Inducing Elasticity in Foundation Models: Post-Training Techniques for Adaptable Inference

Aashiq Muhamed, Jiarui Liu, Mona T. Diab, Virginia Smith
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{amuhamed,jiarui15,mdiab,smithv}@andrew.cmu.edu

Abstract

Large foundation models (LFMs) power a diverse range of applications, but their deployment often requires adapting model size and performance to specific hardware constraints and latency requirements. Existing approaches rely on training independent models of various sizes, leading to storage redundancy, inconsistent behavior across sizes, and limited scalability. This work investigates post-training techniques for inducing elasticity into pre-trained LFMs, enabling dynamic adaptation of model size during inference based on specific needs. We frame this as decomposing LFM weight matrices into sparsely activating factors. While naive decompositions like weight SVD struggle to maintain performance across complex tasks while inducing the desired nested sub-structures, we propose two novel methods: SparseDecomp, which exploits sparse neuron activations in feed-forward networks to conditionally select decoder rows; and RankDecomp, which leverages the basis-agnostic nature of Transformers for low-rank weight decomposition. Integrating SparseDecomp and RankDecomp with GritLM-7B, a state-of-the-art LFM excelling in both generative and embedding tasks, we conduct a comparative analysis. Our results demonstrate that these approaches offer complementary benefits. SparseDecomp maintains robust performance across a wider range of sparsity levels, achieving average speedups of up to 4.6% with 25% sparsity. RankDecomp, conversely, yields more significant latency reduction, reaching a speedup of 22.2% at 25% sparsity, but exhibits greater sensitivity to increasing sparsity. This study provides valuable insights into leveraging post-training weight decomposition for developing efficient and adaptable LFMs, paving the way for future research on creating elastic and resource-aware models.

1 Introduction

Large foundation models (LFMs) [2] have demonstrated remarkable capabilities across a wide range of applications. However, their deployment presents a fundamental challenge: the need for flexible model scaling strategies to adapt to varying hardware and latency constraints. Existing approaches, such as the Llama family of models [37], address this by offering independently trained models at various scales. This, however, leads to increased storage requirements during inference and potential behavioral inconsistencies between models, hindering optimization techniques like speculative decoding [21] and model cascades [40]. Furthermore, the discrete set of model sizes may not cater to the full spectrum of downstream tasks and hardware constraints, forcing compromises on either model accuracy or efficiency.

While model compression techniques like pruning [20, 26, 39], distillation [31, 36, 15], and quantization [6, 42] can reduce model size and improve efficiency, they often require access to the training

4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024).

data, additional computational resources, and may not facilitate dynamic adaptation of model size based on specific inputs or tasks. Recent work like MatFormer [17] tackles this challenge by training a natively elastic Transformer architecture, enabling a single universal model to generate numerous sub-models.

As pretraining a Transformer to induce elasticity is expensive and often infeasible, this work investigates whether post-training techniques can induce elasticity into existing pretrained LFMs, without requiring retraining or access to the original training data. We focus on integrating these techniques with GritLM [28], a state-of-the-art LFM trained using Generative Representational Instruction Tuning (GRIT) that excels in both generative and embedding tasks. Our investigation reveals that finetuning-based approaches (applying MatFormer loss to GritLM) and naive post-training baselines like weight singular value decomposition (SVD), gaussian weight sketching, leverage score sampling, and Discrete Empirical Interpolation Method (DEIM) fail to maintain GritLM's performance (see Appendix C). To address these challenges, we reframe the problem as identifying sparsely activating weight decomposition factors and introduce two post-training methods for adaptable inference:

1. SparseDecomp: Leverages sparsity in pretrained FFN activations to decompose weight matrices into sparse factors, enabling nested substructures through dynamic selection of decoder rows.
2. RankDecomp: Performs low-rank decomposition of weight matrices by leveraging the basis-agnostic nature of the residual stream in Transformers. This allows us to project weights onto a lower-dimensional subspace spanned by dominant activation modes, precomputed offline.

Our comparative analysis on GritLM-7B shows that SparseDecomp and RankDecomp offer complementary strengths in achieving latency-performance trade-offs across generative and retrieval augmented generation (RAG) benchmarks. SparseDecomp maintaind performance across a wide range of sparsity levels achieving an average speed-up of 4.6% with 25% sparsity. RankDecomp, on the other hand, achieves greater latency reductions with an average speed-up of 22.2% at 25% sparsity but shows more sensitivity to sparsity. Our contributions include:

- We propose and study two approaches to enhance model elasticity: SparseDecomp, which exploits sparse neuron activations in FFNs to conditionally select decoder rows, and RankDecomp, which utilizes the basis-agnostic nature of Transformers for weight dimension reduction. These methods successfully induce nested substructures within LFMs while preserving performance across complex tasks. Our experiments also show that traditional training-based and post-training baselines, including weight SVD, leverage score sampling, and DEIM, fail to achieve this dual objective.
- Our comparative analysis on GritLM-7B evaluates SparseDecomp and RankDecomp, demonstrating their complementary nature and detailing their respective strengths and weaknesses with respect to latency and performance on generative and embedding benchmarks.

2 Methodology

2.1 Weight Decomposition for Adaptable Inference

Neural networks are overparameterized, suggesting that their functionality can be captured by fewer *effective parameters* [22]. We hypothesize that decomposing weights into sparsely activating factors can identify these parameters. For a network output $y^L(x, \theta)$ with weights $\theta \in \mathbb{R}^N$, we seek weight factors $P = P_1, \ldots, P_F$, $P_i \in \mathbb{R}^N$, $F \ll N$, such that $y^L(x, \theta) \approx y^L(x, \tilde{\theta})$, where $\tilde{\theta} = \sum_{i=1}^{F} P_i$. We focus on nested decompositions, ranking P_i by performance contribution. This enables dynamic scaling based on task requirements or computational resources. The factor selection process is a submodular optimization:

$$\max_{S \subseteq 1,\ldots,F} \mathcal{P}(S) - \lambda \mathcal{C}(S) \tag{1}$$

where $\mathcal{P}(S)$ is submodel performance using factors indexed by S, $\mathcal{C}(S)$ is computational cost, and λ controls the performance-efficiency trade-off. This work explores post-training techniques for inducing elasticity in pretrained LFMs by decomposing layer weights into sparsely activating factors. By selecting a subset of these factors, we can control efficiency-performance trade-offs across diverse applications.

2.2 MatFormer: A Training-Based Nested Decomposition

MatFormer [18] introduces a training-based approach for nested Transformer structures, embedding hierarchy within FFN layers. It creates k nested Transformer blocks T_j, where $T_1 \subset T_2 \subset \cdots \subset T_k$, enabling submodels with varying computational demands. An FFN block T_j^{FFN} operates as:

$$T_j^{\text{FFN}}(x) = \sigma(x \cdot \mathbf{W}_{\text{up}}[:, : m_j]) \cdot \mathbf{W}_{\text{down}}[: m_j, :] \tag{2}$$

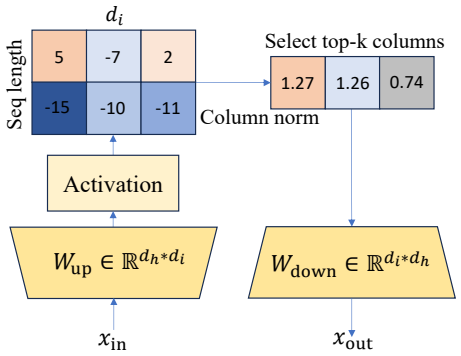
where $\mathbf{W}_{\text{up}} \in \mathbb{R}^{d_h \times d_i}$, $\mathbf{W}_{\text{down}} \in \mathbb{R}^{d_i \times d_h}$, $x \in \mathbb{R}^{d_h}$, and m_j determines columns/rows used based on a pre-defined form factor.

Each column (or row) can be viewed as a weight factor. For $\mathbf{W}_{\text{up}}$, each factor is defined as:

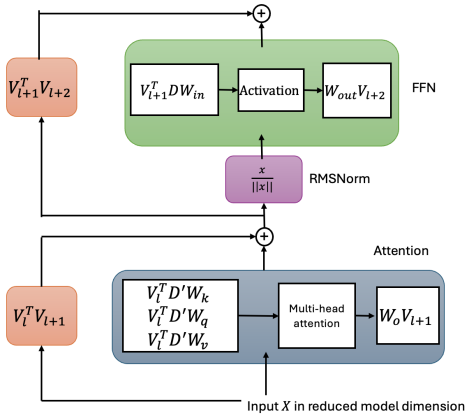
$$P_j = [\mathbf{0}_{d_h \times (j-1)}, \mathbf{W}_{\text{up}}[:, j], \mathbf{0}_{d_h \times (d_i - j)}] \tag{3}$$

where $j \in \{1, ..., d_i\}$ (similarly for $\mathbf{W}_{\text{down}}$). By training with randomly sampled form factors, MatFormer implicitly ranks these factors, creating the nested structure. The same decomposition is applied to all inputs x.

While sparsity is effective at trading off latency with performance, MatFormer's reliance on training from scratch limits its applicability to existing pretrained models. We examine two post-training methods inspired by existing work, SparseDecomp and RankDecomp, which induce nested structures without predefined rankings or retraining.



(a) SparseDecomp applied to an FFN layer. We take L2 norms along the sequence dimension, and select the top-k columns with the largest values. The unselected columns are shown in gray.



(b) RankDecomp applied to a single Transformer layer. The V matrices are obtained from right singular vectors of the activation matrix, and D and D' are scaling factors from the conversion of LayerNorm into RMSNorm.

