DYNSVD: DYNAMIC SVD VIA TOKEN-ADAPTIVE BASIS SELECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing Singular Value Decomposition (SVD)-based compression methods construct static low-rank bases that are fixed across all inputs, even though the contribution of each basis depends jointly on its singular value and the input token. As a result, singular directions with small singular values, which can be highly activated for specific inputs, are permanently discarded by static truncation. To address this issue, we propose **DynSVD**, a training-free, token-adaptive inference framework that selects bases dynamically on top of static SVD. We prove that dynamic selection is optimal for each token and that its error adds to the static truncation error without cross terms, which lets DynSVD complement static SVD methods. Experiments show that, at the same computation budget, DynSVD lowers perplexity by 42% on average relative to static SVD. Within a fixed GPU memory budget, one stored DynSVD model offers a flexible, continuous range of operating points through a single threshold, and with custom Triton kernels it yields up to $1.10\times$ higher decoding throughput at the same perplexity and up to 27% lower perplexity at the same throughput than static SVD.

1 INTRODUCTION

Scaling large language models (LLMs) [18] has led to strong performance across diverse tasks [1, 4, 41, 50], but sharply increases parameter counts and inference cost [6], making model compression [5] essential for practical deployment. During autoregressive decoding, the model generates one token at a time and repeatedly accesses its weights [11]. The cost of each step accumulates over long outputs, including reasoning traces [13, 40], and can be especially restrictive on devices with limited memory bandwidth [21]. This cost can be reduced in two ways: adaptive inference reads less of the model for each token [26, 28], and weight compression makes the model itself smaller [8, 25].

Adaptive inference lets each token use only part of the model. Mixture-of-experts models route each token to a few experts [38, 7], and activation sparsity methods skip weights whose inputs are small [26, 28, 51]. Because the full model stays in memory, these methods reduce the per-token cost but not the storage. This matters most on personal devices, where both budgets are tight: memory is limited [2], and the compute available to the model changes over time because the hardware is shared with other applications [48].

Because decoding is bound by memory bandwidth, reading fewer weight bytes directly speeds it up [11]. Weight compression achieves this by storing fewer or lower-precision parameters, mainly through quantization and pruning. Quantization, though, depends on hardware support for low-precision formats [8, 19, 37, 45], and pruning faces a trade-off between efficiency and accuracy: unstructured sparsity is hard to accelerate on parallel hardware [14, 31], while structured pruning loses accuracy [3, 9, 29].

Low-rank decomposition offers an alternative paradigm that factorizes each weight matrix into two smaller factors through SVD [17, 20]. It runs on standard dense matrix multiplications [22] and can be combined with quantization or channel pruning. Recent methods improve its accuracy by folding calibration statistics into the weights before truncating them once [15, 49, 44, 43]. The result is a fixed SVD factorization whose factors every token uses in full.

However, these static SVD methods still lose much accuracy at high compression. We attribute this persistent performance gap to a fundamental property of LLMs: their pretrained weight matrices are

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

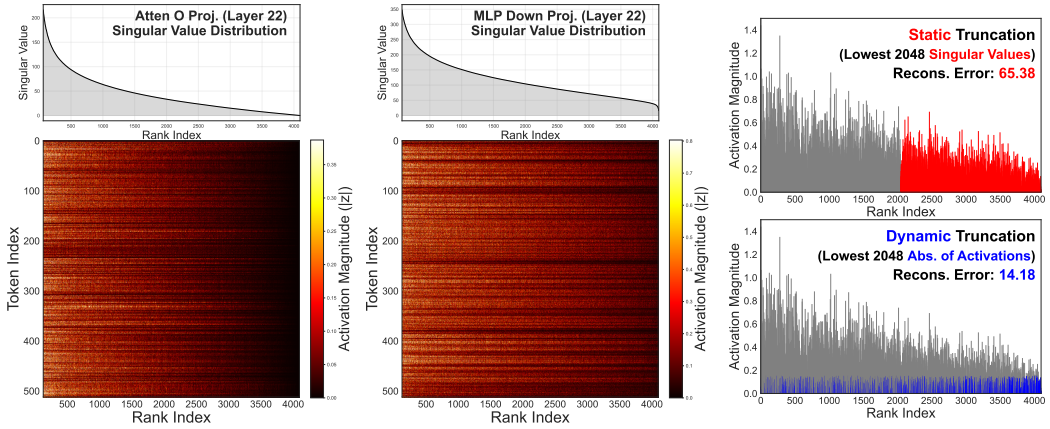

Figure 1: Token-wise intermediate activation magnitudes across ranks on LLaMA-2-7B after SVD-LLM whitening. **Left/Center:** Heatmaps for attention and MLP layers show that (i) different tokens activate different sets of bases and (ii) tokens can strongly activate bases with small singular values. **Right:** Dynamic truncation by activation magnitude has lower reconstruction error than static truncation by singular value at the same active rank (more views in Appendix E).

inherently high-rank [34, 39]. Empirically, LLM weight matrices exhibit heavy-tailed singular value spectra even after activation-aware whitening [44] (see Figure 1 top), indicating that task-relevant information is distributed across a broad range of singular directions rather than being concentrated in a small number of dominant components [33]. As a result, *static* low-rank approximations obtained by truncating a *fixed* set of singular vectors inevitably discard a significant amount of valid information, imposing an intrinsic limit on the achievable accuracy of low-rank compression methods.

At the same time, high-rank weights do not imply that *all singular directions are equally important for every input token*. Each token strongly activates a different set of singular directions, so the directions that matter depend on the input (Figure 1). Directions with small singular values can still be strongly activated for particular tokens, and spectral ordering alone cannot tell which ones to discard. Consequently, while LLMs are high-rank in aggregate, the directions each token needs vary, revealing an opportunity for dynamic low-rank computation at inference.

Dynamic selection over a full-rank decomposition alone provides no compression, since all r components must still be stored and projected. Singular components instead split into three roles, with top components activated universally, the long tail contributing negligibly, and an intermediate band exhibiting the token-dependent activation of Figure 1. This motivates pairing *static truncation* of the negligible tail with *dynamic selection* within the retained subspace, achieving both compression and input-adaptive computation.

Building on these insights, we propose **DynSVD**, a token-adaptive inference framework that selects bases dynamically on top of static SVD compression. Our contributions are threefold:

1. **Token-adaptive inference on static SVD.** DynSVD pairs static low-rank truncation with per-token basis selection driven by intermediate activation magnitudes, so that a compressed model adapts to each token without any training. This selection provably minimizes the per-token error, and the dynamic selection error adds independently to the static truncation error.
2. **Runtime flexibility.** A single selection parameter, r_d for exact top-r_d or θ for the threshold rule, sets how many parameters each token reads and can be changed at run time without recompression. With the shared threshold, the budget also moves to the layers that need it.
3. **Performance and throughput gains.** At the same number of active parameters, DynSVD lowers perplexity by 42% on average relative to static SVD. Within a fixed GPU memory budget, one stored model serves several speed and quality trade-offs, and with custom Triton kernels it reaches up to $1.10\times$ higher batch-one decoding throughput at the same perplexity and up to 27% lower perplexity at the same throughput than static SVD.

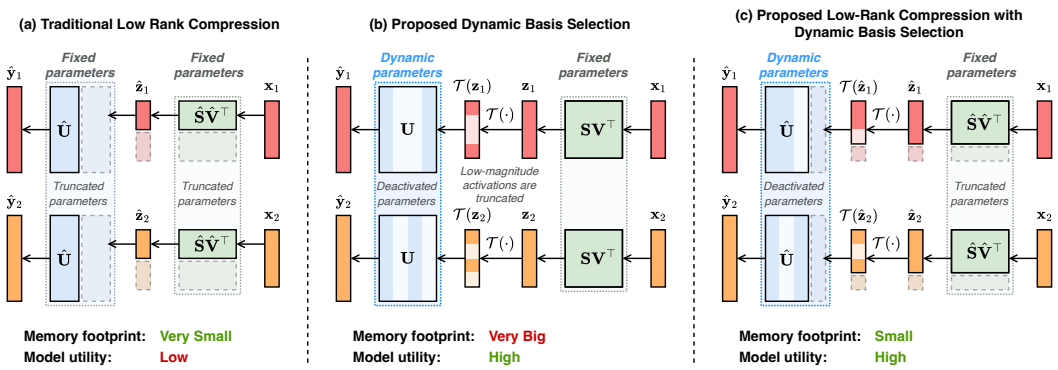

Figure 2: Comparison of decomposition strategies. (a) Traditional low-rank compression uses a fixed truncated subspace. (b) Full-rank with dynamic selection adapts to each input but increases storage. (c) Our approach combines low-rank truncation with dynamic selection, achieving both compression and input-adaptive computation.

2 Related Work

Post-Training SVD Compression. Post-training SVD methods improve upon vanilla weight truncation by incorporating activation information. FWSVD [15] weighs parameters by Fisher information, ASVD [49] rescales weights using input-channel statistics, and Basis Sharing [42] shares bases across layers. SoLA [16] preserves high-activation FFN channels and applies SVD to the remainder, while AdaSVD [23] allocates per-layer retention ratios via input–output cosine similarity. SVD-LLM [44] introduces truncation-aware data whitening that ties reconstruction loss to the discarded singular values, and Dobi-SVD [43] applies incremental PCA directly to activation matrices. Swift-SVD [35] obtains the truncation in closed form from the output activation covariance.

Despite these advances, all prior SVD compression methods produce static compressed weights: the truncated bases are discarded permanently, applying the same reduced subspace to every token regardless of input-specific relevance. Notably, DynSVD is orthogonal to these methods and can be applied on top of the factors they produce.

