its unstructured sparse matrix $\mathbf{S}$ requires reconstructing $\tilde{\mathbf{W}} = \mathbf{BA} + \mathbf{S}$, otherwise it will incur dense-sparse multiplications that are still memory costly (Fig. 3c). This causes additional computing than the full-rank baseline. LoRA/ReLoRA (Hu et al., 2021; Lialin et al., 2023) reduces trainable parameters by freezing a full-rank $\mathbf{W}_0$ and training (at least in a later stage) only low-rank

factors, potentially reducing memory needs. Yet, any compute savings are limited because the forward pass yields a larger compute than its full-rank counterpart, especially when the rank must stay relatively large in pre-training tasks. CoMERA (Yang et al., 2024) achieves higher model compression and FLOP reduction, but its low-rank tensor operations are GPU unfriendly. Similar to matrix-compressed approaches, CoMERA cannot avoid a performance drop either. Some works investigate pure structured sparsity or combined with low-rank factors (Hu et al., 2024; Mozaffari et al., 2024) to achieve speed up, but still show a significant performance drop during the pre-training stage.

Gradient Compression. GaLore (Zhao et al., 2024) reduces memory by projecting gradients into a low-rank space, shrinking optimizer states below the typical $2\times$ AdamW overhead (Loshchilov, 2017). However, it increases computation by adding up/down projections on top of already compute-heavy full-rank training. As shown in Fig. 1, its estimated FLOPs surpass full-rank training on the LLaMA-1B scale. Follow-up work (Chen et al., 2024; Huang et al.; Liao et al., 2024; Hao et al., 2024; Zhu et al., 2024) further explores low-rank gradient projection. While these methods are promising, they are mostly orthogonal to our focus. Crucially, these methods are still computing lower-bounded by the full-rank baseline. Our goal instead is to reduce computing cost to a fraction of full-rank training, thus lowering the demand of computing resources in LLM pre-training.

This paper presents an alternative approach that explores the low-rank property in model **activations** from an architectural perspective. This is conceptually different from the above model compression methods despite the similarity in their formulations. Our approach is mostly orthogonal with gradient compression techniques, meaning that they can be combined to further boost efficiency.

3 CoLA for Efficient LLM Pre-Training

3.1 A Motivating Example

Many previous works have observed the low-rank structure of model activations in deep neural networks (Cui et al., 2020; Huh et al., 2021). We also observe this phenomenon in LLMs, i.e. the *effective rank* of the activations is much smaller than their original dimensionality. To quantify this, we define the *effective rank* $r(\alpha)$ of activation as the minimal number of singular values needed to

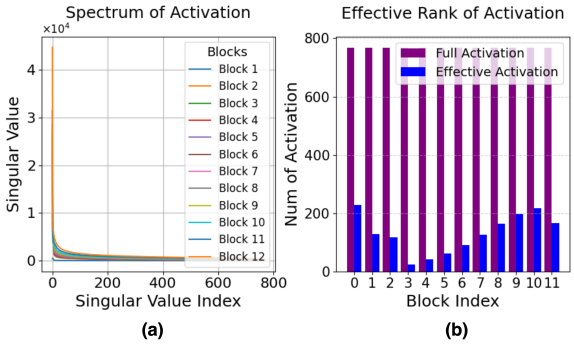

Figure 2: MLP Activation Spectrum of the pre-trained GPT-2 small (Radford et al., 2019). Model activations are evaluated on the WikiText2 dataset. a) The singular value decay across different decoder blocks. b) The full dimension vs. effective rank ($\alpha = 0.95$) per block.

preserve an α-fraction of the total spectral energy. Formally:

$$r(\alpha) \;=\; \min\left\{ k \;\middle|\; \frac{\sum_{i=1}^{k}\sigma_i^2}{\sum_{i=1}^{n}\sigma_i^2} \;\geq\; \alpha \right\}, \qquad (1)$$

where $\sigma_1, \sigma_2, \ldots, \sigma_n$ are the singular values of the activation matrix, and $0 < \alpha \leq 1$ is the desired ratio of preserved information. As shown in our experiments, the rapid decay of singular values [Fig. 2a] leads to much smaller $r(\alpha)$ compared to the full dimension [Fig. 2b]. This highlights the significant low-rank nature in the activations of pre-trained LLMs. More results showing the same pattern can be found in Appendix A.

3.2 Low Rank Weights + Activations

Motivated by the observed low-rank nature of LLM activations, we propose to enforce explicit low-rank activation via injecting non-linearity between factorized weight matrices.

Let $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ be the weight matrix of an arbitrary linear layer followed by a nonlinear activation in the transformer architecture:

$$\mathbf{h} = \sigma\left(\mathbf{W}\mathbf{x}\right), \text{ with } \mathbf{x} \in \mathbb{R}^{d_{\text{in}}}. \qquad (2)$$

We replace $\mathbf{W}$ by two low-rank matrices $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$ and $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$, where rank $r < \min(d_{\text{in,out}})$ is a hyper-parameter, and inject a non-linear activation σ in the middle. This modification results in a transformation consisting of (linear $\circ$ non-linear $\circ$ linear) operations:

$$\mathbf{h}' = \mathbf{B}\,\sigma(\mathbf{A}\mathbf{x}). \qquad (3)$$

During the backward step, we simply apply the chain rule to compute gradients w.r.t each of the

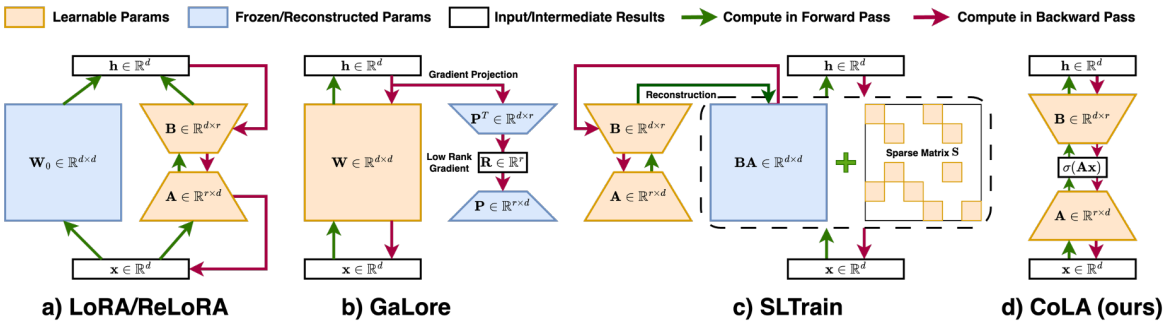

Figure 3: Comparison between CoLA (ours) and other efficient pre-training frameworks. a) LoRA/ReLoRA (Lialin et al., 2023) maintains a full-rank frozen weight; b) GaLore (Zhao et al., 2024) only reduces optimizer states by down and up projecting gradients; c) SLTrain (Han et al., 2024) requires necessary reconstruction of the low-rank and sparse matrices; d) CoLA (ours) is a pure low-rank method involves only rank r weight matrices.