2.3 SparseDecomp: Sparse Decomposition of FFN Weights

SparseDecomp (Fig 1(a)) leverages the inherent sparsity in pretrained FFN activations to decompose weight matrices into sparse factors, enabling efficient nested substructures. This approach is analogous to treating FFNs as sparse autoencoders, where weakly activated features can be removed to compress the representation.

Studies show that only a small subset of neurons within FFN blocks are active for a given input [23, 7]. For inputs $\mathbf{X} \in \mathbb{R}^{B \times S \times d_h}$, the FFN computation is:

$$\mathbf{Z} = \text{FF}_1(\mathbf{X}) = \sigma(\mathbf{X}\mathbf{W}_{\text{up}} + \mathbf{b}_{\text{up}}) \tag{4}$$

$$\text{FF}_2(\mathbf{Z}) = \mathbf{Z}\mathbf{W}_{\text{down}} + \mathbf{b}_{\text{down}} \tag{5}$$

where $\mathbf{Z} \in \mathbb{R}^{B \times S \times d_i}$ is the activation matrix and $\mathbf{W}_{\text{down}} \in \mathbb{R}^{d_i \times d_h}$.

Decoder rows are selected by averaging feature activations across the batch and taking the L2 norm across the sequence dimension denoted $\|\cdot\|_{\text{r}}$: $\mathbf{a} = \left\| \frac{1}{B} \sum_{b=1}^{B} \mathbf{Z}_b \right\|_{\text{r}} \in \mathbb{R}^{d_i}$. Let π be the permutation

that sorts elements of $\mathbf{a}$ in descending order. We select the top-k decoder rows corresponding to the k largest values in $\mathbf{a}$. Sparse factors $P_j \in \mathbb{R}^{d_i \times d_h}$ for $\mathbf{W}_{\text{down}}$ are defined as:

$$P_j = \mathbf{e}_{\pi(j)} \mathbf{W}_{\text{down}}[\pi(j), :]^\top \cdot \mathbf{1}[j \leq k] \tag{6}$$

where $j \in \{1, ..., d_i\}$, $\mathbf{e}_{\pi(j)} \in \mathbb{R}^{d_i \times 1}$ is the $\pi(j)$-th standard basis vector, and $\mathbf{1}[\cdot]$ is the indicator function. The decomposed matrix is constructed as $\hat{\mathbf{W}}_{\text{down}} = \sum_{j=1}^{d_i} P_j$. This construction yields a nested structure, where increasing k incorporates more factors, leading to larger and potentially more accurate submodels but also increasing latency. By dynamically adjusting threshold k for each input batch, SparseDecomp allows for adaptable inference. However, there is an overhead to performing this subset selection at inference time.

2.4 RankDecomp: Basis-Agnostic Low-Rank Decomposition

RankDecomp (Fig 1(b)) leverages the property that the *residual stream* in Transformers lacks a privileged basis [8], to decompose weight matrices into low-rank factors. This property implies that any invertible linear transformation applied to the residual stream can be compensated by corresponding inverse transformations on weight matrices, yielding an equivalent model with a different coordinate system but identical functionality. This stems from attention and FFN layers interacting with the residual stream through arbitrary full-rank linear transformations. RankDecomp exploits this by introducing orthogonal transformations that are aligned with the dominant modes of variation in the activations, to project weights onto a lower-dimensional subspace.

For a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ in a Transformer block that acts on the residual stream, RankDecomp performs low-rank decomposition using SVD on flattened input activations $\mathbf{X} \in \mathbb{R}^{B \times d_{\text{in}}}$, where B is the total number of samples in a tiny calibration dataset. The SVD yields $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, with $\mathbf{V} = [\mathbf{v}_1, ..., \mathbf{v}_{d_{\text{in}}}]$ containing right singular vectors sorted by decreasing singular values. RankDecomp approximates $\mathbf{W}$ by projecting onto the subspace spanned by the top k right singular vectors: $\hat{\mathbf{W}} = \mathbf{V}_k \mathbf{V}_k^\top \mathbf{W} = \sum_{j=1}^{k} P_j$, where $\mathbf{V}_k \in \mathbb{R}^{d_{\text{in}} \times k}$ contains the first k columns of $\mathbf{V}$, and $P_j = \mathbf{v}_j \mathbf{v}_j^\top \mathbf{W}$ are rank-one decomposition factors. This creates nested substructures where adjusting k controls the approximation rank and submodel dimension, enabling adaptive inference by balancing accuracy and efficiency. The basis V and k are determined based on activations from a calibration dataset offline, incurring no additional runtime overhead. The calibration set is typically orders of magnitude smaller than the pretraining dataset.

We apply RankDecomp to the weights in the FFN, Embedding, and Attention layers inspired by several ideas in existing work [1, 11] to ensure we realize efficiency gains from this decomposition.

Basis Invariance: The LayerNorm operation after the attention layer introduces a weak privileged basis by normalizing activations based on their specific coordinate values. To apply RankDecomp to the activations between attention and FFN layers, we convert all LayerNorm operations to RMSNorm, which makes the basis unprivileged. This conversion involves absorbing the scaling and shifting components of LayerNorm into adjacent weight matrices [1].

RankDecomp Transformations: To realize efficiency gains from the weight decomposition, we apply $\mathbf{V}_\ell \in \mathbb{R}^{d_h \times k}$ (right singular vectors) by splitting the computation $\mathbf{V}_k \mathbf{V}_k^\top \mathbf{W}_\ell$ into $\mathbf{V}_k^\top \mathbf{W}_\ell$ at current layer ℓ and $\mathbf{W}_{\ell-1} \mathbf{V}_k$ at the previous layer. These matrices project the activations onto lower-dimensional subspaces spanned by the top k singular vectors which effectively reduces model dimension. Similarly, we apply these transformations to the embedding and FFN weights as follows: $\tilde{\mathbf{W}}_{\text{embd}} = \mathbf{W}_{\text{embd}} \mathbf{V}_1$, $\tilde{\mathbf{W}}_{\text{up}}^\ell = \mathbf{V}_{\ell-1}^\top \mathbf{W}_{\text{up}}^\ell$, $\tilde{\mathbf{W}}_{\text{down}}^\ell = \mathbf{W}_{\text{down}}^\ell \mathbf{V}_\ell$,. This effectively reduces the model dimension in all residual streams to realize efficiency gains.

Residual Connection Correction: As we use different orthogonal transformations at each layer, we introduce correction terms $\mathbf{V}_{\ell-1}^\top \mathbf{V}_\ell$ in the residual connections. These terms account for the change of basis between consecutive layers and incur a small latency overhead.

3 Experiments And Results

3.1 Experimental Setup

To compare the two weight decomposition approaches SparseDecomp and RankDecomp, we use GritLM-7B [28] as our base model. GritLM integrates embedding and generative tasks through instruction tuning, allowing it to perform effectively across both tasks without the need for separate models. It is initialized with Mistral 7B [16] and incorporates Query-Doc Caching, to reduce the number of forward passes, making it more efficient than traditional RAG. SparseDecomp is applied to the FFN decoder in each layer, while RankDecomp is applied to the embedding, attention, and FFN encoder and decoder weights in every layer. We explore varying levels of sparsity, ranging from 5% to 50%. For SparseDecomp, 25% sparsity means selecting the top 75% of decoder rows from the intermediate dimension d_i. For RankDecomp, it corresponds to using the top 75% of singular vectors that span the model dimension.

Following the GritLM evaluation setup in [28], we evaluate our approaches on generative, embedding, and latency benchmarks to comprehensively assess the performance-latency trade-offs. Generative performance is evaluated on MMLU, GSM8K, BBH, and HumanEval [13, 5, 33, 34, 27], using Pass@1 as the metric for HumanEval and exact match for the other tasks. To ensure a realistic scenario where pretraining datasets of GritLM are unavailable during calibration, the RankDecomp experiments employ two calibration datasets: the Alpaca training dataset [35] and the WikiText-2 training dataset [25], each comprising 1024 samples. For embedding performance, we use the Semantic Text Similarity (STS) subset of the MTEB benchmark [29], with Spearman correlation against ground truth as the evaluation metric. All generative and embedding benchmarks are conducted on a single A6000 GPU.

Latency is measured on the Natural Questions dataset [19], consisting of 2,681,468 documents derived from the BEIR NQ corpus [28], under three modes: No RAG, RAG only, and Query Caching. In RAG mode, retrieved context is added to the input, while Query Caching reuses key-value states from the embedding pass to minimize forward passes and storage costs. GPU latency is measured on an NVIDIA L40 46GB GDDR6. Additional details on the benchmarks are provided in Appendix B.

3.2 Results

Dataset	Reference	SparseDecomp						RankDecomp Alpaca					RankDecomp WT
	GritLM 7B	5%	10%	25%	50%	25% Sampling	25% Abs	5%	10%	25%	50%	25% FT	25%
MMLU	0.575	0.575	0.576	0.575	0.574	0.574	0.549	0.567	0.548	0.463	0.307	0.493	0.365
GSM8K	0.575	0.565	0.540	0.530	0.410	0.435	0.000	0.505	0.460	0.225	0.005	0.120	0.055
BBH	0.547	0.562	0.540	0.533	0.514	0.529	0.000	0.543	0.515	0.425	0.149	0.423	0.100
HumanEval	0.305	0.274	0.268	0.256	0.220	0.262	0.000	0.287	0.268	0.110	0.000	0.207	0.000
Average	0.500	0.494	0.481	0.474	0.430	0.450	0.137	0.476	0.448	0.306	0.115	0.311	0.130

Table 1: Generative benchmark: Performance of GritLM-7B, SparseDecomp, and RankDecomp calibrated on Alpaca or WikiText-2 (WT). FT refers to recovery finetuning.

