Beyond FLOPs: Benchmarking Real Inference Acceleration of LLM Pruning under a GEMM-Centric Taxonomy

Haozhe Hu [1,2] **Hao Wu** [1,2] **Anhao Zhao** [2,3] **Longwei Ding** [2] **Peiran Yin** [1,2]
Yunpu Ma[4] **Xiaoyu Shen** [2*]
[1]Shanghai Jiao Tong University
[2]Zhejiang Key Laboratory of Industrial Intelligence and Digital Twin, EIT, Ningbo
[3]Department of Computing, The Hong Kong Polytechnic University
[4]Munich Center for Machine Learning, LMU Munich

Sophie17@sjtu.edu.cn; xyshen@eitech.edu.cn

Abstract

Pruning has emerged as a dominant paradigm for accelerating large language model (LLM) inference, spanning a broad spectrum of methods that remove computation across tokens, layers, heads, dimensions, and attention patterns. Despite sharing the same objective, these pruning approaches induce fundamentally different execution behaviors, causing realized speedups to depend heavily on hardware and kernel implementations. Consequently, the practical acceleration benefits of different pruning families remain poorly understood. In this work, we introduce a GEMM-centric taxonomy that reorganizes existing pruning methods according to the logical **M**, **N**, and **K** dimensions of general matrix multiplication (GEMM). Leveraging this abstraction, we build a unified benchmarking framework that enables implementation-consistent comparison across the pruning design space and systematically characterizes the acceleration–quality Pareto frontier. Our results on Llama3.1-8B show that static depth pruning remains the strongest Pareto-optimal baseline and stays closest to its theoretical acceleration upper bound in memory-bounded scenarios. During prefill, the frontier transitions from static depth at low quality loss (0%–4%), to dynamic depth at moderate loss (5%–16%), and finally to static width pruning at higher loss levels (17%–26%). These findings establish the first unified view of the practical limits of pruning-based LLM acceleration and provide guidance for future pruning research.[1]

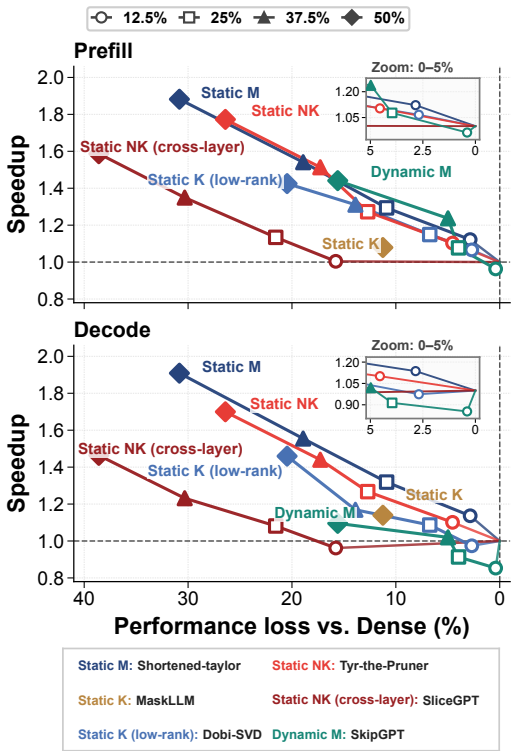

Figure 1: Throughput–quality trade-off of different pruning methods at 12.5%/25%/37.5%/50% sparsity in Llama3.1-8B, with compared methods listed below.

1 Introduction

The rapid advancement of LLMs has driven progress in reasoning, code generation, scientific discovery, and agentic systems (Zhao et al., 2025b; Hu et al., 2025; Team et al., 2026). However, inference efficiency remains a key bottleneck for practical deployment. Acceleration methods span system-level optimizations (e.g., scheduling and kernels (Kwon et al., 2023; Zheng et al., 2024; He et al., 2026)) and model-level compression, among which pruning is widely studied to eliminate redundancy while preserving architectural and deployment compatibility (Zhou et al., 2024), which can naturally extend to Vision Language Models (VLMs) and other multi-modality models (Wu et al., 2025, 2026b,a; Liu et al., 2026; Tan et al., 2026). Existing LLM pruning methods span a diverse design space characterized by execution behavior and structural focus. Behaviorally, approaches divide into *static pruning*, which re-

*Corresponding Author
[1]Code is available at https://github.com/EIT-NLP/LLM-Pruning/tree/main/PruningInferSim

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 31283–31310
October 24-29, 2026 ©2026 Association for Computational Linguistics

moves a fixed amount of computation, and *dynamic pruning*, which adapts computation per input instance (Jiang et al., 2024; Han et al., 2025; Hu et al., 2026). Structurally, pruning targets various architectural levels, including tokens, layers, and sublayers (Men et al., 2025; He et al., 2024; Fan et al., 2026), attention heads (Ma et al., 2023), matrix dimensions (Ashkboos et al., 2023), low-rank approximations (Wang et al., 2024), and attention patterns (Sun et al., 2026).