Operation	FLOPs
Attention: Q, K, V	$6nd^2$
Attention: SDP	$4n^2d$
Attention: Project	$2nd^2$
Feed-forward	$6nd\text{d}_{\text{ff}}$
Total Forward	$8nd^2 + 4n^2d + 6nd\text{d}_{\text{ff}}$
Total Backward	$16nd^2 + 8n^2d + 12nd\text{d}_{\text{ff}}$

Table 2: Breakdown compute of a single LLaMA decoder layer in full-rank training. Lower-order terms such as bias, layer norm, activation are omitted.

Methods	FLOPs
Full-Rank	$C_{\text{Full-Rank}} = 24nd^2 + 12n^2d + 18nd\text{d}_{\text{ff}}$
CoLA	$C_{\text{CoLA}} = 48ndr + 12n^2d + 18nr(d + \text{d}_{\text{ff}})$
(Re)LoRA	$C_{\text{LoRA}} = C_{\text{CoLA}} + 16nd^2 + 12n^2d + 12nd\text{d}_{\text{ff}}$
SLTrain	$C_{\text{SLTrain}} = C_{\text{Full-Rank}} + 24d^2r + 18d\text{d}_{\text{ff}}r$
GaLore	$C_{\text{GaLore}} = C_{\text{Full-Rank}} + 16d^2r + 12d\text{d}_{\text{ff}}r$

Table 3: Estimated compute of a single LLaMA decoder layer for different pre-training methods. Results combine forward, backward and any additional compute occurred at optimizer step.

low rank factors as

$$\nabla_{\mathbf{B}} = \nabla_{\mathbf{h}'}\mathbf{z}^T, \nabla_{\mathbf{z}} = \mathbf{B}^T\nabla_{\mathbf{h}'}, \nabla_{\mathbf{o}} = \nabla_{\mathbf{z}} \odot \sigma'(\mathbf{o}),$$
$$\nabla_{\mathbf{A}} = \nabla_{\mathbf{o}}\mathbf{x}^T, \nabla_{\mathbf{x}} = \mathbf{A}^T\nabla_{\mathbf{o}}, \tag{4}$$

where $\mathbf{o} = \mathbf{A}\mathbf{x}, \mathbf{z} = \sigma(\mathbf{o})$, $\odot$ denote the element-wise product. We empirically find that keeping the original nonlinearity on top of Eq. (3) does not harm the performance, nor necessarily brings benefits. However, applying Eq. (3) to all linear layers regardless of whether being followed by nonlinearity is crucial to boost model performance. We refer more details to the ablation study in Appendix E.

Fig. 4 shows the architecture of each transformer block when adopting CoLA into the LLaMA architecture. We highlight the fact that only the original linear layers and (if any) their follow-up non-linear transformation are modified to the CoLA formulation. Other computations such as the scaled-dot product of the self-attention, as well as residual connections and the element-wise product of LLaMA's feed-forward layers, remain unchanged.

3.3 Computing Efficiency

We analyze and compare the computational complexity of CoLA with other efficient pre-training methods based on the LLaMA architecture. We adopt a similar notion from (Kaplan et al., 2020), where a general matrix multiply (GEMM) between an $M \times N$ matrix and an $N \times K$ matrix involves roughly $2MNK$ add-multiply operations. We denote the model inner width as d, and the inner width of the feed-forward layer as d_{ff}. For simplicity, we only show non-embedding calculations of a single sequence with token batch size of n for each decoder layer. This is because the total computation scales only linearly with the number of layers n_{layer} and the number of sequences n_{seq}. Furthermore, lower-order cheap operations of complexity $\mathcal{O}(nd)$ or $\mathcal{O}(nd_{\text{ff}})$ are omitted, such as bias, layer norm, non-linear function, residual connection, and element-wise product.

We show the detailed cost of the full-rank training in Table. 2. Notice that we apply the $2\times$ rule when calculating the backward cost. This is because for each forward GEMM that Eq. (2) describes, two GEMMs are needed to compute gradients for both the weight matrix $\mathbf{W}$ and the input $\mathbf{x}$, and are of the same cost the forward GEMM, i.e.,

$$\nabla_{\mathbf{x}} = \mathbf{W}^T\nabla_{\mathbf{h}}, \nabla_{\mathbf{W}} = \nabla_{\mathbf{h}}\mathbf{x}^T. \tag{5}$$

We apply the same analysis to all the following

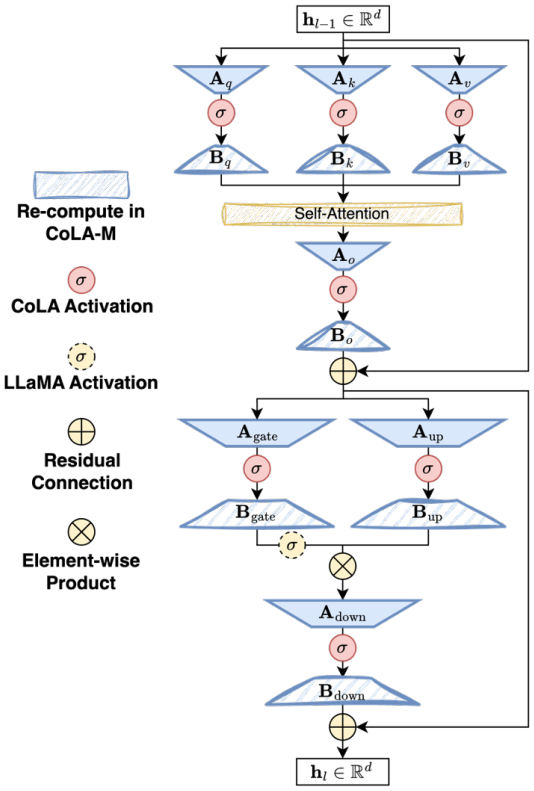

Figure 4: A decoder block in CoLA with LLaMA-like architecture (layer norms, rotary positional embeddings are omitted for simplicity). Every linear layer is replaced by low-rank factorization with a non-linear function in between. Modules painted in sketch are the re-computations during the backward step of CoLA-M (a memory efficient implementation of CoLA).