Generative Performance The results in Table 1 show a consistent decline in performance for both RankDecomp and SparseDecomp as sparsity levels increase. SparseDecomp is more robust to performance degradation with sparsity, with an average performance drop of only 10% even at 50% sparsity. This suggests that SparseDecomp can effectively identify and select activated rows in the FFN decoder. With RankDecomp, using the Alpaca training dataset, the performance decline is more pronounced. At 5% sparsity, the performance drops modestly by 5%, but as sparsity increases to 25%, the decline becomes severe, with a 40% reduction in performance. This trend is particularly noticeable in complex reasoning tasks, where excluding GSM8K, performance drops by 23% at 25% sparsity. At 50% sparsity, the performance drop is even more dramatic, plummeting by approximately 80%.

Embedding Performance As shown in Table 2, the embedding performance of SparseDecomp is also robust to sparsity, exhibiting only minimal average performance degradation, and it remains highly effective for retrieval tasks. Embedding tasks only involve a single forward pass, and are in general more robust to sparsity than generative tasks where errors accumulate. We also observed that the mean pooling is applied over the final hidden state for embedding generation is less sensitive to

Dataset	Reference	SparseDecomp						RankDecomp Alpaca					RankDecomp WT
	GritLM 7B	5%	10%	25%	50%	25% Sampling	25% Abs	5%	10%	25%	50%	25% FT	25%
BIOSSES	86.31	86.38	86.37	86.31	86.49	86.38	86.29	86.68	86.26	85.51	78.39	82.93	86.98
SICK-R	83.13	83.11	83.09	83.01	82.50	82.96	73.09	82.97	82.95	82.28	75.38	81.55	81.55
STS12	77.34	77.32	77.31	77.31	77.07	77.33	74.24	76.14	75.81	74.58	67.28	74.52	74.75
STS13	85.05	85.04	85.03	85.02	84.84	85.03	82.26	84.49	84.66	85.10	79.95	82.98	84.34
STS14	82.91	82.85	82.83	82.70	82.42	82,76	78.64	82.38	81.87	81.11	73.42	77.86	81.10
STS15	88.13	88.12	88.12	88.11	87.84	88.02	88.02	87.16	86.82	86.36	80.81	85.15	86.24
STS16	86.24	86.24	86.26	86.25	86.24	86.27	85.59	85.42	85.14	84.91	80.68	83.53	85.18
STS17	90.15	90.12	90.12	90.08	89.92	90.10	88.53	90.21	90.33	89.63	82.89	89.27	88.83
STS22	68.61	68.61	68.62	68.59	68.61	68.62	68.63	67.85	66.82	64.87	61.02	64.22	65.25
STSBenchmark	85.64	85.62	85.62	85.64	85.68	85.67	84.97	84.61	84.13	82.56	75.17	80.32	83.30
Average	83.35	83.34	83.34	83.30	83.16	83.31	81.03	82.79	82.48	81.69	75.50	80.23	81.75

Table 2: Embedding Benchmark: Performance of GritLM-7B, SparseDecomp, and RankDecomp calibrated on Alpaca or WikiText-2 (WT), on the STS subset of MTEB. We evaluate the cosine similarity between two embeddings against a ground truth continuous score, scored using Spearman correlation.

sparsity than the language modeling head is used for text generation. While RankDecomp shows a more noticeable decline in performance as sparsity increases, the drop is around 2% at 25% sparsity, which is acceptable for most applications.

Latency Performance Figure 1 and Appendix Figures 10 to 13 reveal that increasing sparsity in either SparseDecomp or RankDecomp reduces latency across various RAG configurations, including No RAG, RAG, and Query Caching. The latency reduction observed with SparseDecomp stems from accelerated matrix multiplications involving the activation matrix and $\mathbf{W}_{\text{down}}$. Compared to SparseDecomp, RankDecomp achieves a more pronounced reduction in latency due to the reduced model dimension and associated computational cost of matrix multiplication in both the attention and FFN layers. RankDecomp therefore exhibits higher sensitivity to sparsity than SparseDecomp.

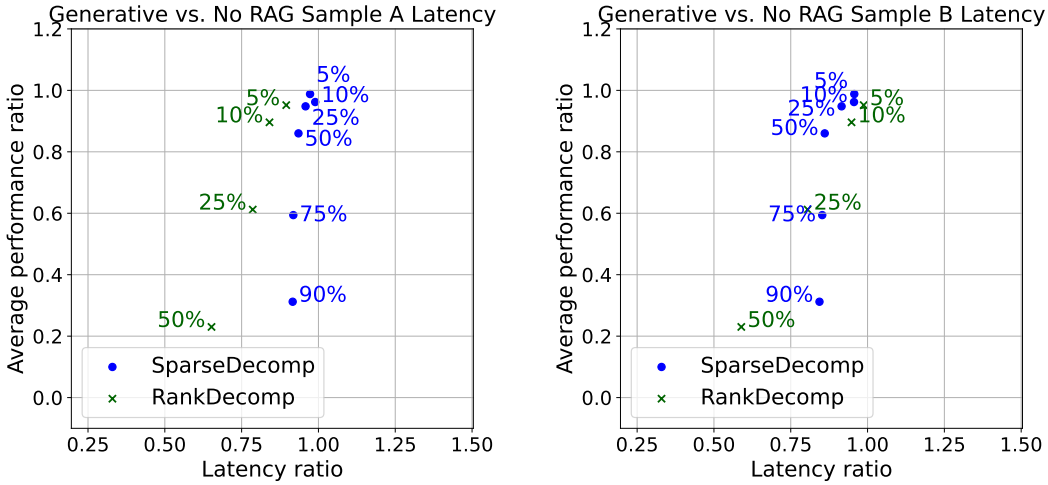


Figure 1: Trade-off between average generative performance and latency ratio under the No RAG mode, evaluated on Sample A (left) and Sample B (right). Sample A has a query of 1 token and a document of 4000 tokens, and sample B is the inverse. For each approach, we generate 16 tokens. The latency ratio is measured by dividing the latency of the model with introduced sparsity by the latency of running GritLM-7B without any sparsity applied. Similarly, the average performance ratio is reported relative to GritLM-7B without any sparsity applied.

3.3 Ablation Studies

SparseDecomp: Sampling-Based Selection As an ablation study, we implemented a sampling-based selection method instead of directly selecting the top-k decoder rows to construct the decomposed matrix. Rows were randomly sampled according to the multinomial distribution of the weights of the activation matrix $\mathbf{Z}$. The results for 25% sparsity are shown in Table 1 and Table 2, denoted as "25% Sampling". We observed a moderate drop in generative performance compared to the top-k

selection method at the same sparsity level, while the embedding performance showed minimal degradation under this change.

SparseDecomp: Absolute Magnitude Selection We explored an alternative method for calculating $\mathbf{a}$ without incorporating inputs $\mathbf{X}$ by using weight matrix $\mathbf{W}_{\text{up}} \in \mathbb{R}^{d_h \times d_i}$. We use the L2 norm along dimension d_h and define $\mathbf{a} = \|\mathbf{W}_{\text{up}}\|_{\text{r}} \in \mathbb{R}^{d_i}$. We then select the top-k rows based on absolute magnitude of a that remain consistent across sequences. However, as shown in Table 1, we find that this significantly reduces generative performance to zero on the GSM8K, BBH, and HumanEval datasets, emphasizing the need to tailor magnitude selection to each input sequence based on feature activations for preserving reasoning performance. In contrast, the MMLU evaluation shows only a 2.6% performance decrease. Here MMLU is treated as a classification task by extracting output logits for each answer choice. For embedding benchmarks, the average performance decline is relatively modest at 2.27%, indicating that the selected embedding activation rows are more universally applicable across different sequences in embedding tasks compared to generative tasks.

RankDecomp: Effect of Calibration Dataset For the generative benchmark, we observed that performance is highly sensitive to the choice of calibration data. Calibration using a dataset designed for instruction tuning, such as Alpaca, led to better outcomes, reducing performance degradation by 40% compared to using a general dataset like WikiText-2, which resulted in a 75% reduction. This improvement is likely because GritLM was tailored for instruction tuning, and complex reasoning tasks require preserving this specialized knowledge within the activations. In contrast, the embedding benchmark exhibited less sensitivity to calibration data, with both Alpaca and WikiText-2 yielding roughly equivalent performance.

RankDecomp: Effect of Post-Calibration Recovery finetuning In the recovery finetuning experiment (denoted FT), we finetuned the model post-sparsification with a GritLM generative loss (autoregressive perplexity) on the Alpaca dataset using 4096 samples at a batch size of 8. All models were finetuned on a single A6000 GPU within 1 to 3 hours. As shown in Table 1, recovery finetuning provided some improvement, recovering about 2% of generative performance. We hypothesize that the effectiveness of this approach could be further enhanced with additional calibration data that more closely aligns with the evaluation data distribution. However, as seen in Table 2, recovery finetuning slightly negatively impacted embedding benchmark performance, reducing it by an additional 1.5% compared to not applying finetuning.