Despite belonging to the broader pruning family, diverse pruning designs are difficult to compare directly. Because these techniques alter a model's shape, sparsity patterns, or execution flow, they directly affect low-level hardware behaviors such as memory access contiguity and kernel scheduling efficiency. As a result, realized speedups become tightly coupled to specific hardware implementations. This dependency severely fragments the evaluation landscape. Since measuring actual latency requires dedicated system engineering for each new method, studies frequently default to theoretical proxies such as FLOPs reduction and single-kernel throughput (Ashkboos et al., 2023; Fang et al., 2024; Raposo et al., 2024; Qinsi et al., 2025). In other cases, they evaluate against inconsistent standards by using different model families (Touvron et al., 2023) or specialized hardware optimizations (Li et al., 2025b). This leaves a critical gap in understanding *how different pruning patterns translate into practical speedups under a unified and comparable setting*.

Motivated by this gap, we revisit LLM pruning through the lens of GEMM-dominated inference. Regardless of their structural focus, existing pruning methods ultimately reduce computation along one or more of the three GEMM logical axes: M, N, and K. Mapping diverse pruning strategies into this shared operational space abstracts away method-specific complexities and successfully decouples algorithms from bespoke hardware dependencies.

Building on this abstraction, we develop a comprehensive suite that captures both taxonomy-level behaviors and method-specialized implementation details, while providing cross-platform kernel baselines through DSLs such as Triton (Tillet et al., 2019) and Tilelang (Wang et al., 2025). Because any pruning technique can be mapped directly to this shared abstraction, methods can finally be evaluated fairly against an identical baseline. This framework also establishes a plug-and-play foundation for future research, enabling new pruning algorithms to be tested for true end-to-end latency out of the box, while offering a taxonomy-based optimization view that can guide further specialized kernel engineering.

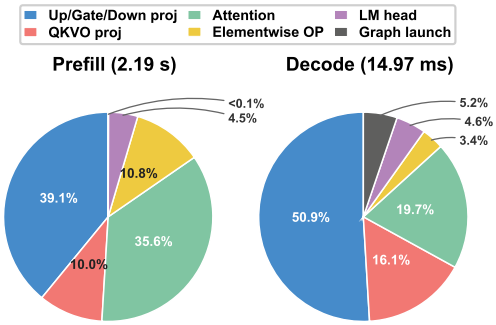

Figure 2: Llama3.1-8B's latency breakdown on each forward step, with batch size 1 and 32,768 context length.

Equipped with this framework, we conduct the first comprehensive cross-design comparison spanning the existing pruning design space. This allows us to systematically benchmark the acceleration potential of each taxonomy in a completely implementation-consistent manner, uncovering the true downstream performance trade-offs of different pruning families for the first time (Figure 1). In summary, our main contributions are as follows:

(1) **Taxonomy.** We present a GEMM-centric taxonomy for LLM pruning that unifies existing methods by their logical **M**, **N**, and **K** pruning dimensions, providing an operation-level view of their effect on propagation and implementation.

(2) **Framework.** We build a unified inference benchmarking framework with hardware-agnostic baseline acceleration implementations for each taxonomy, enabling controlled comparison of their end-to-end and kernel-level speedup, and further hardware-specific optimizations.

(3) **Insights.** Under the same framework and hardware environment, we establish the current Pareto frontier of pruning-based acceleration: static depth remains the strongest speed-oriented baseline, dynamic depth becomes preferable under moderate quality loss, and static width becomes competitive only at higher loss levels.