Adaptive Inference. Adaptive inference methods lower the cost of each token by computing only part of each layer. Deja Vu [26] trains small predictors that choose the attention heads and MLP neurons for each input. Without any training, TEAL [28] skips the weight channels whose inputs are small, and R-Sparse [51] applies the dense weight only to the large inputs and sends the remaining small inputs through fixed low-rank factors. All of them keep the dense weights in memory, so they lower the per-token cost but not the storage.

DynSVD brings adaptive inference to SVD-compressed models: it keeps the truncated factors of a static SVD method and chooses, for each token, which of the stored bases to use. Its storage is set by the number of bases it keeps and stays below that of the dense model, while selection further reduces the parameters read per token. For this reason, we compare DynSVD with static SVD at the same per-token compute rather than with adaptive inference on dense models.

3 Method

3.1 Preliminary

Consider a linear layer with weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$, assuming full rank $r = \min(m, n)$:

$$\mathbf{W} = \mathbf{U}\mathbf{S}\mathbf{V}^\top = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top, \tag{1}$$

with $\sigma_1 \geq \cdots \geq \sigma_r > 0$. Truncating to the top-$r_{\text{static}}$ singular values yields $\hat{\mathbf{W}} = \hat{\mathbf{U}}\hat{\mathbf{S}}\hat{\mathbf{V}}^\top$, the best low-rank approximation by the Eckart–Young–Mirsky theorem (Figure 2-(a)). Writing $\mathbf{W} = \mathbf{L}\mathbf{R}$

with $\mathbf{L} = \mathbf{U}$ and $\mathbf{R} = \mathbf{S}\mathbf{V}^\top$, the layer output for input $\mathbf{x} \in \mathbb{R}^n$ decomposes as $\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{L}\mathbf{z}$ via the intermediate activation $\mathbf{z} = \mathbf{R}\mathbf{x} \in \mathbb{R}^r$, with

$$z_i = \sigma_i \mathbf{v}_i^\top \mathbf{x}, \tag{2}$$

combining the singular value magnitude with the input alignment along the i-th right singular vector.

3.2 Token-Adaptive Basis Selection

We empirically observe that tokens often strongly activate bases with small singular values (Figure 1), at times exceeding those retained under standard truncation, which indicates that the optimal basis subset varies per input.

As shown in Figure 2-(b), we propose to select, for each input, the r_d components with the largest activation magnitudes. Let $\mathcal{T}_{r_d} : \mathbb{R}^r \to \mathbb{R}^r$ retain the top-r_d entries by absolute value and zero out the rest. Given $\mathbf{W} = \mathbf{L}\mathbf{R}$ with $\mathbf{L} = \mathbf{U}$ and $\mathbf{R} = \mathbf{S}\mathbf{V}^\top$, the layer output under dynamic selection is

$$\hat{\mathbf{y}} = \mathbf{L}\,\mathcal{T}_{r_d}(\mathbf{R}\mathbf{x}). \tag{3}$$

The following theorem establishes that this rule minimizes layer-output reconstruction error when $\mathbf{L}$ has orthonormal columns.

Theorem 1 (Optimal Basis Selection via Intermediate Activation Magnitude)**.** *Let $\mathbf{L} \in \mathbb{R}^{m\times r}$ satisfy $\mathbf{L}^\top\mathbf{L} = \mathbf{I}_r$. For any input $\mathbf{x}$ and target r_d, the reconstruction error $\|\mathbf{L}\mathbf{z} - \mathbf{L}\mathcal{T}_{r_d}(\mathbf{z})\|_2^2$ subject to $\|\mathcal{T}_{r_d}(\mathbf{z})\|_0 = r_d$ is minimized when $\mathcal{T}_{r_d}$ retains the r_d entries of $\mathbf{z}$ with the largest absolute values.*

The proof is provided in Appendix D.1. The orthonormality condition holds naturally for $\mathbf{L} = \mathbf{U}$, and is preserved under the whitening transformations used in SVD-LLM, making Theorem 1 directly applicable to whitened decompositions.

3.3 Low-Rank Compression with Dynamic Selection

While dynamic selection over a full-rank decomposition provides input-adaptive computation, it offers no compression. Let $r' \le r = \min(m, n)$ denote the chosen factorization rank: rank-r' factorization stores $r'(m + n)$ parameters, which exceeds the original mn unless $r' < mn/(m + n)$ (e.g., $r' < 0.5r$ for square attention layers and $r' < 0.8r$ for $4\times$ MLP layers). Moreover, computing $\mathbf{z} = \mathbf{R}\mathbf{x}$ over all r dimensions before selection eliminates any compute savings.

As shown in Figure 2-(c), we address both issues by combining dynamic selection with low-rank truncation. The key insight is that singular components occupy distinct roles: those with large singular values are universally important across inputs, those with very small singular values contribute negligibly regardless of input alignment, and those in the intermediate range exhibit token-dependent activation. Static truncation to rank r_s removes negligible components, while dynamic selection within the retained r_s adapts to each input.

Let $\hat{\mathbf{L}} = \mathbf{U}_{r_s} \in \mathbb{R}^{m\times r_s}$ and $\hat{\mathbf{R}} = \mathbf{S}_{r_s}\mathbf{V}_{r_s}^\top \in \mathbb{R}^{r_s\times n}$ denote the rank-r_s truncated factors. The layer output under combined truncation and dynamic selection is

$$\hat{\mathbf{y}} = \hat{\mathbf{L}}\,\mathcal{T}_{r_d}(\hat{\mathbf{R}}\mathbf{x}), \tag{4}$$

with $\hat{\mathbf{z}} = \hat{\mathbf{R}}\mathbf{x} \in \mathbb{R}^{r_s}$ and $r_d < r_s$. Storage reduces to $r_s(m + n)$, with projection $\mathcal{O}(nr_s)$ and output reconstruction $\mathcal{O}(mr_d)$, so decoding one token reads $nr_s + mr_d$ parameters, which we count as its active parameters. The reconstruction error decomposes into two independent terms.

Theorem 2 (Decomposition of Reconstruction Error in Dynamic SVD)**.** *Let $\mathbf{W} = \mathbf{L}\mathbf{R}$ with $\mathbf{L}^\top\mathbf{L} = \mathbf{I}_r$, and let $\hat{\mathbf{L}}\hat{\mathbf{R}}$ be the rank-r_s truncation. For any $\mathcal{T}_{r_d}$ that keeps a subset of the entries of $\hat{\mathbf{z}}$ and sets the rest to zero,*

$$\|\mathbf{W}\mathbf{x} - \hat{\mathbf{L}}\mathcal{T}_{r_d}(\hat{\mathbf{z}})\|_2^2 = \underbrace{\|(\mathbf{W} - \hat{\mathbf{L}}\hat{\mathbf{R}})\mathbf{x}\|_2^2}_{\textit{truncation error}} + \underbrace{\|\hat{\mathbf{L}}(\hat{\mathbf{z}} - \mathcal{T}_{r_d}(\hat{\mathbf{z}}))\|_2^2}_{\textit{selection error}}. \tag{5}$$

The proof is provided in Appendix D.2. Since $\hat{\mathbf{L}}^\top\hat{\mathbf{L}} = \mathbf{I}_{r_s}$, Theorem 1 applies to $\hat{\mathbf{L}}$ and the magnitude-based top-r_d rule remains optimal in the truncated case. More importantly, the two error terms act on

disjoint index sets and can be reduced independently: whitening from SVD-LLM reshapes the input distribution to control the truncation error in expectation over the calibration set, whereas dynamic selection controls the selection error per token. A small singular value indicates that a direction is unimportant on average, but does not preclude strong activation by individual tokens, as illustrated in Figure 1 and Appendix E. Whitening and dynamic selection are therefore orthogonal mechanisms that compose naturally.

The two terms meet only through the budget. At a fixed per-token cost $nr_s + mr_d$, a larger r_s lowers the truncation error but leaves a smaller share of the stored components for each token. This is why DynSVD stores more components than each token uses ($r_s > r_d$). We write $r_d = \alpha \cdot r_s$, where the ratio α sets how many of the r_s stored components each token uses, and choose r_s so that $nr_s + mr_d$ meets the target active parameter budget.

3.4 From Exact Top-r_d to Threshold-Based Selection

Exact top-r_d selection is well-defined and uses exactly the target number of active parameters. However, it has two limitations: (i) every layer and module keeps the same fraction α of its components, although layers, modules, and even tokens differ in how their activation magnitudes spread over the components (Figure 1). A single fixed ratio therefore gives some layers more components than they need and others fewer. (ii) Top-r_d selection is not hardware friendly. It needs a sort or a selection over all of $\hat{\mathbf{z}}$ before any row of $\hat{\mathbf{L}}$ can be read, and such selection is costly on GPUs [46].

We therefore propose a threshold operator $\mathcal{T}_\theta(\cdot)$ that addresses both limitations. It keeps a component when its magnitude is large relative to the token's own average,

$$\mathcal{T}_\theta(\hat{\mathbf{z}})_i = \begin{cases} \hat{z}_i & \text{if } |\hat{z}_i| \geq \theta \cdot \frac{1}{r_s} \sum_{j=1}^{r_s} |\hat{z}_j|, \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

where the mean is computed on the fly for each token and layer. The kept entries are always the largest ones, so Theorem 1 still applies to each token, now with a count that depends on the token.