3.4 Discussion

The results demonstrate the distinct yet complementary strengths of SparseDecomp and RankDecomp across generative, embedding, and latency benchmarks. SparseDecomp shows robustness in generative and embedding tasks, maintaining performance even under increased sparsity levels due to its ability to dynamically select activated columns tailored to each input sequence. However, the lateny reduction achieved by SparseDecomp even as sparsity levels increase, is less pronounced. On the other hand, RankDecomp offers significant latency reductions and maintains competitive performance even at lower sparsity levels by reducing computational cost through dimensionality reduction in both attention and FFN layers. RankDecomp however shows performance degradation at high sparsity levels and is highly sensitive to the choice of calibration dataset, which directly influences its generative performance. Together, these approaches provide a balanced trade-off: SparseDecomp excels in dynamic adaptability, is robust to a wide range of sparsity levels, and requires no calibration, while RankDecomp improves computational efficiency and remains effective with carefully chosen calibration data. Their complementary nature suggests that integrating these methods could optimize performance across various tasks and sparsity conditions, leveraging the dynamic selection strengths of SparseDecomp with the efficiency and tuning sensitivity of RankDecomp.

4 Related Work

Large foundation models must cater to a variety of users with different performance needs and budgets, leading to the costly practice of training, storing, and maintaining numerous user and task-specific models. This challenge has spurred research into developing multiple models with various inference latencies from a single training process, thereby reducing the overhead associated with creating and

managing multiple distinct models. Prior work has investigated training methods that enable subnet extraction methods, such as in CNNs [41, 30, 3] and encoder-only transformers [14, 9, 12]. Among decoder-based models, SortedNet [38] trains multiple sub-models simultaneously through a nested architecture. Alternatively, MatFormer [17] introduces an efficient approach by embedding a nested structure within FFN layers, enabling a single model to generate diverse sub-models without multiple training runs. Although effective at weight decomposition, MatFormer requires training a model from scratch, limiting its application to induce elasticity in existing open-source LLMs. For additional related work see Appendix A.

Unlike existing work, in this paper we explore post-training methods aimed at inducing elasticity without additional training or minimal training (using a calibration set). We first show that full and partial training with MatFormer-like losses are inefficient, while naive projection approaches such as Gaussian projection, leverage score sampling, and weight SVD perform poorly. We then introduce the idea of weight decomposition to unify existing approaches, and introduce SparseDecomp and RankDecomp. These novel methods are inspired by existing work in the inference efficiency literature [7, 1], which we extend for inducing elasticity in pretrained models. Our approach allows for the creation of multiple models with varying inference latencies from a single pretrained model, addressing the need for diverse performance options without the overhead of training and maintaining separate models.

5 Conclusion And Future Work

This work explored two novel post-training methods for inducing elasticity in pretrained LFMs: SparseDecomp, which focuses on exploiting sparse activations in FFNs, and RankDecomp, which leverages the basis-agnostic property of Transformer residual streams for dimension reduction. Our experiments on GritLM-7B demonstrate that both methods can effectively create nested sub-models with varying sparsity levels, enabling adaptable inference without retraining. While SparseDecomp exhibits robustness across generative and embedding tasks, RankDecomp achieves larger latency reductions but shows sensitivity to the calibration dataset. Future work will investigate combining these methods to optimize for specific tasks and desired performance-latency trade-offs. Additionally, exploring per-layer reduction strategies and adaptive routing mechanisms for sub-model selection could further enhance the efficiency and flexibility of elastic LFMs. This research paves the way towards adaptable and resource-aware deployment of powerful foundation models across a wider range of applications.

References

[1] Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. Slicegpt: Compress large language models by deleting rows and columns. *arXiv preprint arXiv:2401.15024*, 2024.

[2] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga,

Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *arXiv preprint arXiv: 2108.07258*, 2021.

[3] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[4] Saifon Chaturantabut and Danny C Sorensen. Nonlinear model reduction via discrete empirical interpolation. *SIAM Journal on Scientific Computing*, 32(5):2737–2764, 2010.

[5] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[6] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023.

[7] Harry Dong, Beidi Chen, and Yuejie Chi. Prompt-prompted mixture of experts for efficient llm generation. *arXiv preprint arXiv:2404.01365*, 2024.

[8] Nelson Elhage, Robert Lasenby, and Christopher Olah. Privileged bases in the transformer residual stream, 2023. *URL https://transformer-circuits. pub/2023/privilegedbasis/index. html. Accessed*, pages 08–07, 2023.

[9] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[10] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv: 2210.17323*, 2022.

[12] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021.

[13] Dan Hendrycks, Nicholas Carlini, John Schulman, and Jacob Steinhardt. Unsolved problems in ml safety. *arXiv preprint arXiv:2109.13916*, 2021.

[14] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33:9782–9793, 2020.

[15] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[16] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[17] Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, Prateek Jain, et al. Matformer: Nested transformer for elastic inference. *arXiv preprint arXiv:2310.07707*, 2023.

[18] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. *Advances in Neural Information Processing Systems*, 35:30233–30249, 2022.

[19] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[20] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[21] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *International Conference on Machine Learning*, 2022.

[22] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018.

[23] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[24] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.

[25] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[26] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32, 2019.

[27] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

[28] Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. Generative representational instruction tuning. *arXiv preprint arXiv:2402.09906*, 2024.

[29] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022.

[30] Elvis Nunez, Maxwell Horton, Anish Prabhu, Anurag Ranjan, Ali Farhadi, and Mohammad Rastegari. Lcs: Learning compressible subspaces for adaptive network compression at inference time. *arXiv preprint arXiv:2110.04252*, 2021.

[31] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[32] Sharath Nittur Sridhar and Anthony Sarah. Undivided attention: Are intermediate layers necessary for bert? *arXiv preprint arXiv:2012.11881*, 2020.

[33] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[34] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

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

[36] Yijun Tian, Yikun Han, Xiusi Chen, Wei Wang, and Nitesh V Chawla. Tinyllm: Learning a small student from multiple large language models. *arXiv preprint arXiv:2402.04616*, 2024.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *ARXIV*, 2023.

[38] Mojtaba Valipour, Mehdi Rezagholizadeh, Hossein Rajabzadeh, Marzieh Tahaei, Boxing Chen, and Ali Ghodsi. Sortednet, a place for every network and every network in its place: Towards a generalized solution for training many-in-one neural networks. *arXiv preprint arXiv:2309.00255*, 2023.

[39] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418*, 2019.

[40] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, page 105–114, New York, NY, USA, 2011. Association for Computing Machinery.

[41] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018.

[42] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

A Additional Related Works

While model families like Llama offer independently trained models at different scales, this approach suffers from collocation overhead, behavioral inconsistencies, and a limited selection of sizes. To address these limitations, researchers have explored various strategies, including:

Pruning: Techniques like Block Pruning [20] remove unnecessary weights or attention heads [26, 39, 32, 24] to reduce model size. However, these methods often require additional training and may be architecture-specific.

Distillation: Knowledge distillation methods [31, 36, 15] transfer knowledge from a larger model to a smaller one, achieving size reduction while preserving performance. However, they still require training a separate smaller model.

Quantization: Quantization techniques [6, 42] represent model weights with lower precision formats, reducing memory footprint and improving efficiency. However, quantization can lead to accuracy degradation.

Mixture of Experts (MoE): MoE models [10] employ a sparsely activated set of experts, allowing for efficient scaling and adaptation to different inputs. However, training and managing MoE models can be complex.

B Experiment Benchmark Details

B.1 Generative Performance Benchmark

Generative performance is evaluated on MMLU [13], GSM8K [5], BBH [33, 34], and HumanEval [27] using exact match for MMLU, GSM8K, and BBH, and Pass@1 for HumanEval [28].

B.2 Embedding Performance Benchmark

Embedding performance is assessed using the Massive Text Embedding Benchmark (MTEB) [29], specifically focusing on the Semantic Text Similarity (STS) task subset. Performance is measured by comparing the cosine similarity of embeddings against ground truth similarity scores, with results reported as Spearman correlation coefficients.

B.3 Latency Benchmark

We benchmark the GritLM model for Retrieval-Augmented Generation (RAG) in three modes: No RAG, RAG only, and Query Caching. In RAG only mode, the retrieved context is placed directly into the language model's input. Although GritLM supports both Query Caching and Query-Doc Caching modes, this study focuses on Query Caching due to the significant storage demands of caching entire documents. Query Caching minimizes the number of forward passes by storing the key-value states from the embedding forward pass in bfloat16 format and reusing them during generation, reducing storage requirements compared to document caching.

Latency measurements use the Natural Questions dataset [19] with an index of 2,681,468 documents derived from the BEIR NQ corpus [28]. Latency is assessed on an "AMD EPYC 7763 64-Core Processor with 128 threads across 2 sockets" and an "NVIDIA L40 46GB GDDR6" GPU. For each approach, two sample configurations are tested: Sample A (1 token query, 4000 tokens document) and Sample B (4000 tokens query, 1 token document), with 16 tokens generated per test. The index is stored in float32 format, and overall storage consists of the index and the passages. Variations in I/O latency across benchmarking nodes necessitate re-evaluation of the reference model (GritLM-7B) latency for each mode.

C Other Baselines

This section explores initial attempts at inducing elasticity in GritLM 7B by training-based and weight sketching approaches which unfortunately resulted in significant performance drops. These methods serve as a foundation for understanding the challenges and motivating the need for alternative approaches.

C.1 Training-based Approaches

Full and Partial Training with Matformer Loss: We explored replacing GritLM's FFN layers with a nested Matformer structure and finetuning the entire model or just the FFN layers using Matformer's joint loss. However, even after training for 2 days, these approaches failed to preserve GritLM's performance, indicating that simply integrating Matformer's structure and loss is both expensive and insufficient.

Initialization Strategies: We experimented with various initialization strategies for the FFN layers, including zero initialization, selective random initialization (where smaller submodels received random weights), and other methods like He and Xavier initialization. None of these strategies led to significant improvements, suggesting that initialization alone cannot effectively address the challenges of inducing elasticity with training based approaches.