2 Background and Definitions

Inference Process of LLM. For a standard dense LLM (Grattafiori et al., 2024), we consider the simplest single-step forward pass, where inference cost mainly comes from GEMM computation, element-wise operations (e.g., normalization, residual ad-

dition), and the associated CPU-side processing overhead. In practice, the computation in each LLM layer is dominated by nine GEMM operations, which are organized across the attention and feed-forward network (FFN) as follows:

$$[Q, K, V] = [X\mathbf{W}_q, X\mathbf{W}_k, X\mathbf{W}_v]$$
$$X_{\text{attn.}} = \left(\text{Softmax}(QK^\top/\sqrt{d})V\right)\mathbf{W}_o$$
$$X_{\text{FFN}} = \left((X\mathbf{W}_{\text{up}}) \odot \sigma(X\mathbf{W}_{\text{gate}})\right)\mathbf{W}_{\text{down}}$$

Specifically, the attention layer contains four linear projections (Q, K, V, and O), together with two GEMMs inside the attention computation ($\mathbf{QK}^\top$ and $\mathbf{PV}$), while the FFN layer contains three linear projections (Up, Gate, and Down). Figure 2 reports the runtime breakdown of these GEMMs and other operations. The breakdown shows that GEMM operations, together with the LM head, account for roughly 90% of the total inference cost and clearly dominate the overall runtime. This suggests that GEMM-oriented optimization remains a central direction for LLM inference acceleration.

LLM Pruning. In the LLM pruning literature, existing approaches are commonly grouped into two categories by pruning dimension: **depth** and **width** (Cheng et al., 2024). **Depth pruning** removes layers or sublayers, which can be viewed as skipping their computation for all tokens (Kim et al., 2024; Men et al., 2025; Ding et al., 2025; Zhong et al., 2025; Fan et al., 2025). **Width pruning**, in contrast, removes neuron groups within weight matrices, such as attention heads (e.g., 128 neurons) (An et al., 2023; Ma et al., 2023; Li et al., 2025a), entire columns (Ashkboos et al., 2023), finer-grained semi-structured 2:4 groups (Frantar and Alistarh, 2023; Sun et al., 2023; Fang et al., 2024), and individual neurons (Xia et al., 2023). Beyond zeroing out neurons, the matrix width can also be reduced by low-rank approximation (Wang et al., 2024; Qinsi et al., 2025), while long-context sparse attention can be regarded as another pruning form that operates over key-value tokens (Gao et al., 2025b; Zhang et al., 2025; Yuan et al., 2025).

Apart from pruning dimension, the dynamics of pruning also affect speed and performance. **Static pruning** eliminates redundant structures through a one-time calibration phase and loads only the retained neurons during inference (Frantar and Alistarh, 2023; Ma et al., 2023), which reduces memory footprint and requires no modifications to the original implementation for acceleration. **Dynamic pruning**, conversely, skips computations on-the-fly during inference. For example, token-wise dynamic pruning employs online routing to determine which computations to skip for each token, which often requires customized kernels for real-world batch serving and usually does not directly reduce the memory footprint (Jiang et al., 2024; Zhao et al., 2025a; Ding et al., 2026). These categories describe pruning granularity and deployment behavior, but they do not reveal which GEMM dimensions are reduced. Since such dimension-level changes directly affect propagation and deployable speedup, we next reinterpret LLM pruning through the M, N, and K axes of GEMM.

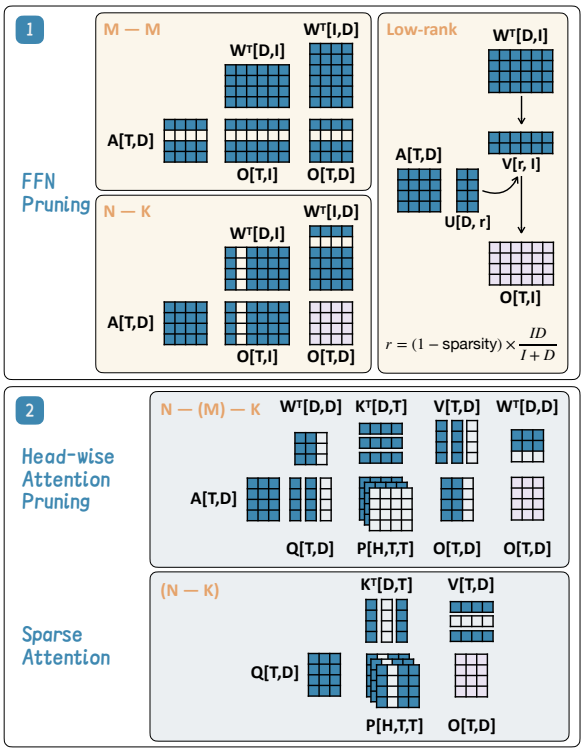

Figure 3: An overview of **MNK**-dimension pruning and their propagation through attention and FFN layers. The **empty** blocks denote zero-filled elements that make no contributions to the computation, **lavender** blocks denote deviations compared to the dense results. The head-wise pruning can also be treated as **M**-dimension pruning over each head with head dim set to 1, where sparse attention can be treated as pruning over key and value **N**, **K** logical dimension, respectively.