Algorithm 1 DynSVD with thresholding

Require: weights $\{\mathbf{W}_l\}$, calibration inputs $\{\mathbf{X}_l\}$, ratio $\alpha = r_d/r_s$, budgets $\{P_l\}$

[Offline]

1: **for** each layer l **do**
2: $\quad \mathbf{C} \leftarrow \texttt{Cholesky}(\mathbf{X}_l \mathbf{X}_l^\top)$
3: $\quad \mathbf{U}\mathbf{S}\mathbf{V}^\top \leftarrow \texttt{SVD}(\mathbf{W}_l \mathbf{C})$
4: $\quad r_s \leftarrow \lfloor P_l/(n_l + \alpha m_l) \rfloor$
5: $\quad r_d \leftarrow \lfloor \alpha\, r_s \rfloor$
6: $\quad \hat{\mathbf{L}}_l \leftarrow \mathbf{U}_{:,1:r_s}$, $\hat{\mathbf{R}}_l \leftarrow \mathbf{S}_{1:r_s} \mathbf{V}_{:,1:r_s}^\top \mathbf{C}^{-1}$
7: **end for**
8: Pick θ with $\sum_l m_l\, \mathbb{E}[k_l(\theta)] \approx \sum_l m_l\, r_{d,l}$

[Decoding] (each token $\mathbf{x}$, each layer)

9: $\hat{\mathbf{z}} \leftarrow \hat{\mathbf{R}}\mathbf{x}$
10: $\mathcal{S} \leftarrow \{\, i : |\hat{z}_i| \geq \theta \cdot \text{mean}_j |\hat{z}_j| \,\}$
11: **return** $\hat{\mathbf{L}}_{:,\mathcal{S}}\, \hat{\mathbf{z}}_{\mathcal{S}}$ ▷ reads only the columns in $\mathcal{S}$

Algorithm 1 summarizes the procedure, where lines 2, 3, and 6 follow the whitened truncation of SVD-LLM [44]. To set θ, we run the model once on calibration tokens and collect, for every layer, each $|\hat{z}_i|$ divided by its token's mean. For any θ, these stored values give the average number of components that each layer keeps, $\mathbb{E}[k_l(\theta)]$. We pick the θ at which the parameters read per token match those of exact top-r_d, $\sum_l m_l\, \mathbb{E}[k_l(\theta)] \approx \sum_l m_l\, r_{d,l}$. A new budget at run time only needs a new θ, read from the same stored values. We share one θ across all layers instead of calibrating one per layer. Per-layer thresholding $\mathcal{T}_{\theta_l}(\cdot)$, with a separate θ_l for each layer l, would hold each layer near its own count $r_{d,l}$, whereas a shared θ fixes only the total budget and lets layers whose magnitudes are spread out keep more components and concentrated layers keep fewer. Section 4.3 studies these selection rules in detail through an ablation.

3.5 Custom Triton Kernel for Decode-Time Inference

Theorem 1 guarantees that only the selected components among the r_s retained ones need to participate in the output reduction, but realizing this saving on the GPU is nontrivial, because the selected columns change from token to token and the product is no longer a plain GEMV (or GEMM). Therefore, we design custom Triton kernels for DynSVD that select components dynamically at decode time. The first computes $\hat{\mathbf{z}} = \hat{\mathbf{R}}\mathbf{x}$, and the second reads only the selected columns of $\hat{\mathbf{L}}$, so unselected columns cost no memory traffic. The kernels provide separate paths for the threshold $\mathcal{T}_\theta(\cdot)$ and for exact top-r_d $\mathcal{T}_{r_d}(\cdot)$. Implementation details are given in Appendix F.

Table 1: Compression results on LLaMA-7B. We report WikiText-2 and C4 perplexity (lower is better) and zero-shot accuracy on five reasoning benchmarks (higher is better). Act./Total denotes active and total parameters in billions, and for DynSVD-T the active count is an average. For fair comparison, SVD-LLM and Dobi-SVD results are reported without finetuning or remapping. DynSVD-K and DynSVD-T are built on SVD-LLM, and DynSVD†-T on Dobi-SVD.

Ratio	Act. / Total	Method	Wiki2↓	C4↓	ARC_e↑	HellaS.↑	OpenB.↑	PIQA↑	WinoG.↑	Avg.↑
–	6.48B / 6.48B	Original	5.68	7.34	0.67	0.56	0.28	0.78	0.67	0.59
20%	5.18B / 5.18B	FWSVD	1727	1511	0.11	0.08	0.09	0.10	0.05	0.09
		ASVD	11.14	15.93	0.53	0.41	0.25	0.68	0.64	0.50
		SoLA	12.56	21.65	0.57	0.37	0.24	0.65	0.62	0.49
		Basis Sharing	7.88	14.98	0.65	0.45	0.28	0.72	0.66	0.55
		SVD-LLM	7.91	16.11	0.62	0.43	0.25	0.69	0.66	0.53
		Dobi-SVD	8.54	**10.01**	0.58	0.43	0.22	0.70	0.63	0.51
		Swift-SVD	7.97	16.43	0.64	0.43	0.26	0.69	0.66	0.54
	5.18B / 6.14B	**DynSVD-K**	7.04	11.81	0.67	0.49	0.28	0.72	0.68	**0.57**
	5.20B / 6.14B	**DynSVD-T**	6.99	11.60	0.68	0.49	0.29	0.72	0.69	**0.57**
	5.20B / 6.14B	**DynSVD†-T**	**6.97**	11.53	0.68	0.49	0.30	0.72	0.68	**0.57**
40%	3.88B / 3.88B	FWSVD	18156	12847	0.05	0.01	0.06	0.05	0.02	0.04
		ASVD	1407	1109	0.05	0.08	0.06	0.05	0.02	0.05
		SoLA	34.28	99.28	0.36	0.29	0.15	0.56	0.53	0.38
		Basis Sharing	12.79	43.97	0.51	0.34	0.23	0.62	0.62	0.47
		SVD-LLM	13.56	54.59	0.45	0.32	0.19	0.60	0.57	0.43
		Dobi-SVD	13.54	23.54	0.41	0.34	0.22	0.61	0.58	0.43
		Swift-SVD	13.86	55.03	0.47	0.32	0.20	0.61	0.58	0.44
	3.88B / 5.44B	**DynSVD-K**	10.16	24.75	0.50	0.38	0.23	0.63	0.60	0.47
	3.91B / 5.44B	**DynSVD-T**	9.45	20.72	0.54	0.40	0.25	0.64	0.60	0.48
	3.91B / 5.44B	**DynSVD†-T**	**9.41**	**20.64**	0.54	0.40	0.23	0.65	0.61	**0.49**
60%	2.59B / 2.59B	FWSVD	32194	29292	0.02	0.01	0.06	0.02	0.01	0.02
		ASVD	57057	43036	0.04	0.09	0.05	0.08	0.06	0.06
		Basis Sharing	53.64	336	0.32	0.28	0.15	0.55	0.54	**0.37**
		SVD-LLM	59.96	382	0.30	0.27	0.12	0.56	0.50	0.35
		Dobi-SVD	46.18	191	0.31	0.28	0.15	0.54	0.52	0.36
		Swift-SVD	64.08	325	0.29	0.27	0.14	0.55	0.51	0.35
	2.59B / 3.63B	**DynSVD-K**	32.22	167	0.32	0.28	0.15	0.55	0.53	**0.37**
	2.60B / 3.63B	**DynSVD-T**	28.27	142	0.34	0.29	0.15	0.57	0.52	**0.37**
	2.60B / 3.62B	**DynSVD†-T**	**26.82**	**130**	0.35	0.29	0.14	0.56	0.53	**0.37**
80%	1.29B / 1.29B	FWSVD	687291	708243	0.01	0.01	0.00	0.01	0.02	0.01
		ASVD	80425	67927	0.03	0.02	0.04	0.01	0.03	0.03
		Basis Sharing	532	2763	0.26	0.26	0.14	0.52	0.51	**0.34**
		SVD-LLM	673	2822	0.26	0.26	0.17	0.52	0.50	**0.34**
		Swift-SVD	6665	9327	0.27	0.26	0.14	0.52	0.49	0.33
	1.29B / 1.81B	**DynSVD-K**	362	1338	0.25	0.26	0.19	0.53	0.50	**0.34**
	1.30B / 1.81B	**DynSVD-T**	**331**	**1257**	0.25	0.26	0.17	0.53	0.50	**0.34**

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Models and Datasets. We evaluate on LLaMA families [41] with scalability studies and additional results on more recent architectures (e.g., LLaMA-3.1-8B [12], Qwen3-8B [47], OLMoE-A1B-7B [32]). Language modeling is measured by perplexity on WikiText-2 [30] and C4 [36], and downstream zero-shot tasks via the lm-evaluation-harness protocol [10]. Multimodal generality is assessed on LLaVA-1.5-7B [27] with visual question answering benchmarks.

Baselines and Implementation. We compare DynSVD against representative post-training SVD baselines: FWSVD [15], ASVD [49], SVD-LLM [44], Dobi-SVD [43], Swift-SVD [35], SoLA [16], Basis Sharing [42], and AdaSVD [23]. Baseline results are obtained from official open-source code, or from values reported in the original publications when code is unavailable. Following ASVD and SVD-LLM, DynSVD adopts the same whitening transformation and uses 128 calibration samples from WikiText-2 with sequence length 2048. We report active-parameter compression ratios from 20% to 80%, where active parameters equal total parameters for all static baselines. Generally, DynSVD uses a fixed dynamic-to-static rank ratio $\alpha = r_d/r_s = 0.5$. At 20%, this ratio would store

more parameters than the original model, so we choose r_s such that DynSVD stores fewer parameters than the original model and set r_d to meet the same active budget.

We evaluate two selection rules on top of SVD-LLM: **DynSVD-K** keeps the top-r_d components of each token, and **DynSVD-T** keeps the components above one threshold θ shared by all layers, which keeps r_d components on average. **DynSVD†-T** applies DynSVD-T to the per-layer ranks released by Dobi-SVD (without remapping) instead of uniform ranks.

