GRASS: Compute Efficient Low-Memory LLM Training with Structured Sparse Gradients

Aashiq Muhamed[1], Oscar Li[2], David Woodruff[3], Mona Diab[1], Virginia Smith[2]
{amuhamed, runlianl, dwoodruf, mdiab, smithv}@andrew.cmu.edu
[1] Language Technologies Institute, [2] Machine Learning Department
[3] Department of Computer Science
Carnegie Mellon University

Abstract

Large language model (LLM) training and finetuning are often bottlenecked by limited GPU memory. While existing projection-based optimization methods address this by projecting gradients into a lower-dimensional subspace to reduce optimizer state memory, they typically rely on *dense* projection matrices, which can introduce computational and memory overheads. In this work, we propose GRASS (GRAdient Stuctured Sparsification), a novel approach that leverages *sparse* projections to transform gradients into structured sparse updates. This design not only significantly reduces memory usage for optimizer states but also minimizes gradient memory footprint, computation, and communication costs, leading to substantial throughput improvements. Extensive experiments on pretraining and finetuning tasks demonstrate that GRASS achieves competitive performance to full-rank training and existing projection-based methods. Notably, GRASS enables half-precision pretraining of a 13B parameter LLaMA model on a single 40GB A100 GPU—a feat infeasible for previous methods—and yields up to a $2\times$ throughput improvement on an 8-GPU system. Code is released here[1].

1 Introduction

Pretraining and finetuning large language models (LLMs) are often memory bottlenecked: storing model parameters, activations, gradients, and optimizer states in GPU memory is prohibitively expensive. As an example, pretraining a LLaMA-13B model from scratch under full bfloat16 precision with a token batch size of 256 requires at least 102 GB memory (24GB for trainable parameters, 49GB for Adam optimizer states, 24GB for weight gradients, and 2GB for activations), making training infeasible even on professional-grade GPUs such as Nvidia A100 with 80GB memory (Choquette et al., 2021). Existing memory efficient system-level techniques like DeepSpeed optimizer sharding/offloading (Rajbhandari et al., 2020) and gradient checkpointing (Chen et al., 2016) trade off throughput for memory advantages which slow down pretraining. As models scale, the memory and compute demands of increasingly large LLMs continue to outpace hardware advancements, highlighting the need for advances in optimization algorithms beyond system-level techniques.

Various optimization techniques have been proposed to enhance the efficiency of LLM training. One prominent approach is parameter-efficient finetuning (PEFT), such as Low-Rank Adaptation (LoRA), which reparameterizes weight matrices using low-rank adaptors (Hu et al., 2021). This significantly reduces the number of trainable parameters, yielding smaller optimizer states and gradients. However, despite its efficiency, LoRA and its derivatives (Sheng et al., 2023; Zhang et al., 2023; Xia et al., 2024) often underperform compared to full-rank finetuning (Biderman et al., 2024). Variants like ReLoRA (Lialin et al., 2023) extend LoRA to pretraining by periodically updating the full matrix with new low-rank updates, but it still requires a costly initial full-rank training warmup which makes it impractical in memory-constrained scenarios.

To allow for full-rank pretraining and finetuning, another approach for memory-efficient LLM training involves designing adaptive optimizers (Shazeer and Stern, 2018). One such class, memory-efficient subspace optimizers, utilizes projection matrices to project high-dimensional gradients into a lower-dimensional space and performs optimization within the subspace. This projection significantly reduces the memory footprint required to store optimizer states. Existing methods such as GALORE (Zhao et al., 2024) and FLORA (Hao et al., 2024) employ dense projection matrices, which introduce additional memory and computational overhead. In contrast, we employ

[1] `https://github.com/aashiqmuhamed/GRASS`

Algorithm 1 Memory-efficient Subspace Optimization

```
Input: Initial weights $W_0 \in \mathbb{R}^{m \times n}$ with $m \leq n$; update frequency $K$; total iterations $T$; subspace rank $r$ with $r \ll m$; an off-the-shelf optimizer function opt to update the optimizer state; scale factor $\alpha$.
Output: Optimized weights $W^{(T)}$
 1: $t \leftarrow 0$
 2: $W^{(0)} \leftarrow W_0$                          ▷ Set initial weights $W_0 \in \mathbb{R}^{m \times n}$
 3: $S^{(0)} \leftarrow$ opt.init($0^{r \times n}$)           ▷ Adam state $\in \mathbb{R}^{2 \times r \times n}$
 4: for $t \leftarrow 0$ to $T$ do
 5:     if $t \equiv 0 \pmod{K}$ then
 6:         // Compute new projection matrix
 7:         $P \leftarrow$ compute$_P$ ($\nabla L(W^{(t)})$)        ▷ $P \in \mathbb{R}^{m \times r}$
 8:         // [Optional] Update optimizer state
 9:         $S^{(t)} \leftarrow$ update_state($S^{(t)}$)
10:     end if
11:     $G_C \leftarrow P^\top \nabla L(W^{(t)})$                     ▷ $G_C \in \mathbb{R}^{r \times n}$
12:     $S^{(t+1)}, \Delta^{(t+1)} \leftarrow$ opt.update($S^{(t)}, G_C$)
13:     $W^{(t+1)} \leftarrow W^{(t)} + \alpha P \Delta^{(t+1)}$              ▷ Apply update
14: end for
```

Algorithm 2 MeSO Implementations

FLORA
Compute dense P: Sample P_{ij} *i.i.d.* from $\mathcal{N}(0, 1/r)$.
Update_state: Updates momentum as $P_{(t+1)} P_{(t)}^\top S^{(t)}$.
Compute G_C: Computes G_C using dense matmul.
Apply update: Updates full W after dense matmul.

GALORE
Compute dense P: Top-r left singular vectors of grad G_W.
Update_state: Maintains optimizer state.
Compute G_C: Computes G_C using dense matmul.
Apply update: Updates full W after a dense matmul.

GRASS (ours)
Compute sparse P: Computes the selection matrix B and the diagonal scaling matrix ρ based on row norms of G_W.
Update_state: Resets $S^{(t)}$ to zero as necessary.
Compute G_C: Uses matrix associativity and sparse matmul.
Apply update: Sparse update W after sparse matmul.

structured sparse matrices for P, demonstrating their advantages in memory, computation, and communication efficiency across both pretraining and finetuning. Our main contributions include:

1. We introduce GRASS, a novel method that enables full parameter training of LLMs with structured sparse gradients. By leveraging sparse projection matrices, GRASS significantly reduces memory consumption and communication overhead compared to existing projection-based optimization techniques. We theoretically motivate and empirically analyze effective ways to construct the sparse projection matrix for GRASS.
2. We conduct extensive experiments on both pretraining and finetuning tasks, demonstrating that GRASS converges faster in wall-clock time than existing projection-based methods due to its additional compute efficiency benefits. GRASS exhibits minimal performance degradation (<0.1 perplexity gap) compared to full-rank training on the 1B parameter LLaMA model while achieving a 2.5× reduction in memory footprint.
3. We present an optimized PyTorch implementation of GRASS for modern hardware, incorporating implementation tricks to enhance training throughput, stability, and scalability. For pretraining a 1B LLaMA model, GRASS achieves a 25% throughput increase on a single GPU and up to a 2× throughput improvement on 8 GPUs over full-rank training and GALORE. Furthermore, GRASS's low memory footprint enables half-precision training of a 13B LLaMA model on a single 40GB A100 GPU, a feat that existing projection-based optimization methods cannot achieve.

2 A Unified View of Memory-efficient Subspace Optimizers (MeSO)

High memory usage of full-rank training. Standard full-rank training of the weight matrix $W \in \mathbb{R}^{m \times n}$ in any linear layer of an LLM involves **1)** computing the full-parameter gradient $G_W := \nabla L(W)$ and **2)** using it to update the model weights and optimizer states:

$$S^{(t+1)}, \Delta W^{(t)} \leftarrow \texttt{opt.update}(S^{(t)}, \nabla L(W^{(t)}))$$
$$W^{(t+1)} \leftarrow W^{(t)} + \Delta W^{(t)} \tag{1}$$

Here, opt.update denotes the optimizer's update function, which uses the current optimizer state $S^{(t)}$ and the gradient to compute the updated state $S^{(t+1)}$ and a learning-rate-adjusted weight update $\Delta W^{(t)}$ (see Appendix A for the pseudocode for the Adam optimizer). However, storing both the gradient and optimizer state incurs significant memory overhead – for example, an additional $3mn$ floats for Adam – motivating the need for more memory-efficient optimization techniques. We discuss these techniques in the following sections, while Appendix C covers additional related work.

Memory-efficient optimization in a subspace. To minimize the memory usage of the optimizer state, memory-efficient subspace optimizers (MeSO) restrict the optimization to a subspace defined by a projection matrix $P \in \mathbb{R}^{m \times r}$ ($r \ll m$) through the following objective: $\min_{A \in \mathbb{R}^{r \times n}} L(W_0 + PA)$. Applying an off-the-shelf optimizer like Adam to learn the smaller matrix A reduces the optimizer state size to $O(rn)$, which can be much smaller than the $O(mn)$ used in full-rank training. We provide the pseudocode of this optimization procedure in Algorithm 1, which unifies both existing methods and our proposed method[2]. We highlight the key parts of this algorithmic framework below.

Computing the projection matrix, `compute`$_P$. Employing a fixed P throughout training confines the search to its column space, limiting the learned model's expressiveness. To address this, MeSO methods periodically recompute P every K iterations with different choices (Algorithm 2): FLORA (Hao et al., 2024) independently samples each entry of P from $\mathcal{N}(0, 1/r)$, whereas GALORE (Zhao et al., 2024) sets P to be the top-r left singular vectors of the full-parameter gradient matrix $\nabla L(W)$ obtained through a Singular Vector Decomposition (SVD). Despite these differences, a commonality among prior works is the choice of *dense* matrices for P. In our work, we explore the use of *sparse* matrices as an alternative and propose several principled choices for such matrices in Section 3.2.

Optimizer state update, `update_state`. Updating P can modify the subspace optimization landscape. Different methods have proposed distinct strategies for updating the existing optimizer state $S^{(t)}$. We describe our strategy in Section 3.3.

Projection of the full gradient, $P^\top \nabla L(W^{(t)})$. MeSO methods require projecting the $m \times n$ full parameter gradient matrix $\nabla L(W^{(t)})$ into a lower-dimensional subspace $r \times n$ via left multiplication with $P^\top$. Existing methods compute this projection by first materializing the full gradient matrix $\nabla L(W^{(t)})$ in memory before performing the left projection multiplication. In contrast, GRASS leverages the associative property of matrix multiplication and the sparse structure of P to compute this projection without materializing the full gradient. This yields considerable computational and memory savings, detailed in Section 3.1. These efficiencies also extend to the weight update step, $W^{(t)} + \alpha P \Delta^{(t+1)}$, due to the sparsity of P. Here, the scale factor α (also used in GALORE) adjusts the effective learning rate of these linear layer weight matrices relative to other trainable model parameters.

[2]This algorithm version never materializes the A matrix, but is equivalent as we show in Appendix B.

3 GRASS: a more-efficient MeSO optimizer

Unlike prior MeSO methods that employ dense projection matrices, GRASS (GRAdient Structured Sparsification) utilizes a sparse projection matrix $P \in \mathbb{R}^{m \times r}$, where each column $p_j \in \mathbb{R}^m$ has at most one non-zero entry ($\|p_j\|_0 \leq 1, \forall j \in [r]$). This structure effectively constrains the subspace optimization to update only r rows of the full weight matrix W, inducing structured row-sparsity in the gradients – hence the name GRASS. By periodically updating P, GRASS learns different rows of W in different iterations, resembling a generalized form of coordinate gradient descent. We dive into the efficiency benefits of this sparse projection and various methods for constructing P in the following subsections.

3.1 Efficiency gains of GRASS

Efficient Storage of P. In GRASS, the sparse projection operator $P^\top \in \mathbb{R}^{r \times m}$ can be expressed as the product $P^\top = \rho B$, with a diagonal scaling matrix $\rho \in \mathbb{R}^{r \times r}$ and a binary selection matrix $B \in \{0, 1\}^{r \times m}$ which selects a single j-th row in G_W for its i-th row $B_{ij} = 1$. Both ρ and B can be efficiently stored using r instead of mr floats, making GRASS more memory-efficient in optimizer-related storage (**Optimizer** column in Table 1).

Efficient Gradient Projection. GRASS avoids computing and storing the full gradient matrix $G_W \in \mathbb{R}^{m \times n}$ for projection ($P^\top G_W$) , unlike existing MeSO methods (Zhao et al., 2024; Hao et al., 2024). Leveraging the chain rule, we express $G_W = (\nabla_y L)^\top X$, where $\nabla_y L \in \mathbb{R}^{b \times m}$ is the gradient of the loss with respect to the layer outputs and $X \in \mathbb{R}^{b \times n}$ represents the input activations, with b being the token batch size. This allows us to apply the associative rule and compute[3] the sparse gradient projection efficiently as $\rho((B \nabla_y L^\top) X)$. This insight yields significant advantages in compute, memory, and communication:

- *Compute savings*: By exploiting this regrouped multiplication, GRASS computes the projection in just $rbn + rn$ FLOPs. In contrast, dense projection methods like GALORE and FLORA require $mbn + rmn$ FLOPs, making GRASS over m/r times more computationally efficient. This significant advantage arises from 1) leveraging the associative rule, 2) the equivalence of left

[3]Implementation-wise, we only need to define a custom backward pass for the PyTorch linear layer.