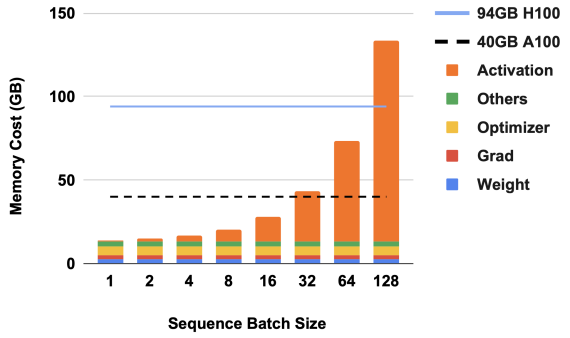

Figure 5: Memory breakdown for LLaMA-1B using fairly large sequence batch sizes in pre-training. The activation memory is at dominant place.

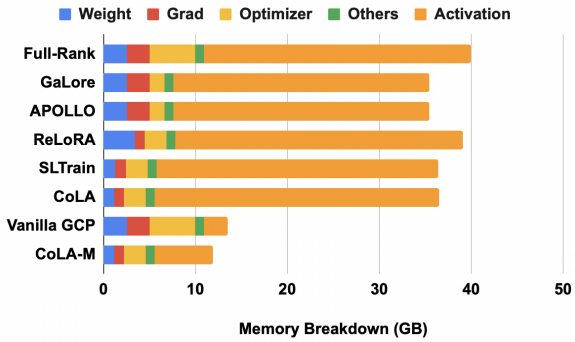

Figure 6: Memory breakdown of pre-training LLaMA-1B on single GPU using different pre-training methods. Activation is at dominance for most methods.

pre-training methods:

- **LoRA/ReLoRA** (Hu et al., 2021; Lialin et al., 2023): $\mathbf{h}_{\text{LoRA}} = \mathbf{W}_0\mathbf{x} + \mathbf{BAx}$, with fixed $\mathbf{W}_0$.
- **SLTrain** (Han et al., 2024): $\mathbf{h}_{\text{SLTrain}} = \mathbf{BAx} + \mathbf{Sx} = (\mathbf{BA} \oplus_{\mathcal{I}} \mathcal{V})\mathbf{x}$, where $\oplus$ denotes the scatter-add operator, $\mathcal{I}$ and $\mathcal{V}$ are the indices and values of non-zero elements in the sparse matrix $\mathbf{S}$.
- **GaLore** (Zhao et al., 2024): $\mathbf{R}_t = \mathbf{P}_t^T\mathbf{G}_t$, $\tilde{\mathbf{G}}_t = \mathbf{PN}_t$, where $\mathbf{P}_t$ projects the gradient $\mathbf{G}_t$ onto a low-rank space, and then projects it back when updating the full-rank weight $\mathbf{W}$.

We summarize the computational costs of these methods in Table 3 and observe that the costs of SLTrain and GaLore are lower bounded by full-rank training, while (Re)LoRA is lower bounded by CoLA when choosing the same rank. In contrast, CoLA reduces the computation from full-rank training when $r < 0.62d$, assuming $d_{\text{ff}} \approx 2.5d$ in LLaMA-like architecture. The default rank choice is set to $r = \frac{1}{4}d$, leading to a reduction in compute to about half of the full-rank training. We refer all details of compute analysis to Appendix B.

4 CoLA-M: A Memory-Efficient Implementation

Although CoLA has more intermediate results from each low-rank projection and the following non-linear function (as shown in Fig. 4), we can choose strategically which ones to save in order to balance re-computations with memory overhead. In this section, we design and develop CoLA-M, a memory-efficient implementation to leverage CoLA's structural advantage to achieve superior memory saving without sacrificing throughput.

4.1 Memory Breakdown in Pre-Training

We assume a common notion that training modern transformers with Adam (or AdamW) involves four key memory components (Zhao et al., 2024; Han et al., 2024): model parameters, gradients, optimizer states, and activations (intermediate forward-pass results). Gradients consume $1\times$ model size, Adam consumes $2\times$, and activations typically consume $1 \sim 4\times$, depending on the batch size.

Methods	Memory	Re-Compute
Full-Rank	$20nd + 2n^2h$	N/A
Vanilla GCP	nd	$23nd^2 + 4n^2d$
CoLA	$17.5nd + 2n^2h + 14nr$	N/A
CoLA-M	$2nd + 7nr$	$18.5ndr + 4n^2d$

Table 4: Memory and re-computation analysis of full-rank training with vanilla GCP vs. CoLA and CoLA-M.

Methods	Memory Saving	Re-Compute
LLAMA-1B w/ vanilla GCP	20.25 GB	1×
CoLA-M-1B	18.94 GB	0.22×

Table 5: Quantitative comparison of memory saving and re-computation between vanilla GCP and CoLA-M.

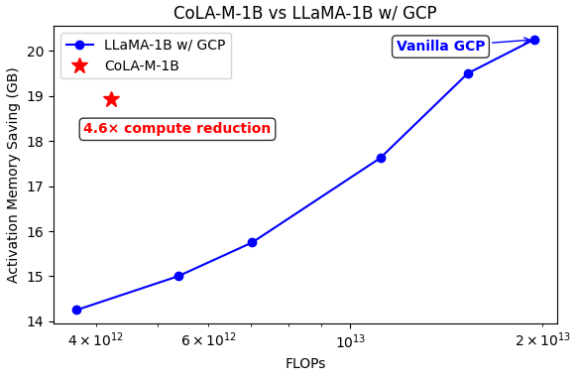

Figure 7: We show how memory reduction scales with the re-computation in full-rank training with GCP and compare with CoLA-M. With similar gains on memory efficiency, CoLA-M effectively reduces re-compute by $4.6\times$, enabling compute efficient checkpointing.

We focus on the scenario where the memory cost determined by the model size is not in the extreme limit of the GPU. We argue that this is rather realistic, since the model size and the minimum required tokens should scale up simultaneously during pre-training (Kaplan et al., 2020; Hoffmann et al., 2022; Krajewski et al., 2024; Kumar et al., 2024). A tiny batch size on a single GPU would be impractical. Therefore, we analyze memory usage on a 40-GB A100 or a 94-GB H100 GPU with a fairly large sequence batch size. Fig. 5 shows that activations dominate memory usage in this setup. Although our method overall consumes less memory than full-rank training (largely due to parameter efficiency), vanilla CoLA allocates more memory to activations. However, we argue that our unique architecture can significantly enhance existing memory-saving techniques and consume only a fraction of the original cost.