4.2 Main Results

Comparison with Existing Methods. We evaluate DynSVD against representative SVD-based compression methods on LLaMA-7B across active compression ratios ranging from 20% to 80%, with results summarized in Table 1. At every ratio, DynSVD achieves the lowest WikiText-2 perplexity and the highest zero-shot accuracy. Averaged over 40% and 60%, both DynSVD-T and DynSVD†-T achieve approximately 42% lower perplexity than SVD-LLM. Even DynSVD-K, which uses neither the threshold rule nor the rank allocation of Dobi-SVD, outperforms all static baselines on WikiText-2 at every ratio, so token-adaptive basis selection is the main source of the improvement.

On Top of Static SVD Methods. DynSVD adds token-adaptive selection on top of existing static SVD models. Built on SVD-LLM, DynSVD-T reduces WikiText-2 perplexity by 12–53% across active compression ratios. Likewise, DynSVD†-T applies the same selection rule to Dobi-SVD and achieves 18–42% lower perplexity. At the same computation budget, these gains demonstrate that DynSVD can be applied orthogonally to existing static SVD compression methods..

Table 2: Perplexity on larger models.

Act. Params Comp. Ratio	Method	Llama-13B		Llama-30B	
		Wiki2 (↓)	C4 (↓)	Wiki2 (↓)	C4 (↓)
–	Original	5.11	6.95	4.08	6.29
20%	SVD-LLM	6.95	13.54	5.65	10.48
	DynSVD-K	6.16	**9.71**	5.14	**8.22**
	DynSVD-T	**5.95**	9.74	**5.02**	**8.22**
40%	SVD-LLM	11.33	35.60	8.03	23.39
	DynSVD-K	8.44	19.09	6.49	13.33
	DynSVD-T	**7.66**	**16.35**	**6.17**	**12.06**
60%	SVD-LLM	39.94	210	19.24	114
	DynSVD-K	22.19	93.97	12.77	53.44
	DynSVD-T	**18.22**	**77.20**	**11.91**	**47.22**
80%	SVD-LLM	297	1318	165	1195
	DynSVD-K	**205**	**972**	107	618
	DynSVD-T	211	1145	**96.06**	**601**

Generalization across Models. Table 2 demonstrates that the benefits of DynSVD extend to larger models. When evaluated on LLaMA-13B and LLaMA-30B, DynSVD-T consistently achieves 11–54% lower perplexity than SVD-LLM across all compression regimes. For instance, on LLaMA-30B at 60% compression, DynSVD-T reduces WikiText-2 perplexity from 19.24 to 11.91, representing a 38% reduction. We further extend the evaluation beyond the legacy models to recent architectures, including Mixture-of-Experts and Vision-Language models, where DynSVD consistently outperforms SVD-LLM, as detailed in Appendix B and Appendix C.

4.3 Ablation Study on Selection Rule

Table 3: Selection rules on the same stored LLaMA-7B models at active compression ratios. tok/s: decoding throughput on one A100 40GB with our Triton kernels (batch=1, prefill=128, gen=128).

Ratio	Selection rule	Wiki2↓	tok/s↑
40%	Exact top-r_d $\mathcal{T}_{r_d}(\cdot)$	10.16	105.5
	Per-layer $\mathcal{T}_{\theta_l}(\cdot)$	9.69	**115.6**
	Shared $\mathcal{T}_{\theta}(\cdot)$	**9.45**	115.4
60%	Exact top-r_d $\mathcal{T}_{r_d}(\cdot)$	32.22	134.0
	Per-layer $\mathcal{T}_{\theta_l}(\cdot)$	29.34	146.5
	Shared $\mathcal{T}_{\theta}(\cdot)$	**28.27**	**147.4**
80%	Exact top-r_d $\mathcal{T}_{r_d}(\cdot)$	362	185.4
	Per-layer $\mathcal{T}_{\theta_l}(\cdot)$	**327**	196.9
	Shared $\mathcal{T}_{\theta}(\cdot)$	331	**197.7**

Exact vs. Threshold Selection. Section 3.4 pointed out two limitations of exact top-r_d: every projection keeps the same share of its bases, and the per-token selection is costly on GPUs. Table 3 tests both on the same stored models at 40%, 60%, and 80% active compression. Removing the selection step makes both threshold rules 6–10% faster. Since each token and projection can now keep a different number of bases, perplexity is also lower. At 40%, it drops from 10.16 to 9.69 with per-layer thresholds and to 9.45 with the shared threshold. As Table 1 shows, a threshold does not have to keep exactly r_d bases, so DynSVD-T uses slightly more active parameters than DynSVD-K (3.91B vs. 3.88B at 40%). Even with the same active parameters, exact top-r_d reaches only 10.02, so what matters is which bases are kept, not how many.

Budget Allocation across Layers. Figure 3 shows how each selection rule spends the budget. DynSVD-K keeps exactly r_d bases in every projection. Per-layer thresholding lets the number of kept bases vary from token to token, but each projection still keeps almost r_d bases. In contrast, the shared threshold keeps a different number of bases in each layer. It keeps fewer in the first layers' query and key projections, where a few bases already carry most of the signal, and more in the MLP down projections, where the signal is spread out. As a result, the selection error of Theorem 2, the share of $\|\hat{\mathbf{z}}\|^2$ in the dropped bases, falls from 1.95% with DynSVD-K to 1.75% with per-layer thresholding and 1.49% with the shared threshold. Both threshold rules gain more over DynSVD-K as $\alpha = r_d/r_s$ shrinks, from at most 0.2% lower perplexity at $\alpha = 0.9$ to 16–25% at $\alpha = 0.3$, since the choice of which bases to keep matters more when fewer are kept.

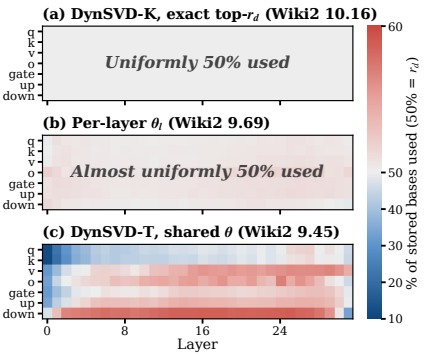

Figure 3: Share of the stored bases each projection uses per token (LLaMA-7B, 40%). Blue and red mean fewer and more than DynSVD-K.

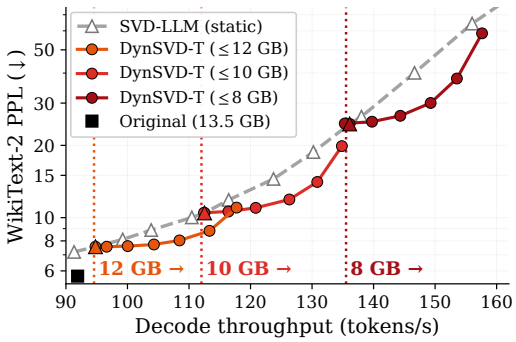

Figure 4: Pareto plot of throughput vs. perplexity for LLaMA-7B under memory budgets.

Table 4: LLaMA-7B under 10 GB (top) and 8 GB (bottom). DynSVD-T runs on the SVD-LLM model above it ($r_d/r_s = 0.8, 0.7, 0.6$), and its speed-up is over that model. Last row: SVD-LLM as fast as DynSVD-T at 0.6.

Model	Memory (GiB)	Act. / Total (B)	tok/s	Wiki2↓
SVD-LLM	9.67	4.43 / 4.43	112.5 (1.00×)	10.46
DynSVD-T	9.67	3.95 / 4.43	116.4 (1.03×)	10.61
	9.67	3.71 / 4.43	120.8 (1.07×)	10.98
	9.67	3.47 / 4.43	126.3 (1.12×)	11.90
SVD-LLM	7.55	3.30 / 3.30	136.1 (1.00×)	24.56
DynSVD-T	7.55	2.94 / 3.30	139.8 (1.03×)	25.09
	7.55	2.76 / 3.30	144.3 (1.06×)	26.56
	7.55	2.58 / 3.30	149.3 (1.10×)	30.03
SVD-LLM	6.74	2.82 / 2.82	149.4 (1.10×)	44.82

4.4 Runtime Flexibility under a Memory Budget

Runtime Flexibility. In a local-device scenario, the model must fit in a fixed amount of memory while the compute available to it changes over time. We study this setting on one A100 40GB GPU (batch 1, prefill 128, gen 128), and report the GPU memory of the whole process. A static SVD model has a single speed for a given memory, and it must be replaced by a smaller model to run faster. In contrast, DynSVD-T can adjust its threshold on the same stored weights and thus change its active parameters. Figure 4 fixes the GPU memory to 12, 10, or 8 GB and loads the largest SVD-LLM model that fits (triangles). DynSVD-T runs on the same stored weights and lowers r_d/r_s from 0.9 to 0.4, so the memory stays the same while the speed increases. From $r_d/r_s = 0.8$ down to 0.4, every DynSVD-T point has lower perplexity than the static model of the same speed, so DynSVD-T is Pareto optimal against the static models in this range: no SVD-LLM model is both faster and more accurate. Under 12 and 10 GB, DynSVD-T reaches up to $1.10\times$ higher throughput at the same perplexity and up to 27% lower perplexity at the same throughput, compared with the SVD-LLM curve interpolated between neighboring measured models.

Same Total Parameters. Table 4 lists the measurements under 10 and 8 GB, where DynSVD-T runs on the stored weights of the SVD-LLM model above it and therefore has the same total parameters. For example, under 10 GB DynSVD-T decodes 130.8 tokens per second at 14.08 perplexity, while the SVD-LLM model with almost the same speed (130.2 tokens per second) reaches 18.76. Under 8 GB, DynSVD-T decodes 149.3 tokens per second at 30.03 perplexity, while the SVD-LLM model with the same speed reaches 44.82, so DynSVD-T is 33% lower at the same speed, although both models already exceed 24 perplexity at this budget.