Method	Memory			FLOPs		Comm
	Weights	Optimizer	Grad	Regular step (Lines 11-13)	$\texttt{compute}_P$ step (Line 7)	
Full	mn	$2mn$	mn	$mbn + mn + Cmn$	0	mn
LoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$	$mbn + 2rmn + C(rm + rn) + rn + rm$	0	$mr + nr$
ReLoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$	$mbn + 2rmn + C(rm + rn) + rn + rm$	$mnr + mn$	$mr + nr$
FLORA	mn	$mr + 2nr$	mn	$mbn + 2rmn + mn + Crn$	mr	mn
GaLore	mn	$mr + 2nr$	mn	$mbn + 2rmn + mn + Crn$	$mn \min(n, m)$	mn
GRASS (ours)	mn	$2r + 2nr$	nr	$rbn + 3rn + Crn$	$mn + m + r$	nr

Table 1: Summary of Memory, FLOPs, and Distributed Communication Volume for the different methods. GRASS improves over existing methods in Memory, FLOPs, and Communication. Weight $W \in \mathbb{R}^{m \times n}$. b is token batch size, r is subspace rank, C cost of optimzer update operations per parameter, $G \in \mathbb{R}^{m \times n}$, $P \in \mathbb{R}^{m \times r}$. Detailed breakdown in Appendix G.

multiplication by ρ to a simple row-wise scaling (costing only nr FLOPs), and 3) the cost-free row selection performed by left multiplication with B.

- *Memory savings*: GRASS's multiplication order eliminates the need to ever materialize the full gradient matrix, directly yielding the projected result. This saves memory by avoiding the storage of mn floats required by other methods (see the **Grad** column in Table 1). Importantly, this memory advantage is independent of and can be combined with layerwise weight update techniques (Lv et al., 2023b; Zhao et al., 2024), which reduce memory by processing gradients one layer at a time.
- *Communication savings:* During distributed training, existing MeSO methods like GALORE and FLORA communicate the full $m \times n$ gradient matrix across workers, leading to a communication cost of $O(mn)$. Since GRASS is implemented in the backward pass, it can directly compute and communicate the $r \times n$ projected gradient without materializing the full gradient, reducing communication volume to $O(rn)$ (**Comm** column in Table 1).

Efficient Weight Update. The weight update step, $W^{(t)} + P\Delta^{(t+1)}$, also benefits from the sparsity of P in GRASS. Instead of constructing the full $m \times n$ update matrix $P\Delta^{(t+1)}$, which is row-sparse, GRASS directly computes and applies the updates to the r nonzero rows. This reduces the computational cost to just $2rn$ FLOPs, compared to the $rmn + mn$ FLOPs required by dense update methods like GALORE and FLORA. The total compute cost of GRASS during a regular step (which includes this efficient weight update, the aforementioned efficient gradient projection, and the optimizer computation) compares favorably against other baselines as shown in the **Regular step** column in Table 1.

3.2 Choices of sparse P

We now discuss concrete choices for $\texttt{compute}_P$ by specifying how to construct ρ and B for $P^\top = \rho B$. To simplify the notation, we denote the index of the only non-zero entry in the j-th row of B by $\sigma_j \in [m]$. We consider both stochastic and deterministic approaches to construct $\{\sigma_j\}_{j=1}^r$ and $\{\rho_{jj}\}_{j=1}^r$.

A. Stochastic construction of P. Since $\sigma_j \in [m]$ is a categorial variable, a natural approach is the with-replacement sampling of $\sigma_j \overset{\text{i.i.d.}}{\sim}$ Multinomial$(1, q)$, with the probability of sampling any integer $k \in [m]$ given by q_k. To ensure the unbiasedness[4] of the reconstructed gradient $\mathbb{E}[PP^\top G_W] = G_W$ for its optimization convergence benefits, we set $\rho_{jj} = \frac{1}{\sqrt{r \cdot q_{\sigma_j}}}$ after sampling σ_j. To set the multinomial distribution parameter q, we consider two different principles:

- *The Variance-reduction principle*: Here we want to minimize the total variance of the gradient estimate $PP^\top G_W$. The optimal q is given by the following theorem (proof in Appendix E):

  Theorem 3.1. *Among all the Multinomial$(1, q)$ distributions, the one that is proportional to the row norms of G with $q_k = \frac{\|G_k\|_2}{\sum_{i=1}^m \|G_i\|_2}$ minimizes the total variance of the gradient estimate $PP^\top G$.*

  We call this method **Multinomial-Norm**.
- *The Subspace-preservation principle*: When P is fixed for a large K number of iterations and the gradient is low-rank (Zhao et al., 2024), reducing the variance of the gradient estimate could be less important than preserving the low-rank subspace of G_W upon projection. To achieve this, we set q_k proportional to the *squared* row norms of G_W ($q_k \propto \|G_k\|^2$) and call this method **Multinomial-Norm**^{2}. This q distribution gives us approximate leverage score sampling (Magdon-Ismail, 2010), which ensures

[4]See the proof of this statement in Appendix D.

high probability preservation of the low-rank subspace with little additive error (see Appendix F). In addition to these two principled unbiased sampling with replacement methods, we also experiment with the **Uniform Distribution** with $q_k = 1/m$ as a baseline. Furthermore, we explore the non-replacement sampling counterparts (**-NR**) for each of the three distributions. Since it is analytically intractable to guarantee unbiasedness in this case, we set $\rho_{jj} = 1$ for the **NR** methods.

B. Deterministic construction of P. We consider minimizing the gradient reconstruction error in Frobenius norm $\|PP^\top G_W - G_W\|_F^2$ as the principle to choose P. One minimizing solution sets all $\rho_{jj} = 1$ and $\{\sigma_j\}_{j=1}^r$ to be the indices of rows of G_W with largest row-norms. We call this $\texttt{compute}_P$ method **Top**-r.

Compute cost. Unlike GaLore, Grass only requires computing row norms of G_W but not an SVD in the update step. ($\texttt{compute}_P$ column in Table 1). Furthermore, no additional memory is consumed for SVD as in GaLore.

3.3 Implementation Details

Updating the Optimizer State. Updating the projection matrix P in Grass can lead to significant shifts in the selected rows of the parameter matrix W between iterations. Since different rows of W may have distinct gradient moment statistics, we reset the optimizer states to zero during the `update_state` step. To further stabilize training after such updates, we implement a learning rate warmup phase. This combined approach effectively mitigates training instabilities, particularly those observed in smaller models during pretraining.

Distributed Training. Since Grass updates the projection matrix during each worker's backward pass in distributed training, synchronizing the selected indices across workers is necessary. To minimize communication overhead, we first compute the gradient G_W and then sketch it by sampling r columns based on their norms, resulting in a sketched matrix $G_{comm} \in \mathbb{R}^{m\times r}$. An all-reduce operation is performed on G_{comm}, ensuring all workers access a consistent version of the sketch before sampling indices. Furthermore, we implement custom modifications to prevent PyTorch DDP (Paszke et al., 2019) from allocating memory for full gradients in our Grass implementation (see Appendix H for details).

4 Experiments

4.1 Pretraining Performance

Experimental setup. We compare[5] Grass against Full-rank (without gradient projection) and GaLore by pretraining LLaMA-based models (Touvron et al., 2023) in BF16 on the cleaned C4 subset of Dolma (Soldaini et al., 2024). We train without data repetition over a sufficiently large amount of data, across a diverse range of model sizes (60M, 350M, 1B). We adopt a LLaMA-based architecture with RMSNorm and SwiGLU activations (Touvron et al., 2023; Shazeer, 2020; Zhang and Sennrich, 2019). For both Grass and GaLore, we fix the frequency K at 200 iterations, α at 0.25, use a consistent rank r, and project the linear layers within the attention and feed-forward layers. P is applied to project the smaller dimension of G_W to achieve the best memory-performance tradeoff (Zhao et al., 2024). We use the same batch size and tune the learning rate individually for each method (see Appendix I).

Model size	**60M**	**350M**	**1B**
Full-Rank	36.97	18.71	18.12
GaLore	37.09	19.38	19.23
Grass (Top-r)	37.24	19.49	19.04
$\mathbf{r/d_{model}}$	128 / 512	128 / 1024	256 / 2048
Tokens	1.0B	5.4B	8.8B

Table 2: Train perplexity of LLaMA models on the C4 subset of Dolma. Grass (Top-r) is competitive with GaLore, but with lower memory footprint and higher training throughput.

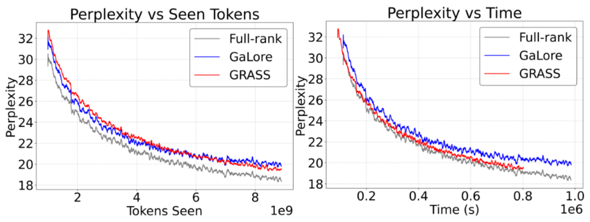

Figure 1: Pretraining 1B LLaMA on 8.8B tokens of C4 with Grass (Top-r), Full-rank and GaLore. (Left) Train perplexity vs seen tokens. (Right) Train perplexity vs wall-clock time. Grass outperforms GaLore and shows < 0.01 perplexity gap with Full-rank loss curve in wall-clock time.

Results. As shown in Table 2, Grass matches GaLore and approaches Full-rank's performance within a perplexity gap of less than 1 even when $r/d_{model} = 8$. In Figure 1, for the 1B model we see that this gap disappears when we look at perplexity vs. training time (as opposed to tokens seen)

[5]We compare against Flora in Section 4.2 and 4.3 as it was primarily intended for finetuning in the original work.

on a single A100 GPU, where due to increased pretraining throughput GRASS closely follows the Full-rank loss curve with < 0.1 perplexity gap.

4.2 Finetuning Performance

Experimental setup. We evaluate GRASS, LoRA, Full-rank, GALORE, and FLORA on the GLUE NLU benchmark (Wang et al., 2018a) by finetuning a pretrained RoBERTa-Base model (Liu et al., 2019) with a sequence length of 128 in float32 (results on the dev set). For all the optimization methods, we restrict them to only optimize the linear layers in the attention and MLP layers for three epochs with individually tuned learning rates. We set rank $r = 8$ for all the low-rank methods. For the MeSO methods, we set the update frequency $K = 100$ and tune the scale factor α for each method. (See more details in Appendix I.)

Results. In Table 3, GRASS Top-r performs competitively with LoRA, FLORA, and GALORE even though GRASS exhibits a reduced memory footprint and improved training throughput compared to these methods as we show in Section 4.4.

4.3 Instruction-finetuning Performance

Experimental setup. We compare GRASS against Full finetuning, GALORE, FLORA, and LoRA on instruction finetuning using a LLaMA-7B model (Touvron et al., 2023) pretrained on 1T tokens. We finetune on Alpaca (Taori et al., 2023) (52k samples) and a 100k sample subset of FLAN v2 (Wei et al., 2021) from Tulu (Wang et al., 2023) (due to FLAN v2's scale), using BF16 precision, batch size 64, and a source and target sequence length of 512. All methods, except for Full finetuning which updates all parameters, are restricted to only update the linear layers in the attention and MLP layers with rank $r = 64$. We finetune for 1000 steps on Alpaca (1.26 epochs) and 1500 steps on Flan v2 (1.08 epochs). Additional hyperparameters are in Appendix I. Following prior work (Touvron et al., 2023; Dettmers et al., 2023), we assess the instruction-tuned models' average 5-shot test performance on the MMLU benchmark (Hendrycks et al., 2020) (57 tasks).

Results. As shown in Table 4, GRASS performs competitively with full-parameter finetuning, FLORA, GALORE, and LoRA during instruction finetuning on both Alpaca and Flan v2. Furthermore, Section 4.4 demonstrates that, at $r = 64$, GRASS not only matches LoRA's performance but also boasts a lower memory footprint and an 18% throughput increase. Because GRASS can perform higher rank training with multiple projection matrix updates, it is expected to further outperform the rank-constrained LoRA in more challenging tasks with larger datasets.

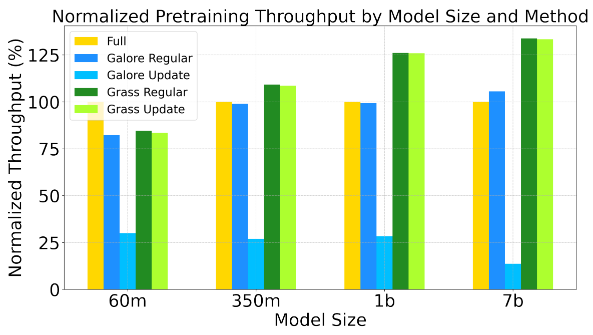

Figure 2: Normalized pretraining throughput at $r = 64$ for GRASS (Top-r), Full-rank, and GALORE relative to Full-rank. For model size > 60m, GRASS throughput exceeds Full and GALORE throughput by $> 25\%$.

