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]

Abstract

Large language model (LLM) training and finetuning are often severely constrained by limited GPU memory. While parameter-efficient finetuning techniques like LoRA address this by learning low-rank weight updates, they frequently underperform compared to full-rank training, especially during pretraining. We propose GRASS (GRAdient Stuctured Sparsification), a novel approach that slashes LLM training memory and compute requirements without compromising performance. GRASS leverages sparse projections to transform gradients into structurally sparse gradients, significantly lowering memory usage for both optimizer states and gradient communication. This compression, in turn, unlocks substantial throughput improvements. Extensive experiments on pretraining and finetuning tasks demonstrate that GRASS achieves competitive performance to existing projection-based optimizers and full-rank training. Notably, GRASS enables pretraining a 13B parameter LLaMA model on a single 40GB A100 GPU—a feat infeasible for previous methods—and yields up to a 2× throughput improvement on an 8-GPU system. Code can be found at https://github.com/aashiqmuhamed/GRASS.

1. Introduction

Pretraining and finetuning large language models (LLMs) are often memory bottlenecked: storing model parameters, gradients, and optimizer states in GPU memory is prohibitively expensive. As an example, pre-training a LLaMA-13B model from scratch under pure 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 industry-standard 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) techniques, such as Low-Rank Adaptation (LoRA), that reparameterizes a weight matrix $w \in \mathbb{R}^{m\times n}$ as $w = w_0 + BA$, where w_0 is a fixed full-rank matrix, and $B \in \mathbb{R}^{m\times r}$ and $A \in \mathbb{R}^{r\times n}$ serve as additive low-rank adaptors. As rank $r < \min(m, n)$, A and B include a reduced number of trainable parameters, yielding smaller optimizer states and gradients. Variants of LoRA, such as ReLoRA, are employed in pre-training by periodically updating w using previously learned low-rank adaptors. Despite their widespread use, 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). ReLoRA also requires an initial full-rank model warm-up for pretraining, which is costly and impractical. This is because the optimal weight matrices are not necessarily low-rank, and reparameterization alters the gradient training dynamics, failing to recover the original training behavior even when A and B are full-rank (Lialin et al., 2023).

Another class of memory efficient methods directly design efficient adaptive optimizers (Shazeer & Stern, 2018); we look at one class of such optimizers called *gradient projection-based adaptive optimizers* in this work. As detailed in Algorithm 1, these algorithms compute projection matrices $P_t \in \mathbb{R}^{m\times r}$ to project the gradient matrix G into a low dimensional G_c. The memory cost of the optimization states which rely on the projected gradient statistics can be significantly reduced. As using a fixed P_t throughout training limits the expressiveness of the model, recent research

[1]Language Technologies Institute [2]Machine Learning Department [3]Department of Computer Science, Carnegie Mellon University. Correspondence to: Aashiq Muhamed <amuhamed@andrew.cmu.edu>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

Algorithm 1 Memory Efficient Subspace Optimization With Adam Optimizer. While existing methods consider dense projection matrix P, we instead consider sparse projection matrices to enjoy further computation and memory efficiency.

Input: Initial parameters $w_0 \in \mathbb{R}^{m\times n}$ with $m \leq n$, learning rate γ, update frequency T, rank r, projection function $P(\cdot)$, decay rates β_1, β_2, gradient func $\nabla L(\cdot)$, scale factor $\alpha \in \mathbb{R}$.
Output: Optimized parameters w_T

Initialize optimizer state:
$t \leftarrow 0$
$M_0, V_0 \leftarrow 0^{r\times n}, 0^{r\times n}$ ▷ *Initialize moments*
$s_0 \leftarrow$ initial random seed (optional)
Training procedure:
while not converged **do**
  $G \leftarrow \nabla L(w_t)$ ▷ *Compute full gradient*
  if $t \bmod T = 0$ **then**
    $s_{t+1} \leftarrow$ new random seed (optional)
    $P_{t+1} \leftarrow P(G, s_{t+1})$ ▷ *Update projection*
  else
    $s_{t+1} \leftarrow s_t$
    $P_{t+1} \leftarrow P_t$
  end if
  $G_c \leftarrow P_{t+1}^\top G$ ▷ *Compress gradient*
  $M_{t+1} \leftarrow \beta_1 M_t + (1-\beta_1) G_c$ ▷ *Update momentum*
  $V_{t+1} \leftarrow \beta_2 V_t + (1-\beta_2) G_c^2$ ▷ *Update variance*
  $\hat{M} \leftarrow M_{t+1}/(1-\beta_1^{t+1})$
  $\hat{V} \leftarrow V_{t+1}/(1-\beta_2^{t+1})$
  $G_r \leftarrow \alpha P_{t+1} \hat{M}/(\sqrt{\hat{V}} + \epsilon)$ ▷ *Decompress gradient*
  $w_{t+1} \leftarrow w_t - \gamma G_r$ ▷ *Update parameters*
  $t \leftarrow t+1$
end while

proposes periodically updating P_t every T iterations. Different methods vary in the choice of P: FLORA (Hao et al., 2024) resamples each entry of P_t from *i.i.d.* $\mathcal{N}(0, 1/r)$, while GALORE (Zhao et al., 2024) applies singular value decomposition (SVD) to the full parameter gradient G every T-th iteration and selects P as the top-r left singular vectors. The selected P_t in these works, however, are *dense* matrices which introduces additional memory and compute overhead to apply. In our work, we explore *structured sparse* matrices as an alternative choice for P_t and demonstrate its advantage in computation, communication, and memory efficiency. Our main contributions include:

1. We introduce GRASS, a novel method that enables full parameter training of LLMs with structurally 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 sampling distributions for constructing GRASS.
2. We conduct extensive experiments on both pretraining and finetuning tasks, demonstrating that GRASS achieves comparable performance to existing projection-based methods at the same iteration complexity. GRASS also exhibits minimal performance degradation (<0.1 perplexity gap) compared to full-rank training on the 1B LlaMA model while achieving a 2.5× reduction in memory footprint.
3. We present an efficient PyTorch implementation of GRASS optimized 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 the Full model and GALORE. Furthermore, GRASS's low memory footprint enables training a 13B LlaMA model with rank 768 on a single 40GB A100 GPU, a feat that existing projection based optimization methods cannot achieve.