4.2 CoLA Enables Efficient Checkpointing

Gradient checkpointing (GCP) (Chen et al., 2016) is a system-level technique that reduces memory usage by selectively storing ("checkpointing") only a subset of intermediate activations during the forward pass. When the backward pass begins, the missing activations are recomputed on the fly instead of being stored in memory, thereby lowering the memory cost. A vanilla (also the most effective) implementation of GCP in LLM pre-training is to save merely the input and output of each transformer block, and re-compute everything within each block during the backward step. Some works have investigated the optimal selection of checkpoints through both empirical and compiler view (Feng and Huang, 2021; He and Yu, 2023). Such techniques can also be developed for CoLA, and are beyond the scope of this paper.

Motivated by the bottleneck structure of CoLA, we implement CoLA-M as saving only the low-rank activations (red circles in Fig. 4), and re-compute the up projections, and (if applicable) the self-attention (painted in sketch in Fig. 4) during the backward pass. This reduces the re-computation cost to half of the CoLA forward. We analyze the memory and re-computation cost using the same notions as in Section 3.3 and denote h as the number of attention heads. We further simplify the analysis under LLaMA architecture by uniformly assuming $d_{\text{ff}} \approx 2.5d$. The memory and re-computation overhead are shown in Table 4. We refer the detailed analysis to Appendix C.

Although delicate optimizations of GCP is beyond our scope, we show in Table 5 and in Fig. 7 the quantitative results and scaling behavior of GCP on LLaMA-1B when applying a heuristic checkpointing strategy. We observe that CoLA-M greatly reduces re-computation cost while achieving similar memory saving as vanilla GCP.

5 Experiments

5.1 Pre-Training LLaMA on C4

We validate our proposed methods by extensively pre-training LLaMA-like LLMs from 60M to 7B scales. Experiments were performed on NVIDIA A100/H100 GPUs. We closely follow the experiment settings of (Zhao et al., 2024; Han et al., 2024), and directly compare CoLA with their reported results. We use the C4 dataset (Raffel et al.,

Table 6: Comparison across various efficient pre-training methods of validation perplexity (PPL (↓)), number of parameters in millions (Param), and the estimated memory usage (Mem) including model, gradient and optimizer states based on BF16 precision. We pre-train LLaMA models from 60M to 1B on the C4 dataset (Raffel et al., 2020) following the same setup and compare results directly against those reported in (Zhao et al., 2024; Han et al., 2024).

	60M			130M			350M			1B		
r / d	128 / 512			256 / 768			256 / 1024			512 / 2048		
Tokens	1.1B			2.2B			6.4B			13.1B		
	PPL	Param (M)	Mem (GB)	PPL	Param (M)	Mem (GB)	PPL	Param (M)	Mem (GB)	PPL	Param (M)	Mem (GB)
Full-rank	34.06	58	0.43	**24.36**	134	1.00	**18.80**	368	2.74	15.56	1339	9.98
ReLoRA	37.04	58	0.37	29.37	134	0.86	29.08	368	1.94	18.33	1339	6.79
GaLore	34.88	58	0.36	25.36	134	0.79	18.95	368	1.90	15.64	1339	6.60
SLTrain	34.15	44	0.32	26.04	97	0.72	19.42	194	1.45	16.14	646	4.81
CoLA	**34.04**	**43**	**0.32**	24.48	**94**	**0.70**	19.40	**185**	**1.38**	**15.52**	**609**	**4.54**

	Mem (GB)	10k	40k	65k	80k
8-bit Adam	72.59	N/A	18.09	N/A	15.47
8-bit GaLore	65.16	26.87	17.94	N/A	15.39
SLTrain	60.91	27.59	N/A		
CoLA-M	**28.82**	**22.76**	**16.21**	**14.59**	**13.82**

Table 7: Validation perplexity of LLaMA-7B pre-trained on C4 dataset. 8-bit Adam/GaLore are collected from (Zhao et al., 2024). SLTrain is collected from (Han et al., 2024). No results of BF16 Adam reported. We emphasize that CoLA outperforms other methods at 150k steps (14.61 & 14.65) by using only 65k steps.

	60M		130M		350M	
	PPL	FLOPs	PPL	FLOPs	PPL	FLOPs
Full-Rank	34.06	1×	24.36	1×	18.80	1×
Control	37.73	0.4×	27.05	0.5×	20.53	0.4×
CoLA	34.04	0.4×	24.48	0.5×	19.40	0.4×
	31.52	0.7×	**23.97**	0.7×	**18.32**	0.7×

Table 8: Scaling behavior of CoLA and full-rank training. Control represents scaling down the full-rank training cost to be similar with CoLA in default, by reducing number of layers and/or size down model width.

2020), which is a colossal, cleaned version of Common Crawl's web crawl corpus. C4 dataset has been widely used for pre-training LLMs. Trainings were done without data repetition on a sufficiently large amount of tokens. We compare CoLA with baselines including **full-rank** pre-training, **ReLoRA** (Hu et al., 2021), **GaLore** (Zhao et al., 2024), and **SLTrain** (Han et al., 2024), with a focus on methods that explore model efficiency.

We implement CoLA and its memory efficient variant CoLA-M by parameterizing all linear layers into the proposed linear-nonlinear-linear composition [i.e. Eq. (3)], and keep all other parameters and operations unchanged. We use AdamW optimizer and cosine annealing learning rate scheduler (Loshchilov and Hutter, 2016) with warm-up. We remark that CoLA is NO more sensitive to optimizer-related hyper-parameters. We refer more details to Appendix D.

Table 6 compares our methods and other efficient pre-training techniques in terms of validation perplexity, parameter size, and estimated memory usage of model, gradients and optimizer states. CoLA has the smallest model size, thereby consumes the least memory, and demonstrates on-par validation perplexity compared to the full-rank baselines. Table 7 compares the validation perplexity on the 7B model. We highlight the fact that CoLA outperforms 8-bit Adam/GaLore at their 150k steps (14.61 & 14.65) using only 65k steps. Meanwhile the total memory cost of CoLA-M is less than half of the other methods. We also emphasize our comparison with GaLore and SLTrain. Our proposed method has an overall better performance, whilst further reduces model size on top of the reduction of SLTrain. More importantly, we follow our discussion in Section 3.3 and remark that CoLA has uniformly fewer computations than GaLore and SLTrain when applying the same rank.

5.2 Scaling Behavior

We briefly discuss how CoLA might scale differently compared to full-rank training. A comprehensive investigation of this topic is beyond the scope of this work due to the huge computing cost.