4.4 Efficiency analysis

Pretraining Throughput. Figure 2 compares the BF16 pretraining throughput (tokens/s) of GRASS and GALORE relative to Full-rank, across model sizes, for both regular and projection update[6] steps. We use rank $r = 64$ on attention and feedforward layers, sequence length 256, and total batch size 1024 on a single 80GB A100 GPU. See Appendix I for detailed settings. We did not employ activation checkpointing, memory offloading, or optimizer state partitioning in our experiments.

While GRASS exhibits lower throughput than Full-rank at 60M parameters (due to customized matrix multiplication overhead), GRASS significantly outperforms both at 1B and 7B parameters, achieving 26% and 33.8% higher throughput than Full-rank, and 27% and 26.7% higher than GALORE (for the regular step). GRASS's projection update overhead is minimal, unlike GALORE's costly SVD computations. The throughput advantage for GRASS is expected to grow with larger batch sizes, benefiting further from its lower memory footprint compared to other methods. Appendix Figure 11 provides further throughput comparisons across different ranks, showing that GRASS achieves its highest relative throughput gains at rank ($r = 64$), with diminishing returns as rank increases or model size decreases.

[6]The regular update iteration doesn't invoke $\texttt{compute}_P$ but only updates the parameters, while the projection update step performs both.

Model	COLA	MNLI	MRPC	QNLI	QQP	RTE	SST2	STSB	WNLI	Average
Full-rank	59.62	87.36	91.51	92.60	90.43	79.03	94.49	90.38	56.34	82.42
LoRA	58.36	86.80	90.09	92.49	89.43	75.09	94.49	90.22	56.34	81.48
GaLore	57.64	87.40	88.97	92.86	88.94	76.17	94.49	89.76	56.34	81.40
Flora	59.65	86.65	89.82	92.09	88.61	76.34	94.27	90.06	56.34	81.53
Grass (Top-r)	59.16	86.92	89.60	92.42	88.65	76.37	94.15	90.13	56.34	81.53
Grass (Multi-Norm^2-NR)	58.87	86.08	89.94	91.69	83.36	76.17	94.73	90.00	56.34	81.35
Grass (Multi-Norm-R)	57.81	86.25	87.58	91.80	88.06	68.59	94.27	89.73	56.34	80.05
Grass (Uni-NR)	49.66	85.70	78.01	90.94	87.56	57.76	93.35	84.86	56.34	76.02

Table 3: Evaluating Full-rank and different memory-efficient optimization methods on the GLUE benchmark using RoBERTa-Base. Grass is competitive with LoRA and Flora but with a lower memory footprint. Values in blue represent the top three results in each column.

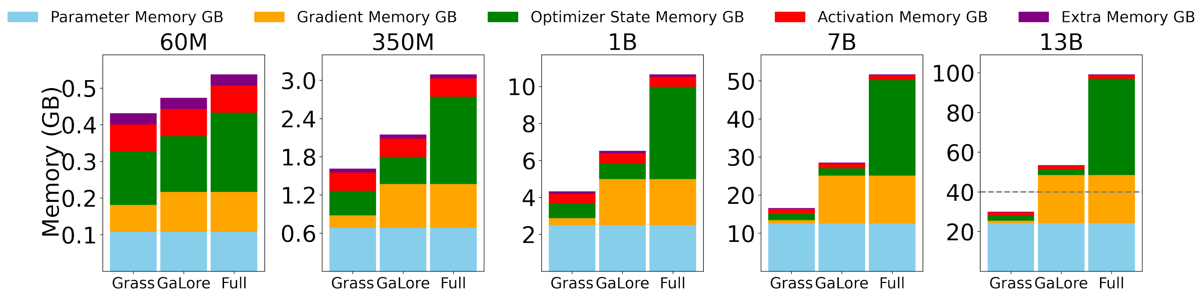

Figure 3: Pretraining memory footprint for Grass (Top-r), GaLore, and Full across model sizes for a regular (non projection update step) and $r = 128$. Grass has a lower memory footprint across all model sizes and the reduction is greater at larger model sizes.

Model		MMLU Acc (%)	
LLaMA-7b	Trainable Params	Alpaca	FLAN v2
Full	6898.3M	38.12	35.85
LoRA	159.90M	38.21	34.98
GaLore	6476.0M	37.93	34.72
Flora	6476.0M	37.86	35.16
Grass (Top-r)	6476.0M	**38.37**	**36.88**

Table 4: Average 5-shot MMLU accuracy for LLaMA-7B models finetuned with various methods across Alpaca and FLAN v2. Grass, Flora, GaLore, and LoRA were applied to attention and MLP layers using rank 64. Grass not only competes effectively with full training but also offers advantages in terms of lower memory usage and higher throughput compared to all baseline methods.

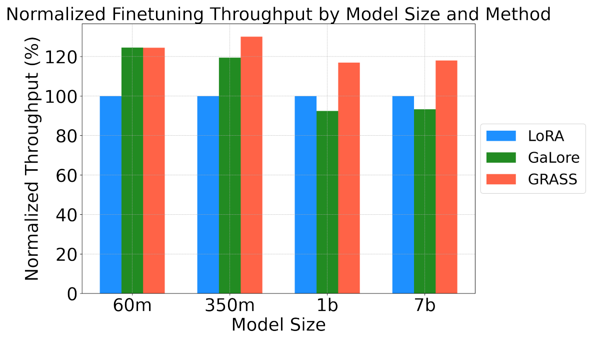

Figure 4: Normalized LLaMA finetuning throughput of Grass (Top-r), GaLore, and LoRA relative to LoRA. We use rank $r = 64$. Grass is $> 18\%$ faster than LoRA.

Finetuning Throughput. Figure 4 compares the BF16 finetuning throughput of Grass, GaLore, and LoRA across various LLaMA model sizes, focusing on the regular step. Unlike the pretraining throughput benchmark, we finetune only the attention and MLP layers using $r = 64$. We maintain a uniform local batch size, sequence length 256, and total batch size of 1024 across all methods (detailed hyperparameters are provided in Appendix I). For the 7B parameter model, Grass achieves throughput improvements of 26% and 18% over GaLore and LoRA, respectively. Appendix Figure 12 provides further throughput comparisons and analysis across ranks 8, 16, 32, and 64, demonstrating that Grass consistently maintains its throughput advantage across these ranks.

Pretraining Memory. Figure 3 benchmarks the BF16 memory footprint of pretraining Grass against Full-rank and GaLore across various model sizes (token batch size 256, rank r=128), focusing on the regular training step. We benchmark memory efficiency with a token batch size of 256, consistent with (Zhao et al., 2024, Fig 1) since activation memory can be controlled through microbatching while reducing gradient and optimizer state memory requires techniques like Grass.

Grass consistently exhibits a lower memory footprint than both Full-rank and GaLore, with the memory reduction increasing with model size. This advantage stems from Grass's reduced

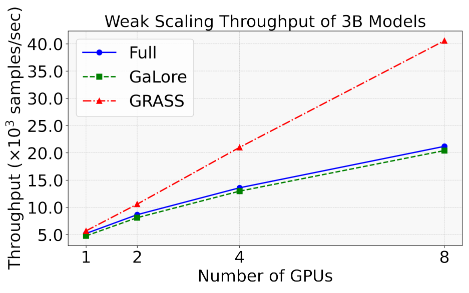

Figure 5: Communication Efficiency: Weak Scaling Throughput Comparison for 3B LLaMA pretraining using GRASS (Top-r), Full-rank, and GALORE. GRASS shows 2× higher throughput over Full and GALORE at 8 GPUs.

gradient and optimizer memory (due to its sparse projection matrices). At 13B parameters, GRASS uses 70% less memory than Full-rank and 45% less than GALORE.

Beyond the memory advantage in the regular update iteration, Grass is also more memory efficient in the projection update iteration compared to its counterpart GALORE: GALORE requires converting the full gradient to float32 for SVD computation when computing the projection matrix, making it unable to pretrain the 13B LlaMA model in BF16 at rank (r = 128) on an 80GB GPU. In contrast, GRASS is capable of pretraining the 13B model on ranks up to $r = 768$ on a 40GB GPU and up to $r = 1024$ on a 48GB GPU.

Finetuning Memory. Appendix Figure 9 and Figure 10 compare the memory footprint of GRASS and LoRA during LLaMA finetuning. GRASS demonstrates a memory advantage of roughly 1GB over LoRA when finetuning the 7B parameter model in BF16 at rank (r=64). However, as the batch size increases, activations dominate the memory footprint, and the memory usage of GRASS and LoRA becomes comparable.

Communication. Figure 5 benchmarks the (weak scaling (Gustafson, 1988)) throughput (tokens/sec) of training a 3B parameter LLaMA model on a multi-GPU L40 compute node with a peak all-reduce bandwidth of 8.64 GB/s as we scale the number of participating GPUs. We use a token batch size of 4096 per worker (local batch size 16, sequence length 256). GRASS, by communicating only the projected gradients, achieves significantly higher throughput (2× on 8 GPUs) compared to both Full-rank and GALORE.

4.5 Ablations

Effect of Rank. Figure 6 presents ablations on the impact of the subspace rank r for GRASS during pretraining of a 350M parameter LLaMA model on the C4 subset of Dolma. Increasing the rank generally leads to better training losses for the same number of updates, but with diminishing returns. Additionally, since GRASS enables full-parameter training, we observe that training at rank $r = 128$ for 80k steps is more effective than training at rank $r = 512$ for 40k steps. GRASS can therefore be used to trade-off memory and computational cost where in a memory-constrained setting one could select a lower rank and train longer.

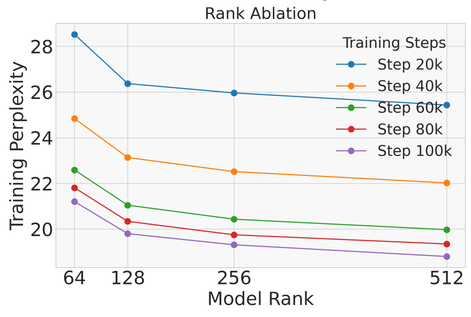

Figure 6: GRASS (Top-r) rank ablations for 350M LLaMA training. We report perplexity on Dolma C4 across various ranks and training steps. Loss is averaged over a window of 50 steps.

Effect of Update Frequency. Figure 7 analyzes the impact of update frequency on the convergence of GRASS during pretraining of a 60M-parameter LLaMA model on the Realnews subset of C4 (Raffel et al., 2020) at various steps during training. Both overly frequent and infrequent updates to the projection matrix hinder convergence. Optimal convergence is achieved within an update frequency range of 200 to 500 iterations.

compute$_P$ Methods. Table 5 evaluates our proposed methods to compute the sparse projection

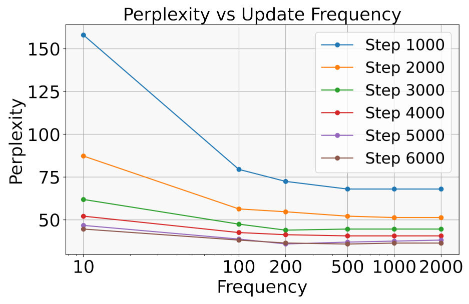

Figure 7: GRASS(Top-r) Update Frequency vs. Training Perplexity for 60M LLaMA pretraining on Realnews subset of C4 evaluated at various points during training. A frequency of 200 is near optimal.

Sampling Method	Eval perp
Frozen Top-r	34.78
Uniform-R	32.46
Uniform-NR	31.06
Multinomial-Norm-R	31.32
Multinomial-Norm-NR	30.93
Multinomial-Norm^2-R	31.85
Multinomial-Norm^2-NR	30.91
Top-r	**30.88**
GaLore	30.67
Full-rank	30.27

Table 5: Comparison of GRASS Sampling Methods on Evaluation Perplexity during 60M LLaMA Pretraining on the RealNews Subset of C4. Best sampling strategy is bolded.

P matrix (in Section 3.2) for GRASS during pretraining of a 60M LLaMA model on 500M tokens from the RealNews subset of C4. We additionally consider the Frozen Top-r method as a baseline by computing top indices once only at iteration 0. We notice that stochastic strategies employing non-replacement biased (NR) sampling generally surpass their with replacement unbiased (R) counterparts. Within the unbiased strategies (R), the variance reduction approach (Multinomial-Norm-R) outperforms the subspace preservation method (Multinomial-Norm^2-R), while their biased (NR) counterparts exhibit comparable performance. Both Multinomial-Norm^2-NR and Top-r are competitive with GaLore, while Uniform sampling underperforms. Similar trends in performance across sampling methods are observed during finetuning (Table 3). We find that uniform sampling is more effective for pretraining than finetuning, likely because the norm distribution is more uniform at the onset of pretraining.

5 Conclusion And Future Work

In this work, we introduce GRASS, a novel memory-efficient subspace optimization method for LLM pretraining and fine-tuning by leveraging sparse gradient projections. GRASS significantly reduces the memory footprint of optimizer states and gradients and eliminates the need to materialize the full gradients during the projection step, leading to substantial computational efficiency gains. Our experimental results demonstrate that GRASS achieves comparable performance to full-rank training and existing projection-based methods while offering a substantial memory reduction and throughput increase across various model sizes and tasks. Future work will explore extending GRASS to utilize diverse structured sparsity patterns and investigating strategies for dynamically adjusting the projection rank based on hardware and model size.