2. Methodology

2.1. GRASS: Subspace Optimization with Structured Sparse Gradients

In GRASS, we consider the projection matrix $P_t \in \mathbb{R}^{m\times r}$ defined in Algorithm 1 to be a column sparse matrix, where every column p_j has at most one non-zero entry: $\|p_j\|_0 \leq 1, \forall j \in [r]$. Since P_t can have at most r nonzero rows, the subspace optimization problem is constrained to a coordinate-aligned low rank subspace. Therefore, updates with a sparse P_t matrix can be interpreted as performing generalized coordinate descent modifying only the selected rows of the weight matrix w.

Formally, each column p_j of the sparse projection matrix P_t is constructed during the *Update Projection* step as: $p_j = e_{\sigma(j)}\rho_j$ for $j \in [r]$. Here $e_i \in \mathbb{R}^m$ denotes the one-hot vector whose i-th coordinate is 1. $\sigma\colon [r] \to [m]$ is a selection function which maps each of the r selections to the selected row index in $\{1, \ldots, m\}$, and ρ_j is a real valued constant which scales the j-th column's one-hot vector. One way to construct the σ and ρ is through random sampling. Theorem 2.1 below provides a way to construct the parameterized sampling distribution to minimize variance:

Theorem 2.1 (Optimal Unbiased Projection Sampling Matrix from Multinomial Distribution)**.** *Given a general gradient matrix $G \in \mathbb{R}^{m\times n}$, let each selection index $\sigma(j)$ ($\forall j \in [r]$) be sampled with replacement (i.i.d.) from the multinomial (categorical) distribution with the probability of sampling index $i \in [m]$ given by $p_i \in [0,1]$. If we correspondingly let $\rho_j := \frac{1}{\sqrt{r p_{\sigma(j)}}}$, then the induced random projection matrix P whose column is $p_j = e_{\sigma(j)}\rho_j$ gives an unbiased gradient estimate $PP^\top G$ with $\mathbb{E}[PP^\top G] = G$. In addition, among all the parameterized multinomial distributions of p, the one that is proportional to the row norm of G with $p_i = \frac{\|G_i\|_2}{\sum_{k=1}^m \|G_k\|_2}$ minimizes the total variance of the gradient estimate $PP^\top G$. Proof in Appendix A.*

Method	Memory			FLOPs		Comm
	Weights	Opt	Grad	Projection, opt, w update	Update P_t	
Full	mn	$2mn$	mn	$mnb + mn + Cmn$	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	mn	$2r + 2nr$	nr	$rbn + 3rn + Crn$	$mn + m + r$	nr

Table 1: Summary of Memory Requirements, Gradient Communication Volume, and FLOPs Analysis for Various Methods. We report cost to construct gradients, update optimizer states and update weight $w \in \mathbb{R}^{m\times n}$. b is token batch size, C cost of optimizer operations per parameter, $G \in \mathbb{R}^{m\times n}$, $P_t \in \mathbb{R}^{m\times r}$. Detailed breakdown in Appendix C.

Theorem 2.1 suggests that the distribution $\mathcal{D}$ should be derived from the row norms of G. As we freeze the projection matrix for T iterations in GRASS, we are interested in accurately capturing the low-rank subspace of G. In particular, we use the squared row norms of G since their interpretation as approximate leverage scores ensures that the low-rank subspace of G is preserved upon projection, with an additive error, with high probability (See Appendix B for a proof). Based on this additional insight, we investigate the following distributions in this work:

- **Multinomial Distribution:** Indices are sampled with or without replacement based on $p_k \propto \|G_k\|^2$ where G_k is the k-th row of G. When sampling without replacement $\rho_i = \frac{1}{\sqrt{r p_{\sigma(i)}}}$, while $\rho_i = 1$ otherwise.
- **Top-k Distribution:** Top-k indices corresponding to the rows of G with largest row-norms are chosen and $\rho_i = 1$.
- **Uniform Distribution:** Each index has an equal probability of being chosen, and explored with and without replacement. When sampling with replacement, the normalization constant is set to $\rho_i = \frac{1}{\sqrt{r}}$, while $\rho_i = 1$ when sampling without replacement.

The unbiased methods (sampling with replacement) are scaled so that $\mathbb{E}[PP^T] = I$, while the biased methods (Top-k, sampling without replacement) are unscaled. We find that the biased sampling methods perform better than the unbiased methods. P_t is applied to the project the smaller dimension of G to achieve the best memory-performance tradeoff (Zhao et al., 2024).

2.2. Benefits of GRASS

In Table 1, we see that GRASS consumes less memory, FLOPs and communication than other methods. Additionally, GRASS preserves the sparse structure of the gradients and can be even more beneficial in such applications.

FLOPs In GRASS, the projection operator $P_t^\top \in \mathbb{R}^{r\times m}$ matrix can be composed into the product of a diagonal scaling matrix $\rho \in \mathbb{R}^{r\times r}$ and a slicing operation $S \in \mathbb{R}^{r\times m}$, which can be efficiently applied to G. The slicing matrix $S \in \{0, 1\}^{r\times m}$ selects r specific rows from G with $S_{ij} = 1$ to select row j of G as the i-th row of SG. Unlike existing works (Zhao et al., 2024; Hao et al., 2024), we integrate P_t during the backward pass instead of within the optimizer, thereby eliminating the need to construct or compute the full gradient $G \in \mathbb{R}^{m\times n}$ for every weight. Here, G can be represented as $G = AB$, where $A \in \mathbb{R}^{m\times b}$ is the gradient of the layer output, $B \in \mathbb{R}^{b\times n}$ is the input activations, and b is the token batch size. The application of the projection matrix as $\rho((SA)B)$ results in considerably fewer FLOPs compared to dense matrix multiplication. Additionally, since PyTorch tensors are stored in row-major order, slicing matrix rows offers faster memory access. The cost of weight update is also cheaper than a dense P_t, since only the rows corresponding to the nonzero rows of P_t of the parameter matrix w needs to be updated. Unlike GALORE, our method does not require to compute SVD periodically when updating P_t.

