2025 IEEE International Symposium on Circuits and Systems (ISCAS)

GASA: Rank-Sliced GAther-Scatter Activations and Application to Sparsity-Preserving Parameter-Efficient Fine-Tuning

H. T. Kung
kung@harvard.edu
Harvard University

Andrew Sabot
asabot@g.harvard.edu
Harvard University

***Abstract*—We present a novel *rank-sliced* GAther-Scatter Activation (GASA) algorithm to minimize I/O costs in computing neural network layer activations (XW) between a data matrix X, and a singular value decomposition (SVD) of a weight matrix $W = U\Sigma V^T$. We maintain high accuracy with ResNet-18 [1] on the CIFAR-10 [2] dataset and Deberta-V3-base [3] on the IMDB dataset [4] at high sparsities (i.e., up to 85% sparsity for U and V) by using rank pruning for W and UV pruning for each rank slice. Furthermore, with rank-sliced computation, we can perform parameter-efficient fine-tuning on the resulting sparse networks while *preserving sparsity* to retain the sparsity-induced computational efficiency for inference. That is, our rank-sliced weight update preserves the original sparsity structure of each W. Our sparsity-preserving fine-tuning maintains model accuracy under adapter ranks as low as 8, compared to the rank of 150 of the pre-trained pruned model.**

1. Introduction

In this paper, we describe GAther-Scatter Activation (GASA) for computing activations in a neural network layer. For a given data matrix X and a weight matrix W in the singular value decomposition (SVD) representation, $W = U\Sigma V^T$, GASA is a novel algorithm that computes XW in a *rank-sliced* manner, where each rank slice (or simply slice) S of W is the rank-1 product of a column vector u and a row vector v in U and V^T, respectively.

In GASA, we compute XS for each rank slice S using a *gather-scatter* method that gathers columns of X into a gathering vector x_g according to u and then scatters the gathering vector to columns of XS according to v. We then compute XW by computing XS for all rank slices. We sparsify W by performing *rank prunning* of W and *UV pruning* of U and V matrices.

Rank pruning followed by rank-sliced pruning can prune a high percentage of model parameters, e.g., using 50% rank-sliced pruning after 70% rank pruning (Table 1). If we perform $a\%$ pruning on Σ, followed by $b\%$ rank-sliced UV pruning on U and V, the compounded pruning rate is: $1-(1-a)(1-b) = a+b-ab$. For $a = 70\%$ and $b = 50\%$, the compounded pruning rate is $0.85 = 0.70+0.50-0.70\cdot 0.50$.

The GASA rank-sliced computation lends itself to *sparsity-preserving* parameter-efficient fine-tuning, which preserves the sparsity of W created with rank pruning and rank-sliced UV pruning. The popular low-rank fine-tuning method LoRA [5] does not preserve sparsity since the product of its low-rank update matrices A and B is generally not sparse and is not guaranteed to match the sparsity pattern of the original weight matrix W. In contrast, GASA preserves sparsity patterns, improving inference efficiency. Figure 3(c) illustrates a rank-2 fine-tuning involving two rank-1 adapter slices S_f and S_g, which have the same sparsity structures as existing slices S_1 and S_2, respectively. After fine-tuning, slices S_f and S_g can be added to slices S_1 and S_2, respectively, keeping the same sparsity structure. Thus, GASA's rank-sliced fine-tuning does not introduce inference latency because fine-tuning updates are merged with the original weights. Also, the rank-sliced activation computation based on Figures 1 and 2 can take advantage of the same sparsity for the fine-tuned slices.

Additionally, GASA is I/O efficient. Suppose that matrices X and W are $n \times n$, and the local memory can hold $3n$ vectors (gathering, u and v vectors) plus some additional vectors to support data streaming. Then GASA uses the minimum possible IO in the sense that it reads and writes each column vector of X and XW from and to the external memory at most once, respectively (Section 2).

In summary, the contributions of this paper are:

- A rank-sliced gather-scatter method GASA for computing activations (Section 2 and Figure 1).
- Analysis of how GASA achieves the minimum possible I/O (Section 2.2).
- A method for sparsifying the weight matrix with rank pruning of W and rank-sliced UV pruning, along with empirical performance results for the resulting sparse weight matrix (Section 4, Table 1).
- Rank-sliced sparsity-preserving parameter-efficient fine-tuning, a different take on the LoRA low-rank approach, on the rank pruned and UV-pruned model Section 5), along with empirical performance results on Deberta V3 base (Figure 5) where we use adapters with ranks as low as 8.