Acknowledgements

This work was supported in part by the National Science Foundation grants IIS2145670 and CCF2107024, and funding from Amazon, Apple, Google, Intel, Meta, and the CyLab Security and Privacy Institute. D. Woodruff would like to acknowledge support from a Simons Investigator Award, NSF CCF-2335412, and Office of Naval Research (ONR) award number N000142112647. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of these funding agencies.

Limitations

While GRASS offers compelling advantages in memory efficiency and training throughput, there are several aspects that warrant further investigation and potential improvements.

Implementation Complexity. Unlike drop-in optimizer replacements, GRASS requires integrating custom linear layers into the Transformer architecture, as the sparse projection operations occur during the backward pass. While this involves minimal code modifications, it introduces a slight complexity barrier for adoption compared to simply switching optimizers. Nonetheless, the significant gains in performance and memory efficiency outweigh this minor overhead.

Scalability to Larger Models. Our empirical evaluation primarily focused on model scales up to 13B parameters. The effectiveness of GRASS for significantly larger LLMs, exceeding hundreds of billions of parameters, requires further examination. Similarly, as batch sizes increase, the memory savings from sparse projection might become less prominent compared to the activation memory footprint. Exploring strategies to mitigate this potential issue, such as combining GRASS with activation checkpointing techniques, would be beneficial.

Hyperparameter Sensitivity. GRASS's performance depends on hyperparameters like rank (r) and update frequency (K). While our experiments provide insights into suitable ranges for these hyperparameters, a more comprehensive analysis of their impact on training dynamics, particularly as model scales increase, is crucial for maximizing performance and generalizability. Developing methods to automatically and adaptively tune these hyperparameters could further enhance GRASS's applicability.

Ethical Considerations

We acknowledge the potential ethical implications associated with large language models. These include:

Misuse Potential. LLMs, being powerful text generation tools, can be misused to create harmful or misleading content, including disinformation, hate speech, and spam. While our work focuses on improving training efficiency, we strongly advocate for responsible use of LLMs and encourage further research on safeguards against malicious applications.

Bias Amplification. LLMs are trained on massive text corpora, which can inherently contain biases and stereotypes. These biases can be amplified during training, leading to potentially discriminatory or unfair outputs. While GRASS is unlikely to exacerbate this bias, we recognize the importance of addressing this issue through careful data curation, bias mitigation techniques, and ongoing monitoring of LLM behavior.

Environmental Impact. Training large LLMs requires significant computational resources, which can have a substantial environmental footprint. Our work aims to reduce the computational cost and energy consumption of LLM training, contributing to more sustainable and environmentally responsible practices in NLP research.

Data and Licensing Considerations. We have carefully considered the ethical implications of the datasets used in this work which are publicly released and have followed accepted privacy practices at creation time.

- MMLU and GLUE are released under the permissive MIT license, allowing for broad research use.
- Alpaca is also distributed under the MIT license.
- FLAN uses the Apache license, which permits both academic and commercial applications.
- Dolma utilizes the ODC Attribution License, promoting open data sharing and reuse.

We strictly adhere to the license terms and intended use of these datasets, ensuring responsible handling of data and compliance with ethical guidelines. We acknowledge the ongoing need for critical assessment and transparency regarding data sources, potential biases, and licensing implications in LLM research.

References

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. 2019. Memory efficient adaptive optimization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed optimisation for non-convex problems. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 560–569. PMLR.

Dan Biderman, Jose Gonzalez Ortiz, Jacob Portes, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John P. Cunningham. 2024. Lora learns less and forgets less. *arXiv preprint arXiv: 2405.09673*.

Daniel Cer, Mona Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. 2017. SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 1–14, Vancouver, Canada. Association for Computational Linguistics.

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

Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35.

Tim Dettmers, M. Lewis, Sam Shleifer, and Luke Zettlemoyer. 2021. 8-bit optimizers via block-wise quantization. *International Conference on Learning Representations*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *NEURIPS*.

Bill Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*. Asia Federation of Natural Language Processing.

John L. Gustafson. 1988. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533.

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

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, D. Song, and J. Steinhardt. 2020. Measuring massive multitask language understanding. *International Conference on Learning Representations*.

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*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Hector Levesque, Ernest Davis, and Leora Morgenstern. 2012. The winograd schema challenge. In *Thirteenth international conference on the principles of knowledge representation and reasoning*.

Bingrui Li, Jianfei Chen, and Jun Zhu. 2023. Memory efficient optimizers with 4-bit states. In *Advances in Neural Information Processing Systems*, volume 36, pages 15136–15171. Curran Associates, Inc.

Pengxiang Li, Lu Yin, Xiaowei Gao, and Shiwei Liu. 2024. Owlore: Outlier-weighed layerwise sampled low-rank projection for memory-efficient llm finetuning. *arXiv preprint arXiv:2405.18380*.

Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. 2023. ReloRA: High-rank training through low-rank updates. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@NeurIPS 2023)*.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv: 1907.11692*.

Qijun Luo, Hengxu Yu, and Xiao Li. 2024. Badam: A memory efficient full parameter training method for large language models. *arXiv preprint arXiv:2404.02827*.

Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. 2023a. Adalomo: Low-memory optimization with adaptive learning rate. *arXiv preprint arXiv: 2310.10195*.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. 2023b. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv: 2306.09782*.

Malik Magdon-Ismail. 2010. Row sampling for matrix algorithms via a non-commutative bernstein bound. *arXiv preprint arXiv: 1008.0587*.

Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. 2024. Lisa: Layerwise importance sampling for memory-efficient large language model fine-tuning. *arXiv preprint arXiv:2403.17919*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, N. Gimelshein, L. Antiga, Alban Desmaison, Andreas Köpf, E. Yang, Zach DeVito, Martin Raison, A. Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *Neural Information Processing Systems*.

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. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *Conference on Empirical Methods in Natural Language Processing*.

Cedric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. Sparcml: high-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*.

Noam Shazeer. 2020. Glu variants improve transformer. *arXiv preprint arXiv: 2002.05202*.

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2023. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv: 2311.03285*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA. Association for Computational Linguistics.

Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Pete Walsh, Luke Zettlemoyer, Noah A. Smith, Hannaneh Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. 2024. Dolma: an open corpus of three trillion tokens for language model pretraining research. *arXiv preprint arXiv: 2402.00159*.

Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. 2019. Compressing gradient optimizers via count-sketches. *International Conference on Machine Learning*.

Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 2021. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10118–10129. PMLR.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv: 2307.09288*.

Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018a. Glue: A multi-task benchmark and analysis platform for natural language understanding. *BLACKBOXNLP@EMNLP*.

Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018b. Atomo: Communication-efficient learning via atomic sparsification. *Advances in neural information processing systems*, 31.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. 2023. How far can camels go? exploring the state of instruction tuning on open resources. *Neural Information Processing Systems*.

Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. 2018. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv: 2109.01652*.

Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. 2017. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv: 1704.05426*.

David P. Woodruff. 2014. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*.

Wenhan Xia, Chengwei Qin, and Elad Hazan. 2024. Chain of lora: Efficient fine-tuning of language models via residual learning. *arXiv preprint arXiv: 2401.04151*.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2023. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv: 2308.03303*.

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*.

A Optimizer Functions

In Equation (1) and Algorithm 1, we use functions `opt.init` and `opt.update` to abstractly represent any stateful optimizer's initialization and update function. Here we provide concrete implementations of these functions for Adam (Kingma and Ba, 2014) in Algorithm 3 and 4.[7] We assume the parameter matrix Z and its gradient $\nabla_Z L$ is of generic shape $\mathbb{R}^{c\times d}$.

Algorithm 3 Initialization of the Adam optimizer, `adam.init`

Input: $Z \in \mathbb{R}^{c\times d}$ (technically, Adam only requires knowing the shape of the parameter)
Output: $S \in \mathbb{R}^{2\times c\times d}$

```
1: $M \leftarrow \mathbf{0}_{c\times d}$    ▷ First gradient moment statistics
2: $V \leftarrow \mathbf{0}_{c\times d}$    ▷ Second gradient moment statistics
3: $S \leftarrow (M, V)$
```

Algorithm 4 Update of the Adam optimizer, `adam.update`. $\beta_1, \beta_2 \in [0, 1)$ are the exponential decay rates for the first and second gradient moment estimates. t is the current iteration. $\eta > 0$ is the current iteration's learning rate. ϵ is a small constant used for numerical stability in division.

Input: $S \in \mathbb{R}^{2\times c\times d}$ the most recent optimizer state
$\nabla L(Z) \in \mathbb{R}^{c\times d}$ the current gradient of Z
Output: $S_{\text{new}} \in \mathbb{R}^{2\times c\times d}$ the updated optimizer state
$U \in \mathbb{R}^{c\times d}$ the additive update matrix

```
1: $M, V \leftarrow S$    ▷ Unpack the states $M, V \in \mathbb{R}^{c\times d}$
2: $M_{\text{new}} \leftarrow \beta_1 \cdot M + (1-\beta_1)\cdot \nabla L(Z)$
3: $V_{\text{new}} \leftarrow \beta_2 \cdot V + (1-\beta_2)\cdot \nabla L(Z)^{\circ 2}$
4: $S_{\text{new}} \leftarrow (M_{\text{new}}, V_{\text{new}})$
5: $M_\star \leftarrow M_{\text{new}}/(1-\beta_1^t)$
6: $V_\star \leftarrow V_{\text{new}}/(1-\beta_2^t)$
7: $U \leftarrow -\eta \cdot M_\star \oslash (V_\star^{\circ\frac{1}{2}} + \epsilon \cdot \mathbf{1}_{c\times d})$
```

B Derivation of the Unified Algorithm of Memory-efficient Subspace Optimizers

As we have described in Section 2, MeSO optimizers solve the subspace optimization problem under the projection matrix $P \in \mathbb{R}^{m\times r}$:

$$\min_{A\in\mathbb{R}^{r\times n}} L(W_0 + PA) \tag{2}$$

by applying an off-the-shelf optimizer `opt`. Since we want to start at the initial weight matrix W_0, A is initialized to be the zero matrix:

$$A^{(0)} \leftarrow 0_{r\times n} \tag{3}$$

$$S^{(0)} \leftarrow \texttt{opt.update}(A^{(0)}) \tag{4}$$

and updated through

$$S^{(t+1)}, \Delta^{(t+1)} \leftarrow \texttt{opt.update}(S^{(t)}, \frac{d}{dA}L(W_0 + PA^{(t)})) \tag{5}$$

$$A^{(t+1)} \leftarrow A^{(t)} + \Delta^{(t+1)} \tag{6}$$

By chain rule, we have $\frac{d}{dA}L(W_0 + PA^{(t)}) = P^\top \nabla L(W_0 + PA^{(t)})$.

When MeSO updates the projection matrix to be P_{new}, we can treat the new subspace optimization as having its $W_0^{\text{new}} = W_0^{\text{old}} + P_{\text{old}}A^{(t)}$ and re-initializing $A^{(t)}$ at $0_{r\times n}$ in addition to an optimizer state update using `update_state`. The pseudocode of this algorithm where we maintain the value of the A matrix is given in Algorithm 5.

Algorithm 5 Memory-efficient subspace optimization (MeSO) with an instantiated A matrix

Input: Initial weights $W_0 \in \mathbb{R}^{m\times n}$ with $m \le n$; update frequency K; total iterations T; subspace rank r with $r \ll m$, an off-the-shelf optimizer `opt`; function to update the optimizer state, scale factor α.
Output: Optimized weights $W^{(T)}$

```
1: $t \leftarrow 0$
2: $A^{(0)} \leftarrow 0_{r\times n}$
3: $S^{(0)} \leftarrow$ opt.init($A^{(0)}$)    ▷ Adam state $\in \mathbb{R}^{r\times n}$
4: while $t \le T$ do
5:     if $t \equiv 0 \pmod K$ then
6:         $W_0 \leftarrow W_0 + PA^{(t)}$    ▷ record progress
7:         $A^{(t)} \leftarrow 0_{r\times n}$    ▷ reinitialize $A$
8:         // Compute new projection matrix
9:         $P \leftarrow$ compute$_P$ ($\nabla L(W_0)$)    ▷ $P \in \mathbb{R}^{m\times r}$
10:        // [Optional] Update optimizer state
11:        $S^{(t)} \leftarrow$ update_state($S^{(t)}$)
12:    end if
13:    $G_C \leftarrow P^\top \nabla L(W_0 + PA^{(t)})$    ▷ $G_C \in \mathbb{R}^{r\times n}$
14:    $S^{(t+1)}, \Delta^{(t+1)} \leftarrow$ opt.update($S^{(t)}, G_C$)
15:    $A^{(t+1)} \leftarrow A^{(t)} + \alpha\Delta^{(t+1)}$    ▷ Apply Update
16:    $t \leftarrow t + 1$
17: end while
```

By defining $W^{(t)} := W_0 + PA^{(t)}$, we can easily see that Algorithm 5 is equivalent to Algorithm 1 presented in the main paper.