Memory savings GRASS achieves significant memory savings over Full training from the reduced dimensionality of optimizer states. The savings over GALORE and FLORA are from the gradient and projection matrices. GRASS does not require saving or computing the full gradient G and can operate with the reduced gradient $G_c \in \mathbb{R}^{r\times n}$ unlike existing methods which require updating the entire dense w. The projection matrix P per weight matrix also only requires saving linear indices and scaling factors which is $2r$ compared to mr in GALORE and FLORA. GRASS also eliminates the need for constructing full gradients like in LoRA during the non-update steps (Dettmers et al., 2023) and avoids allocating additional memory for intermediate matrix products during projection and weight updates. During projection update, no additional memory is consumed for SVD as in GALORE. The memory savings in GRASS are orthogonal to the layerwise trick (Lv et al., 2023; Zhao et al., 2024) which is prohibitively expensive on multi-GPU, and can slow throughput down by 9% on 1 GPU (Zhao et al., 2024).

Communication As our GRASS implementation does not require constructing the full gradient, communication volume is also significantly lower than GALORE and FLORA.

2.3. Specific 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 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 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 D for details).

3. Experiments And Results

3.1. Pretraining Performance

Model size	60M	350M	1B
Full-Rank	36.97	18.71	18.12
GALORE	37.09	19.38	19.23
GRASS	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 is competitive with GALORE, but with lower memory footprint and higher training throughput.

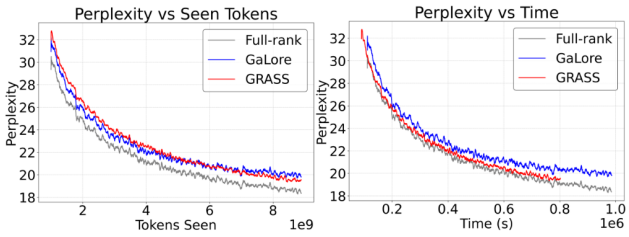

Figure 1: Pretraining 1B LLaMA on 8.8B tokens of C4 with GRASS, Full-rank and GALORE. (Left) Train perplexity vs seen tokens. (Right) Train perplexity vs wall-clock time. GRASS outperforms GALORE and shows < 0.01 perplexity gap with Full-rank loss curve in wall-clock time.

We compare GRASS against full-rank training and GALORE by pretraining LLaMA-based large language models on the cleaned C4 subset of Dolma (Soldaini et al., 2024), without data repetition over a sufficiently large amount of data, across a diverse range of model sizes (60M, 350M, 1B). Like GALORE, we adopt a LLaMA-based architecture with RMSNorm and SwiGLU activations (Touvron et al., 2023; Shazeer, 2020; Zhang & Sennrich, 2019). For both GALORE and GRASS, we fix the frequency T at 200 and the scale factor α at 0.25, maintain a consistent rank r, and project attention and feed-forward layers. All experiments are conducted in BF16 with identical batch sizes, and the learning rate is independently tuned for each method (see Appendix E). As shown in Table 2, GRASS matches GALORE in iteration complexity and approaches the full model's performance within a perplexity gap of less than one 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) on a single A100 GPU, where due to increased pretraining throughput GRASS closely follows the Full loss curve with < 0.1 perplexity gap.

3.2. Finetuning Performance

We apply GRASS and other efficient methods to finetune the pre-trained RoBERTA-Base model (Liu et al., 2019) on the GLUE NLU benchmark (Wang et al., 2018). Each method is implemented on the same transformer layers with a rank of $r = 8$ and trained over 3 epochs with a sequence length of 128. We tuned the learning rate and scale factor across all methods, detailed in Appendix E. We see in Table 3, that GRASS performs competitively with GALORE and LoRA, even though GRASS exhibits a reduced memory footprint and improved training throughput compared to these methods as we show in the next section. Among the GRASS sampling strategies, TopK and Multinomial with no replacement (Multi-NR) perform comparably to LoRA, whereas Uniform sampling falls short.

3.3. Efficiency analysis

Throughput In Figure 3, we compare the pretraining throughput (tokens/s) of GRASS and GALORE relative to the Full model, across a range of model sizes, focusing on both regular and projection update steps. We consistently use a rank of $r = 64$ for both the attention and feedforward layers, with a uniform local batch size across all methods, sequence length 256, and a total batch size of 1024. Benchmarking was conducted using a single 80GB A100 GPU and an AMD EPYC 7763 64-Core Processor. Detailed hyperparameter settings are provided in Appendix E. We did not employ activation checkpointing, memory offloading, or optimizer state partitioning in our experiments.

Our analysis reveals that at a model size of 60M, the throughput of GRASS and GALORE is lower than that of the Full model, likely due to the overhead from the custom backward pass implementation. However, at larger model sizes (1B and 7B), throughput significantly surpasses the Full model, with improvements of 26% and 33.8% respectively, and comparative gains over GALORE of 27%

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 (Topk)	59.16	86.92	89.60	92.42	88.65	76.37	94.15	90.13	56.34	81.53
GRASS (Multi-NR)	58.87	86.08	89.94	91.69	83.36	76.17	94.73	90.00	56.34	81.35
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 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.

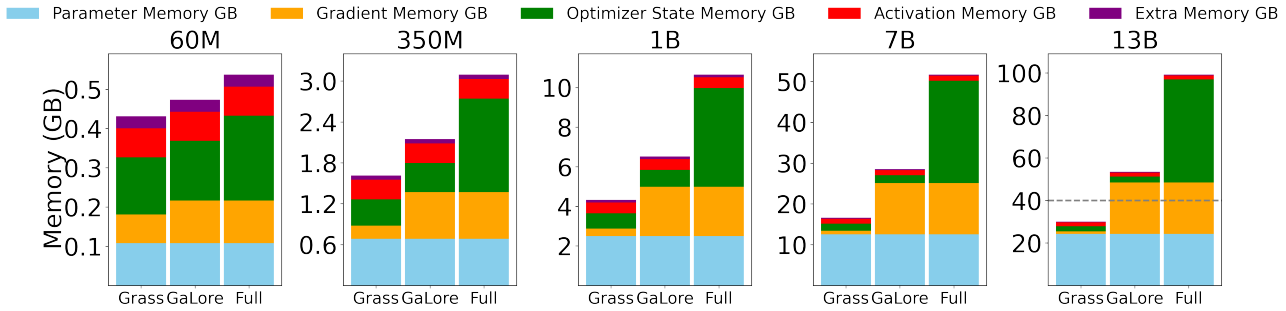

Figure 2: Pretraining memory footprint for GRASS, GALORE, and Full across model sizes for a regular (non projection update step) and $r = 128$. GRASS has a lower memory footprint across all model sizes and the reduction is greater at larger model sizes.