C.2 Post-training Approaches

Singular Value Decomposition (SVD): SVD decomposes weight matrices W into $W = U\Sigma V\top$ and creates lower-rank approximations $W_k = U_k\Sigma_k V_k\top$ by selecting the top k singular values. While this reduces parameters and computations, applying SVD to GritLM led to significant performance drops, demonstrating the limitations of linear approximations.

Leverage Score Sampling: This data-driven approach selects important rows of a matrix based on leverage scores $l_i = \|U_i\|^2$, where U_i is the i-th row of U from the SVD. Despite its theoretical appeal, leverage score sampling also failed to maintain performance, suggesting that row importance might not directly correlate with contributions to downstream tasks.

Gaussian Random Projection: This method projects weight matrices onto a lower-dimensional subspace using random Gaussian matrices R: $W_k = WR$. While efficient, this approach also resulted in performance degradation.

Discrete Empirical Interpolation Method (DEIM): DEIM attempts to address nonlinearities by selecting a subset of informative entries within the activation states using Proper Orthogonal Decomposition (POD) and DEIM indices (see Alg 1). However, in our experiments, DEIM accumulated residual errors and underperformed weight SVD on the embedding benchmark.

Overall, these baseline methods, demonstrate the difficulties of inducing elasticity while maintaining performance emphasizing the need for alternative approaches.

C.3 Discrete Empirical Interpolation Method

The performance degradation of these post-training approaches can be attributed to the significant nonlinearity present in GritLM, primarily due to the activation functions within the FFN layers. These functions introduce non-linear transformations that are crucial for the model's expressive power and performance. The linear reduction techniques we explored fail to account for these nonlinearities, leading to discrepancies between the reduced models and the original GritLM. To effectively induce elasticity while preserving performance, we need to consider alternative approaches that focus on the activation states within the FFN layers and their modes of variation. By selectively removing less important modes, we can reduce the dimensionality of the activation states without sacrificing the model's ability to capture essential information. We explore this strategy next.

To address the computational challenges posed by the nonlinearities within GritLM's activation states, we draw inspiration from Reduced-Order Modeling (ROM) and specifically implement the Discrete Empirical Interpolation Method (DEIM) [4]. This approach reduces the dimensionality of the activation states, leading to faster and more adaptable model variants while maintaining accuracy.

DEIM mitigates the computational complexity associated with nonlinearities by strategically selecting a subset of the most informative entries within these high-dimensional activation states. The method uses Proper Orthogonal Decomposition (POD), which identifies the dominant modes of variation within the data. Mathematically, let $Y \in \mathbb{R}^{b\times s\times d_h}$ represent a batch of input activation states to the FFN, where b is the batch size, s is the sequence length, and d_h is the hidden dimension. As depicted in Algorithm 1, during offline calibration over a calibration dataset, we construct a covariance matrix Z by accumulating the outer product of the activations over multiple calibration steps. Here $Y_{act}^{(n)}$ represents the activation state at the n-th calibration step, obtained after the first MLP and nonlinearity in the FFN. f denotes the activation function, W_{gate} and W_{up} are projection matrices,

and $\odot$ represents element-wise multiplication. Subsequently, we perform eigendecomposition on Z: where Λ is a diagonal matrix containing the eigenvalues and V is a matrix whose columns are the corresponding eigenvectors. We select the top k eigenmodes based on the magnitude of the eigenvalues.

The crucial step in DEIM is the determination of DEIM indices, which pinpoint the most representative entries within the selected eigenmodes. This can be accomplished using either a greedy selection algorithm or leverage score selection. These indices guide the selection of a subset of entries from the activation states during online inference, effectively reducing the dimensionality and computational cost. We opted for the leverage score selection in the current experiments, as solving the least squares problem for every FFN was prohibitively expensive. By incorporating DEIM, we transform the original operation: $Y_{final} = W_{down}(f(W_{gate}Y) \odot W_{up}Y)$ into a more efficient form: $Y_{final} = W_{down}W_{pre}(f(W_{gate}[P,:]Y) \odot W_{up}[P,:]Y)$ where P represents the DEIM selection matrix.

The motivation behind using DEIM stems from the need to address the computational bottleneck presented by the nonlinearities in GritLM that prevent naive slicing. In our experiments however we found that DEIM accumulates residual errors and underperforms weight SVD on the embedding benchmark. We highlight a few directions to investigate this in future work.

D Experiments and Results

D.1 Experimental Details

GritLM-7B GritLM [28], effectively combines embedding and generative tasks through instruction tuning. Unlike traditional Retrieval Augmented Generation (RAG) systems that rely on separate models for these tasks, GritLM leverages a single model initialized with Mistral 7B [16]. Despite its unified nature, GritLM achieves performance comparable to models specifically designed for either embedding or generation tasks. The key to GritLM's efficiency lies in its ability to reduce the computational cost associated with processing user queries and context. By employing "Query-Doc Caching," GritLM stores and reuses relevant computations, effectively halving the number of forward passes required compared to RAG systems. Furthermore, depending on the task, GritLM utilizes different attention mechanisms: bidirectional attention for embedding tasks to capture comprehensive context, and causal attention for generation tasks to produce coherent and contextually relevant text.

Training based approaches Based on the open-sourced pretrained GritLM model,[1] we reimplement its FFN layers by introducing the nested structure, and assign equal weights of partial losses for all submodels. Following the MatFormer implementation, we set form factors for four submodels as $1, 2, 4, 8$, and train these submodels simultaneously. We train the model on the embedding dataset Medi2 [2] and the generation dataset Tulu2 [3] [28].

Post training approaches Gaussian Random Projection, Leverage Score Sampling, and Weight SVD do not involve any training. DEIM uses a subsampled (25%) calibration set from the evaluation set without labels in all experiments.

D.2 Latency Benchmark

We report the latency benchmark for baselines in Table 3.

D.3 Embedding Benchmark

The performance of the baselines on the embedding benchmark is listed in Table 4.

[1] `https://huggingface.co/GritLM/GritLM-7B`

[2] `https://huggingface.co/datasets/GritLM/MEDI2`

[3] `https://huggingface.co/datasets/GritLM/tulu2`

Algorithm 1 Discrete Empirical Interpolation Method For Inducing Elasticity

Input: Batch size b, sequence length s, hidden dimension d_h, intermediate dimension d_i, activation function f, number of calibration steps N_c, truncation number k, FFN input $Y \in \mathbb{R}^{b \times s \times d_h}$.
Output: Model optimized with DEIM for efficient inference.
procedure OFFLINECALIBRATION(Y) ▷ Repeat for every layer
  for $n = 1$ to N_c **do** ▷ Iterate over calibration steps
    $Y_{proj} = W_{gate} \cdot Y^{(n)}$ ▷ Project using gate weights
    $Y_{up} = W_{up} \cdot Y^{(n)}$ ▷ Project using up weights
    $Y_{act} = f(Y_{proj}) \odot Y_{up}$ ▷ Apply activation and element-wise multiplication
    Update covariance matrix: $Z \mathrel{+}= Y_{act} Y_{act}^{\top}$ ▷ Batch matrix multiply and accumulate over batch dimension
  end for
  $\Lambda, V = \text{eig}(Z)$ ▷ Eigendecomposition of Z
  Identify top k eigenmodes:
  $idx = \text{argsort}(-\Lambda)[:k]$
  $V_f = V[:, idx]$
  Determine DEIM indices: $P = \text{GreedySelection}(V_f, k)$ **or** $P = \text{LeverageScoreSelection}(V_f, k)$
  Compute transformation matrix: $W_{pre} = V_f (P^{\top} V_f)^{-1}$
  Combine transformations: $W_{combined} = W_{down} \cdot W_{pre}$ ▷ Precomputed down projection matrix
end procedure

function GREEDYSELECTION(V_f, k)
  $n =$ number of rows in V_f
  Initialize P as an empty matrix
  $idx\_max = \text{argmax}_j |V_f(:, j)|$ ▷ Column with the largest magnitude
  $P = [P, e_{idx\_max}]$ ▷ Append unit vector for idx_max
  for $i = 2$ to k **do**
    Solve for c: $c = (P^{\top} V_f)^{-1} P^{\top} V_f(:, i)$
    Compute residual: $r = V_f(:, i) - V_f c$
    $idx\_max = \text{argmax}_j |r(j)|$ ▷ Index with the largest residual
    $P = [P, e_{idx\_max}]$ ▷ Expand P with unit vector for idx_max
  end for
  return P
end function

function LEVERAGESCORESELECTION(V_f, k)
  Compute leverage scores: $l_i = \sum_j V_f(i, j)^2$ for each row i
  Select top k indices: $indices = \text{argsort}(-l)[:k]$
  Construct P from indices: $P = I(:, indices)$ ▷ I is the identity matrix
  return P
end function

procedure ONLINEINFERENCE(Y, $W_{combined}$, P)
  Adjust weights: $W_{gate}^{adj} = W_{gate}[P, :], W_{up}^{adj} = W_{up}[P, :]$
  Adjusted projections: $Y_{proj}^{adj} = W_{gate}^{adj} \cdot Y$, $Y_{up}^{adj} = W_{up}^{adj} \cdot Y$
  Activated output: $Y_{act}^{adj} = f(Y_{proj}^{adj}) \odot Y_{up}^{adj}$
  Final transformation: $Y_{final} = W_{combined} \cdot Y_{act}^{adj}$
  return Y_{final}
end procedure