Table 8 shows a few experiments on how CoLA might be improved when computing is scaled up. The default rank choice reduces the compute cost to about a half of full-rank training, without significantly hurting the model performance. Meanwhile, if we relax the computing restriction and increase the rank to be greater than one quarter of d, then CoLA outperforms full-rank training in all three scales, while still being able to reduce the computing cost. One might argue that full-rank training can also be scaled down to a similar compute of

	1B (BZ = 64)			7B (BZ = 16)		
	Mem (GB)	Throughput	FLOPs	Mem (GB)	Throughput	FLOPs
Full-Rank	69.84	12,365	1×	84.94	5,810	1×
Vanilla GCP	**14.89**	8,799	1.68×	52.49	4,357	1.67×
CoLA	66.46	**22,979**	**0.40×**	55.52	**9,638**	**0.40×**
CoLA-M	17.33	16,617	0.55×	**26.82**	7,026	0.54×

Table 9: Detailed measurements and comparison of CoLA and CoLA-M against full-rank and vanilla GCP on a 94 GB H100 GPU. CoLA-M consumes only one third of the memory while achieving higher throughput than full-rank training with only about half its compute.

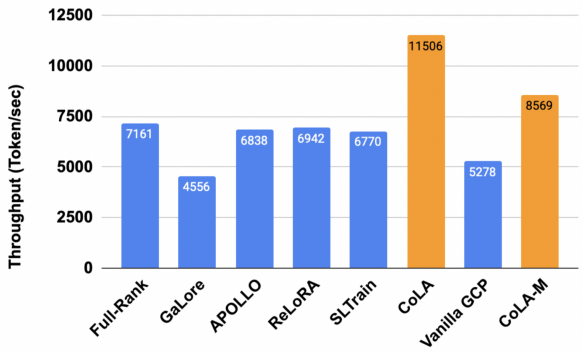

Figure 8: Comparison of throughput measured when pre-training a LLaMA-1B on a 40 GB A100 GPU with sequence batch size of 16 for different methods.

CoLA and might perform similarly. We implement such baselines in Table 8 and refer this setup to "Control". We typically reduce the number of layers or the model width of full-rank models to scale down their computing cost. We find empirically that they will tend to reduce performance quickly and dramatically underperform CoLA.

5.3 System Measurements

We further investigate the efficiency of CoLA from a more practical perspective. It is often observed that a theoretically less expensive method can have worse system-level performance due to poorly designed hardware implementation or lack of system-aware optimizations. We show that this is NOT the case for CoLA, by illustrating that its out-of-the-box system performance already significantly outperforms the full-rank training and other efficient training methods. We focus on the actual memory usage and the pre-training throughput.

Fig. 8 shows the measured throughput for pre-training the LLaMA/CoLA 1B model. The sequence batch size is set as 16, which fully utilizes the A100 GPU. Among all these methods, CoLA and CoLA-M are the only two that show higher GPU throughput than the full-rank baseline, while all other methods downgrade throughput due to their computing overhead. In particular, CoLA-M, the memory-efficient CoLA that significantly reduces overall GPU memory, still shows higher training throughput despite the recomputation overhead. Meanwhile, vanilla GCP, which uses a similar idea of trading compute for memory, reduces throughput from the full-rank baseline by 26%. We show further the details of these measurements in Table 9, and also compare their estimated FLOPs. At both 1B and 7B scale, CoLA-M manages to almost halve the computing cost and reduce the memory usage by two thirds, achieving a great balance between computing and memory efficiency. We refer more details of our system profiling to Appendix. F.

	1B (BZ=32)		7B (BZ=32)	
	Mem (GB)	Throughput	Mem (GB)	Throughput
Full-rank	5.74	21,109	18.15	11,086
SLTrain	4.18	20,096	12.70	9,968
CoLA	**3.84**	**34,697**	**10.87**	**16,012**

Table 10: Comparison of memory (GB) and throughput (Token/sec) at inference time on an A100 GPU.

5.4 Inference Performance

We highlight the fact that CoLA not only reduces pre-training resources but also speeds up inference and reduces its memory cost. Table 10 shows that CoLA improves inference throughput by up to **1.64×** while reducing memory cost by up to **1.67×**.

6 Conclusions

We propose CoLA, and its memory efficient variant CoLA-M, to achieve collectively parameter, compute and memory efficiency at both pre-training and inference time for large foundation models. CoLA effectively reduces 2× model size while preserving full-rank level performance. More importantly, we show the reduction does not come with additional compute. Instead, CoLA halves compute and almost doubles training throughput from its full-rank counterpart. When memory is of higher concerns, CoLA-M trades only minimum compute for state-of-the-art memory reduction during pre-training, meanwhile still reducing compute and improving throughput. We hope our work will inspire the community to further investigate the architecture efficiency that has been overlooked and under-discovered for large foundation models.

7 Limitations

This work limits the study of our proposed formulation under LLaMA-like architectures. The adaptation of CoLA to other architectures is conceptually trivial, but their performance is to be evaluated via real experiments. In this work, we only pre-train each model to be roughly compute-optimal (for original LLaMA models, not CoLA), while industry-produced LLMs (that are of similar scales) are often over-trained. It is worth investigating the performance of CoLA when significantly over-trained. We leave this computing-expensive research for future work.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Viktoriia Chekalina, Georgii Novikov, Julia Gusak, Ivan Oseledets, and Alexander Panchenko. 2023. Efficient gpt model pre-training using tensor train matrix representation. *arXiv preprint arXiv:2306.02697*.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Xi Chen, Kaituo Feng, Changsheng Li, Xunhao Lai, Xiangyu Yue, Ye Yuan, and Guoren Wang. 2024. Fira: Can we achieve full-rank training of llms under low-rank constraint? *arXiv preprint arXiv:2410.01623*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Chunfeng Cui, Kaiqi Zhang, Talgat Daulbaev, Julia Gusak, Ivan Oseledets, and Zheng Zhang. 2020. Active subspace of neural networks: Structural analysis and universal attacks. *SIAM Journal on Mathematics of Data Science*, 2(4):1096–1122.

Tri Dao, Beidi Chen, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. 2021. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Jianwei Feng and Dong Huang. 2021. Optimal gradient checkpoint search for arbitrary computation graphs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11433–11442.

Andi Han, Jiaxiang Li, Wei Huang, Mingyi Hong, Akiko Takeda, Pratik Jawanpuria, and Bamdev Mishra. 2024. Sltrain: a sparse plus low-rank approach for parameter and memory efficient pretraining. *arXiv preprint arXiv:2406.02214*.

Yongchang Hao, Yanshuai Cao, and Lili Mou. 2024. Flora: Low-rank adapters are secretly gradient compressors. *arXiv preprint arXiv:2402.03293*.

Horace He and Shangdi Yu. 2023. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. *Proceedings of Machine Learning and Systems*, 5:414–427.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Yuezhou Hu, Kang Zhao, Weiyu Huang, Jianfei Chen, and Jun Zhu. 2024. Accelerating transformer pre-training with 2: 4 sparsity. *arXiv preprint arXiv:2404.01847*.