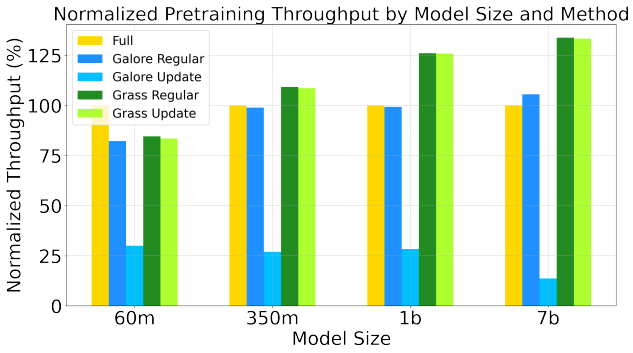

Figure 3: Normalized pretraining throughput at $r = 64$ for GRASS, Full-rank, and GALORE relative to Full-rank. GRASS throughput exceeds Full and GALORE throughput by $> 25\%$.

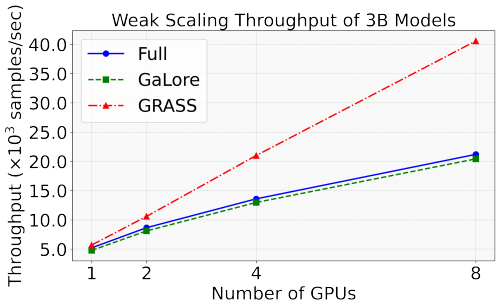

Figure 4: Communication Efficiency: Weak Scaling Throughput Comparison for 3B LLaMA pretraining using GRASS, Full-rank, and GALORE. GRASS shows $2\times$ higher throughput over Full and GALORE at 8 GPUs.

and 26.7%. Importantly, the overhead from the projection update step is minimal for GRASS, in contrast to GALORE, which experiences considerable slowdowns due to SVD costs. This throughput advantage for GALORE should only increase if we increase the batch size over other methods, since it has a lower memory footprint.

Further throughput comparisons by rank are detailed in Appendix Figure 9, showing that GRASS achieves maximum relative throughput gains at the lower rank of $r = 64$, with diminishing returns both as rank increases or model size decreases.

Memory In Figure 2, we benchmark the memory footprint of GRASS against Full and GALORE using a token batch size of 256 and rank 128 across model sizes for a regular (non projection update) step. We see that GRASS is more memory efficient than both the Full model and GALORE for pretraining due to the reduced cost of gradient and optimizer (projection matrix) memory and that this improvement increases with model size. At 13B parameters, GRASS uses 70% less memory than Full-rank and 45% less than GALORE. Importantly, we find that the update step in GRASS is also much more memory efficient than GALORE which needs to convert the full gradient into float32 to perform SVD, as a result, GALORE cannot train a 13B model on an 80GB A100 like GRASS. Appendix Figure 7 shows a similar advantage for finetuning over LoRA at rank 64.

Communication In Figure 4, we benchmark the throughput (tokens/sec) for weak scaling of 3B LlaMA in a multi-GPU L40 node with peak all-reduce bandwidth 8.64 GB/s. We use a token batch size of 4096 per worker (local batch

size 16, sequence length 256) and find that communicating the projected gradients results in significantly higher throughput in a multi-GPU setting over Full and GALORE.

3.4. Ablations

Sampling methods In Table 4, we compare various sampling strategies for indices during the pretraining of a 60M LlaMA model on 500M tokens from the RealNews subset of C4 (Raffel et al., 2020). The Frozen Multinomial method samples indices at iteration 0 and does not resample in subsequent iterations. Sampling with and without replacement are denoted as R and NR, respectively. Our findings reveal that the NR versions of these strategies outperform the R versions, and biased sampling strategies prove to be more effective. Both Multinomial-NR and Top-k strategies are competitive with the GALORE method. Notably, the Uniform-NR strategy performs significantly better in pretraining than finetuning, likely because the norm distribution is closer to uniform at the start of pretraining.

Sampling Method	Eval perp
Frozen Top-k	34.78
Uniform	32.46
Uniform-NR	31.06
Multinomial-R	31.85
Multinomial-NR	30.91
Top-k	**30.88**
GALORE	30.67
Full-rank	30.27

Table 4: Comparison of GRASS sampling methods on eval perplexity for 60M LlaMA on the RealNews subset of C4. Best sampling strategy bolded.

Coverage of indices In Figure 5, we plot the coverage defined as the union of indices sampled over k iterations divided by the total indices per layer. We plot the coverage for the 60M LlaMA model pretrained on the RealNews subset of C4, at $k = 15$ iterations. Here the rank $r = 128$ and matrix dimension is 512 indicating that 97.66% is the theoretical coverage for uniform sampling with replacement (see Appendix J). We find that all sampling methods exhibit good coverage with the Multinomial-NR being close to uniform. Top-K and Multinomial oversample indices in certain layers, suggesting potential areas for further investigation into their utility in pruning strategies.

4. Conclusion And Future Work

In this work, we introduce GRASS, a novel optimization technique to reduce the memory bottleneck associated with training LLMs while providing additional computation efficiency gains. Specifically, GRASS utilizes sparse matrices to project gradients into a low-dimensional subspace to reduce the size of the optimizer states. As part of the projection step, GRASS also avoids computing the full parameter's gradient matrix, which as a result improves over existing methods both in terms of computational efficiency and gradient memory. Experimentally, these design choices have directly translated to a significantly smaller memory footprint, decreased communication volume, and enhanced throughput during LLM training.