3 A GEMM-Centric View

Viewed through GEMM operations, diverse pruning methods exhibit recurring dependency patterns despite their differences in target modules, atomic pruning units, and affected dimensions. In particular, the dimensions pruned in one GEMM can

Categories	Methods	Strategies	Raw Dimension		MNK-pruning Dimension					
			Attn.	FFN	Q&K&V	$QK^\top$	PV	O	Up & Gate	Down
Static **M**	ShortenedLLaMA	depth	layer		**M**	**M**	**M**	**M**	**M**	**M**
	BlockPruner	depth	layer	layer	**M**	**M**	**M**	**M**	**M**	**M**
Static **K**	SVD-LLM†	low-rank	col	col	**K**	–	–	**K**	**K**	**K**
	Dobi-SVD†	low-rank	col	col	**K**	–	–	**K**	**K**	**K**
	SparseGPT	semi-structured	col	col	**K**	–	–	**K**	**K**	**K**
	MaskLLM	semi-structured	col	col	**K**	–	–	**K**	**K**	**K**
Static **NK**	FLAP	width	head	row&col	**N**	**M**	**M**	**K**	**N**	**K**
	LLMPruner	width	head	row&col	**N**	**M**	**M**	**K**	**N**	**K**
	Týr-the-Pruner	width	head	row&col	**N**	**M**	**M**	**K**	**N**	**K**
	SliceGPT‡	width	row&col	row&col	**K**	–	–	**N**	**K**	**N**
Dynamic **M**	MoD	depth	layer		**M**	**M**	**M**	**M**	**M**	**M**
	SkipGPT	depth	layer	layer	**M**	**M**	**M**	**M**	**M**	**M**
Dynamic **NK**	SeerAttention	sparse attention	key&value		–	**N**	**K**	–	–	–
	BLASST	sparse attention	value		–	–	**K**	–	–	–

Table 1: Original pruning strategies, raw dimension, and corresponding pruned **MNK**-based dimension over each GEMM group. † for static **K** denotes the low-rank variant, and ‡ for static **NK** denotes the cross-layer variant.

determine how sparsity propagates to subsequent GEMMs within a layer. This motivates a unified GEMM-centric view for characterizing both inter-family differences and intra-family commonalities in implementation behavior.

In general, consider a GEMM formed as $\mathbf{O} = \mathbf{A}\mathbf{W}^\top$ with input tensor $\mathbf{A} \in \mathbb{R}^{\mathbf{T}\times\mathbf{D}}$, weight $\mathbf{W} \in \mathbb{R}^{\mathbf{I}\times\mathbf{D}}$, and output tensor $\mathbf{O} \in \mathbb{R}^{\mathbf{T}\times\mathbf{I}}$. The token dimension **T**, output feature dimension **I**, and input feature dimension **D** correspond to the logical GEMM dimensions **M**, **N**, and **K**, respectively. Although pruning may change the physical shape [**T, I, D**], this correspondence remains unchanged. Following this mapping, Table 1 summarizes representative pruning methods by their original pruning units and corresponding MNK dimensions.

Properties of MNK-dimension Pruning. Different pruning dimensions affect not only individual operators, but also the computational flow across an entire layer and even the whole model. For structured zeroing-based pruning, the pruning effect can be summarized in a mask-based manner:

$$\mathbf{M}: \begin{cases} \mathbf{O} = (\mathbf{A} \odot \mathcal{M})\mathbf{W}^\top = (\mathbf{A}\mathbf{W}^\top) \odot \mathcal{M} \\ \text{where } \mathcal{M} \in \{0,1\}^{T\times 1} \end{cases}$$

$$\mathbf{N}: \begin{cases} \mathbf{O} = \mathbf{A}(\mathbf{W}^\top \odot \mathcal{M}) = (\mathbf{A}\mathbf{W}^\top) \odot \mathcal{M} \\ \text{where } \mathcal{M} \in \{0,1\}^{1\times I} \end{cases}$$

$$\mathbf{K}: \begin{cases} \mathbf{O} = (\mathbf{A} \odot \mathcal{M})\mathbf{W}^\top = \mathbf{A}(\mathbf{W} \odot \mathcal{M})^\top \\ \text{where } \mathcal{M} \in \{0,1\}^{1\times D} \end{cases}$$

Here, each mask is broadcast along its length-1 dimension. A key distinction follows immediately: unlike the **M** and **N** dimensions, **K**-dimension pruning acts on the reduction dimension and therefore has no direct masking equivalent on **O**.