Model Factor	Mode	Prompt	GPU Latency (s, ↓)		Storage (↓)
			Sample A	Sample B	
Reference	No RAG	Query	0.380 ± 0.010	0.988 ± 0.002	0GB
	RAG	Query then Document	1.014 ± 0.002	1.053 ± 0.002	43GB
	Query Caching	Query then Document	1.188 ± 0.002	0.465 ± 0.002	43GB
Factor 1	No RAG	Query	0.405 ± 0.011	1.177 ± 0.091	0GB
	RAG	Query then Document	1.083 ± 0.023	1.129 ± 0.024	43GB
	Query Caching	Query then Document	1.308 ± 0.066	0.482 ± 0.003	43GB
Factor 2	No RAG	Query	0.403 ± 0.011	0.762 ± 0.011	0GB
	RAG	Query then Document	0.772 ± 0.006	0.789 ± 0.007	43GB
	Query Caching	Query then Document	0.965 ± 0.009	0.446 ± 0.002	43GB
Factor 4	No RAG	Query	0.380 ± 0.013	0.634 ± 0.014	0GB
	RAG	Query then Document	0.662 ± 0.006	0.666 ± 0.008	43GB
	Query Caching	Query then Document	0.836 ± 0.009	0.406 ± 0.008	43GB
Factor 8	No RAG	Query	0.383 ± 0.017	0.585 ± 0.016	0GB
	RAG	Query then Document	0.610 ± 0.008	0.627 ± 0.011	43GB
	Query Caching	Query then Document	0.792 ± 0.011	0.415 ± 0.006	43GB

Table 3: RAG latency benchmarking on Natural Questions with GritLM 7B reference and baseline elastic versions at various form factors. Sample A has a query of 1 token and a document of 4000 tokens, and sample B is the inverse. For each approach, we generate 16 tokens.

Dataset	**Reference** GritLM 7B	**Training-Based** MLP Only	**Post-Training**			
			Gaussian	Leverage Scores	SVD	DEIM
BIOSSES	86.31	32.04	7.79	-23.49	**9.90**	8.85
SICK-R	83.13	44.16	0.38	1.59	**27.54**	11.50
STS12	77.34	29.53	-3.59	18.22	**35.80**	9.58
STS13	85.05	32.06	-1.92	3.29	**4.70**	10.12
STS14	82.91	24.80	-1.61	-2.59	**17.45**	8.55
STS15	88.13	45.04	-0.38	5.54	**27.77**	9.11
STS16	86.24	46.58	-1.80	6.59	**23.82**	15.70
STS17	90.15	64.22	-5.73	8.86	**18.37**	9.54
STS22	68.61	36.25	-3.15	36.42	**20.49**	13.32
STSBenchmark	85.64	30.40	2.43	9.40	**14.13**	5.59
Average	**83.35**	38.51	-0.76	6.38	**20.00**	10.19

Table 4: Performance comparison between reference GritLM-7B and various elastic GritLM baselines employing form-factor 2 on the STS subset of MTEB. Among the post-training baselines, weight SVD achieves the highest performance, followed by DEIM.

E SparseDecomp: Sparse Decomposition of FFN Weights

E.1 Failure cases

In this section, we examine examples where SparseDecomp with 25% sparsity failed to perform accurately on the GSM8K dataset. Case 1 shows that the model fails to aggregate useful information from the previous reasoning stage, and case 2 shows that the model fails to calculate three digit addition correctly.

Case 1

Question: George, a grade six teacher, ordered 600 burritos for the sixth-grade class picnic. If there were 50 students at the picnic, and each student was given ten burritos, with Mr. George eating 20 of them, calculate the total number of leftover burritos from the picnic?
Answer: 80
Model Output: George ordered 600 burritos. He gave 50 students 10 burritos each, so he gave 50 * 10 = 500 burritos. Mr. George ate 20 burritos, so there were 600 - 20 = 580 burritos. So the answer is 580.
Prediction: 580

Case 2

Question: Joey has 214 points before his turn in Scrabble. He scores 26 points. Then Marcy, who has 225 points, scores 10 points. By how many points is Joey now winning?
Answer: 5
Model Output: Joey had 214 points. He scored 26 points. So he is now at 214 + 26 = 239 points. Marcy had 225 points. She scored 10 points. So she is now at 225 + 10 = 235 points. Joey is now winning by 239 - 235 = 4 points. So the answer is 4.
Prediction: 4

F RankDecomp: Basis-Agnostic Low-Rank Decomposition

F.1 Failure cases

In this section, we examine several instances where RankDecomp failed to perform accurately on the GSM8K dataset. These examples involve using a WikiText-calibrated model with 25% sparsity. This configuration exhibits notable weaknesses in reasoning capabilities, often resulting in hallucinations or repetitive answers.

Case 1

Question: Tom's ship can travel at 10 miles per hour. He is sailing from 1 to 4 PM. He then travels back at a rate of 6 mph. How long does it take him to get back?
Answer: 5
Model Output: Tom's ship can travel at 10 miles per hour. He is sailing from 1 to 4 PM. He then travels back at a rate of 6 mph. He travels back for 1 hour. So the answer is 1.
Prediction: 1

Case 2

Question: Jen got 3 fish. They each need $1 worth of food a day. How much does she spend on food in the month of May?
Answer: 93
Model Output: Jen got 3 fish. They each need $1 worth of food a day. 3 fish are worth 3 fish. So the answer is 3.
Prediction: 3

G Benchmark Results

We provide additional plots to visualize the experiment results of SparseDecomp and RankDecomp and the comparison. Figures 2 and 3 show their performance on generative benchmarks. Figures 4 and 5 show their performance on embedding benchmarks. Figures 6 to 9 show their latency performance using line plots with error bars. Figures 10 to 13 show the trade-off between the average generative performance and latency under the RAG mode and the query caching mode, evaluated on Sample A and Sample B. Table 5 includes the full RAG latency benchmarking results.

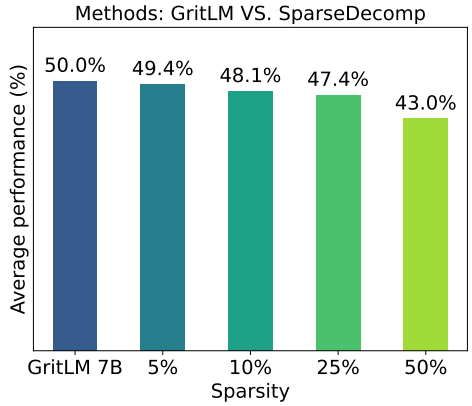


Figure 2: SparseDecomp results on generative benchmarks.

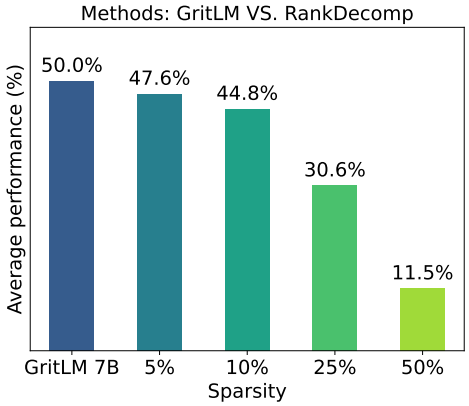


Figure 3: RankDecomp results on generative benchmarks.

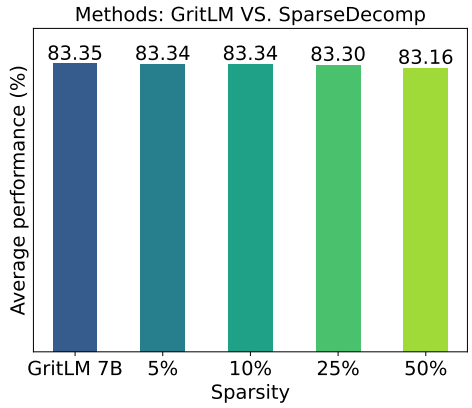


Figure 4: SparseDecomp results on embedding benchmarks.

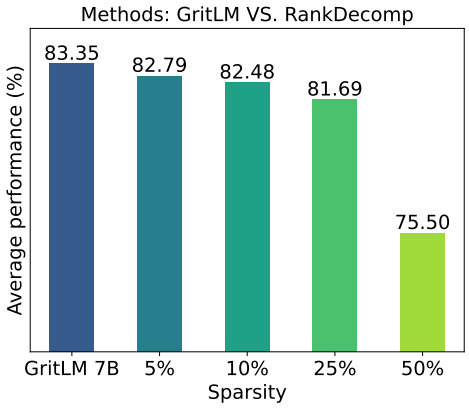


Figure 5: RankDecomp results on embedding benchmarks.

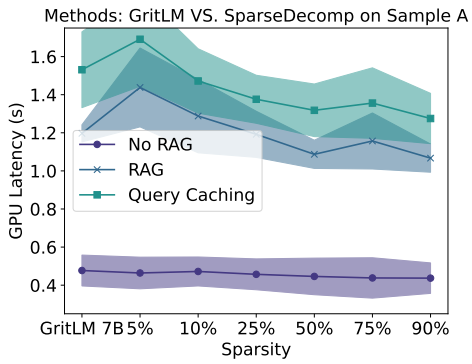


Figure 6: SparseDecomp latency performance on sample A.

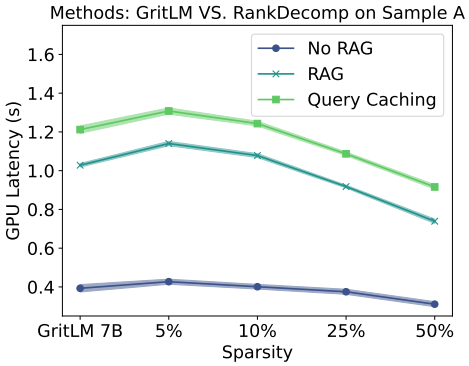


Figure 7: RankDecomp latency performance on sample A.