Weihao Huang, Zhenyu Zhang, Yushun Zhang, Zhi-Quan Luo, Ruoyu Sun, and Zhangyang Wang. Galore-mini: Low rank gradient learning with fewer learning rates. In *NeurIPS 2024 Workshop on Fine-Tuning in Modern Machine Learning: Principles and Scalability*.

Minyoung Huh, Brian Cheung, Jeremy Bernstein, Phillip Isola, and Pulkit Agrawal. 2024. Training neural networks from scratch with parallel low-rank adapters. *arXiv preprint arXiv:2402.16828*.

Minyoung Huh, Hossein Mobahi, Richard Zhang, Brian Cheung, Pulkit Agrawal, and Phillip Isola. 2021. The low-rank simplicity bias in deep networks. *arXiv preprint arXiv:2103.10427*.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*.

Siddhartha Rao Kamalakara, Acyr Locatelli, Bharat Venkitesh, Jimmy Ba, Yarin Gal, and Aidan N Gomez. 2022. Exploring low rank training of deep neural networks. *arXiv preprint arXiv:2209.13569*.

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

Mikhail Khodak, Neil Tenenholtz, Lester Mackey, and Nicolo Fusi. 2021. Initialization and regularization of factorized neural layers. *arXiv preprint arXiv:2105.01029*.

Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, et al. 2024. Scaling laws for fine-grained mixture of experts. *arXiv preprint arXiv:2402.07871*.

Tanishq Kumar, Zachary Ankner, Benjamin F Spector, Blake Bordelon, Niklas Muennighoff, Mansheej Paul, Cengiz Pehlevan, Christopher Ré, and Aditi Raghunathan. 2024. Scaling laws for precision. *arXiv preprint arXiv:2411.04330*.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2014. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*.

Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. 2023. Relora: High-rank training through low-rank updates. In *The Twelfth International Conference on Learning Representations*.

Xutao Liao, Shaohui Li, Yuhui Xu, Zhi Li, Yu Liu, and You He. 2024. Galore +: Boosting low-rank adaptation for llms with cross-head projection. *arXiv preprint arXiv:2412.19820*.

I Loshchilov. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Mohammad Mozaffari, Amir Yazdanbakhsh, Zhao Zhang, and Maryam Mehri Dehnavi. 2024. Slope: Double-pruned sparse plus lazy low-rank adapter pretraining of llms. *arXiv preprint arXiv:2405.16325*.

Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. 2015. Tensorizing neural networks. *Advances in neural information processing systems*, 28.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

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

Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. 2013. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6655–6659. IEEE.

Yang Sui, Miao Yin, Yu Gong, Jinqi Xiao, Huy Phan, and Bo Yuan. 2024. Elrt: Efficient low-rank training for compact convolutional neural networks. *arXiv preprint arXiv:2401.10341*.

Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. 2017. Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4451–4458. IEEE.

Zi Yang, Ziyue Liu, Samridhi Choudhary, Xinfeng Xie, Cao Gao, Siegfried Kunzmann, and Zheng Zhang. 2024. Comera: Computing-and memory-efficient training via rank-adaptive tensor optimization. *arXiv preprint arXiv:2405.14377*.

Qiaozhe Zhang, Ruijie Zhang, Jun Sun, and Yingzhuang Liu. 2024. How sparse can we prune a deep network: A fundamental limit perspective. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*.

Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, Vikas Chandra, Bo Long, David Z Pan, Zhangyang Wang, and Jinwon Lee. 2024. Apollo: Sgd-like memory, adamw-level performance. *arXiv preprint arXiv:2412.05270*.

A Observation of Low-Rank Activation in Pre-Trained GPT2

In this section, we further show the low-rank structure in model activations evaluated on a pre-trained GPT-2 (Radford et al., 2019) small. The evaluation is conducted with sequence batch size of 64 and sequence length of 1024. We fix $\alpha = 0.95$ throughout this section. Similar patterns are observed from the attention layers (Fig. 9, 10, 11). The low-rank nature of activations is evident across all the different components of the model. This suggests that despite the high-dimensional representations, the effective dimensionality of the activations remains constrained.

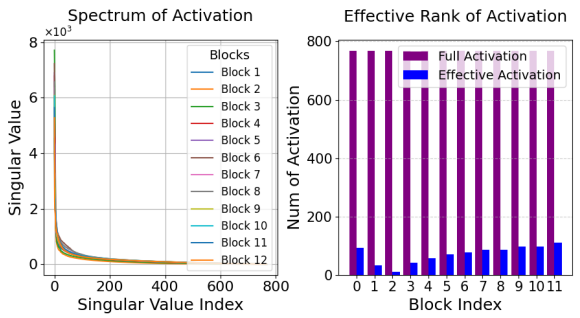

Figure 9: Activation Spectrum of Attention Layer (Q)

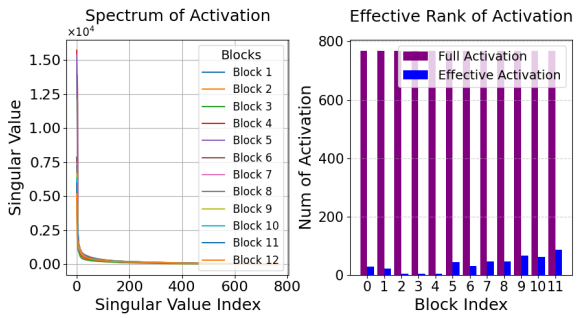

Figure 10: Activation Spectrum of Attention Layer (K)

B Detailed Compute Analysis

According to Table. 2, the total compute of full-rank training is simply combining forward and backward as

$$C_{\text{Full-Rank}} = 24nd^2 + 12n^2d + 18ndd_{\text{ff}}. \quad (6)$$

In our proposed architecture, every single linear layer is replaced by low rank matrices $\mathbf{A}$, $\mathbf{B}$, and an activation function sandwiched in between. The activation only introduces trivial compute thus can be omitted in the calculation. For each d^2 and dd_{ff} in Eq. (6), CoLA effectively converts them into $2dr$ and $r(d + d_{\text{ff}})$. Therefore the total compute of CoLA is

$$C_{\text{CoLA}} = 48ndr + 12n^2d + 18nr(d + d_{\text{ff}}). \quad (7)$$

Plugging in an actual setting of LLaMA/CoLA-1B, in which $r = \frac{1}{4}d$ and $r \approx \frac{1}{10}d_{\text{ff}}$, we achieve a compute reduction from Eq. (6) to approximately

$$C_{\text{CoLA-1B}} = 16.5nd^2 + 12n^2d + 1.8ndd_{\text{ff}}. \quad (8)$$

We now discuss and compare CoLA with other efficient pre-training methods in terms of their compute complexity. We start with LoRA (Hu et al., 2021) and ReLoRA (Lialin et al., 2023). They share the same architecture that's shown in Fig. 3