C Additional Related Work

Memory-Efficient Optimization. Several works aim to reduce the memory footprint of adaptive optimizer states. Techniques include factorizing second-order moment statistics (Shazeer and Stern, 2018), layerwise sampling (Li et al., 2024; Pan et al., 2024; Luo et al., 2024), quantizing optimizer states (Dettmers et al., 2021; Anil et al., 2019; Dettmers et al., 2023; Li et al., 2023), and fusing backward operations with optimizer updates to minimize gradient storage (Lv et al., 2023a). GRASS is orthogonal to these approaches and proposes a

[7]For any matrix $Z \in \mathbb{R}^{c\times d}$, we have $Z^{\circ 2}$ and $Z^{\circ\frac{1}{2}}$ to respectively denote the matrix which is the elementwise square and elementwise square root of Z.

gradient projection-based adaptive optimizer that significantly reduces memory costs by relying on projected gradient statistics.

Gradient Compression. In distributed and federated training, several gradient compression methods have been introduced to reduce the volume of transmitted gradient data. Common approaches include:

1. **Quantization:** Quantization aims to reduce the bit precision of gradient elements. Examples include 1-bit SGD (Seide et al., 2014), SignSGD (Bernstein et al., 2018), 1-bit Adam (Tang et al., 2021), TernGrad (Wen et al., 2017), and QSGD (Alistarh et al., 2017).
2. **Sparsification:** This involves transmitting only a small subset of significant gradient elements. Random-k and Top-k element select k random or largest-magnitude elements, respectively to transmit. Top-k generally exhibits better convergence (Stich et al., 2018), and requires communicating both values and indices (Lin et al., 2018; Renggli et al., 2019).
3. **Low-Rank Decomposition:** This involves factorizing a gradient matrix $M \in \mathbb{R}^{n\times m}$ as $M \approx PQ^\top$ for transmission, where $P \in \mathbb{R}^{n\times r}$ and $Q \in \mathbb{R}^{m\times r}$ with $r \ll \min(n,m)$. ATOMO (Wang et al., 2018b) employs SVD for decomposition, while Power-SGD (Vogels et al., 2019) utilizes power iteration for more efficient low-rank factorization.

Unlike existing methods, GRASS introduces a novel approach by employing sparse projection of gradients to enhance memory efficiency in both local and distributed training contexts.

D Ensuring Unbiased Gradient Reconstruction

In this section, we formally state the theorem that gives the form of the sampling distribution for σ_j and ρ_{jj} that ensures the reconstructed gradient $PP^\top G_W$ is unbiased which we describe in Section 3.2.

Theorem D.1. *Let $B \in \{0,1\}^{r\times m}$ be the sparse binary matrix with the unique non-zero index of j-th row being $\sigma_j \in [m]$. Let $\sigma_j \overset{i.i.d.}{\sim}$ Multinomial$(1,q)$) ($q \in \mathbb{R}^m$ with the probability of sampling integer $k \in [m]$ being q_k). If we correspondingly let the diagonal value of the diagonal matrix ρ to be $\rho_{jj} := \frac{1}{\sqrt{rq_{\sigma_j}}}$, then for the random projection matrix $P = (\rho B)^\top$, we have $\mathbb{E}[PP^\top G] = G$ for any (gradient) matrix $G \in \mathbb{R}^{m\times n}$.*

Proof. Here we first write down the form of the random matrix product $PP^\top$. Let $e_j \in \mathbb{R}^m$ be the unit column vector with j-th coordinate being 1 and all other coordinates being zero. Then by definition, the j-th row vector of B is $e_{\sigma_j}^\top$.

$$PP^\top \tag{7}$$
$$=B^\top\rho^\top\rho B \tag{8}$$
$$=\begin{bmatrix} | & & | \\ e_{\sigma_1} & \cdots & e_{\sigma_r} \\ | & & | \end{bmatrix}_{m\times r} \times \text{diag}(\frac{1}{r\cdot q_{\sigma_1}},\ldots,\frac{1}{r\cdot q_{\sigma_r}}) \times \begin{bmatrix} -e_{\sigma_1}^\top- \\ \vdots \\ -e_{\sigma_r}^\top- \end{bmatrix}_{r\times m} \tag{9}$$
$$=\frac{1}{r}\sum_{i=1}^r \frac{1}{q_{\sigma_i}} e_{\sigma_i} e_{\sigma_i}^\top \tag{10}$$

In Equation 10, we have decomposed the matrix $PP^\top$ into the average of r random rank-1 matrices each of which depends on on the randomness of a unique σ_i. By linearity of expectation and the *i.i.d.* property of $\{\sigma_i\}_{i=1}^r$, we have

$$\mathbb{E}[PP^\top] = \frac{1}{r}\sum_{i=1}^r \mathbb{E}[\frac{1}{q_{\sigma_i}} e_{\sigma_i} e_{\sigma_i}^\top] \tag{11}$$
$$=\mathbb{E}[\frac{1}{q_{\sigma_1}} e_{\sigma_1} e_{\sigma_1}^\top] \tag{12}$$

Since σ_1 have a probability of q_k to take the value of integer $k \in [m]$, we have

$$\mathbb{E}[\frac{1}{q_{\sigma_1}} e_{\sigma_1} e_{\sigma_1}^\top] \tag{13}$$
$$=\sum_{k=1}^m q_k \cdot \frac{1}{q_k} e_k e_k^\top \tag{14}$$
$$=I_{m\times m} \tag{15}$$

Thus we have proved that $\mathbb{E}[PP^\top] = I_{m\times m}$. By linearity of expectation, for any matrix $G \in \mathbb{R}^{m\times n}$, we thus have $\mathbb{E}[PP^\top G] = G$ and the proof is complete. □

E Proof of Theorem 3.1

Here we restate the complete version of Theorem 3.1 and then present its proof.

Theorem (Complete statement of Theorem 3.1)**.** *Let $B \in \{0,1\}^{r\times m}$ be the sparse binary matrix with the unique non-zero index of j-th row being $\sigma_j \in [m]$. Let $\sigma_j \overset{i.i.d.}{\sim}$ Multinomial$(1,q)$ ($q \in \mathbb{R}^m$ with the probability of sampling integer $k \in [m]$ being q_k). Given σ_j, we correspondingly set the diagonal value of the diagonal matrix ρ to be $\rho_{jj} := \frac{1}{\sqrt{rq_{\sigma_j}}}$ and define $P = (\rho B)^\top$. This induces an unbiased gradient estimator of $G \in \mathbb{R}^{m\times n}$: PP^TG. Among all these gradient estimators induced by different parameter value q of the multinomial distribution, the one that is proportional to the row norms of G with $q_k = \frac{\|G_k\|_2}{\sum_{i=1}^m \|G_i\|_2}$ minimizes the total variance of the gradient estimate $PP^\top G$.*

Proof. We first write down the total variance of the estimator $PP^\top G$:

$$\mathbb{E}\,\text{tr}[(PP^\top G)^\top (PP^\top G)] - \text{tr}[\mathbb{E}[(PP^\top G)]\mathbb{E}[(PP^\top G)]^\top] \tag{16}$$

$$=\text{tr}[G^\top \mathbb{E}[PP^\top PP^\top]G] - \text{tr}[GG^\top] \tag{17}$$

Since only the first term in Equation 17 is a function of P and thus depends on the value of q, we first focus on analytically deriving the form of $\mathbb{E}[PP^\top PP^\top]$.

By the expression in Equation 10, we have:

$$PP^\top PP^\top \tag{18}$$

$$=\frac{1}{r^2}\sum_{i=1}^r\sum_{j=1}^r \frac{1}{q_{\sigma_i}}\frac{1}{q_{\sigma_j}} e_{\sigma_i}e_{\sigma_i}^\top e_{\sigma_j}e_{\sigma_j}^\top \tag{19}$$

$$=\frac{1}{r^2}\sum_{i=1}^r \frac{1}{q_{\sigma_i}^2} e_{\sigma_i}e_{\sigma_i}^\top e_{\sigma_i}e_{\sigma_i}^\top + \frac{1}{r^2}\sum_{i=1,j=1,i\neq j}^r [\frac{1}{q_{\sigma_i}} e_{\sigma_i}e_{\sigma_i}^\top][\frac{1}{q_{\sigma_j}} e_{\sigma_j}e_{\sigma_j}^\top] \tag{20}$$

$$=\frac{1}{r^2}\sum_{i=1}^r \frac{1}{q_{\sigma_i}^2} e_{\sigma_i}e_{\sigma_i}^\top + \frac{1}{r^2}\sum_{i=1,j=1,i\neq j}^r [\frac{1}{q_{\sigma_i}} e_{\sigma_i}e_{\sigma_i}^\top][\frac{1}{q_{\sigma_j}} e_{\sigma_j}e_{\sigma_j}^\top] \tag{21}$$

In the last step, we use the fact that for any i, $e_{\sigma_i}^\top e_{\sigma_i} = 1$. Now we take the expectation of Equation 21. By applying linearity of expectation and the *i.i.d.* property of $\{\sigma_j\}$, we have

$$\mathbb{E}[PP^\top PP^\top] \tag{22}$$

$$=\frac{1}{r}\text{diag}(\frac{1}{q_1},\dots,\frac{1}{q_m}) + \frac{r-1}{r}\cdot I_{m\times m} \tag{23}$$

As a result, we can express the first term in Equation 17 as

$$\text{tr}[G^\top \mathbb{E}[PP^\top PP^\top]G] \tag{24}$$

$$=\frac{1}{r}\text{tr}[G^\top \text{diag}(\frac{1}{q_1},\dots,\frac{1}{q_m})G] + \frac{r-1}{r}\text{tr}[GG^\top] \tag{25}$$

If we represent the rows of G as column vectors $\{G_k\}_{k=1}^m$, then the only term in Equation 25 that depends on q can be expressed as

$$\text{tr}[G^\top \text{diag}(\frac{1}{q_1},\dots,\frac{1}{q_m})G] \tag{26}$$

$$=\text{tr}[\sum_{k=1}^m \frac{1}{q_k}G_kG_k^\top] \tag{27}$$

$$=\sum_{k=1}^m \frac{1}{q_k}\text{tr}[G_kG_k^\top] \tag{28}$$

$$=\sum_{k=1}^m \frac{\|G_k\|_2^2}{q_k} \tag{29}$$

Based on these derivations, to minimize the total variance is therefore equivalent to minimize Equation 29. From now on, we denote $\lambda_i := \|G_i\|_2$ as the 2-norm of the i-th row of matrix G.

Solving the variance-minimization problem: As we have shown, minimizing the total variance of $PP^\top G$ leads to the following optimization problem:

$$\min_p \sum_{i=1}^m \frac{\lambda_i^2}{q_i} \tag{30}$$

$$\text{subject to} \quad \sum_{i=1}^n q_i = 1, \quad q_i \geq 0 \text{ for all } i.$$

Here we first ignore the inequality constraint $q_i \geq 0$ and solve the relaxed problem:

$$\min_p \sum_{i=1}^m \frac{\lambda_i^2}{q_i} \tag{31}$$

$$\text{subject to} \quad \sum_{i=1}^n q_i = 1$$

The Lagrangian L for this relaxed constrained optimization is:

$$L(q, \mu) = \sum_{i=1}^{m} \frac{\lambda_i^2}{q_i} + \mu \left(\sum_{i=1}^{m} q_i - 1 \right)$$

where μ is the Lagrange multiplier for the equality constraint. The stationary condition for the Lagrangian gives us

$$\frac{\partial L}{\partial q_i} = -\frac{\lambda_i^2}{q_i^2} + \mu = 0, \quad \forall i \in [m] \tag{32}$$

$$\sum_{i=1}^{m} q_i = 1 \tag{33}$$

Assuming not all λ_i are zero, this gives us

$$q_i^* = \frac{\lambda_i}{\sum_{j=1}^{m} \lambda_j}$$

Since this optimal solution to Equation 31 also lies in the constraint space of Equation 30, this is also the optimal solution of the optimization we care about.

Thus we have shown that the distribution parameter q that minimizes the total variance of the gradient estimate is proportional to the row 2-norm of G.

□

F Row Norms and Subspace Embedding Property

The following proof is from Magdon-Ismail (2010) which can be roughly stated as sampling with squared row-norms preserves subspaces up to additive error with high probability.