2. Computing Activations With Gather-Scatter

Given an $n \times n$ data matrix X and an SVD of an $n \times n$ weight matrix $W = U\Sigma V^T$, with rank-sliced computation,

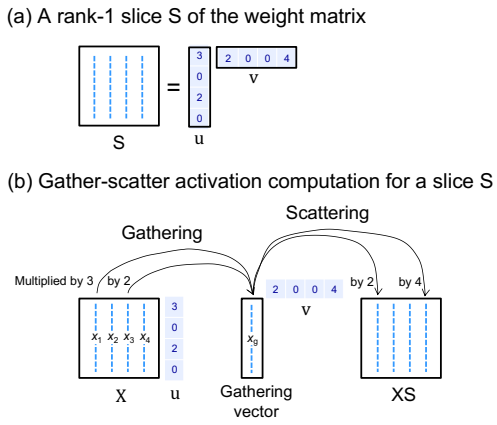

Figure 1: The **gather-scatter algorithm** of computing activations for a rank-1 slice S of a weight matrix W. (a) A rank-1 slice S of W is the product of a column vector u and a row vector v. (b) For an activation data matrix X, we compute XS for a slice S in two stages. In the first stage, we gather column vectors of X to form a gathering vector x_g, according to u. In the second scattering stage, we distribute x_g to form columns of the result matrix XS, according to v.

we can compute activations XW with efficient use of I/O resources, as illustrated in Figure 1. We assume W is rank-deficient and is *rank pruned* to rank r (where $r < n$). For illustrative simplicity in Section 2, we assume that $r = 3$. Note that:

$$W = \sigma_1 S_1 + \sigma_2 S_2 + \sigma_3 S_3$$

with each S_i being a rank-1 matrix. We call each S_i a rank slice or simply a slice of W, see Figure 1(a) for an illustration of a S slice. Note that each slice is the product uv of some column vector u and row vector v. We assume in this example that u and v are 50% sparse, i.e., sparsity $\alpha = 50\%$.

2.1. Gather-and-Scatter Activations for a Single Rank Slice

Consider gather-and-scatter computing of XS for a slice S as depicted in Figure 1(b). Suppose that $X = [x_1, x_2, x_3, x_4]$ where each x_i is a column vector and $S = uv$ where u is a column vector containing $(3, 0, 2, 0)$ and v is a row vector containing $(2, 0, 0, 4)$. We compute XS in two stages:

1) Gather columns of X according to u to form a gathering vector x_g. That is, x_g is a weighted sum of the gathered vectors. In our example, $x_g = 3x_1 + 2x_3$.
2) Scatter the gathering vector x_g to multiple columns according to v. That is, each of these columns is a scaled copy of x_g. In our example, the scattered columns form a matrix $[2x_g, 0, 0, 4x_g]$.

The gather-scatter algorithm (see Figure 1(b)) allows column-skipping when there is sparsity of any pattern in u and v. In the gathering stage, we can skip loading and multiplying the column vectors corresponding to zero entries in u. Similarly, in the scattering stage, we can skip multiplying the gathering vector and storing the resulting vectors corresponding to zero entries in v.

We now count multiplications in the gather-scatter algorithm. Let α be the sparsity of u and v for all slices of W with rank r. Suppose that X is $n \times n$. Then for each slice, the gathering and scattering stages each incur $(1 - \alpha)n^2$ multiplications. If there are r slices, then the total number of multiplications is $2(1 - \alpha)rn^2$. The factor of 2 comes from the multiplications with U and V. We can similarly count additions.

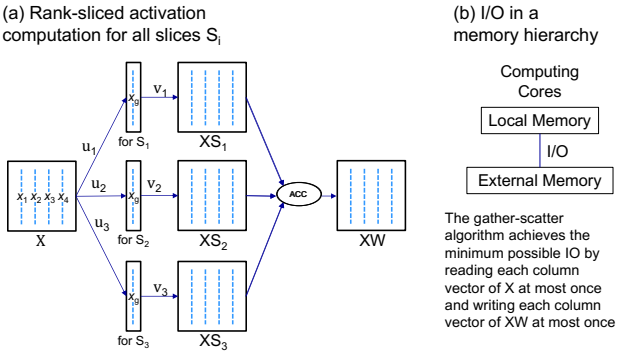

Figure 2: (a) We compute activations XW, given a data matrix X and weight matrix W, by computing XS_i for all slices S_i of W. Here, W consists of three slices: S_1, S_2 and S_3, where $S_i = u_i v_i$. We first compute XS_1, XS_2, and XS_3, and then perform weighted accumulation of these results with σ_1, σ_2 and σ_3 being the weighting coefficients, as denoted in the ACC circle in the diagram. (b) We can minimize I/O on a memory hierarchy. Suppose that X, as well as u_i and v_i, for all slices, are initially stored in the external memory and that computed results XW are to be written to the external memory. Suppose further that the local memory can hold the gathering vectors x_g for all slices plus a few intermediate vectors to support data streaming. Then by scheduling the gather-scatter algorithm achieves the minimum possible I/O.