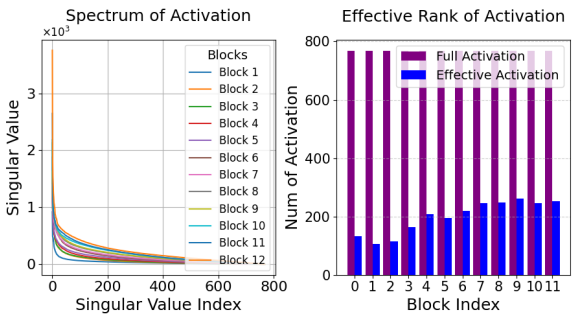

Figure 11: Activation Spectrum of Attention Layer (V)

a), in which low rank matrices $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$ and $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$ are adapted onto a full rank matrix $\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$. Hence modifies Eq. (2) into

$$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \mathbf{BAx}. \quad (9)$$

This yields a consistently more expensive forward step than the full-rank training regardless the choice of r. During the backward step, since gradient does not flow into $\mathbf{W}_0$, only one GEMM that computes gradient w.r.t $\mathbf{x}$ is involved with the full-rank component $\mathbf{W}_0\mathbf{x}$. Combining together both full-rank and low-rank components in both forward and backward step, the total compute of LoRA is

$$C_{\text{LoRA}} = 16nd^2 + 12n^2d + 12ndd_{\text{ff}} + \underbrace{48ndr + 18nr(d + d_{\text{ff}})}_{C_{\text{CoLA}}}. \quad (10)$$

When choosing the same r for LoRA and CoLA, we have $C_{\text{LoRA}} > C_{\text{CoLA}}$ always true.

In ReLoRA (Lialin et al., 2023), the hybrid strategy that warms up with the full-rank training arises more uncertainties in analyzing its complexity. And such strategy needs delicate tuning of hyper-parameters such as the full rank warm-up ratio, the restart frequency of optimizer, etc, and the choice of rank might also be affected by these strategy-level hyper-parameters. Therefore, we follow the same notion in (Zhao et al., 2024) that only consider the pure low-rank training of ReLoRA, which simplifies the compute analysis of ReLoRA to be the same as LoRA.

SLTrain (Han et al., 2024) proposes a low-rank + sparse parameterization instead of having a fixed full-rank matrix $\mathbf{W}_0$. The architecture of SLTrain is shown in Fig. 3 c). We continue using the notation for the low-rank matrices, and denote the sparse matrix as $\mathbf{S}$, with the sparsity level as δ. This modifies Eq. (2) into

$$\mathbf{h} = \mathbf{BAx} + \mathbf{Sx} = (\mathbf{BA} \oplus_{\mathcal{I}} \mathcal{V})\mathbf{x}, \quad (11)$$

where $\oplus$ denotes the scatter-add operator, $\mathcal{I}$ and $\mathcal{V}$ denote the indices and values of non-zero elements in $\mathbf{S}$. This implementation avoids instantiating a full sized $\mathbf{S}$, instead keeping only the non-zero elements. However, this introduces non-trivial reconstruction cost of $\mathbf{BA}$ in every step. And if we further denote $\tilde{\mathbf{W}} = \mathbf{BA} \oplus_{\mathcal{I}} \mathcal{V}$, then the forward data-flow that starts from $\tilde{\mathbf{W}}$ is the same as in the full-rank training, as well as the backward data-flow that ends at $\tilde{\mathbf{W}}$. Therefore, the total compute of SLTrain should be $C_{\text{full-rank}}$ plus reconstructing $\tilde{\mathbf{W}}$, and its corresponding $2\times$ compute during backward, i.e.,

$$C_{\text{SLTrain}} = C_{\text{full-rank}} + 24d^2r + 18dd_{\text{ff}}r. \quad (12)$$

For the last class of method to discuss, GaLore (Zhao et al., 2024) and it's follow-ups such as Fira (Chen et al., 2024) and APOLLO (Zhu et al., 2024), all investigate the memory efficiency associated with the AdamW optimizer. We only show the data-flow GaLore in Fig. 3 b), others are similar except some minor differences in how to manipulate gradients. The model architecture is kept unchanged in all these methods. Therefore, the complexity analysis is on the additional compute for projecting gradients into a low-rank space. GaLore proposes the following update rules:

$$\begin{aligned} \mathbf{R}_t &= \mathbf{P}_t^T \mathbf{G}_t, \tilde{\mathbf{G}}_t = \alpha \cdot \mathbf{P}\mathbf{N}_t, \\ \mathbf{W}_t &= \mathbf{W}_{t-1} + \eta \cdot \tilde{\mathbf{G}}_t, \end{aligned} \quad (13)$$

where the projector $\mathbf{P}_t \in \mathbb{R}^{d\times r}$ at time t is computed by decomposing $\mathbf{G}_t \in \mathbb{R}^{d\times d}$ via singular value decomposition (SVD) and is updated periodically, $\mathbf{N}_t \in \mathbb{R}^{d\times r}$ is the low-rank optimizer states, α is a scaling factor and η is the learning rate. Therefore, the total compute of GaLore is

$$C_{\text{GaLore}} = C_{\text{full-rank}} + 16d^2r + 12dd_{\text{ff}}r. \quad (14)$$

We remark that the compute analysis for the additional cost of SLTrain and GaLore (and its variants) is of limited scope and does not necessarily reflect their actual overhead. The actual cost will be dependent on other practical considerations on both algorithm and system level, such as the specific use case of these methods (e.g., pre-training, fine-tuning, etc), the actual number of the optimizer steps performed, the actual number of forward and backward steps performed when fixing total training tokens (i.e., if the hardware can afford larger batch sizes then the actual steps are fewer). It is almost impossible to give a unified notion while being fair when comparing between them. Hence we follow the similar setup used in (Zhao et al., 2024; Han et al., 2024; Chen et al., 2024; Zhu et al., 2024) when they analyze memory efficiency and measure system-level performance. However, it is rather safe to conclude that the overall cost introduced by GaLore and its variants will be diluted in real practices of pre-training due to the optimizer step is not frequent as forward and backward steps, hence are less expensive than SLTrain. Nonetheless, we highlight the fact that all the aforementioned methods are non-trivially more expensive than CoLA in terms of compute, and are all (except LoRA/ReLoRA) lower bounded by the full-rank training.