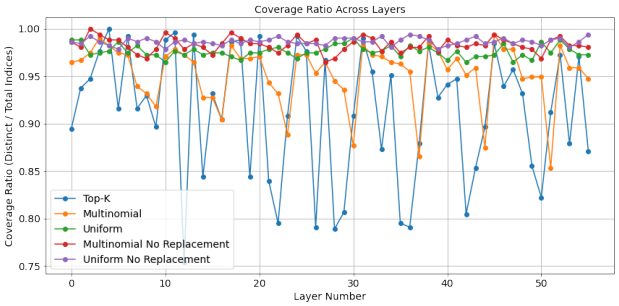

Figure 5: Per layer indices coverage (Distinct/Total) for the sampling strategies across 100 pretraining iterations.

GRASS opens up exciting avenues for future research. Extending GRASS to explore various structured sparsity patterns, beyond simple column sparsity, could further enhance compression and throughput. Additionally, a more detailed investigation of the relationship between GRASS's rank parameter, training time, and performance could reveal opportunities to not only match but potentially surpass the performance of full-gradient training in certain scenarios.

References

Biderman, D., Ortiz, J. G., Portes, J., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., Blakeney, C., and Cunningham, J. P. Lora learns less and forgets less. *arXiv preprint arXiv: 2405.09673*, 2024.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv: 1604.06174*, 2016.

Choquette, J., Gandhi, W., Giroux, O., Stam, N., and Krashinsky, R. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021. doi: 10.1109/MM.2021.3061394.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *NEURIPS*, 2023.

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

Lialin, V., Shivagunde, N., Muckatira, S., and Rumshisky, A. Relora: High-rank training through low-rank updates.

2023. URL https://api.semanticscholar.org/CorpusID:259836974.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv: 1907.11692*, 2019.

Lv, K., Yang, Y., Liu, T., Gao, Q., Guo, Q., and Qiu, X. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv: 2306.09782*, 2023.

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *Neural Information Processing Systems*, 2019.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024.

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

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4603–4611. PMLR, 2018. URL http://proceedings.mlr.press/v80/shazeer18a.html.

Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J. E., and Stoica, I. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv: 2311.03285*, 2023.

Soldaini, L., Kinney, R., Bhagia, A., Schwenk, D., Atkinson, D., Authur, R., Bogin, B., Chandu, K., Dumas, J., Elazar, Y., Hofmann, V., Jha, A. H., Kumar, S., Lucy, L., Lyu, X., Lambert, N., Magnusson, I., Morrison, J., Muennighoff, N., Naik, A., Nam, C., Peters, M. E., Ravichander, A., Richardson, K., Shen, Z., Strubell, E., Subramani, N., Tafjord, O., Walsh, P., Zettlemoyer, L., Smith, N. A., Hajishirzi, H., Beltagy, I., Groeneveld, D., Dodge, J., and Lo, K. Dolma: an open corpus of three trillion tokens for language model pretraining research. *arXiv preprint arXiv: 2402.00159*, 2024.

Spring, R., Kyrillidis, A., Mohan, V., and Shrivastava, A. Compressing gradient optimizers via count-sketches. *International Conference on Machine Learning*, 2019.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv: 2307.09288*, 2023.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *BLACKBOXNLP@EMNLP*, 2018. doi: 10.18653/v1/W18-5446.

Woodruff, D. P. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 2014. doi: 10.1561/0400000060.

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

Zhang, B. and Sennrich, R. Root mean square layer normalization. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/1e8a19426224ca89e83cef47f1e7f53b-Paper.pdf.

Zhang, L., Zhang, L., Shi, S., Chu, X., and Li, B. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv: 2308.03303*, 2023.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

A. Proof of Theorem 2.1

We introduce a gradient approximation method that utilizes a multinomial sampling strategy to construct an unbiased gradient estimator. A general gradient G can be expressed through an atomic decomposition:

$$G = \sum_{i=1}^{n} \lambda_i a_i,$$

where each a_i is a unit-norm atom, and λ_i are the corresponding coefficients. This decomposition can take forms such as element-wise decomposition and singular value decomposition (SVD) for matrices.

Let P be a sampling matrix for the rows, where each row P_i has a single non-zero entry. The matrix P is formed such that the sampling index for each row P_i is chosen based on multinomial sampling using the probability vector p. Thus, PP^T is a diagonal matrix.

To approximate G under strict memory constraints, we use multinomial sampling to select exactly k rows. The approximate gradient G_r is then defined as:

$$G_r = PP^T G = \sum_{i=1}^{k} P_i P_i^T G = \sum_{i=1}^{n} \frac{\lambda_i t_i}{\alpha_i} a_i,$$

where a_i is an atom with only a single non-zero row of G and normalized to have $\|a_i\| = 1$ with a suitable λ_i, t_i is drawn from the multinomial distribution Multinomial(p, k), where p is a probability vector of length n and k is the number of draws. The normalization factor $\alpha_i = kp_i$ ensures that $\mathbb{E}[G_r] = G$. We can show that the expected value of G_r equals the original gradient G. Since $\mathbb{E}[t_i] = kp_i$, the expectation of G_r becomes:

$$\mathbb{E}[G_r] = \mathbb{E}\left[\sum_{i=1}^{n} \frac{\lambda_i t_i}{\alpha_i} a_i\right] = \sum_{i=1}^{n} \lambda_i a_i = G.$$

Similarly

$$\mathbb{E}[\|G_r\|^2] = \sum_{i=1}^{n} \lambda_i^2 \left(\frac{1-p_i}{kp_i} + 1\right) - \frac{1}{k} \sum_{i=1}^{n} \sum_{j \neq i}^{n} \lambda_i \lambda_j a_i^T a_j.$$

Optimization Problem: The objective is to minimize the variance of G_r while ensuring that exactly k components are selected. This leads to the following optimization problem:

$$\min_{p} \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} \quad \text{subject to} \quad \sum_{i=1}^{n} p_i = 1, \quad 0 < p_i \leq 1 \text{ for all } i.$$

The Lagrangian L for this constrained optimization is:

$$L(p, \mu, \gamma) = \sum_{i=1}^{n} \frac{\lambda_i^2}{p_i} + \mu \left(\sum_{i=1}^{n} p_i - 1\right) - \sum_{i=1}^{n} \gamma_i p_i,$$

where μ is the Lagrange multiplier for the equality constraint, and γ_i are the multipliers for the inequality constraints ensuring $p_i \geq 0$.