Theorem F.1 (Subspace Preservation)**.** *Let* $\mathbf{A} \in \mathbb{R}^{m \times d_1}$ *with rows* $\mathbf{a}_t$*. Define a sampling matrix* $\mathbf{Q} \in \mathbb{R}^{m \times m}$ *using row-sampling probabilities:*

$$p_t \geq \frac{\|\mathbf{a}_t\|^2}{\|\mathbf{A}\|_F^2}.$$

If $r \geq \frac{4p_A \ln \frac{2d_1}{\delta}}{\beta^2}$*, then with probability at least* $1-\delta$*, it follows that:*

$$\|\mathbf{A}^\top \mathbf{A} - \tilde{\mathbf{A}}^\top \tilde{\mathbf{A}}\| \leq \epsilon \|\mathbf{A}\|^2.$$

Proof. Considering the singular value decompositions (SVDs) of $\mathbf{A}$ and $\mathbf{B}$, we have:

$$\|\mathbf{A}^\top \mathbf{B} - \mathbf{A}^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{B}\| = \|\mathbf{V}_A \mathbf{S}_A \mathbf{U}_A^\top \mathbf{U}_B \mathbf{S}_B \mathbf{V}_B^\top - \mathbf{V}_A \mathbf{S}_A \mathbf{U}_A^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{U}_B \mathbf{S}_B \mathbf{V}_B^\top\|.$$

We may now directly apply Lemma F.2, with respect to the appropriate sampling probabilities. One can verify that the sampling probabilities are proportional to the sum of the rescaled squared norms of the rows of $\mathbf{A}$ and $\mathbf{B}$. □

Lemma F.2 (Sampling in Orthogonal Spaces)**.** *Let* $\mathbf{W} \in \mathbb{R}^{m \times d_1}$ *and* $\mathbf{V} \in \mathbb{R}^{m \times d_2}$ *be orthogonal matrices, and let* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *be positive diagonal matrices in* $\mathbb{R}^{d_1 \times d_1}$ *and* $\mathbb{R}^{d_2 \times d_2}$*, respectively. Consider row sampling probabilities:*

$$p_t \geq \frac{1}{\|\mathbf{S}_1\|_F^2} \mathbf{W}^\top \mathbf{S}_1^2 \mathbf{W}_t + \frac{1}{\|\mathbf{S}_2\|_F^2} \mathbf{V}^\top \mathbf{S}_2^2 \mathbf{V}_t.$$

If $r \geq \left(8(p_1 + p_2)/\beta^2\right) \ln \frac{2(d_1+d_2)}{\delta}$*, then with probability at least* $1 - \delta$*, it holds that:*

$$\|\mathbf{S}_1 \mathbf{W}^\top \mathbf{V} \mathbf{S}_2 - \mathbf{S}_1 \mathbf{W}^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{V} \mathbf{S}_2\| \leq \epsilon \|\mathbf{S}_1\| \|\mathbf{S}_2\|.$$

G Detailed Breakdown of Compute, Memory, and Communication Volume

In this section we provide detailed breakdown of the compute, memory, and communication volume for different optimization methods. We focus our discussion to a single weight matrix $W \in \mathbb{R}^{m \times n}$ (we assume $m \leq n$) and its gradient $G \in \mathbb{R}^{m \times n}$. We describe the relevant notation and parameter shape below:

- By chain rule, we have $G = (\nabla_y L)^\top X$, where $\nabla_y L$ is a $b \times m$ matrix denoting the gradient of the loss with respect to the layer's output activations y, and X is an $b \times n$ matrix denoting the input activations. b is the token batch size, which in practice is usually much larger than both m and n. Here we assume X is computed and stored during the forward pass through this layer and $\nabla_y L$ is computed during the backprop pass from the previous layer. We are interested in the memory, floating-point operations, and communication volume to construct the gradients G, update the optimizer state, and update the parameter weights.
- P is an $m \times r$ projection matrix with $r \ll m$.
- C is the number of optimizer operations per gradient element.
- For GRASS, we can decompose $P^\top = \rho B$ where ρ is a $r \times r$ diagonal scaling matrix, $B \in 0, 1^{r \times m}$ is a sparse binary row selection matrix. Both left multiplication by ρ and B can be computed efficiently.

Method	Regular Step Component	Cost	Projection Update Cost
Full	compute $G_W = (\nabla_y L)^\top X$	mbn	N/A
	optimizer `opt.update`	Cmn	
	weight update $W^{(t+1)} \leftarrow W^{(t)} + \Delta W^{(t)}$	mn	
LoRA $(W = W_0 + BA)$	compute $G_W = (\nabla_y L)^\top X$	mbn	N/A
	compute gradient $\nabla_B L$ and $\nabla_B L$	$2rmn$	
	optimizer `opt.update`	$C(rm + rn)$	
	weight update $B^{(t+1)} \leftarrow B^{(t)} + \Delta B^{(t)}$ $A^{(t+1)} \leftarrow A^{(t)} + \Delta A^{(t)}$	$rn + rm$	
ReLoRA $(W = W_0 + BA)$	Compute $G_W = (\nabla_y L)^\top X$	mbn	merge weights $W_0 \leftarrow W_0 + B^{(t)} A^{(t)}$ $mnr + mn$
	compute gradient for LoRA weights	$2rmn$	
	optimizer `opt.update`	$C(rm + rn)$	
	weight update $B^{(t+1)} \leftarrow B^{(t)} + \Delta B^{(t)}$ $A^{(t+1)} \leftarrow A^{(t)} + \Delta A^{(t)}$	$rn + rm$	
GaLore	compute $G = (\nabla_y L)^\top X$	mbn	SVD of G_W $mn \min(n, m)$
	compute $P^\top G$	rmn	
	optimizer `opt.update`	Crn	
	compute $\alpha P \Delta^{(t+1)}$	rmn	
	weight update $W^{(t+1)} \leftarrow W^{(t)} + \alpha P \Delta^{(t+1)}$	mn	
Flora	compute $G = (\nabla_y L)^\top X$	mbn	sample the Gaussian matrix $P_{ij} \overset{i.i.d.}{\sim} \mathcal{N}(0, 1/r)$ mr
	compute $P^\top G$	rmn	
	optimizer `opt.update`	Crn	
	compute $\alpha P \Delta^{(t+1)}$	rmn	
	weight update $W^{(t+1)} \leftarrow W^{(t)} + \alpha P \Delta^{(t+1)}$	mn	
Efficient GaLore	compute $P^\top (\nabla_y L)^\top$	rmb	SVD of G_W $mn \min(n, m))$
	compute $(P^\top (\nabla_y L)^\top) X$	rbn	
	optimizer `opt.update`	Crn	
	compute $\alpha P \Delta^{(t+1)}$	rmn	
	weight update $W^{(t+1)} \leftarrow W^{(t)} + \alpha P \Delta^{(t+1)}$	mn	
Grass (ours)	compute $B(\nabla_y L)^\top$	0	compute row norms and perform multinomial sampling* $mn + m + r^\dagger$
	compute $(B(\nabla_y L)^\top) X$	rbn	
	compute $\rho((B(\nabla_y L)^\top) X)$	rn	
	optimizer `opt.update`	Crn	
	compute $\alpha P \Delta^{(t+1)}$ (only need to compute the non-zero rows)	rn	
	parameter update $W^{(t+1)} \leftarrow W^{(t)} + \alpha P \Delta^{(t+1)}$ (only need to compute the non-zero rows)	rn	

Table 6: Detailed FLOPs Analysis for Various Methods. †This is the complexity of Alias Method for multinomial sampling. For the deterministic method Top-r, the total complexity would be $mn + m \log r$ using a heap.

We compare various optimization strategies: **Full**, **GaLore**, **LoRA**, **ReLoRA**, **Flora**, and our proposed method **Grass**. All numbers for each method are computed based on the implementation original papers. We additionally consider **Efficient GaLore**, which combines GaLore with our proposed efficient regrouped matrix multiplication implementation for reduced FLOPs and a custom hook for reduced communication. As we shall see, even compared to this more efficient implementation of GaLore, our method Grass still enjoys competitive advantages.

Compute Requirements

Table 6 details the FLOPs (per worker) calculation for the baselines and Grass. We provide a breakdown of the computation cost of each step in the Regular optimization step as well as the computation cost of computing the new projection matrix. As we can see, Grass is considerably more compute-efficient than all other methods – most importantly, its compute cost does not contain the most expensive term mbn unlike all the other published methods. Although Efficient GaLore also avoids full parameter gradient computation mbn by using our proposed multiplication rule, it still pays a much higher cost when it computes and per-

Method	Weights	Optimizer State	Gradient Memory
Full	mn	$2mn$	mn
LoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$
ReLoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$
GaLore	mn	$mr + 2nr$	mn
Flora	mn	$mr + 2nr$	mn
Efficient GaLore	mn	$mr + 2nr$	nr
Grass (ours)	mn	$2r + 2nr$	nr

Table 7: Memory Requirements for Various Methods. Note that memory cost for the projection update step is intermittent and not included.

Method	Comm Volume
Full	mn
LoRA	$mr + nr$
ReLoRA	$mr + nr$
GaLore	mn^*
Flora	mn^*
Efficient GaLore	nr
Grass (ours)	nr

Table 8: Gradient Communication Volume for Various Optimizers. * Note that GaLore and Flora communication volume can be reduced to nr using a communication hook.

forms the weight update $(rmn + mn)$ compared to Grass $(2rn)$.

Memory Requirements

Table 7 summarizes the memory requirements for the various baselines and Grass when we use Adam as the (internal) optimizer for each method.

- In terms of storing the weight parameters, every method needs to store the full parameter matrix of shape $m \times n$, while LoRA and ReLoRA also requires storing the low-rank updateable parameters (the B and A matrix)
- In terms of the optimizer state, LoRA and ReLoRA needs to store both the first and second moment estimates for its B and A matrix. For all the MeSO methods, the optimizer state of the implicit A matrix needs to be stored. Besides, these methods also need to store the projection matrix P. Here, unlike the other MeSO methods which employ dense P matrices, Grass can store its sparse projection matrix P using $2r$ numbers instead of mr numbers.
- In terms of the gradient memory, with our proposed regrouped matrix multiplication implementation, Grass never materializes the full parameter's gradient matrix, thus reducing the gradient memory size to only the projection result of shape $r \times n$.

Communication Volume

Table 8 summarizes the communication volume of gradients (per device) for various methods when we use distributed data parallel (DDP) training. Here all the existing methods perform all-reduce on the full-parameter gradient. In contrast, Grass never materializes the full paramater gradient and performs all-reduce directly on the projected matrix, saving the communication volume from mn to nr.

H Distributed Data Parallel Implementation

To optimize memory usage in PyTorch's Distributed Data Parallel (DDP) framework (Paszke et al., 2019), we implement strategic modifications to our model architecture aimed at enhancing distributed training efficiency (see Algorithm 6). Specifically, we designate the weights in the linear layers as non-trainable to circumvent the default memory allocation for full-sized gradient matrices. Instead, we introduce virtual, trainable parameters— occupying merely 1 byte each—linked to each weight matrix. These virtual parameters hold the compressed gradient of the corresponding weight matrix in the `wgrad` attribute. This method capitalizes on DDP's asynchronous all-reduce capabilities while preventing unnecessary memory allocation.

I Experiment Hyperparameters

I.1 Pretraining

We introduce details of the LLaMA architecture and hyperparameters used for pretraining. Table 9 shows the dimensions of LLaMA models across model sizes. We pretrain models on the C4 subset of Dolma [8]. C4 is a colossal, clean version of Common Crawl designed to pretrain language models

[8] https://huggingface.co/datasets/allenai/dolma

Algorithm 6 Distributed GRASS Training with PyTorch DDP illustrated for a single layer

```
Input: Initial weights $W_0 \in \mathbb{R}^{m \times n}$, total iterations $T$, subspace rank $r$, world size $p$, learning rate scale $\alpha$, update frequency $K$
Output: Optimized weights $W^{(T)}$
1: Initialize distributed environment (e.g., NCCL)
2: $W \leftarrow W_0$                                   ▷ Set weights as non-trainable
3: Introduce virtual trainable parameter vparams $\in \mathbb{R}^{1\times 1}$, linked to each weight matrix
4: vparams.wgrad $\leftarrow \emptyset$                  ▷ Initialize storage for compressed gradients
5: Initialize a DDP model with custom gradient hooks
6: for $t = 0$ to $T - 1$ do
7:     Compute local loss $L$ for the current mini-batch
8:     $output \leftarrow$ Forward pass using $W$
9:     if $t \equiv 0 \pmod{K}$ then
10:        Compute backward pass to obtain full gradient $G_W$
11:        // Sketch gradient using column norms and select top-$r$
12:        $G_{sketch} \leftarrow$ ToprColumns$(G_W, r)$
13:        // All-reduce and update the sketched matrix
14:        $G_{sketch} \leftarrow$ AllReduceMean$(G_{sketch})$
15:        Update projection matrix $P$ using $G_{sketch}$, compute and store compressed gradient $G_C$ in vparams.grad. Release
   memory for full gradient $G_W$.
16:    else
17:        Compute backward pass, capturing compressed gradients $G_C$ in vparams.grad
18:        Perform all-reduce on vparams.grad across all workers
19:    end if
20:    Update $W$ using vparams.grad
21: end for
22: return $W$
23:
24: function TOPRCOLUMNS($grad$, $r$)
25:     $indices \leftarrow$ argsort(|colnorms($grad$)|)[$-r$ :]     ▷ Identify indices of top-r column norms
26:     return $grad[:, indices]$
27: end function
```

and word representations in English (Raffel et al., 2019).

For pretraining all models we use a max sequence length of 256 for all models, with a batch size of 262144 tokens. For all baseline experiments, we adopt learning rate warmup for the first 1000 steps, and use cosine annealing for the learning rate schedule, decaying to 10% of the initial learning rate. GRASS, GALORE and FLORA use a projection matrix update frequency of 200. GRASS uses an additional warmup at each update for 200 steps and resets optimizer states for the 60M and 350M training runs, while the 1B run does not require resetting optimizer states. Both 60M and 350M GRASS pretraining jobs uses Top-r selectionwhile the 1B job uses Multinomial sampling without replacement.