Now consider a subsequent GEMM that takes **O** as input, namely $\mathbf{O}\hat{\mathbf{W}}^\top$ with $\hat{\mathbf{W}} \in \mathbb{R}^{\mathbf{D}\times\mathbf{I}}$. For **M** and **N** dimension pruning, the masked output becomes $((\mathbf{A}\mathbf{W}^\top) \odot \mathcal{M})\hat{\mathbf{W}}^\top$. In the **N**-pruning case, the mask applied to the out-feature of the first GEMM becomes a mask on the in-feature dimension of the next GEMM; that is, it reappears as **K**-dimension pruning in the subsequent operator. This yields the propagation behavior shown in Figure 3: ***M**-dimension pruning persists across consecutive GEMMs; **N**-dimension pruning propagates as **K**-dimension pruning in the next GEMM, forming an **NK** pattern; **K**-dimension pruning exhibits no quantifiable propagation.*

Low-rank pruning differs from the zeroing-based cases above in that it compresses the weight matrix through low-rank factorization rather than explicit masking. Although the output **N**-dimension remains unchanged, the reduced latent dimension lowers computation without structural propagation across subsequent GEMMs. We therefore treat it as a variant of static **K**.

Overall, this GEMM-centric view shows why nominal sparsity is an unreliable predictor of realized acceleration: pruning dimensions differ in the computations they remove and the implementation constraints they impose, motivating our unified implementation and benchmarking framework.

4 Inference Implementation

After establishing the pruning taxonomy, we study how each sparsity pattern translates into practical

acceleration. For generality and fair comparison, we build a taxonomy-level acceleration framework rather than method-specific implementations. The framework provides a unified operator-replacement interface, reusable optimization patterns for each pruning family, and specialized kernels written in high-level DSLs such as Triton (Tillet et al., 2019) and TileLang (Wang et al., 2025).

Static K. For static **K**, the implementation is straightforward since neither method causes structural propagation. In both cases, we replace the original operator with a weight-level branch, implemented via low-rank factorization or Sparse Matrix Multiplication (SpMM). Moreover, the naive PyTorch implementation of semi-structured sparsity incurs substantial CPU overhead, so we replace it with a JIT interface that directly invokes cuSPARSELt, reducing initialization latency by 94.4% ($726\mu s \rightarrow 40\mu s$) with improved architecture compatibility. We further run auto-tuning for each logical **M** size to select the best SpMM configuration (Appendix B).

Static M and NK. For static methods with structural propagation, the key implementation issue is to maintain dimension consistency along dependent operators, rather than to redesign kernels. Static **M** pruning removes entire layers or sublayers, whereas static **NK** pruning couples consecutive modules through **N**-to-**K** propagation. We thus implement them with a unified wrapper that rewires each dependent module group jointly. Nevertheless, not all static **NK** share the same path. The cross-layer static **NK**, which is represented by SliceGPT (Ashkboos et al., 2023), begins at token embedding and attention output projection, extends to the LM head, but it also introduces an extra residual projection and excluding propagation through attention computations (Table 1). Accordingly, we apply pruning in a coarse-to-fine order and treat each coupled group as a single unit. The overall pipeline is given as Algorithm 1. Here, the GEMMs outside the attention are divided into four groups by data dependencies (Table 1), where the GEMMs in each group share the same pruning dimension.

Dynamic M and NK. For dynamic pruning, we convert online sparsity into speedup through mask preprocessing and intra-kernel skipping, avoiding method-specific pipeline abstraction. Dynamic **M** and dynamic **NK** thus share a tile-based execution principle: the former derives sparse masks from token-wise routing, while the latter derives them from KV-page masking. For dynamic **M**, a naive implementation performs mask-to-indices conversion and gather–scatter fusion with indices $\mathcal{I}$[2]:

$$\mathcal{K}(X, \mathbf{W}, \mathcal{I}) = \mathrm{Scatter}_{\mathcal{I}}(\mathrm{OP}(\mathrm{Gather}_{\mathcal{I}}(X), \mathbf{W}))$$

Thus an additional mask-reordering preprocessing scheme is critical for such tile-based intra-kernel skipping (Figure 4). Beyond these hardware-agnostic optimizations, several decoding-specific optimizations are also integrated to ensure a fair comparison. Details are provided in Appendix B.