4.5 Decoding Throughput across Lengths and Batch Sizes

Figure 5 compares the decoding throughput of DynSVD-T with that of the original model. At batch size 1, DynSVD-T decodes 1.21–2.16× faster across 40–80% compression and both context lengths. In a batch, selection helps less because different tokens select different components, and on LLaMA-7B at 40% a batch of 8 sequences together selects 96% of the stored components. Even so, for batch sizes from 8 to 512 on LLaMA-2-7B and 64 to 256 on LLaMA-7B, DynSVD-T with $\alpha = r_d/r_s = 0.5$ decodes 1.20–1.33× faster than the original model at 60% and 1.65–1.96× at 80%, and at about the same speed at 40% (0.93–1.00×, Figure 5 and Appendix G). Selection can also be turned off for large batches, since keeping all stored components ($r_d = r_s$) gives exactly the static SVD model with the same stored weights. Our selection rule and Triton kernels target local decoding, where they minimize the computation of each token, so optimizing them for large batches is left for future work.

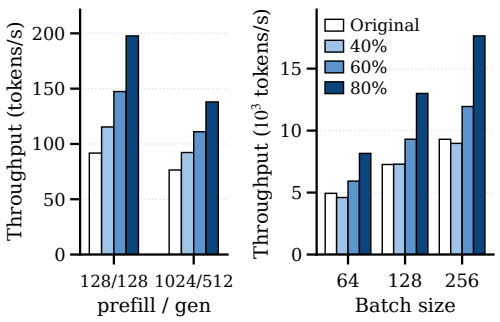

Figure 5: Decoding throughput of the original model and DynSVD-T on LLaMA-7B (A100). Left: batch 1, prefill/gen 128/128 and 1024/512. Right: prefill 4, gen 32.

4.6 Combination with Quantization

Low-rank compression is often combined with quantization to further reduce memory footprint, but the resulting approximation and quantization errors can amplify each other, leading to substantial performance degradation. Following AdaSVD [23], we apply 4-bit GPTQ quantization [8] to our methods and baselines, recalibrating θ on the quantized model. As shown in Table 5, while SVD-LLM and AdaSVD incur a 39–53% relative increase in perplexity, DynSVD exhibits a markedly smaller degradation of 25–26% at the same compression ratio. Notably, Dobi-SVD shows the smallest relative degradation (+9.5%), but DynSVD†-T achieves the best absolute perplexity both before quantization (9.97 vs. 17.38) and after (12.46 vs. 19.03). Dobi-SVD, however, learns its per-layer ranks with differentiable parameters for 10 hours on LLaMA-7B, whereas DynSVD adds under two minutes of threshold calibration to SVD-LLM (Appendix H). We hypothesize that the smaller degradation comes from dynamic selection, since activating fewer components per token lets fewer quantization-noise sources reach the output (Appendix A).

Table 5: Wiki2 PPL on LLaMA-2-7B at 40% active compression, before and after quantization.

Method	Wikitext2 PPL (↓)	
GPTQ 4-bit	X	O
SVD-LLM	17.79	24.74 (+39%)
AdaSVD	14.76	22.55 (+53%)
Dobi-SVD	17.38	19.03 **(+9.5%)**
DynSVD-K	11.42	14.43 (+26%)
DynSVD-T	10.24	12.90 (+26%)
DynSVD†-T	**9.97**	**12.46** (+25%)

5 Conclusion

We presented DynSVD, a training-free, token-adaptive inference framework that selects bases dynamically on top of static SVD. The selection is optimal for each token, and its error adds to the truncation error without interaction, so DynSVD complements existing static SVD methods such as SVD-LLM and Dobi-SVD. At the same active parameters, it lowers perplexity by 42% on average relative to static SVD, with consistent gains on larger and more recent models and after quantization. A static SVD model has one operating point per stored model, whereas one DynSVD model moves between speed and quality at run time by changing a single threshold. Within a fixed GPU memory budget, it therefore reaches up to 1.10× higher decoding throughput at the same perplexity and up to 27% lower perplexity at the same throughput than static SVD, and our Triton kernels make batch size one decoding 1.21–2.16× faster than the original model. The gains are smaller for large batches, where tokens together use almost all stored components, and DynSVD stores more parameters than a static model with the same active parameters. We discuss these limitations and future work in Appendix A.

AI USE STATEMENT

In this work, we used generative AI tools to help draft and refine the text, including grammar correction. We have reviewed all AI-assisted suggestions. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

[1] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I. et al. (2023) GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

[2] Alizadeh, K., Mirzadeh, I., Belenko, D. et al. (2024) LLM in a flash: Efficient large language model inference with limited memory. In *Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[3] Ashkboos, S., Croci, M.L., do Nascimento, M.G., Hoefler, T. & Hensman, J. (2024) SliceGPT: Compress large language models by deleting rows and columns. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[4] Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K. et al. (2023) Qwen technical report. *arXiv preprint arXiv:2309.16609*.

[5] Deng, L., Li, G., Han, S., Shi, L. & Xie, Y. (2020) Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE* **108**(4):485–532.

[6] Dettmers, T., Lewis, M., Belkada, Y. & Zettlemoyer, L. (2022) LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)* **35**, pp. 30318–30332.

[7] Fedus, W., Zoph, B. & Shazeer, N. (2022) Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* **23**(120):1–39.

[8] Frantar, E., Ashkboos, S., Hoefler, T. & Alistarh, D. (2022) GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

[9] Frantar, E. & Alistarh, D. (2023) SparseGPT: Massive language models can be accurately pruned in one-shot. In *Proc. of the International Conference on Machine Learning (ICML)*, pp. 10323–10337.

[10] Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S. et al. (2023) A framework for few-shot language model evaluation. *Zenodo*, v0.4.0. `https://zenodo.org/records/10256836`.

[11] Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M.W. & Keutzer, K. (2024) AI and memory wall. *IEEE Micro* **44**(3):33–39.

[12] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A. et al. (2024) The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

[13] Guo, D., Yang, D., Zhang, H., Song, J., Wang, P. et al. (2025) DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* **645**(8081):633–638.

[14] Han, S., Mao, H. & Dally, W.J. (2016) Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[15] Hsu, Y.-C., Hua, T., Chang, S., Lou, Q., Shen, Y. & Jin, H. (2022) Language model compression with weighted low-rank factorization. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[16] Huang, X., Huang, Y.-L. & Wen, Z. (2025) SoLA: Leveraging soft activation sparsity and low-rank decomposition for large language model compression. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)* **39**(16), pp. 17494–17502.

[17] Ioannou, Y., Robertson, D., Shotton, J., Cipolla, R. & Criminisi, A. (2016) Training CNNs with low-rank filters for efficient image classification. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[18] Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J. & Amodei, D. (2020) Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

[19] Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M.W. & Keutzer, K. (2024) SqueezeLLM: Dense-and-sparse quantization. In *Proc. of the International Conference on Machine Learning (ICML)*, pp. 23901–23923.

[20] Klema, V. & Laub, A. (1980) The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control* **25**(2):164–176.

[21] Laskaridis, S., Katevas, K., Minto, L. et al. (2024) MELTing point: Mobile evaluation of language transformers. In *Proc. of the Annual International Conference on Mobile Computing and Networking (MobiCom)*.

[22] Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I. & Lempitsky, V. (2015) Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[23] Li, Z., Xia, M., Zhang, J., Hui, Z., Qin, H., Kong, L., Zhang, Y. & Yang, X. (2026) AdaSVD: Singular value decomposition with adaptive mechanisms for large multimodal models. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 26358–26368.

[24] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P. & Zitnick, C.L. (2014) Microsoft COCO: Common objects in context. In *Proc. of the European Conference on Computer Vision (ECCV)*, pp. 740–755.

[25] Lin, J., Tang, J., Tang, H., Yang, S. et al. (2024) AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In *Proc. of Machine Learning and Systems (MLSys)*.

[26] Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B. et al. (2023) Deja Vu: Contextual sparsity for efficient LLMs at inference time. In *Proc. of the International Conference on Machine Learning (ICML)*, pp. 22137–22176.

[27] Liu, H., Li, C., Li, Y. & Lee, Y.J. (2024) Improved baselines with visual instruction tuning. In *Proc. of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 26296–26306.

[28] Liu, J., Ponnusamy, P., Cai, T., Guo, H., Kim, Y. & Athiwaratkun, B. (2025) Training-free activation sparsity in large language models. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[29] Ma, X., Fang, G. & Wang, X. (2023) LLM-Pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)* **36**, pp. 21702–21720.

[30] Merity, S., Xiong, C., Bradbury, J. & Socher, R. (2017) Pointer sentinel mixture models. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[31] Mishra, A., Latorre, J.A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C. & Micikevicius, P. (2021) Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*.

[32] Muennighoff, N., Soldaini, L., Groeneveld, D., Lo, K., Morrison, J. et al. (2025) OLMoE: Open mixture-of-experts language models. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[33] Nait Saada, T., Naderi, A. & Tanner, J. (2025) Mind the gap: A spectral analysis of rank collapse and signal propagation in attention layers. In *Proc. of the International Conference on Machine Learning (ICML)*, pp. 45561–45587.

[34] Noci, L., Anagnostidis, S., Biggio, L., Orvieto, A., Singh, S.P. & Lucchi, A. (2022) Signal propagation in transformers: Theoretical perspectives and the role of rank collapse. In *Advances in Neural Information Processing Systems (NeurIPS)* **35**, pp. 27198–27211.

[35] Qi, R., Liu, Y., Wu, X., Wang, X., Li, M., Chen, C., Chen, J., Chen, Y. & Weng, Q. (2026) Swift-SVD: Theoretical optimality meets practical efficiency in low-rank LLM compression. In *Proc. of the International Conference on Machine Learning (ICML)*.