For all methods on each size of models, we tune learning rate from a set of {0.01, 0.005, 0.001, 0.0005, 0.0001}, and the best learning rate is chosen based on the validation perplexity (or train perplexity when a validation does not exist as in Dolma). All MeSO models use a scale factor $\alpha = 0.25$. We find that GALORE is sensitive to hyperparameters and exhibits loss spikes and divergence at the prescribed learning rates in the paper (0.01) particularly at the 1B scale, and as a result we have to train using reduced learning rates where we no longer observe such spikes. The learning rates of GRASS and GALORE are higher than the full model which would display instability at values greater than 0.001. Unless otherwise specified, we average losses using a window of 15 steps. We use Adam with the default hyperparameters ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$).

All models were trained on four 80GB A100 GPUs. The training times were as follows: 100 GPU hours for the 60M model, 200 GPU hours for the 250M model, and 650 GPU hours for the 1B model.

I.2 Finetuning

We finetune the pretrained RoBERTa-Base[9] model (Liu et al., 2019) on the GLUE benchmark[10] (Wang et al., 2018a) using the pretrained model on Hugging Face. GLUE is a natural language understanding benchmark and includes a variety of tasks, including single sentence tasks like CoLA (Warstadt et al., 2018), SST-2 (Socher et al., 2013); similarity and paraphrase tasks like MRPC (Dolan and

[9]https://huggingface.co/FacebookAI/roberta-base

[10]https://huggingface.co/datasets/nyu-mll/glue

Params	Hidden	Intermediate	Heads	Layers	Steps	Data amount
60M	512	1376	8	8	3.8K	1.0B
350M	1024	2736	16	24	20.6K	5.4B
1B	2048	5461	24	32	33.6K	8.8B
7B	4096	11008	32	32	-	-
13B	5120	13824	40	40	-	-

Table 9: Model dimensions for the various LLaMA models. We report the training steps and data amount in tokens for the 60M, 350M, and 1B models.

	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B
Batch Size	32	32	32	32	32	32	32	32
# Epochs	3	3	3	3	3	3	3	3
Learning Rate	2E-05	2E-05	3E-05	2E-05	2E-05	2E-05	2E-05	2E-05
Rank Config.	$r = 8$	$r = 8$	$r = 8$	$r = 8$	$r = 8$	$r = 8$	$r = 8$	$r = 8$
α	2	2	2	2	2	2	2	2
Max Seq. Len.	128	128	128	128	128	128	128	128

Table 10: Hyperparameters of finetuning RoBERTa base for GRASS.

Brockett, 2005), QQP, STS-B (Cer et al., 2017); and inference tasks such as MNLI (Williams et al., 2017), QNLI (Rajpurkar et al., 2016), RTE and WNLI (Levesque et al., 2012).

We report accuracy for SST-2, MNLI, QNLI and RTE. For CoLA and STS-B, we use Matthew's Correlation and Pearson-Spearman Correlation as the metrics, respectively. For MRPC and QQP, we report the average of F1 score and accuracy. We report the best performance out of three seeds due to the instability of the method. We train all models for 3 epochs using a max sequence length of 128, and a batch size of 32. We report the best performance at the end of an epoch. We use a projection update frequency of 100 for all methods. We tuned the learning rate and scale factor α for GALORE, FLORA, LoRA and GRASS from $\{1e-5, 2e-5, 3e-5, 4e-5, 5e-5\}$ and scale factors $\{1, 2, 4, 8, 16\}$. We apply the projection matrices or LoRA to target modules "query", "value", "key", "intermediate.dense" and "output.dense" and use a rank $r = 8$. We use Adam with the default hyperparameters ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$). All experiments were run on a single A100 GPU in under 24 hours.

Table 10 shows the hyperparameters used for finetuning RoBERTa-Base for GRASS.

I.3 Instruction Tuning

We finetune the pretrained LLaMA 7B [11] model from HuggingFace on the 52k samples from Alpaca [12], and the 100k samples from Flan-v2 in Tulu [13]. We evaluate the finetuned model on the MMLU [14] benchmark (Hendrycks et al., 2020), which covers 57 tasks including elementary mathematics, US history, computer science, and law.

Method	Alpaca	Flan
LoRA	1×10^{-4}	1×10^{-4}
GRASS	1×10^{-6}	5×10^{-6}
Full	1×10^{-5}	1×10^{-5}
GALORE	1×10^{-6}	1×10^{-6}
FLORA	1×10^{-6}	1×10^{-6}

Table 11: Learning rates for the different methods for instruction finetuning on Alpaca and Flan-v2.

We use a constant learning rate that we tune in $\{1e-5, 2e-5, 3e-5, 4e-5, 5e-5\}$ for each method and use a constant scale factor $\alpha = 16$. (see Table 11). We use Adam with the default hyperparameters ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$). Additionally, we use a source and target sequence length of 512.

All experiments use 4 A100 80GB GPUs and take about 48 GPU hours overall.

Alpaca Prompt Format The Alpaca prompt format is designed to generate context-dependent text completions. Here, the prompt consists of a task description followed by specific input providing further context. An example of the structured prompt

[11] https://huggingface.co/huggyLLaMA/LLaMA-7b

[12] https://huggingface.co/datasets/tatsu-lab/alpaca

[13] https://huggingface.co/datasets/arazd/tulu_flan/

[14] https://huggingface.co/datasets/cais/mmlu

in Alpaca is provided below:

```
ALPACA_PROMPT_DICT = {
"prompt_input": (
    "Below is an instruction that describes a
    task, paired with an input that provides
    further context. Write a response that
    appropriately completes the request.
    \n\n### Instruction:\n{instruction}\n\n
    ### Input:\n{input}\n\n### Response: "
),
"prompt_no_input": (
    "Below is an instruction that describes a
    task. Write a response that appropriately
    completes the request.\n\n###
    Instruction:\n{instruction}
    \n\n### Response: "
),
}
```

Flan Prompt Format The FLAN-v2 dataset in the JSON Lines format contains detailed conversational exchanges between a user and an assistant. Each line in the raw file represents a single conversational instance, encapsulated as a JSON object with multiple messages. Our processing script reads these lines and formats them:

- iterates over each line in the file, parsing the JSON to extract the conversation.
- collects and concatenates all user messages to form the input text for each instance.
- extracts the assistant's response to form the corresponding output text.
- outputs a simplified JSON structure with 'input' and 'output' fields for each conversational instance.

I.4 Throughput Benchmarking

We benchmark pretraining throughput on a single 80GB A100 GPU and AMD EPYC 7763 64-Core Processor using a total batch size of 1024, rank 64, and a sequence length of 256 across models. We use the following per device batch sizes: 60M (256), 350M (64), 1B (16), 7B (16), 13B (1). The 7B model runs into OOM when training with Full rank so the estimated throughput is only for the forward and backward pass without an optimizer update (overestimate). GaLore and Full unlike Grass cannot train 13B model on the 80GB GPU so we skip this data point. The throughput estimate is based on 200 iterations on the C4 dataset.

We benchmark finetuning throughput on a single 80GB A100 GPU using a total batch size of 1024, rank 64, and a sequence length 256 across models. We use the following per device batch sizes: 60M (256), 350M (64), 1B (16), 7B (16), 13B (1). Grass, GaLore, and LoRA are only applied to the attention and MLP linear layers while the other weights are set as non-trainable. The throughput estimate is based on 200 iterations.

I.5 Communication Benchmarking

For the weak scaling throughput experiments we use a local batch size of 16, a total batch size of 16 $\times$ num_workers and a projection rank of 256 across all methods and model sizes.

I.6 Ablations

For the ablation experiments **Effect of Update Frequency** and **`compute`$_P$** Methods, we pretrain using 500M tokens from the RealNews subset of C4 (Raffel et al., 2020). The RealNews subset[15] contains 1.81M lines in the train set and 13.9K lines in the validation set.

J Experiments: Pretraining Memory

For estimating memory for pretraining we use a token batch size of 256 and a rank $r = 128$ across models. We do not use the layerwise trick in Zhao et al. (2024) since this is currently inefficient during distributed training. As the GPU memory usage for a specific component is hard to measure directly, we estimate the memory usage of the weight parameters and optimizer states for each method on different model sizes. The estimation is based on the number of original parameters, the model dimensions, and the number of low-rank parameters, all trained in BF16 format.

As an example, to estimate the memory requirements for the 13B model, we compute memory consumption across different components: activations, parameters, gradients, and optimizer states.

Parameter Definitions Let the following variables define our 13B model's configuration:

- L: sequence length (256)
- B: batch size (1)
- D: model hidden size (5120)
- N: number of layers (40)
- H: number of attention heads (40)
- V: vocabulary size (32000)
- r: rank (128)

J.1 Activation Memory Calculation

The activation memory calculation is conducted by accounting for each significant computation within the model layers, including attention mechanisms and feed-forward networks. Each term in Figure 8

[15] https://huggingface.co/datasets/allenai/c4

$$
\begin{aligned}
\text{Layer Normalization} &= B \cdot L \cdot D \cdot 2 \\
\text{Embedding Elements} &= B \cdot L \cdot D \\
\text{QKV} &= \text{Embedding Elements} \cdot 2 \\
\text{QKT} &= 2 \cdot \text{Embedding Elements} \cdot 2 \\
\text{Softmax} &= B \cdot H \cdot L^2 \cdot 2 \\
\text{PV} &= \frac{\text{Softmax}}{2} + \text{Embedding Elements} \cdot 2 \\
\text{Out Projection} &= \text{Embedding Elements} \cdot 2 \\
\text{Attention Block Activation} &= \text{Layer Normalization} + \text{QKV} + \text{QKT} + \text{Softmax} + \text{PV} + \text{Out Projection} \\
\text{FF1} &= \text{Embedding Elements} \cdot 2 \\
\text{GELU} &= \text{Embedding Elements} \cdot 4 \cdot 2 \\
\text{FF2} &= \text{Embedding Elements} \cdot 4 \cdot 2 \\
\text{Feed-Forward Activation} &= \text{Layer Normalization} + \text{FF1} + \text{GELU} + \text{FF2} \\
\text{Final Layer Activation} &= \text{Embedding Elements} \cdot 2 \\
\text{Model Activations} &= \text{Layer Normalization} + (N \cdot (\text{Attention Block Activation} + \text{Feed-Forward Activation})) \\
&\quad + \text{Final Layer Activation} \\
\text{Cross-Entropy Loss} &= B \cdot L \cdot V \cdot 2 + B \cdot L \cdot V \cdot 4 \\
\text{Total Cross-Entropy} &= \text{Cross-Entropy Loss} \\
\text{Total Activation Memory} &= \text{Model Activations} + \text{Total Cross-Entropy}
\end{aligned}
$$

Figure 8: Activation memory estimation for the different baselines.

considers the BF16 precision used for storing the activations.

J.2 Memory Calculation for Parameters and Gradients

Memory for parameters and gradients is estimated as follows:

- Total number of parameters across all layers: Computed by summing up all parameter tensors within the model.
- Parameter memory in bytes: Total number of parameters multiplied by 2 (assuming BF16 precision).
- Gradient memory: For Full-rank and GaLore this equals the parameter memory if all parameters are trainable and gradients are stored in BF16. For Grass this equals the projected gradient memory corresponding to the trainable parameters.

J.3 Optimizer State Memory Calculation

- The Adam optimizer in pure BF16 precision stores the first and second moment estimates for each parameter, requiring $2mn$ floats for a weight matrix with dimensions $m \times n$.
- MeSO methods, including Grass, reduce optimizer state memory by projecting gradients into a lower-dimensional subspace. Grass, using sparse projections, needs $2r+2nr$ floats to store the first and second moment estimates of the compressed gradient ($G_C \in \mathbb{R}^{r\times n}$) and the sparse projection matrix ($P \in \mathbb{R}^{m\times r}$). GaLore and Flora, which use dense projection matrices, require $mr + 2nr$ floats for the optimizer states.

J.4 Total Memory Estimation

The total memory required for the model during training is calculated by summing the memory for parameters, gradients, activations, and optimizer states, along with any additional memory overhead as per the adaptation method used.