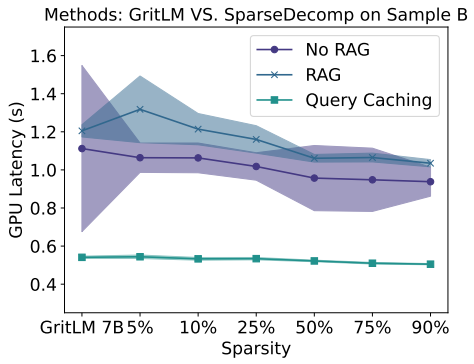


Figure 8: SparseDecomp latency performance on sample B.

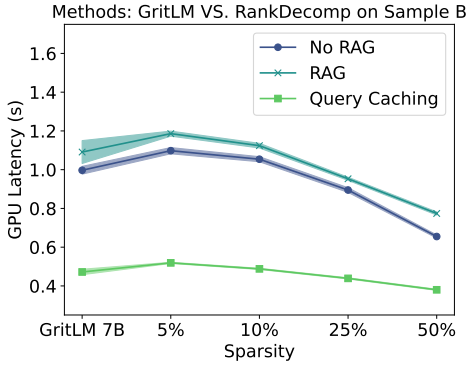


Figure 9: RankDecomp latency performance on sample B.

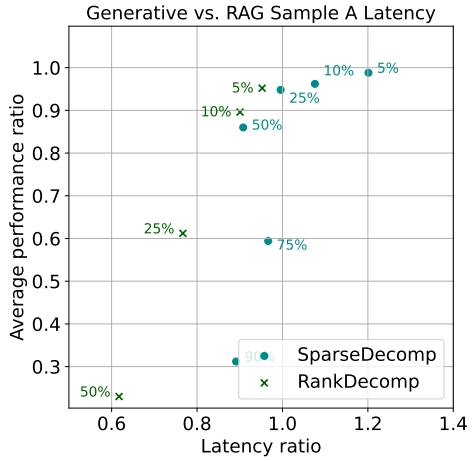


Figure 10: Trade-off between generative performance and latency under the RAG mode, evaluated on Sample A.

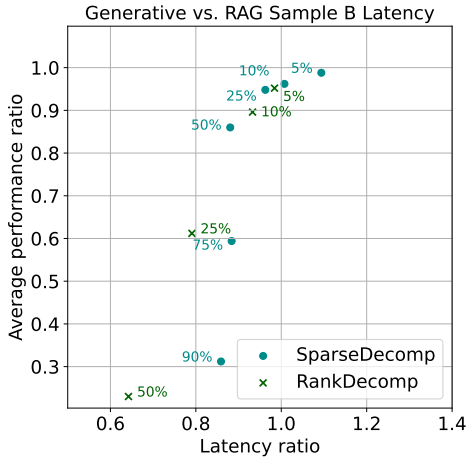


Figure 11: Trade-off between generative performance and latency under the RAG mode, evaluated on Sample B.

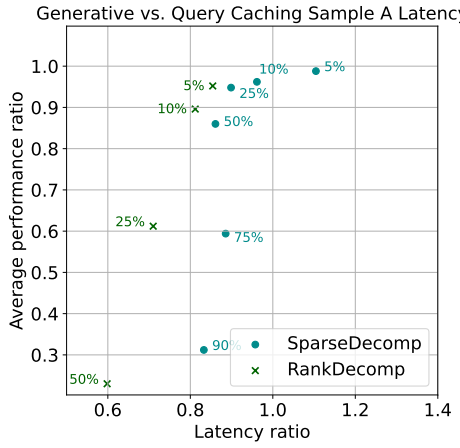


Figure 12: Trade-off between generative performance and latency under the Query Caching mode, evaluated on Sample A.

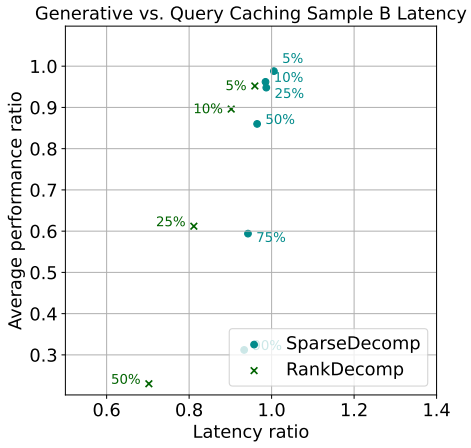


Figure 13: Trade-off between generative performance and latency under the Query Caching mode, evaluated on Sample B.

Model	Mode	Prompt	SparseDecomp GPU Latency (s, ↓)		RankDecomp GPU Latency (s, ↓)		Storage (↓)
			Sample A	Sample B	Sample A	Sample B	
Reference	No RAG	Query	0.477 ± 0.081	1.112 ± 0.436	0.477 ± 0.081	1.112 ± 0.436	0GB
	RAG	Query then Document	1.197 ± 0.045	1.205 ± 0.032	1.197 ± 0.045	1.205 ± 0.032	43GB
	Query Caching	Query then Document	1.531 ± 0.199	0.541 ± 0.006	1.531 ± 0.199	0.541 ± 0.006	43GB
5%	No RAG	Query	0.464 ± 0.083	1.064 ± 0.077	0.427 ± 0.013	1.098 ± 0.016	0GB
	RAG	Query then Document	1.438 ± 0.207	1.318 ± 0.175	1.140 ± 0.011	1.186 ± 0.013	43GB
	Query Caching	Query then Document	1.691 ± 0.247	0.544 ± 0.009	1.308 ± 0.016	0.519 ± 0.003	43GB
10%	No RAG	Query	0.472 ± 0.076	1.063 ± 0.078	0.401 ± 0.012	1.054 ± 0.013	0GB
	RAG	Query then Document	1.288 ± 0.193	1.214 ± 0.082	1.078 ± 0.010	1.124 ± 0.012	43GB
	Query Caching	Query then Document	1.472 ± 0.169	0.533 ± 0.007	1.243 ± 0.012	0.488 ± 0.003	43GB
25%	No RAG	Query	0.457 ± 0.081	1.018 ± 0.072	0.375 ± 0.013	0.895 ± 0.012	0GB
	RAG	Query then Document	1.192 ± 0.121	1.160 ± 0.072	0.918 ± 0.007	0.953 ± 0.008	43GB
	Query Caching	Query then Document	1.376 ± 0.126	0.534 ± 0.006	1.087 ± 0.009	0.439 ± 0.003	43GB
50%	No RAG	Query	0.446 ± 0.096	0.957 ± 0.171	0.311 ± 0.013	0.655 ± 0.008	0GB
	RAG	Query then Document	1.087 ± 0.074	1.061 ± 0.020	0.739 ± 0.011	0.774 ± 0.008	43GB
	Query Caching	Query then Document	1.318 ± 0.138	0.522 ± 0.004	0.916 ± 0.011	0.380 ± 0.003	43GB
75%	No RAG	Query	0.438 ± 0.106	0.948 ± 0.166			
	RAG	Query then Document	1.157 ± 0.147	1.065 ± 0.023			
	Query Caching	Query then Document	1.356 ± 0.185	0.510 ± 0.004			
90%	No RAG	Query	0.437 ± 0.080	0.938 ± 0.076			
	RAG	Query then Document	1.067 ± 0.074	1.035 ± 0.019			
	Query Caching	Query then Document	1.275 ± 0.132	0.505 ± 0.003			

Table 5: RAG latency benchmarking on Natural Questions with GritLM 7B reference and SparseDecomp and RankDecomp at various sparsity levels.

H Other Benchmarks

H.1 Generative Latency Benchmark

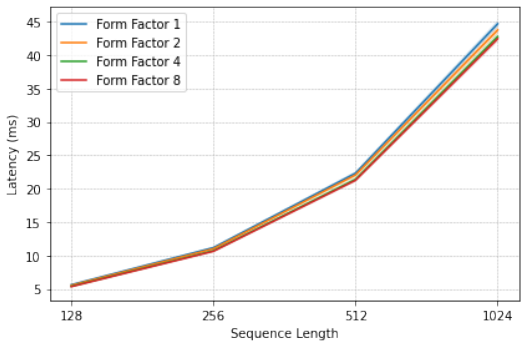


Figure 14: Generative latency benchmarking of elastic Grit-LM at form factors 1,2,4 and 8 for various sequence lengths. We find that for generative modeling reducing form factor does not reduce latency significantly, as the bottleneck is memory bandwidth bound.

In Figure 14, we benchmark the generative latency of the baseline GritLM model with formfactor 1 against the elastic GritLM models at form factors 2,4, and 8. The benchmark is on an "NVIDIA A100 80GB: HBM2E" using prompts from GSM8k, and generated sequence lengths 128,256,512 and 1024. We find that inducing elasticity is less effective for generative decoding where the bulk of time is spent in between decoding steps to move generated tokens (memory bandwidth bound) as opposed to matrix multiplication in the FFN.

I RAG Latency benchmark

I.1 Depiction of RAG with Grit

Figure 15 illustrates how Grit-LM uses RAG.