[36] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. & Liu, P.J. (2020) Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* **21**(140):1–67.

[37] Sharify, S., Saxena, U., Xu, Z., Yazar, W., Soloveychik, I. & Wang, X. (2024) Post-training quantization of large language models with microscaling formats. In *Proc. of the 4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP)*, PMLR **262**, pp. 241–258.

[38] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G. & Dean, J. (2017) Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[39] Shuttleworth, R., Andreas, J., Torralba, A. & Sharma, P. (2025) LoRA vs full fine-tuning: An illusion of equivalence. In *Advances in Neural Information Processing Systems (NeurIPS)* **38**, pp. 174627–174662.

[40] Snell, C., Lee, J., Xu, K. & Kumar, A. (2025) Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[41] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. et al. (2023) LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

[42] Wang, J., Chen, Y.-G., Lin, I.-C., Li, B. & Zhang, G.L. (2025a) Basis Sharing: Cross-layer parameter sharing for large language model compression. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[43] Wang, Q., Ke, J., Tomizuka, M., Keutzer, K. & Xu, C. (2025b) Dobi-SVD: Differentiable SVD for LLM compression and some new perspectives. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[44] Wang, X., Zheng, Y., Wan, Z. & Zhang, M. (2025c) SVD-LLM: Truncation-aware singular value decomposition for large language model compression. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[45] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J. & Han, S. (2023) SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proc. of the International Conference on Machine Learning (ICML)*, pp. 38087–38099.

[46] Xie, X., Luo, Y., Peng, H. & Ding, C. (2025) RTop-K: Ultra-fast row-wise top-k selection for neural network acceleration on GPUs. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[47] Yang, A., Li, A., Yang, B., Zhang, B., Hui, B. et al. (2025) Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

[48] Yang, S., Fan, X., Pan, M., Xi, H., Wang, Z. et al. (2026) FreeToken: Efficient edge-native MoE serving with bandwidth-adaptive execution. *arXiv preprint arXiv:2608.16157*.

[49] Yuan, Z., Shang, Y., Song, Y., Yang, D., Wu, Q., Yan, Y. & Sun, G. (2023) ASVD: Activation-aware singular value decomposition for compressing large language models. *arXiv preprint arXiv:2312.05821*.

[50] Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X.V. et al. (2022) OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

[51] Zhang, Z., Liu, Z., Tian, Y., Khaitan, H., Wang, Z. & Li, S. (2025) R-Sparse: Rank-aware activation sparsity for efficient LLM inference. In *Proc. of the International Conference on Learning Representations (ICLR)*.

A LIMITATIONS

Quantization-noise reduction under dynamic selection. Section 4.6 observed that DynSVD degrades less than SVD-LLM and AdaSVD after GPTQ 4-bit quantization, although Dobi-SVD degrades even less. We hypothesize that this gap comes from a structural property of dynamic selection in which activating only r_d of the r_s stored components per token bounds the number of quantization-noise sources that can reach the output, while magnitude-based selection biases the surviving directions toward high-signal components. Quantitative validation of this mechanism remains future work.

Kernel optimization scope. The custom Triton kernels in Section 3.5 target batch-size-one decode, where bandwidth-bound execution makes per-component selection profitable. In larger batches, different tokens select different components, so the batch together reads most of the stored components and the speed follows the stored rather than the active parameters (Section 4.5).

B ADDITIONAL RESULTS: MODERN ARCHITECTURES

Table 6: Perplexity comparison of SVD-LLM, DynSVD-K, and DynSVD-T on Llama and OLMoE models under various compression ratios. On OLMoE-1B-7B, every method compresses only the expert FFN layers.

Ratio	Method	Llama-2-7B		Llama-3.1-8B		OLMoE-1B-7B	
		Wiki2↓	C4↓	Wiki2↓	C4↓	Wiki2↓	C4↓
–	Original	5.49	7.42	6.19	9.82	7.40	11.72
20%	SVD-LLM	8.82	20.43	15.24	80.11	10.00	15.96
	DynSVD-K	7.33	13.32	9.96	28.35	9.20	14.41
	DynSVD-T	**7.25**	**13.04**	**9.84**	**27.55**	**9.18**	**14.34**
40%	SVD-LLM	17.79	78.53	95.71	887.72	13.54	23.72
	DynSVD-K	11.41	32.49	35.02	227.49	**12.31**	20.73
	DynSVD-T	**10.24**	**26.49**	**29.89**	**182.21**	12.37	**20.42**
60%	SVD-LLM	94.54	695.01	871.91	3316.31	24.91	53.56
	DynSVD-K	43.94	233.34	331.19	1765.86	**22.48**	45.26
	DynSVD-T	**37.85**	**195.08**	**317.90**	**1668.86**	22.83	**44.39**

Table 7: Perplexity comparison of SVD-LLM, DynSVD-K, and DynSVD-T on Qwen models under various compression ratios.

Ratio	Method	Qwen2-7B		Qwen2.5-14B		Qwen3-8B	
		Wiki2↓	C4↓	Wiki2↓	C4↓	Wiki2↓	C4↓
–	Original	7.10	11.01	5.25	10.54	9.72	15.78
20%	SVD-LLM	10.83	39.62	9.03	30.04	15.02	37.65
	DynSVD-K	8.75	20.46	7.47	17.74	12.87	25.69
	DynSVD-T	**8.70**	**20.22**	**7.37**	**17.38**	**12.80**	**25.23**
40%	SVD-LLM	30.45	166.57	16.78	174.61	26.80	125.38
	DynSVD-K	17.85	92.80	11.80	68.80	20.20	80.14
	DynSVD-T	**16.27**	**78.22**	**11.24**	**61.65**	**18.43**	**66.98**
60%	SVD-LLM	136.29	708.84	124.20	742.09	131.05	964.48
	DynSVD-K	**103.20**	536.98	65.64	519.50	**80.28**	554.33
	DynSVD-T	103.82	**529.41**	**60.64**	**475.81**	83.32	**517.69**

C Additional Results: Vision-Language Model

Table 8: LLaVA-1.5-7B with its language model compressed (vision encoder and projector unchanged; whitening and threshold calibration on 32 COCO [24] image–caption samples). TextVQA (val), GQA (test-dev balanced), MMBench (English dev, CircularEval) and ScienceQA-IMG (test): accuracy (%), greedy decoding with the LLaVA-1.5 prompts. Act. / Total: decoder-linear parameters of the language model in billions; DynSVD's active count is measured over all benchmark tokens. DynSVD uses exactly the nominal active parameters of SVD-LLM at the same ratio; at 20% it stores fewer parameters than the Original. DynSVD-T scales its threshold by one factor per benchmark, chosen without labels on 20% of that benchmark's inputs, so that the measured active count matches (within 0.1%). SVD-LLM at 40% no longer answers the questions (it replies with short image captions).

Ratio	Method	Act. / Total (B)	TextVQA↑	GQA↑	MMBench↑	SQA-I↑
–	Original	6.48 / 6.48	57.3	61.3	64.5	67.2
20%	SVD-LLM	5.18 / 5.18	39.7	52.0	43.4	54.5
	DynSVD-K	5.18 / 6.14	48.4	57.4	53.8	61.3
	DynSVD-T	5.18 / 6.14	**48.7**	**57.5**	**55.3**	**62.8**
40%	SVD-LLM	3.88 / 3.88	0.5	0.0	0.1	10.1
	DynSVD-K	3.88 / 5.44	17.2	24.7	17.8	41.9
	DynSVD-T	3.88 / 5.44	**35.2**	**48.4**	**38.3**	**52.1**

D Proofs of Theorems

D.1 Proof of Theorem 1 (Optimal Basis Selection via Intermediate Activation Magnitude)

For brevity, we write $\mathcal{T} := \mathcal{T}_{r_d}$. Let $\mathcal{S} \subseteq \{1, \dots, r\}$ denote the selected indices with $|\mathcal{S}| = r_d$, and let $\boldsymbol{\ell}_i$ denote the i-th column of $\mathbf{L}$. The reconstruction error decomposes as

$$\|\mathbf{L}\mathbf{z} - \mathbf{L}\mathcal{T}(\mathbf{z})\|_2^2 = \Big\| \sum_{i=1}^{r} z_i \boldsymbol{\ell}_i - \sum_{i \in \mathcal{S}} z_i \boldsymbol{\ell}_i \Big\|_2^2 = \Big\| \sum_{i \notin \mathcal{S}} z_i \boldsymbol{\ell}_i \Big\|_2^2. \tag{7}$$

Applying $\boldsymbol{\ell}_i^\top \boldsymbol{\ell}_j = \delta_{ij}$, all cross terms vanish:

$$\Big\| \sum_{i \notin \mathcal{S}} z_i \boldsymbol{\ell}_i \Big\|_2^2 = \sum_{i \notin \mathcal{S}} \sum_{j \notin \mathcal{S}} z_i z_j \boldsymbol{\ell}_i^\top \boldsymbol{\ell}_j = \sum_{i \notin \mathcal{S}} z_i^2. \tag{8}$$

Minimizing this sum subject to $|\mathcal{S}| = r_d$ excludes the indices with the smallest $|z_i|$, equivalently retaining the r_d entries with the largest activation magnitudes. □

D.2 Proof of Theorem 2 (Decomposition of Reconstruction Error in Dynamic SVD)