Algorithm 1: Propagated Structure Pruning

Input : model $\mathcal{M}$, pruning config $\mathcal{P}$
Output : pruned model $\mathcal{M}'$
foreach *layer* ℓ *in* $\mathcal{M}$ **do**
  foreach *sublayer* $s \in \{Attention, MLP\}$ **do**
    if $\textsc{ShouldPruneM}(s_\ell, \mathcal{P})$ **then**
      $\textsc{PruneM}(\mathcal{M}, s_\ell)$;
$\mathcal{C} \leftarrow \textsc{GetGroups}(\mathcal{M}, \mathcal{P})$;
if $\textsc{CrossLayer}(\mathcal{P})$ **then**
  $\mathcal{C} \leftarrow [embedding,\ \mathcal{C},\ lm\ head]$;
foreach (src, dst) *in* $\mathcal{C}$ **do**
  if $\textsc{Prunable}(src, dst, \mathcal{P})$ **then**
    $\mathcal{I} \leftarrow \textsc{GetIndices}(src, dst, \mathcal{P})$;
    $\textsc{PruneNK}(\mathcal{M}, src, dst, \mathcal{I})$;
    if $\textsc{CrossLayer}(\mathcal{P})$ *and* $\textsc{HasResidual}(src, dst)$ **then**
      $\textsc{AddProjection}(src, \mathcal{M})$;
return $\mathcal{M}$;

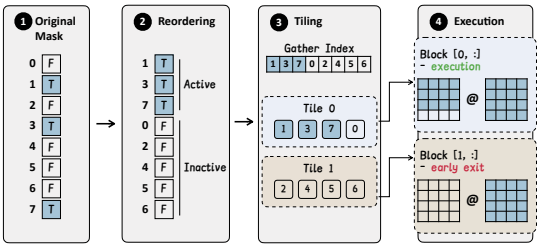

Figure 4: The execution pipeline of dynamic **M**, with the number of tokens set to 8, and M-axis tile size to 4, blocks that contain any active tokens will be executed.

[2]Mask-to-indices conversion triggers dynamic memory allocation at runtime, which breaks the static graph.

Taxonomy	25% sparsity				50% sparsity			
	WikiText2	Acc. Gap %	Prefill Speedup	Decode Speedup	WikiText2	Acc. Gap %	Prefill Speedup	Decode Speedup
Dense	7.54	0.00	1.00x [1.00x, 1.00x]	1.00x [1.00x, 1.00x]	7.54	0.00	1.00x [1.00x, 1.00x]	1.00x [1.00x, 1.00x]
Static **M**	15.52	10.59	**1.29x** [1.28x, 1.34x]	**1.32x** [1.31x, 1.34x]	33.93	30.28	**1.88x** [1.85x, 1.92x]	**1.91x** [1.89x, 1.94x]
Static **K**†	10.14	<u>6.52</u>	1.15x [1.11x, 1.19x]	1.09x [0.97x, 1.23x]	15.44	19.83	1.43x [1.31x, 1.53x]	1.46x [1.38x, 1.57x]
Static **K**	–	–	–	–	<u>11.45</u>	**10.67**	1.08x [1.03x, 1.20x]	1.14x [1.08x, 1.26x]
Static **NK**	12.46	12.63	<u>1.27x</u> [1.24x, 1.35x]	<u>1.27x</u> [1.23x, 1.39x]	19.80	26.10	<u>1.77x</u> [1.62x, 1.84x]	<u>1.70x</u> [1.62x, 1.84x]
Static **NK**‡	38.96	21.16	1.13x [1.12x, 1.16x]	1.08x [1.02x, 1.17x]	68.71	38.36	1.59x [1.37x, 1.76x]	1.46x [1.26x, 1.59x]
Dynamic **M**	<u>9.25</u>	**3.94**	1.08x [0.98x, 1.12x]	0.91x [0.75x, 1.05x]	13.14	<u>15.44</u>	1.44x [1.13x, 1.56x]	1.10x [0.76x, 1.44x]
Dynamic **NK**∗	**7.72**	33.40	1.02x [1.00x, 1.06x]	1.02x [1.00x, 1.04x]	**7.76**	34.45	1.05x [0.98x, 1.20x]	1.04x [1.01x, 1.10x]

Table 2: Overview of WikiText2 perplexity, average accuracy gap, and speedup over the dense model under different pruning taxonomies. Throughput speedups are reported in *mean [min, max]*. The 1st and 2nd results are highlighted in **bold** and <u>underline</u>, respectively. ∗ for dynamic **NK** denotes to force 25%/50% sparsity in all tasks.