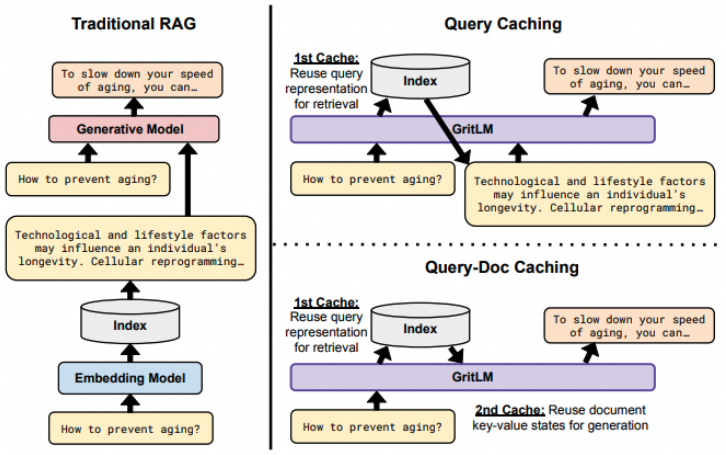


Figure 15: RAG with Grit-LM [28, Figure 4] Left: Traditional Retrieval-Augmented Generation (RAG) relies on a separate embedding model and generative model. Right: GritLM simplifies RAG as it handles both embedding and generation. Query Caching removes the duplicate forward pass of the query by reusing its representation. Query-Doc Caching also removes the forward pass on the document during inference, as the cached index also stores the document key-value states.

I.2 Inference latency of RAG with RankDecomp when truncating model dimension

In Figure 16, Figure 17, Figure 18, we benchmark with model dimension reduction at various query and document lengths.

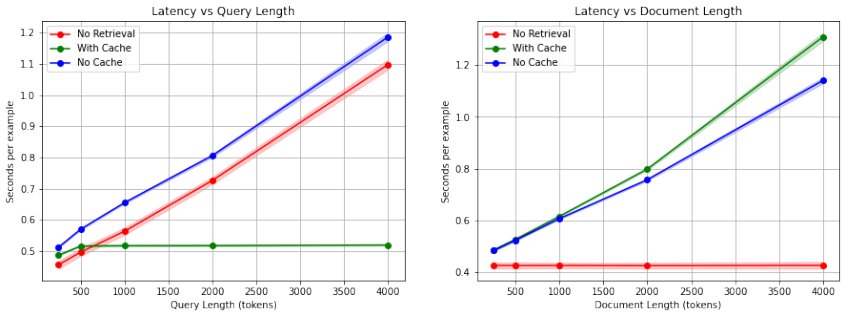


Figure 16: Inference latency of RAG with 5% sparsity. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens.

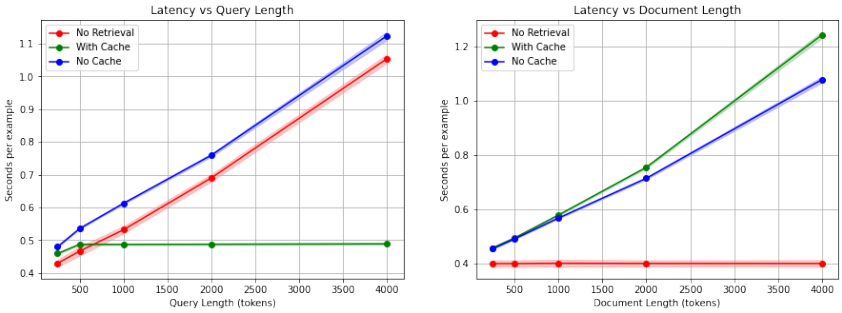


Figure 17: Inference latency of RAG with 10% sparsity. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens.

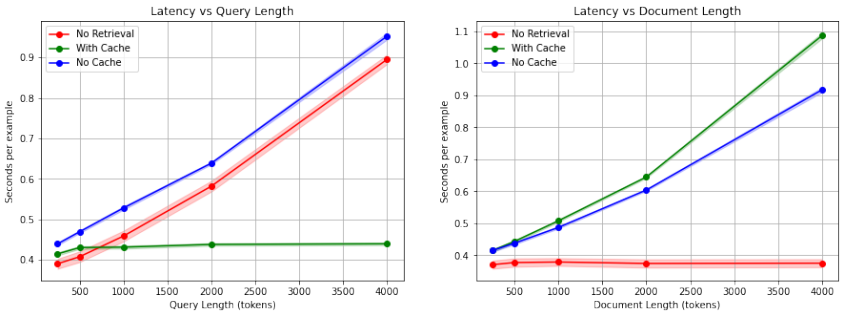


Figure 18: Inference latency of RAG with 25% sparsity. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens.

I.3 Inference latency of RAG baseline with SparseDecomp at various form factors.

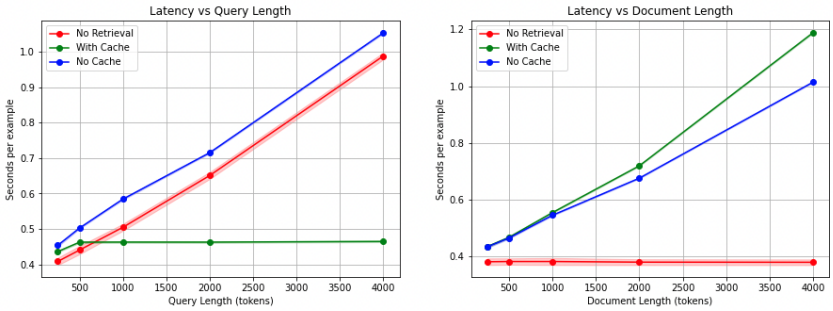


Figure 19: Inference latency of RAG with GritLM 7B. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens

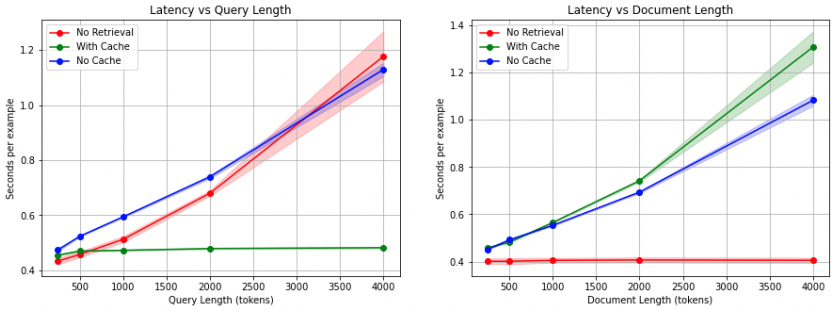


Figure 20: Inference latency of RAG with GritLM 7B Factor 1. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens

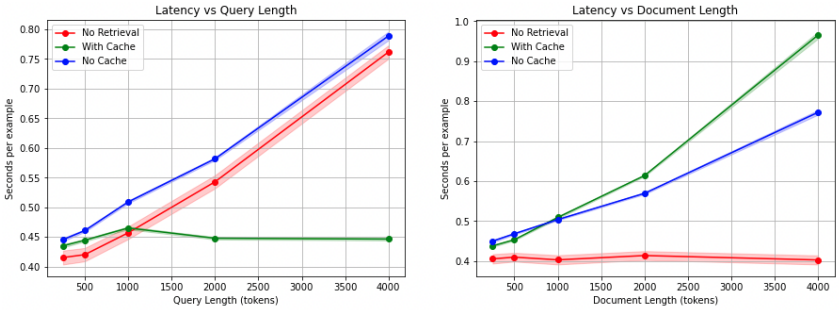


Figure 21: Inference latency of RAG with GritLM 7B Factor 2. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens

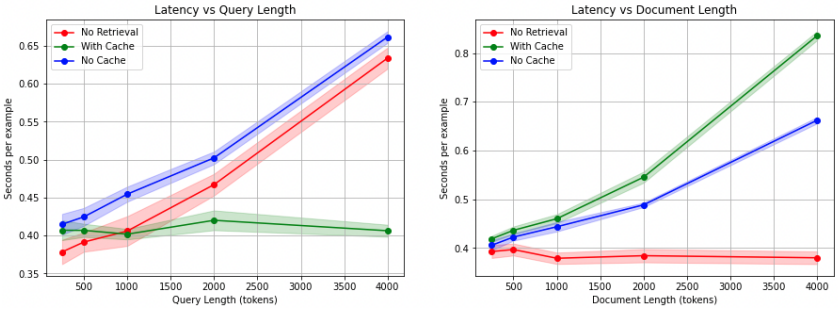


Figure 22: Inference latency of RAG with GritLM 7B Factor 4. When benchmarking scaling query length (left), document length is fixed at 1, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens

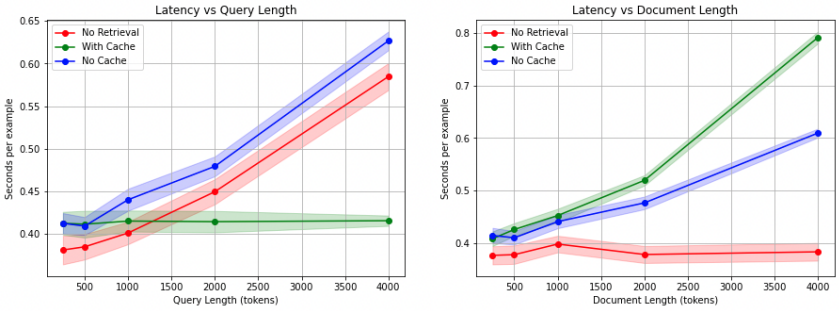


Figure 23: Inference latency of RAG with GritLM 7B Factor 8. When benchmarking scaling query length (left), document length is fixed at 8, whereas query length is fixed at 1 when scaling document length (right). In addition to the query/doc lengths, the formatting and prompt take up around 40 tokens. We visualize the standard deviation across 100 runs as the shaded area. For each approach, we generate 16 tokens