Write $\mathcal{T} := \mathcal{T}_{r_d}$. Adding and subtracting $\hat{\mathbf{L}}\hat{\mathbf{R}}\mathbf{x}$ yields

$$\mathbf{W}\mathbf{x} - \hat{\mathbf{L}}\mathcal{T}(\hat{\mathbf{z}}) = \underbrace{(\mathbf{W} - \hat{\mathbf{L}}\hat{\mathbf{R}})\mathbf{x}}_{\mathcal{E}_{\text{trunc}}} + \underbrace{\hat{\mathbf{L}}(\hat{\mathbf{z}} - \mathcal{T}(\hat{\mathbf{z}}))}_{\mathcal{E}_{\text{select}}}, \tag{9}$$

where $\hat{\mathbf{z}} = \hat{\mathbf{R}}\mathbf{x}$. With $\mathbf{W} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ and $z_i = \sigma_i \mathbf{v}_i^\top \mathbf{x}$,

$$\mathcal{E}_{\text{trunc}} = \sum_{i > r_s} z_i \mathbf{u}_i, \qquad \mathcal{E}_{\text{select}} = \sum_{\substack{i \le r_s \\ i \notin \mathcal{S}}} z_i \mathbf{u}_i, \tag{10}$$

where $\mathcal{S}$ is the index set selected by $\mathcal{T}$. The two index sets are disjoint; by orthonormality of $\{\mathbf{u}_i\}$, $\mathcal{E}_{\text{trunc}}^\top \mathcal{E}_{\text{select}} = 0$, hence

$$\|\mathbf{W}\mathbf{x} - \hat{\mathbf{L}}\mathcal{T}(\hat{\mathbf{z}})\|_2^2 = \|\mathcal{E}_{\text{trunc}}\|_2^2 + \|\mathcal{E}_{\text{select}}\|_2^2. \qquad \square \tag{11}$$

E TOKEN-WISE INTERMEDIATE ACTIVATION MAGNITUDES ACROSS RANKS

To complement Figure 1, we show two further views of the per-token intermediate activation magnitudes $|z_i|$ on LLaMA-7B after SVD-LLM whitening. In every panel, colored bars denote components retained by each strategy and black regions denote truncated components. Static and dynamic strategies retain *the same number* of components per token but differ entirely in *which* bases they keep. The static truncation discards components in ascending order of singular value, removing a contiguous suffix of the rank axis regardless of input, whereas dynamic selection retains the top-r_d entries with the largest $|z_i|$ per token, distributing the truncation pattern across the full rank range. Consequently, dynamic selection can keep low-singular-value components when their per-token activations are large, an option that is structurally unavailable to static truncation.

Block-wise comparison at a fixed budget. Figure 6 shows the attention value projection at Blocks 5, 15, and 25 with r_d=2048 out of r=4096. The dynamic panels instead show black bars scattered across the entire rank axis on a per-token basis. Therefore, low-singular-value components are frequently activated for specific tokens and are therefore retained, while higher-ranked but inactive components are truncated. The pattern is consistent across early, middle, and late blocks, indicating that token-adaptive activation is a structural property of the model rather than a localized phenomenon.

Budget-wise comparison at a fixed layer. Figure 7 shows the MLP up projection at Layer 5 across rank budgets $r_d \in \{512, 1024, 2048\}$. As the budget shrinks, static truncation discards a longer suffix of low-singular-value components, eliminating all activations beyond rank index r_d. Dynamic selection instead concentrates the retained budget on the largest-magnitude entries for each token, preserving the strongest peaks across the full rank range even at r_d=512.

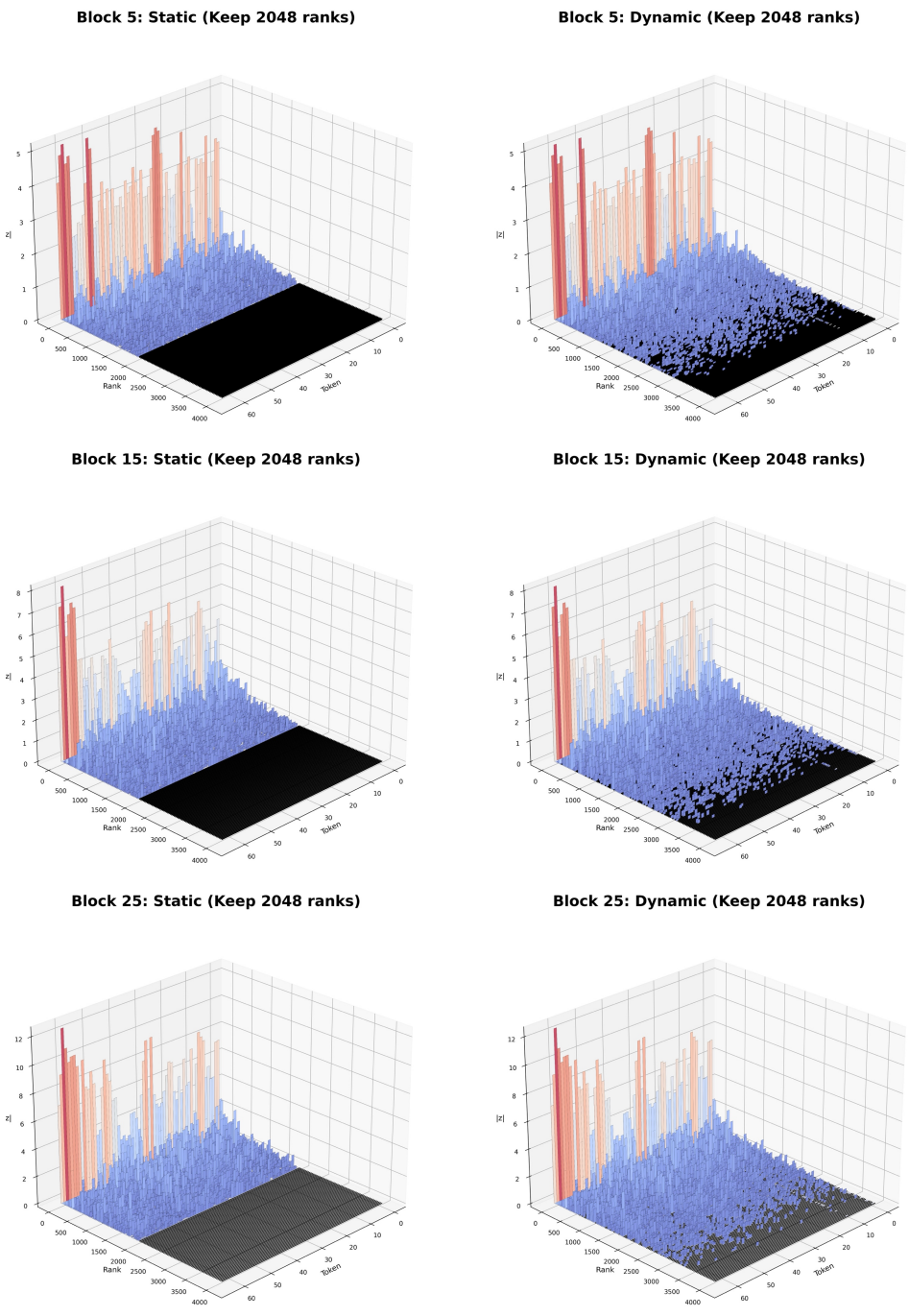

Figure 6: Token-wise intermediate activation magnitudes $|z|$ for the attention value projection on LLaMA-7B at Blocks 5, 15, and 25 (after whitening), with r_d=2048 retained out of r=4096. **Left:** static truncation discards the lower half of the spectrum (uniform black band at higher rank indices) for every token. **Right:** dynamic selection retains the top-r_d entries by $|z_i|$ per token, so the truncation pattern (black) varies by token and is distributed across the rank axis.