5 Experiments

This section studies the practical deployment behavior of different pruning methods beyond nominal sparsity. We organize the evaluation around three questions: *(1) which pruning taxonomies form the throughput–quality Pareto frontier; (2) how realized prefill and decode speedups differ under a unified setup; and (3) where the gap between theoretical acceleration bounds and realized speedup comes from.* Finally, we distill these findings into taxonomy-level implications, outlining the practical role, current bottlenecks, and remaining optimization headroom of each pruning family.

5.1 Experimental Setup

We select Llama3.1-8B (Grattafiori et al., 2024) and Qwen3-4B (Yang et al., 2025) as the baseline, then apply a unified LoRA (Low-rank Adaptation) (Hu et al., 2021) fine-tuning recipe to each method after calibration. Unless otherwise noted, training runs for 5,000 steps with batch size 16, sequence length 2,048, and LoRA rank 16 on a subset of RedPajama (Weber et al., 2024). For dynamic **M**, we extend training to 10,000 steps for convergence. For static **K**, continued tuning is infeasible because its semi-structured sparsity pattern does not support vanilla LoRA merging. At each sparsity level, we select the method with the best LoRA performance as the representative of its taxonomy. More details about the baseline selection are provided in Appendix B.

For throughput simulation, the hardware is set to RTX Pro 6000 Blackwell. All profiling experiments are integrated with CUDA graph to minimize CPU-side overhead. For each pruning strategy, we randomly sample their atomic units to simulate based on target sparsity budget. Throughput is computed as the total number of processed tokens

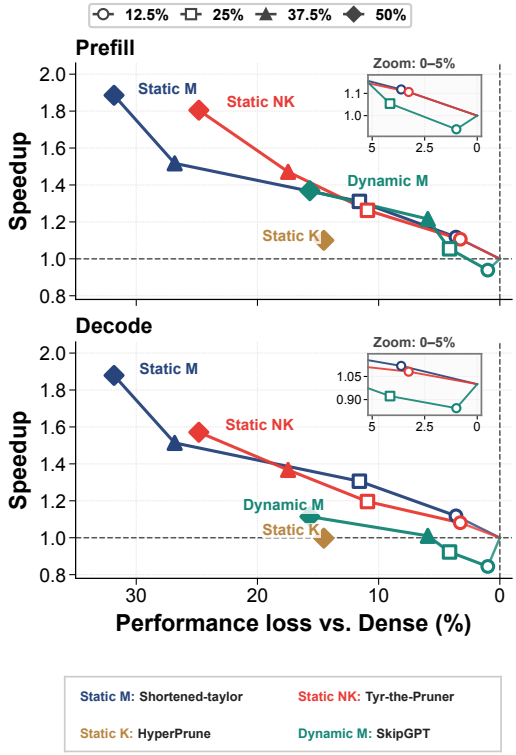

Figure 5: Throughput–quality trade-off of different pruning methods at 12.5%/25%/37.5%/50% sparsity in Qwen3-4B, with compared methods listed below.

divided by the elapsed wall-clock time:

$$\text{Token/s} = \frac{\mathbf{B} \times \mathbf{T}_q}{\textbf{TTFT or TPOT}}$$

where $\mathbf{B}$ is the batch size, $\mathbf{T}_q$ is the number of tokens for each input sequence, TTFT (Time-to-First-Token) and TPOT (Time-Per-Output-Token) denote the execution time. Additionally, all width-pruned dimensions are aligned to multiples of 16 to ensure maximum speedup (Table 5 and 6).