For Grass applied to the 13B model, the memory costs are detailed as follows:

- Total Parameters: Approximately 13 Billion
- Activation Memory: 1936.25 MB
- Parameter Memory: 24825.79 MB
- Gradient Memory: 1230.79 MB
- Optimizer State Memory: 2461.72 MB
- Extra Memory (for largest parameter tensor): 312.50 MB
- Total Memory: 30767.05 MB

K Experiment: Finetuning Memory

In Figure 9 and Figure 10, we compare the finetuning memory footprint of Grass and LoRA when finetuning a LLaMA model at various scales (350M, 1B, 7B) using token batch sizes of 256 and

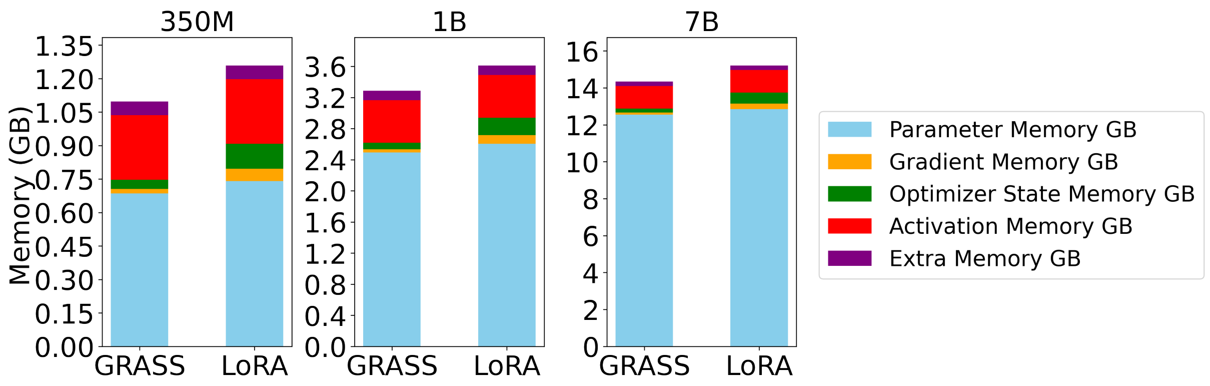

Figure 9: LLaMA finetuning memory footprint of GRASS (Top-r) and LoRA for rank $r = 64$, sequence length 256, batch size 1.

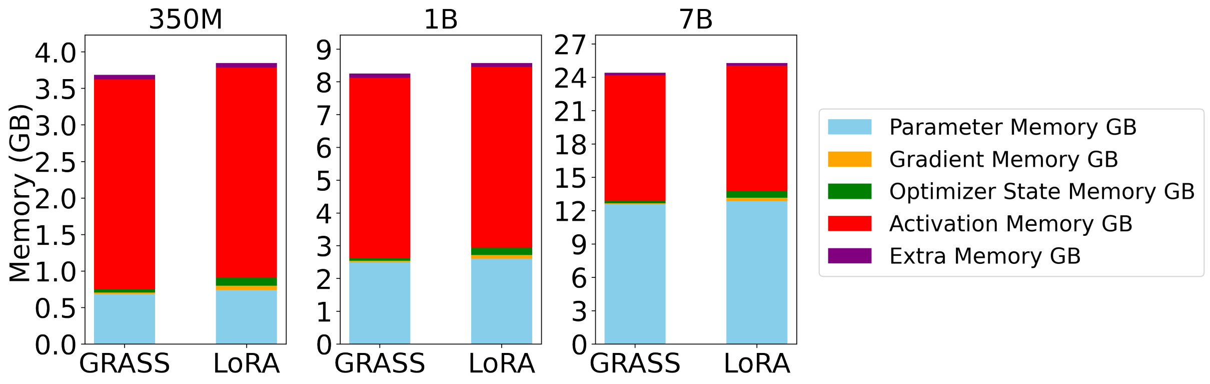

Figure 10: LLaMA finetuning memory footprint of GRASS (Top-r) and LoRA for rank $r = 64$, sequence length 512, batch size 4.

2048 (4×512), respectively. Both methods are applied to all linear layers with a fixed rank of 64. Our analysis reveals that at larger batch sizes, activations predominantly contribute to the memory footprint, resulting in comparable memory usage between GRASS and LoRA.

We estimate memory requirements for finetuning using the same aproach from Section J but only accounting for the gradients and optimizer states corresponding to the trainable (instead of all the) parameters. Furthermore, LoRA requires storing in addition to X (the input to the layer), the activations corresponding to the low-rank input XA to compute the gradient of B, where A and B are the low-rank adapters (Zhang et al., 2023). This results in an additional memory requirement for LoRA of $2BLr$ bytes per linear layer.

L Experiments: Throughput

Figure 11 compares the normalized pretraining throughput (using the Full model) of GRASS and GALORE across 60M, 350M, and 1B model sizes. We find that the throughput advantage of GRASS over GALORE and Full is $> 25\%$ for the 1B model at rank 64. The throughput approaches that of the full model, as model size decreases or projection rank increases.

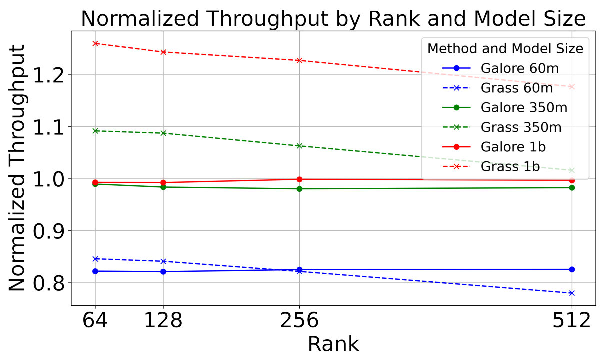

Figure 11: Rank vs. Pretraining Throughput for GRASS (Top-r), LoRA and GALORE across 60M, 350M, 1B, and 7B model sizes.

Figure 12 compares the finetuning throughput across ranks 8, 16,32, and 64 for the GRASS, GALORE, and LoRA baselines. For the ranks commonly used for finetuning (8-64) the throughput advantage of GRASS remains about the same.

We note that for the same rank, GRASS updates fewer parameters per iteration than LoRA, as it

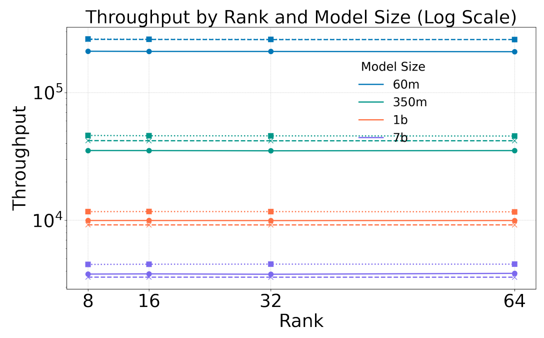

Figure 12: Rank vs LoRA Normalized Finetuning Throughput for GRASS (Top-r) and GALORE across 60M, 350M, and 1B model sizes.

	Train Perp	Eval Perp
Full-Rank	33.48	31.41
GRASS (Top-r)	**33.52**	32.17
GALORE	33.68	**32.10**
ReLoRA	34.30	34.19
FLORA	35.91	35.62
CountSketch	36.97	36.93

Table 12: Comparison of various baselines using 1B LLaMA model validation perplexity. All models are pretrained on 500M tokens of the RealNews subset of C4. r/d_{model} is 256/2048. Best baseline is bolded.

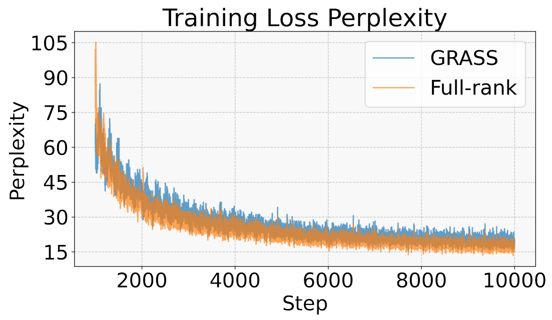

Figure 13: Pretraining LLaMA 1B on Realnews C4 subset with Adafactor. We compare GRASS (Top-r) and Full-rank.

only updates nr parameters while LoRA updates $nr + mr$ parameters (see Table 1). The higher throughput of GRASS is attributed to a combination of reduced FLOPs, fewer memory transfers, better kernel utilization, and a balanced compute-memory workload. LoRA incurs additional overhead from two extra matrix multiplications per layer during the forward pass, as well as the need to construct the full gradient in the backward pass, which increases data movement and computational cost. GRASS avoids these overheads by not requiring the full gradient during regular steps.

M Experiments: Additional Ablations

Comparison with other baselines In Table 12, we report the validation perplexity of various other baselines on a LLaMA 1B pretraining task on the RealNews subset of C4. The attention and feedforward layers in all models are projected to a rank of 256, or use low rank adapters of this rank. We find that the training perplexities are lower while the validation perplexities are higher than in Table 5 for the 60M model due to overfitting on the RealNews dataset. All models use an update frequency of 200, and we tune the learning rate and scale factor α per model.

In addition to GRASS and GALORE, we also include the ReLoRA baseline (Lialin et al., 2023) without any full-rank training warmup, the FLORA baseline where P has entries drawn from $\mathcal{N}(0, 1/r)$, and the CountSketch baseline where $P^\top$ is a CountSketch matrix with r rows with one nonzero entry from $\{\pm 1\}$ per column. The CountSketch projection has been previously applied to embedding layer gradients which are sparse in prior work (Spring et al., 2019), but shows larger variance and poorer convergence rates for dense gradients.

We see that GRASS is competitive with GALORE, while ReLoRA, FLORA, and CountSketch fall short. One way to interpret this is in terms of variance of the gradient sketches— GRASS being data dependent and based on row norms can better approximate the gradient low rank subspace than a data agnostic sketch like FLORA or CountSketch (Woodruff, 2014).

GRASS with Adafactor We pretrain the LLaMA 1B model with GRASS and Full-rank in BF16 on the Realnews subset of C4 using the Adafactor optimizer (Shazeer and Stern, 2018) as an alternative to Adam for opt. Adafactor achieves sub-linear memory cost by factorizing the second-order statistics using a row-column outer product.

For GRASS we use learning rate 0.005, $\alpha = 0.25$, $r = 256$, $K = 200$, batch size 512, optimizer restart with a restart warmup of 100 steps and no initial warmup. For Full-rank training, we use learning rate 0.0005, batch size 512, and 1000 initial linear learning rate warmup steps.

In Figure 13 we report the train perplexity and see that GRASS is within 1 perplexity point of Full-rank, demonstrating its ability to work with other

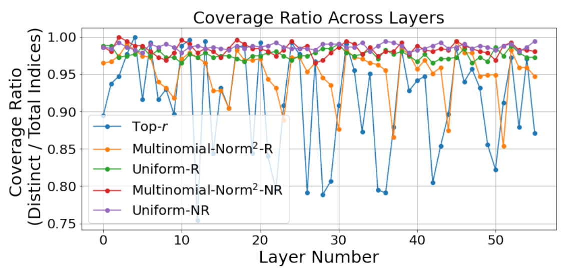

Figure 14: Per layer indices coverage (Distinct/Total) for the sampling strategies across 100 pretraining iterations.

inner off-the-shelf optimizers beyond Adam.

Coverage of indices. In Figure 14, we plot the coverage defined as the union of indices selected over n update projection steps divided by the total indices per layer. We plot the coverage for the 60M LLaMA model pretrained on the C4 RealNews subset, for $n = 15$ updates with $K = 200$ steps between updates. Here, with the rank 128 and the the number of rows $m = 512$, a uniform sampling with replacement over 15 iterations should on average cover $1 - \left(\left(1 - \frac{1}{512}\right)^{128}\right)^{15} \approx 97.66\%$ of all the 512 indices in each layer. Empirically, all sampling methods exhibit good coverage with the Multinomial-Norm^2-NR being close to uniform. Top-r and Multinomial-Norm^2-R oversample indices in certain layers, suggesting potential areas for further investigation into their utility in pruning strategies.

In Figure 15 and Figure 16 we plot the aggregated sampled indices over 15 iterations of 60M LLaMA pretraining on the RealNews subset of C4. We see that while Multinomial-Norm^2-NR and Top-r attain similar performance in terms of perplexity, the sampled indices can be quite different, with Top-r tending to oversample indices in particular layers.

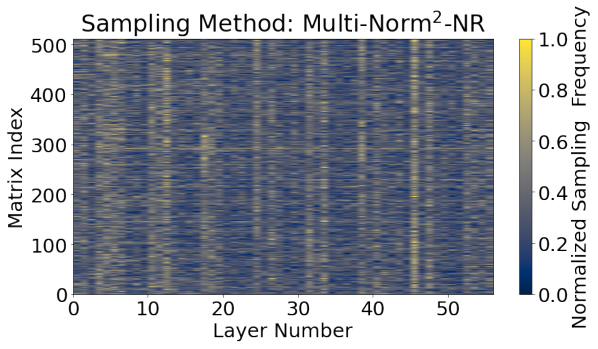

Figure 15: Multinomial-Norm^2 Sampling without Replacement: Heatmap of indices sampled for the different layers across 15 iterations of LLaMA 60M C4 pretraining.

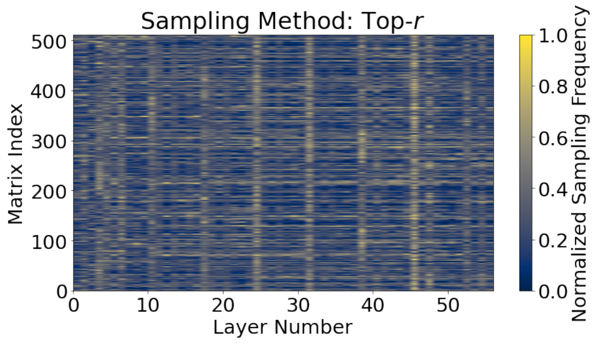

Figure 16: Top-r Selection: Heatmap of indices sampled for the different layers across 15 iterations of LLaMA 60M C4 pretraining.