2.2. Gather-and-Scatter Activations for all Rank Slices

We compute activations XW by computing XS_1, XS_2, XS_3 for all rank slices S_i and accumulating the result $\sigma_1 XS_1 + \sigma_2 XS_2 + \sigma_3 XS_3$. The computation uses a working set of r vectors of size n, where r is the rank of W, plus a few additional vectors for intermediate results to support data streaming. We assume that we have access to a local memory to hold the working set, as depicted in Figure 2(b).

The computation can minimize I/O as described below. When computing the 3 gathering vectors for XS_1, XS_2, and XS_3 in the gathering stage, we will use local memory

to cache the intermediate vectors. After a column vector of X is read from the external memory, scaled copies of this column vector are added to intermediate gathering vectors that correspond to the destination gathering vectors required for this column vector. This implies that we only need to read each column vector of X from the external memory at most once. Therefore, a total of at most n vector reads are required for the gathering operations for all XS_i's.

In terms of data movement, the scattering operation is the reverse of the gathering operation. Before a result vector is written to external memory, it will accumulate *all* the required scaled gathering vectors. This means that we only need to write each column of the result matrix to external memory once. Therefore, a total of at most n vector writes are required for the scattering operations for all XS_i's.

For dense S_i's with dense u and v vectors, these n vector reads and writes are theoretically the minimum required. In contrast, the inner-product-based matrix multiplication incurs $O(n^2)$ vector reads and n vector writes when performed on a local memory of the same size.

From the analysis in Section 2, we summarize the following results for the Gather-and-Scatter method.

- Suppose that we have access to a small local memory that can hold r vectors of size n, where r is the rank of W, plus a few intermediate vectors. Then we can compute activations XW with n vector reads and writes from and to the external memory, respectively.
- Suppose that W has rank r and, for each slice S_i, the associated u and v each have α percent of entries being zeros. Then we can compute XW with $2(1-\alpha)rn^2$ multiplications and additions.

3. Two Types of Pruning

Rank Pruning: (Figure 3(a)) Given a weight matrix W of a pretrained model, we decompose W with SVD: $W = U\Sigma V^T$ [6], [7]. We can prune W by dropping some number of the lowest singular values in Σ. **UV Pruning:** (Figure 3(c)) To increase sparsity, we can further prune u and v vectors of U and V, respectively —independent of Σ.

4. Sparsity-Preserving Low-Rank Fine-Tuning

We ensure the weight and low-rank update matrices have the same sparsity structures, so the sum of these matrices preserves the original weight matrix sparsity, as shown in Figure 3(c). To this end, we perform rank-1 updates on a subset of the weight matrix slices and ensure each slice update is sparsity-preserving. E.g., for a rank-8 update, we update the slices corresponding to the 8 largest singular values. This allows us to control the rank of the updates, similar to LoRA.

Suppose that S is any of these slices to be updated and $S = uv$ for some sparse column vector u and row vector v. Then we will learn a low-rank update ΔS for S by learning updates Δu and Δv for u and v, respectively. We preserve the sparsity structure of S for $S+\Delta S$ by choosing the same sparsities of u and v for Δu and Δv, respectively.

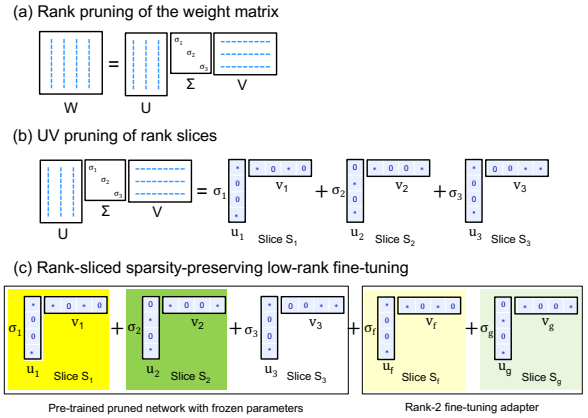

Figure 3: In GASA, the initial weight matrix W is first decomposed into the SVD representation $W = U\Sigma V^T$. The rank of W is then pruned to create a low-rank representation. Diagram (a) depicts an example of this low-rank representation with a rank equal to three and singular values being σ_1, σ_2 and σ_3. Diagram (b) shows that $U\Sigma V^T$ is a weighted sum of three rank-1 products u_1v_1, u_2v_2, and u_3v_3 for slices S_1, S_2 and S_3, respectively, where u and v vectors have sparsity $\alpha = 50\%$. Diagram (c) illustrates our proposed rank-sliced sparsity-preserving low-rank fine-tuning. The two rank-1 fine-tuning slices S_f and S_g preserve the sparsity structures of slices S_1 and S_2, respectively.