The Karush-Kuhn-Tucker (KKT) conditions for this problem are:

1. **Stationarity**: $\frac{\partial L}{\partial p_i} = -\frac{\lambda_i^2}{p_i^2} + \mu - \gamma_i = 0$
2. **Primal Feasibility**: $\sum_{i=1}^{n} p_i = 1, \quad 0 < p_i \leq 1$
3. **Dual Feasibility**: $\gamma_i \geq 0$
4. **Complementary Slackness**: $\gamma_i p_i = 0$

Assuming $p_i > 0$ and $\gamma_i = 0$ due to complementary slackness, the stationarity condition simplifies to $\mu = \frac{\lambda_i^2}{p_i^2}$. Therefore, $p_i = \sqrt{\frac{\lambda_i^2}{\mu}}$.

Applying the primal feasibility condition:

$$\sum_{i=1}^{n} \sqrt{\frac{\lambda_i^2}{\mu}} = 1 \quad \Rightarrow \quad \mu = \left(\sum_{i=1}^{n} |\lambda_i| \right)^2$$

Thus, the optimal probabilities p_i are:

$$p_i = \frac{|\lambda_i|}{\sum_{j=1}^{n} |\lambda_j|}$$

Thus p_i is proportional to the magnitude of λ_i, normalized by the sum of the magnitudes of all λ values, which satisfies $\sum_{i=1}^{n} p_i = 1$ and minimizes the objective function.

B. Row Norms and Subspace Embedding Property

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

Theorem B.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 B.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 B.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:

$$\|\mathbf{S}_1 \mathbf{W}^\top \mathbf{V} \mathbf{S}_2 - \mathbf{S_1} \mathbf{W}^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{V} \mathbf{S}_2\| \leq \epsilon \|\mathbf{S}_1\| \|\mathbf{S}_2\|.$$

C. Memory, FLOPs and Communication Volume

In this section we report the Memory, FLOPs and Communication Volume for the various methods corresponding to a single $m \times n$ weight w and its gradient G.

Notes:

- Let $G = AB^T$, where A is an $m \times b$ matrix, B is an $n \times b$ matrix, where $m \leq n$ and b is the token batch size usually much larger than m, n.
- Let P be an $m \times r$ projection matrix.
- Here we assume A and B are constructed ahead of time and we are interested in the memory, floating-point operations, and communication volume to construct the gradients G, update the optimizer state, and updating weights $w = PP^TG$.
- C is the number of optimizer operations per gradient element.

- All numbers are computed based on the original papers.
- For GRASS, $P^T = RS$ where R is a $r \times r$ diagonal scaling matrix, S is a sparse $r \times m$ row selection matrix. Both R, S can be applied efficiently.

We compare various optimization strategies: **Full**, **GALORE**, **LoRA**, **ReLoRA**, **FLORA**, and **GRASS** (our approach). These strategies are analyzed based on memory requirements, communication volume, and floating-point operations (FLOPs).

FLOPs per Worker

Method	Regular Step Cost	Projection Update Cost
Full	Compute AB (mnb), optimizer state update (Cmn), reprojection update (mn).	0
GALORE	Compute AB (mbn), compute P^TAB (rmn), optimizer state update ($C \cdot rn$), reprojection update (rmn), parameter update (mn).	SVD cost ($mn \min(n, m)$)
LoRA	Compute AB (mbn), compute gradient for LoRA weights ($2rmn$), optimizer update ($C(rm + rn)$), weight update ($rn + rm$).	0
ReLoRA	Compute AB (mbn), compute gradient for LoRA weights ($2rmn$), optimizer update ($C(rm + rn)$), weight update ($rn + rm$).	Merging weights ($mnr + mn$)
FLORA	Compute AB (mbn), compute PAB (rmn), optimizer state update ($C \cdot rn$), reprojection update (rmn), parameter update (mn).	Sampling Gaussians (mr)
GRASS (Ours)	Compute $(P^TA)B$ ($rbn + rn$), optimizer state update ($C \cdot rn$), reprojection and weight update ($2rn$).	Computing row norms and sampling matrix* ($mn + m + r$)

Table 5: Detailed FLOPs Analysis for Various Methods. *This is the complexity of Alias Method for multinomial sampling. Top-k complexity would be $m \log r$ using a heap.

Memory Requirements

Method	Weights	Optimizer State	Gradient Memory
Full	mn	$2mn$	mn
GALORE	mn	$mr + 2nr$	mn
LoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$
ReLoRA	$mn + mr + nr$	$2mr + 2nr$	$mr + nr$
FLORA	mn	$mr + 2nr$	mn
GRASS	mn	$2r + 2nr$	nr

Table 6: Memory Requirements for Various Methods. Note that memory cost for the update step is intermittent.

Communication Volume

Method	Comm Volume
Full	mn
GALORE	mn^*
LoRA	$mr + nr$
ReLoRA	$mr + nr$
FLORA	mn^*
GRASS	nr

Table 7: Gradient Communication Volume for Various Optimizers. * Note that GALORE and FLORA communication volume can be reduced to nr using a communication hook.