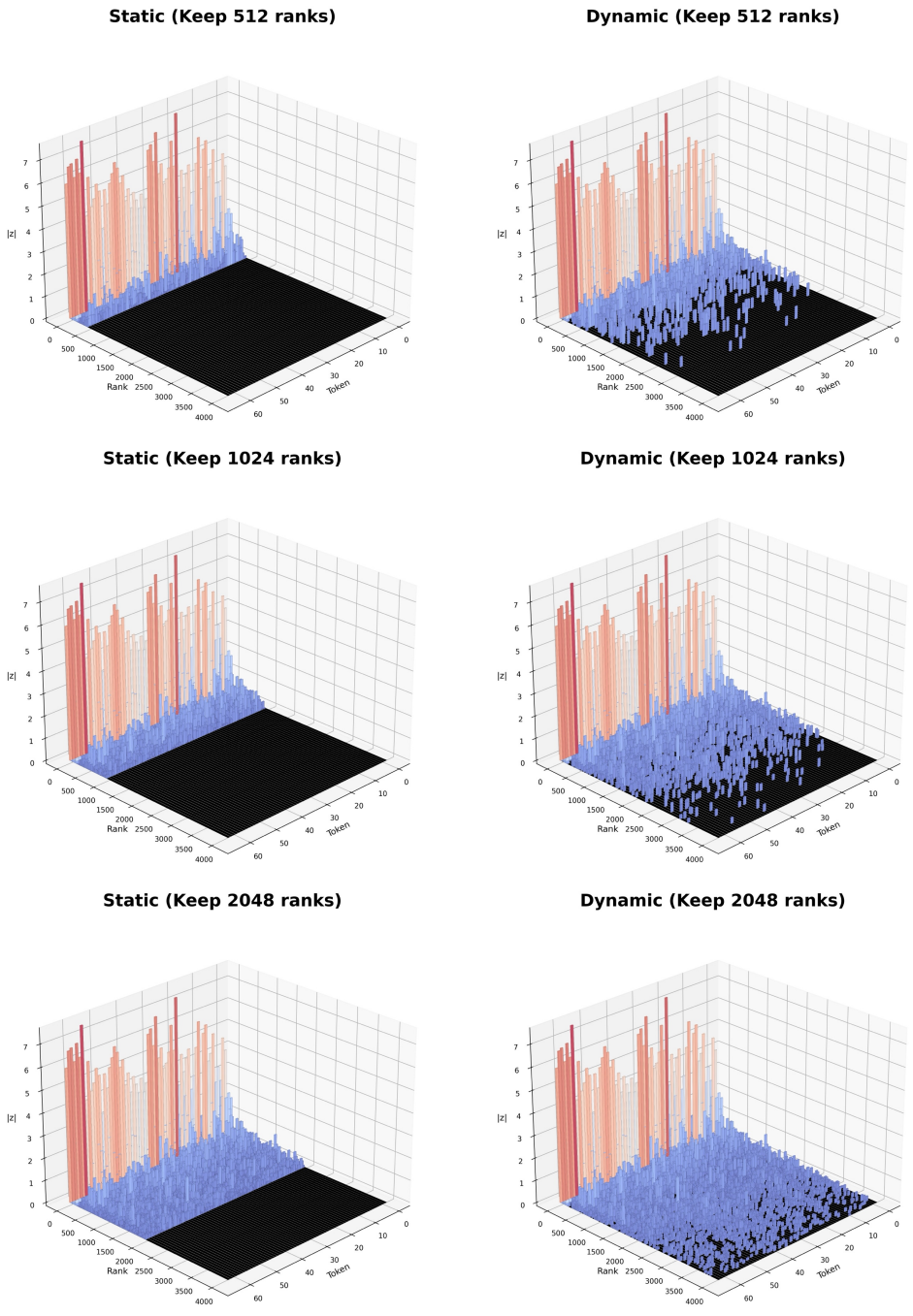

Figure 7: Token-wise intermediate activation magnitudes $|z|$ for the MLP up projection at Layer 5 of LLaMA-7B (after whitening), across rank budgets $r_d \in \{512, 1024, 2048\}$. **Left:** static truncation removes a contiguous suffix of low-singular-value components; the black region grows as r_d shrinks. **Right:** dynamic selection retains the top-r_d components by $|z_i|$ per token, preserving the largest-magnitude peaks across the full rank range even under aggressive budgets.

F TRITON KERNEL IMPLEMENTATION DETAILS

Runtime. All speeds are measured at batch size 1 with a static KV cache, and the whole decoding step is captured in one CUDA graph.

First kernel. Each compressed projection runs two Triton kernels, and projections that read the same input (query, key, and value, or gate and up) share one launch. The first computes $\hat{\mathbf{z}} = \hat{\mathbf{R}}\mathbf{x}$ with FP32 accumulation. For DynSVD-K, it also finds the exact top-r_d entries without sorting. With the sign bit cleared, FP16 bit patterns sort in the same order as magnitudes, so the thread blocks count the entries in a 256-bin histogram of their top 8 bits. The last block to finish reads the histogram to find the bin that holds the r_d-th largest value, locates the value inside the bin in 7 bisection steps, and writes the indices of the kept entries. This step makes DynSVD-K 6–10% slower than DynSVD-T (Table 3).

Second kernel. $\hat{\mathbf{L}}$ is stored transposed, so each entry of $\hat{\mathbf{z}}$ owns one contiguous row, and a dropped entry costs no memory reads. For DynSVD-T, each thread block computes the threshold of Eq. 6 from $\hat{\mathbf{z}}$ itself and loads a row only when its entry passes, so no sorting or index list is needed. For DynSVD-K, it loads the rows listed by the first kernel. Tensor Cores multiply the FP16 values and sum them in FP32, and the result is converted back to FP16 once at the end. The partial sums are added in a fixed order, so the output is the same in every run.

Contribution and run-time control. Figure 8 adds one part of the runtime at a time on LLaMA-2-7B at 40%. The PyTorch reference zeroes the dropped entries but still multiplies by every column of $\hat{\mathbf{L}}$, so it saves no memory reads. It decodes at 0.40× the speed of the original model without a CUDA graph and at 0.63× with one. Our Triton kernels raise this to 1.07× with exact top-r_d and to 1.23× with the threshold rule. The threshold θ is read from GPU memory at every step, so writing a new value between two steps changes it for all projections without reloading the model or capturing a new CUDA graph. Changing it every 16 tokens adds less than 0.04% to the run time.

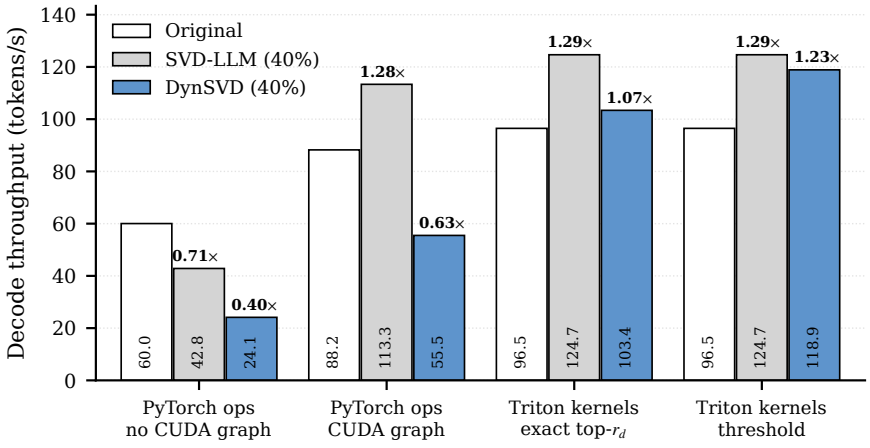

Figure 8: Decoding throughput of LLaMA-2-7B at 40% compression ($r_d = r_s/2$) as each part of our runtime is added, measured in one session on one A100 40GB GPU (batch size 1, prefill 18, gen 128). From left to right: the PyTorch operations of the reference implementation without and with a CUDA graph, our Triton kernels with exact top-r_d selection, and the threshold rule ($\theta = 0.65$, keeping 0.99–1.02 times r_d components) instead of exact top-r_d. The original model and SVD-LLM at 40% use cuBLAS with the PyTorch operations and our Triton GEMV with the Triton kernels. In the bars: tokens/s. Above the bars: speed-up over the original model in the same setting. The run used an earlier version of our kernels. With the final kernels (prefill 128, gen 128), exact top-r_d and the calibrated global threshold reach 108.5 and 118.0 tokens/s against 95.2 for the original model.

G Extended Throughput Results

Table 9 lists the decoding throughput of LLaMA-2-7B from batch size 1 to 512, with SVD-LLM at the same active parameters for reference. Figure 5 in Section 4.5 shows LLaMA-7B.

Table 9: Decoding throughput (tokens/s) of LLaMA-2-7B from batch size 1 to 512 on one A100 40GB GPU (prefill 4, gen 32, CUDA graphs). Each model uses its fastest measured implementation: our Triton kernels at batch size 1 (Appendix F) and Triton or cuBLAS matrix products at larger batch sizes. In parentheses: speed-up over the original model at the same batch size. DynSVD-T at 20% uses $r_d = r_s/2$ and stores more parameters than the original model (7.26B against 6.48B in the linear layers), unlike the 20% model of Table 1.

		Batch size						
Method	Ratio	1	8	32	64	128	256	512
Original	0%	98	722	2669	4956	7388	9485	10326
SVD-LLM	20%	103 (1.06)	787 (1.09)	2935 (1.10)	5382 (1.09)	8328 (1.13)	9748 (1.03)	11288 (1.09)
	40%	127 (1.31)	937 (1.30)	3584 (1.34)	6602 (1.33)	9668 (1.31)	12221 (1.29)	13738 (1.33)
	60%	166 (1.70)	1219 (1.69)	4677 (1.75)	8320 (1.68)	12681 (1.72)	15969 (1.68)	18081 (1.75)
	80%	233 (2.39)	1698 (2.35)	6257 (2.34)	11243 (2.27)	16634 (2.25)	21807 (2.30)	24690 (2.39)
DynSVD-T	20%	93 (0.95)	554 (0.77)	2019 (0.76)	3729 (0.75)	5952 (0.81)	6860 (0.72)	8167 (0.79)
	40%	118 (1.21)	688 (0.95)	2497 (0.94)	4592 (0.93)	7300 (0.99)	8955 (0.94)	10268 (0.99)
	60%	155 (1.59)	885 (1.23)	3219 (1.21)	5923 (1.20)	9306 (1.26)	11894 (1.25)	13720 (1.33)
	80%	216 (2.21)	1251 (1.73)	4515 (1.69)	8199 (1.65)	13101 (1.77)	17677 (1.86)	20195 (1.96)

H Compression Time

Table 10 reports the offline compression-time overhead of each method on a single 80GB NVIDIA A100 GPU. DynSVD-K introduces only a modest increase relative to SVD-LLM, requiring approximately 4–13% additional time. DynSVD-T uses the same factors and adds only the calibration of the global threshold θ, which takes under two minutes for each model in Table 10. This overhead is a one-time, offline cost, while the resulting accuracy and runtime benefits are realized repeatedly during inference.

We additionally compare against Dobi-SVD, which compresses LLaMA-13B in approximately 16 hours. DynSVD-T compresses the same model in about 24 minutes. Dobi-SVD's much higher compression cost stems from learning its truncation positions through differentiable SVD, whereas DynSVD requires no training.

Table 10: Compression time on one A100 80GB GPU. The θ calibration of DynSVD-T was measured on one A100 40GB GPU.

Method	Training-free	LLaMA-7B	LLaMA-13B	LLaMA-30B
SVD-LLM	O	7m 52s	21m 02s	74m 30s
Dobi-SVD	X	∼10h	∼16h	OOM
DynSVD-K	O	8m 18s	23m 49s	77m 09s
DynSVD-T	O	DynSVD-K + θ calibration (<2 min)		