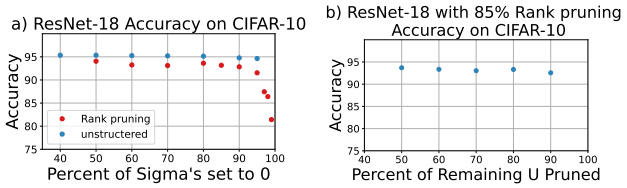

Figure 4: a) Accuracy of ResNet-18 on CIFAR-10 under various levels of rank pruning. b) Further pruning the U matrix after applying 85% rank pruning. The graph is a result of the process in Section 3 and Figure 3.

5. Results for Sparsity-Preserving Low-Rank Fine-Tuning

In this section we demonstrate that the weight matrices of both convolution and transformer networks are amenable to rank pruning and UV matrix pruning as shown in Figure 3(a) and (b). In Figure 4(a) we show the accuracy

Rank Pruned / Rank	No U/V Pruning	50% U	50% U & V
0% 768	0.92008	0.88856	0.88592
50% 384	0.90688	0.88892	0.87624
60% 307	0.89652	0.8798	0.8646
70% 230	0.88652	0.86688	0.85052
80% 153	0.8608	0.85268	0.835

TABLE 1: Accuracy of Deberta-v3-base [3] on the IMDB dataset [4] under various rank-sliced pruning configurations. Note the initial rank of the model is 768.

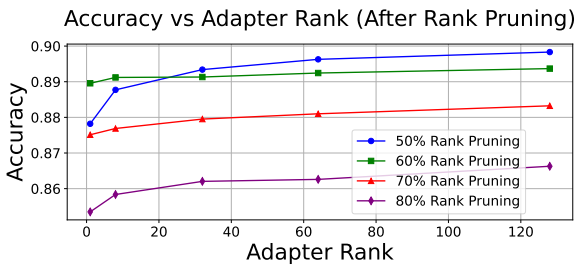

Figure 5: Accuracy of Deberta-v3-base on the IMDB dataset when fine tuning with a low-rank adapter. Our method, illustrated in Figure 3(c) achieves close to the performance shown in Table 1 while using adapters of small rank, e.g., 20 as opposed to the pretrained pruned model with rank 150.

of ResNet-18 [1] on CIFAR-10 [2] with rank pruning. In Figure 4(b) we can see that 85% rank pruning and up to 90% U pruning can achieve high accuracy.

We use Deberta-v3-base [3] and the IMDB dataset [4] to demonstrate how this fine tuning method is applicable to transformers. Each run started with a Deberta-v3-base model, converted it to SVD form and trained it for one epoch. Training using decomposition-based layers for one epoch before rank pruning significantly improved accuracy compared to pruning immediately after SVD. Then the SVD rank was pruned (iteratively for higher sparsity). Finally, the U and V matrices were pruned together.

Table 1 shows a slight drop in accuracy as the rank pruning increases. There is also a 1-2% drop in accuracy when combining the pruning of the U and V matrices. In Figure 5, we present the results of incorporating a low-rank adapter within the Deberta model. The model undergoes rank pruning, followed by fine-tuning with the integration of a low-rank adapter, as illustrated by Figure 3(c). This approach substantially reduces the number of trainable parameters needed. For example, in the case of Deberta-v3-base, the feedforward matrix in each Transformer block initially has an SVD rank of 768. After rank pruning, the matrix rank is reduced to $\approx$150 (See Table 1 for reduced ranks). During fine-tuning, we freeze this matrix and train low-rank adapter matrices on the IMDB dataset. Figure 5 reports accuracy results of our sparsity-preserving low-rank fine-tuning using adapters with ranks ranging from 1 to 128.

6. Background and Related Work

6.1. Gather-Scatter

Gather-scatter algorithms are widely used in fields such as signal processing, scientific computing, and parallel computing. They operate in two primary stages: a gather phase, where data is read from external memory, and a scatter phase, where the computed data is written back to the memory system or an output matrix [8], [9]. In this work, we extend the gather-scatter approach to a new area, that is, efficient computation of activations between a data matrix X and a weight matrix represented in SVD form.