D. Distributed Data Parallel Implementation

Algorithm 2 Distributed GRASS Training with PyTorch DDP

Input: Initial weights $W_0 \in \mathbb{R}^{m \times n}$, total iterations T, subspace rank r, world size p, learning rate scale α, update frequency K
Output: Optimized weights $W^{(T)}$

Initialize distributed environment (e.g., NCCL)
$W \leftarrow W_0$ ▷ *Set weights as non-trainable*
Introduce virtual trainable parameter `vparams` $\in \mathbb{R}^{1\times 1}$, linked to each weight matrix
`vparams.wgrad` $\leftarrow \emptyset$ ▷ *Initialize storage for compressed gradients*
Initialize a DDP model with custom gradient hooks
for $t = 0$ to $T - 1$ **do**
  Compute local loss L for the current mini-batch
  $output \leftarrow$ Forward pass using W
  if $t \equiv 0 \pmod K$ **then**
    Compute backward pass to obtain full gradient G
    // Sketch gradient using column norms and select top-r
    $G_{sketch} \leftarrow$ `ToprColumns`(G, r) ▷ *Using Algorithm 3*
    // All-reduce and update the sketched matrix
    $G_{sketch} \leftarrow$ `AllReduceMean`(G_{sketch})
    Update projection matrix P using G_{sketch}, compute and store compressed gradient G_C in `vparams.grad`
  else
    Compute backward pass, capturing compressed gradients G_C in `vparams.grad`
    Perform all-reduce on `vparams.grad` across all workers
  end if
  Update W using `vparams.grad`
end for
return W

Algorithm 3 ToprColumns Function

Input: Gradient matrix $grad$, subspace rank r
Output: Sketched gradient matrix with top-r column norms
$indices \leftarrow$ `argsort`$(|$`colnorms`$(grad)|)[-r:]$ ▷ *Identify indices of top-r column norms*
return $grad[:, indices]$

To optimize memory usage in PyTorch's Distributed Data Parallel (DDP) framework (Paszke et al., 2019), we implement strategic modifications to our model architecture ti improve distributed training efficiency (see Algorithm 2). 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 uses DDP's asynchronous all-reduce capabilities while preventing unnecessary memory allocation.

E. Experiments: Hyperparameters

Pretraining We introduce details of the LLaMA architecture and hyperparameters used for pre-training. Table 8 shows the dimensions of LLaMA models across model sizes.

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

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 when resetting optimizer states for the 60M and 350M training jobs, while the 1B job did not require resetting optimizer states. Both 60M and 350M GRASS pretraining jobs uses Top-K sampling while 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 models used a scale factor $\alpha = 0.25$. We found that GALORE was sensitive to hyperparameters and exhibited loss spikes and divergence at the prescribed learning rates in the paper (0.01) particularly at the 1B scale, and as a result we had to train using reduced learning rates where we did not observe such spikes. The learning rates of GRASS and GaloRE were higher than the full model which showed instability at values greater than 0.001. Unless otherwise specified we average losses using a window of 15 steps.

Finetuning We finetune the pre-trained RoBERTa-Base model on the GLUE benchmark using the pretrained model on Hugging Face. 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 used a projection update frequency of 100 for all methods. We tuned the learning rate and scale factor 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 $r = 8$.

Table 9 shows the hyperparameters used for finetuning RoBERTa-Base for GRASS.

	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 9: Hyperparameters of finetuning RoBERTa base for GRASS.

Throughput benchmarking We benchmark throughput on a single 80GB A100 GPU using a total batch size of 1024 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). GRASS can train a 13B model on the 80GB GPU unlike GALORE and Full.

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.

F. 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)

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

Figure 6: Activation memory estimation for the different baselines.

F.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 6 considers the BF16 precision used for storing the activations.

F.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.

F.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.

F.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

G. Experiment: Finetuning Memory

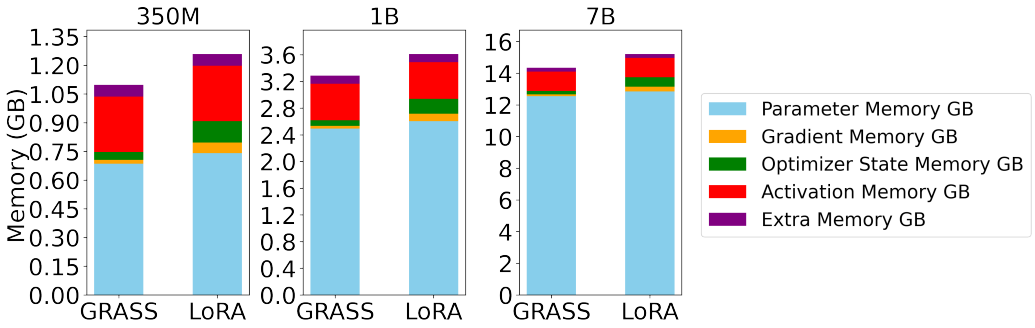

Figure 7: LLaMA finetuning memory footprint of GRASS and LoRA for rank $r = 64$, sequence length 256, batch size 1.

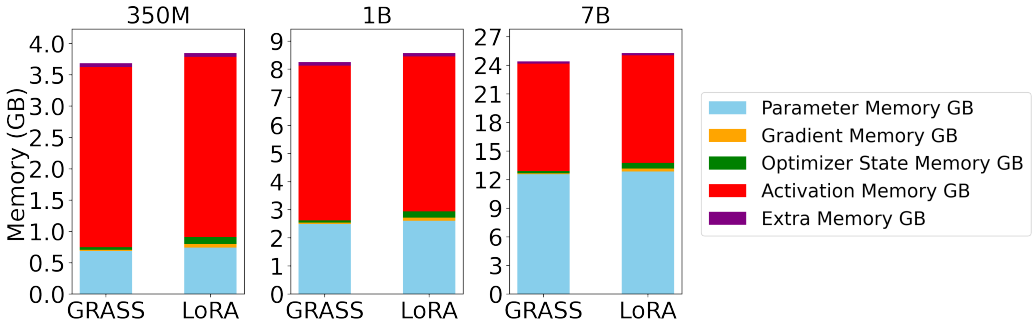

Figure 8: LLaMA finetuning memory footprint of GRASS and LoRA for rank $r = 64$, sequence length 512, batch size 4.

In Figure 7 and Figure 8, 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 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 F 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.

H. Experiments: Throughput

Figure 9 compares the normalized 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 $> 25\%$ at for the 1B model at rank 64. The throughput approaches that of the full model, as model size decreases or projection rank increases.