C Detailed Memory Analysis

We continue using the notions defined in Section. 4.2 and start with the activation memory of full-rank training:

$$\begin{aligned} M_{\text{full-rank}} = &\underbrace{3nd}_{\mathbf{Q},\mathbf{K},\mathbf{V}} + \underbrace{2n^2h + 2nd}_{\text{attention}} + \underbrace{11nd}_{\text{ffw}} \\ &\underbrace{2nd}_{\text{residual connection}} + \underbrace{2nd}_{\text{layer norm}} = 20nd + 2n^2h. \end{aligned} \quad (15)$$

When applying vanilla GCP, only the output of each block is saved, and all other activations are re-computed when needed. This dramatically reduces the total activation memory to only

$$M_{\text{vanilla-GCP}} = nd. \quad (16)$$

However, such benefit comes with a cost equal to almost an entire forward step. From Table. 2, we have the cost of vanilla-GCP as

$$C_{\text{vanilla-GCP}} = C_{\text{full-rank}} + 23nd^2 + 4n^2d. \quad (17)$$

Although we mentioned that delicate optimization of vanilla-GCP is beyond the scope of our discussion, we show a heuristic strategy when selecting checkpoints. Refer to Eq. (15), activations that associated with minimal re-compute are: layer norm, residual connection, and non-linear function (included in the ffw term). Then intuitively these activations should always be re-computed when trying to save memory. In fact this can save a fair amount of memory. Note in this paper we analyze compute in pure theoretical notion that lower order terms does not bring noticeable effect hence are omitted. In practice, however, re-computation brings latency

even for theoretically trivial operations, and will lower the overall GPU throughput. Other terms in Eq. (15) are all significant components when mapping to FLOPs change. One can gradually add more operations into the re-compute list and trade for more memory savings. We show the trend how they scale in Fig. 7.

Now we discuss CoLA and how it enables compute efficient checkpointing. We first evaluate how much memory overhead introduced by the low-rank activations. Compared to Eq. (15), CoLA adds $2nr$ for each of the low-rank layers, i.e., nr for $\mathbf{Ax}$, another nr for $\sigma(\mathbf{Ax})$, thereby

$$M_{\text{CoLA}} = M_{\text{full-rank}} + \underbrace{14nr}_{\text{low-rank } \sigma} - \underbrace{2.5nd}_{\text{remove original } \sigma} \tag{18}$$

We notice that when model scales up, the original LLaMA activation no longer brings benefit to model performance, hence can be removed, which corresponds to $2.5nd$ less activations.

As shown in Figure. 4, CoLA has multiple nonlinear functions injected along the normal dataflow. This partitions the previously longer path, i.e., the whole block, to significantly shorter paths bounded by these low-rank activations. This provides a natural selection of checkpoints that are of r-dimensional instead of d. More importantly, these shorter paths halve the re-compute steps. We show in Figure. 4 that only the weights that are painted in sketch need re-computation during the backward step of CoLA-M. This reduces significantly the cost of implementing GCP in CoLA-like architecture, results in the cost of only

$$C_{\text{CoLA-M}} = C_{\text{CoLA}} + 18.5ndr + 4n^2d. \tag{19}$$

Meanwhile, the memory saving of CoLA-M is still significant. We have the activation memory of CoLA-M as

$$M_{\text{CoLA-M}} = 2nd + 7nr. \tag{20}$$

D Hyper-Parameters

For optimizer related hyper-parameters, we empirically found 0.003 is a balanced choice of learning rate for most of the models we trained, this is similar to the settings in (Han et al., 2024). For CoLA-1B, this learning rate triggers a unstable loss curve, thereby is reduced to 0.002, and is further reduced to 0.001 for CoLA-7B as a conservative practice. For smaller models like CoLA-60M, an even larger learning rate such 0.006 can be adopted. For the warm-up ratio, weight decay and gradient clipping, we found the commonly adopted settings, 0.1, 0.01, 0.5, are proper choices for CoLA. Other than the standard optimizer parameters, one needs to pre-define a rank r when initializing CoLA. A default choice is set to approximately one quarter of the model inner width, i.e., $r = \frac{1}{4}d$.

	60M	130M	350M
CoLA w/ Both σ	**34.04**	**24.48**	19.56
CoLA w/ Only Low-Rank σ	34.35	25.20	**19.40**
CoLA w/ Only Low-Rank σ – Reduced	35.41	25.90	20.50
CoLA w/ Only Full-Rank σ	36.26	26.85	21.18

Table 11: Ablation study regarding where to place the low-rank non-linear functions.

E Ablation Study

We empirically found that keeping the original LLaMA nonlinearity on top of our proposed formulation Eq. (3) helps improve the model performance at smaller scales, such as 60M and 130M. However, when scaling up to 350M we no longer observe such a benefit. Therefore, the default setting of pre-training CoLA-1B/7B is set to use only low-rank nonlinearity. We found also evident that applying low-rank nonlinearity (i.e., Eq. (3)) regardless of whether the original linear layer being followed by nonlinearity is crucial to boost model performance. Results are shown in Table. 11, in which "CoLA w/ Both σ" means keeping the original nonlinearity on top of proposed low-rank nonlinearity, "CoLA w/ Only Low-Rank σ" means applying Eq. (3) in an agnostic way to all linear layers, "CoLA w/ Only Low-Rank σ – Reduced" means only applying Eq. (3) to the linear layers that are originally followed by nonlinearity, "CoLA w/ Only Full-Rank σ" means keeping the low-rank factorization but does not apply low-rank nonlinearity.

F Detailed Profiling Setting

This section provides a detailed explanation of the experimental setup for system-level measurements. For the memory breakdown in Fig. 6, we use a sequence batch size of 32. For throughput measurement in Fig. 8, we use a sequence batch size of 16 because the full-rank model cannot fit into 40GB A100 when using a sequence batch size of 32. Throughput is measured incorporating one for-

ward pass, one backward pass, and one optimizer step. This setup reflects a realistic training scenario, particularly in a multi-GPU environment, such as an 8x A100 cluster utilizing simple data parallelism. For a fair comparison, we set the update step in GaLore/APOLLO to 200, ensuring that the computationally expensive SVD/random projection is performed only once every 200 optimizer steps and is distributed across a single optimizer step. All experiments are conducted on a single GPU to isolate the effected of FLOP reduction on throughput improvement, without being influenced by multi-GPU framework settings or communication overhead. For Table. 7, memory consumption is measured on a 94GB H100 with a sequence batch size of 16. For Table. 10, inference is performed using the same configuration as pre-training, with a sequence batch size of 32.