6.2. Parameter-Efficient Fine Tuning

Parameter-Efficient Fine-Tuning (PEFT) techniques aim to minimize the number of parameters that need to be fine-tuned. Methods like Low-Rank Adaptation (LoRA) [5] fall under PEFT by adding trainable low-rank matrices to the weights, reducing the computational load of full fine-tuning without adding inference delay.

Masked LoRA [10] introduces a mask over the adaptation matrices to enforce sparsity. By applying a masking technique, it ensures that only a subset of weights are updated during fine-tuning, further reducing the overhead of adapting large models. SORA (Sparse Zero-Rank Adaptation) [11] zeros out specific ranks during LoRA updates. The approach selectively sets certain ranks in the LoRA updates to zero, effectively reducing the update dimensionality.

RoseLoRA [12] introduces sparsity directly into the A and B matrices used in LoRA updates. A and B are multiplied to provide a sparse update to the original weight matrix. This approach results in a sparse update, but the sparsity pattern may not be compatible with the original weight matrix, decreasing final sparsity.

DoRA [13] a weight-decomposed low-rank adaptation. It seperates the magnatude and direction of the weights and creates a low-rank adapter for the direction of the weights while leaving the magnitude unfrozen. LoRA-XS [14] uses an SVD of the original weight matrix to create a frozen low-rank adapters and an unfrozen low-rank update. This approach demonstrates the benefits of SVD but does not address sparsity of the SVD matrices.

None of these PEFT methods have the goal of preserving sparsity structures without introducing additional masks.

7. Conclusion

In this paper, we introduced the GAther-Scatter Activation (GASA) algorithm, a novel method for efficiently computing activations in neural network layers with weight matrices represented using SVD. GASA utilizes a *rank-sliced* approach, which computes matrix products in a gather-scatter manner, allowing skipping column computations due to sparsity in u and v vectors and sparsity-preserving fine-tuning. We demonstrate that GASA can significantly reduce I/O costs, achieving the theoretical minimum by reading and writing each matrix column at most once.

A key contribution of GASA is its ability to enable rank-sliced, *sparsity-preserving* low-rank fine-tuning, providing an alternative to LoRA that generate updates, not guaranteed to be compatible with the original networks sparsity. In summary, GASA offers a framework for efficient activation computation and low-rank fine-tuning in neural networks, preserving sparsity while minimizing I/O.

8. Acknowledgments

This research was supported in part by the Air Force Research Laboratory under award number FA8750-22-1-0500, and in part by Meta Platforms Technologies under award number A51540.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:18268744

[3] P. He, J. Gao, and W. Chen, "Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing," 2023. [Online]. Available: https://arxiv.org/abs/2111.09543

[4] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, D. Lin, Y. Matsumoto, and R. Mihalcea, Eds. Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. [Online]. Available: https://aclanthology.org/P11-1015

[5] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[6] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numer. Math.*, vol. 14, no. 5, p. 403–420, Apr. 1970. [Online]. Available: https://doi.org/10.1007/BF02163027

[7] G. Strang, *Linear algebra and learning from data*. SIAM, 2019.

[8] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[9] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[10] J. Wang, G. Yang, W. Chen, H. Yi, X. Wu, Z. Lin, and Q. Lao, "Mlae: Masked lora experts for visual parameter-efficient fine-tuning," 2024. [Online]. Available: https://arxiv.org/abs/2405.18897

[11] N. Ding, X. Lv, Q. Wang, Y. Chen, B. Zhou, Z. Liu, and M. Sun, "Sparse low-rank adaptation of pre-trained language models," 2023. [Online]. Available: https://arxiv.org/abs/2311.11696

[12] H. Wang, T. Liu, R. Li, M. Cheng, T. Zhao, and J. Gao, "Roselora: Row and column-wise sparse low-rank adaptation of pre-trained language model for knowledge editing and fine-tuning," 2024. [Online]. Available: https://arxiv.org/abs/2406.10777

[13] S.-Y. Liu, C.-Y. Wang, H. Yin, P. Molchanov, Y.-C. F. Wang, K.-T. Cheng, and M.-H. Chen, "Dora: Weight-decomposed low-rank adaptation," 2024. [Online]. Available: https://arxiv.org/abs/2402.09353

[14] K. Bałazy, M. Banaei, K. Aberer, and J. Tabor, "Lora-xs: Low-rank adaptation with extremely small number of parameters," 2024. [Online]. Available: https://arxiv.org/abs/2405.17604