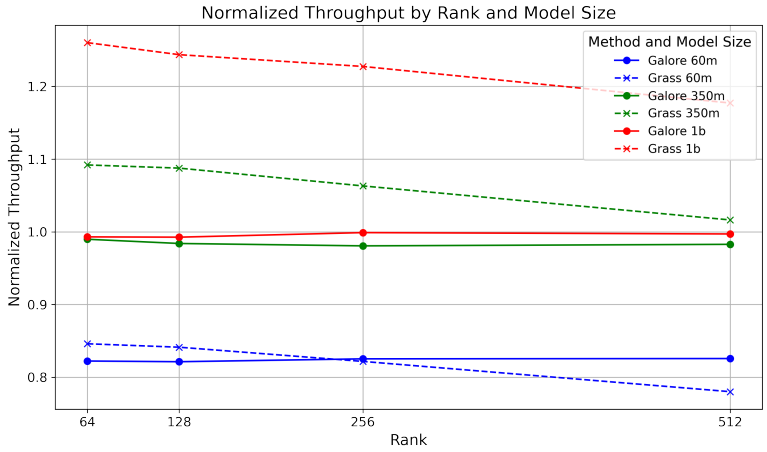

Figure 9: Rank vs Normalized Throughput for GRASS and GALORE across 60M, 350M, and 1B model sizes

I. Experiments: Ablations

Rank sweep In Figure 10, we perform rank ablations on GRASS for pretraining a 350M LlaMA model on the C4 subset of Dolma. We find that between 256 and rank 512 increasing the rank only slightly affects the rate of convergence. Additionally as GRASS allows full-parameter training, we find that training a model at rank 128 for 80k steps is much more effective than training a model at rank 512 for 40k steps. GRASS like GALORE (Zhao et al., 2024) can therefore be used to trade-off memory and computational cost where in a memory-constrained setting one could pick a lower rank and train longer.

Comparison with other baselines In Table 10, 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 4 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^T is a CountSketch matrix with r rows with one nonzero entry from $\{\pm 1\}$ per column. The CountSketch projection has been 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.

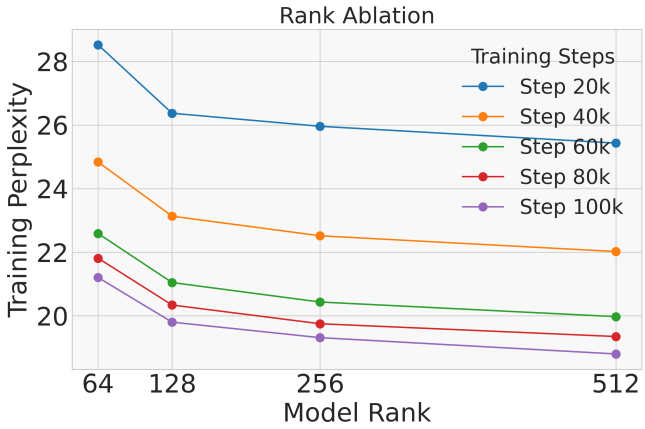

Figure 10: GRASS rank ablations for 350M LLaMA training. We report perplexity on Dolma C4 across various ranks and training steps. Loss is averaged over a window of 50 steps.

	Train Perp	Eval Perp
Full-Rank	33.48	31.41
GRASS	**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 10: 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.

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 leverage scores or row importance norms can better approximate the gradient low rank subspace than a data agnostic sketch like FLORA or CountSketch (Woodruff, 2014).

Coverage In Figure 11 and Figure 12 we plot the aggregated sampled indices over 15 iterations of 60M LlaMA pretraining on the RealNews subset of C4. We see that while Multinomial with no replacement and Top-k attain similar performance in terms of perplexity the sampled indices can be quire different, with top-k tending to oversample indices in particular layers.

J. Analyzing Coverage

We analyze the coverage of indices for a uniform sampling process with replacement. Here 128 indices (rank r) are randomly chosen from a total of 512 possible indices (model dimension d), with this process being repeated across 15 iterations (number of iterations k).

The probability $P(i)$ that a specific index i is not chosen in one individual selection from 512 indices is $P(i) = 1 - \frac{1}{512}$ This reflects the independent probability for each draw within an iteration. Given that each iteration comprises 128 selections, the probability $P_{128}(i)$ that index i is not picked during one full iteration is: $P_{128}(i) = \left(1 - \frac{1}{512}\right)^{128}$ Extending this to 15 iterations, the probability $P_{15\times128}(i)$ that index i is never selected during the entire sampling process is: $P_{15\times128}(i) = \left(\left(1 - \frac{1}{512}\right)^{128}\right)^{15}$ Thus, the probability that an index is selected at least once throughout the 15 iterations is given by: $P_{\text{selected}}(i) = 1 - P_{15\times128}(i)$ Thus 97.66% of the indices are expected to be sampled at least once over the course of 15 iterations.

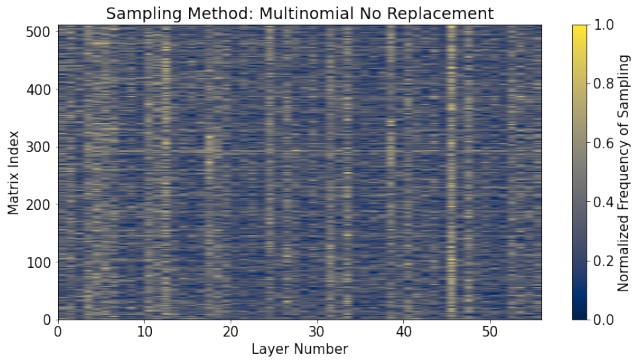

Figure 11: Multinomial Sampling without Replacement: Heatmap of indices sampled for the different layers across 15 iterations of LlaMA 60M C4 pretraining.

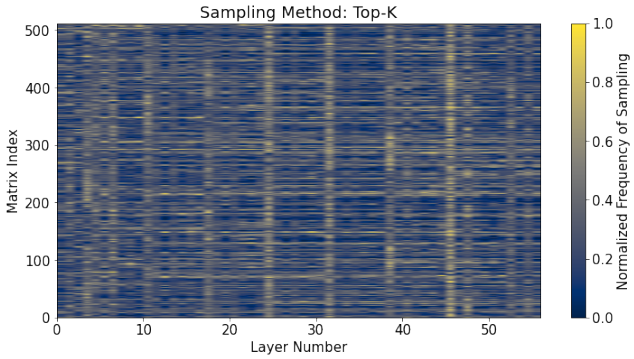

Figure 12: Top-K Sampling: Heatmap of indices sampled for the different layers across 15 iterations of LlaMA 60M C4 pretraining.