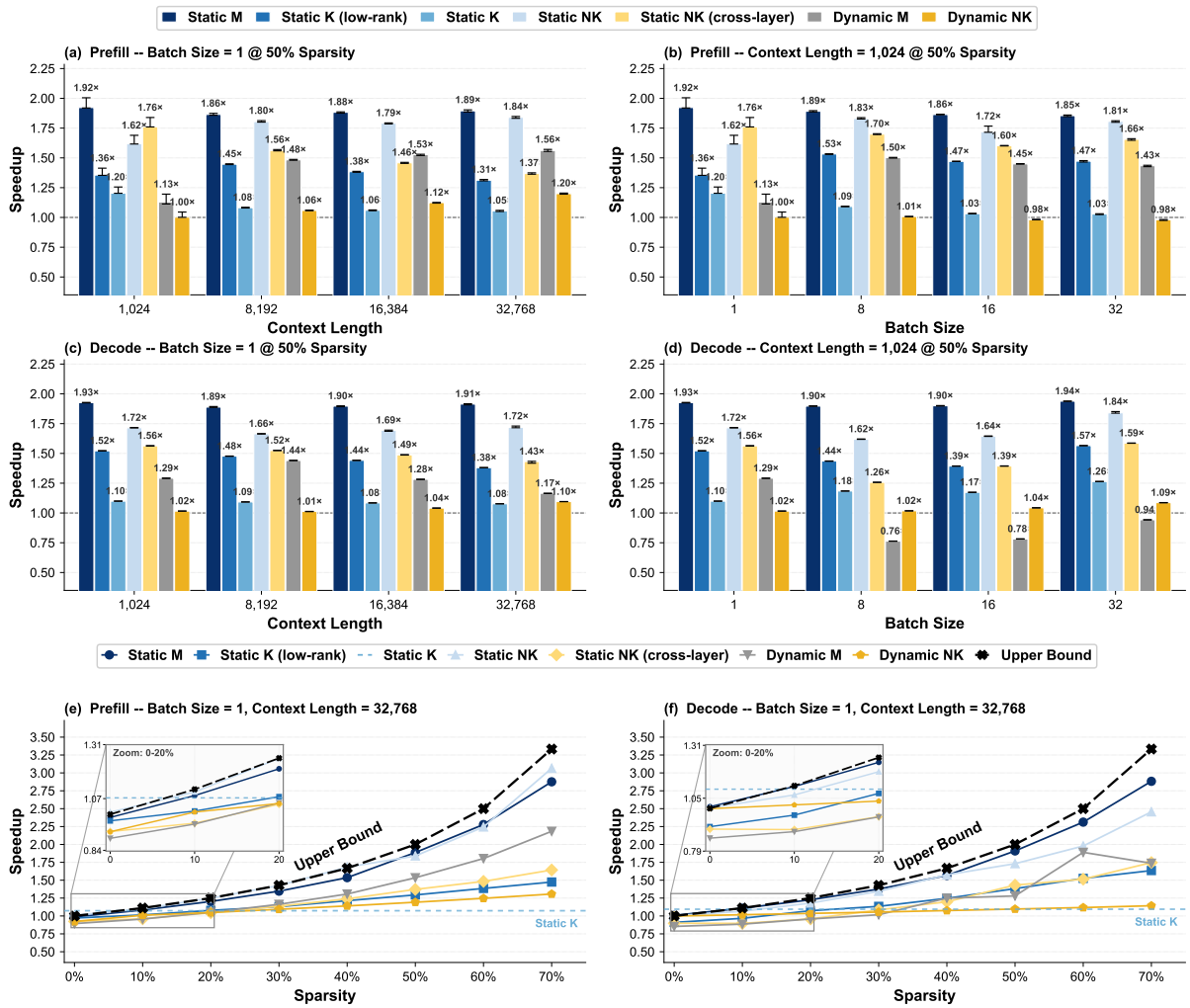

Figure 6: (a-d) Prefill and decode speedup on Llama3.1-8B under 50% sparsity. (e-f) Speedup over different sparsity, and their gap with the theoretical upper bound. Context in decoding denotes the KV cache length.

5.2 Throughput–Quality Trade-off

> **Finding 1:** The Pareto frontier shifts with pruning taxonomy and quality budget: static **M** dominates low loss, dynamic **M** leads moderate loss, and static **NK** becomes competitive at higher loss.

The results reveal that pruning does not yield a single universal throughput–quality frontier; instead, the frontier shifts with pruning taxonomy and quality budget. As shown in Figure 1, Table 2, and Table 12, static **M** provides the strongest trade-off at the low-loss end, achieving a 1.12x speedup across evaluated scenarios with only 2.76% performance loss under 12.5% sparsity. As the quality budget relaxes, dynamic **M** is most competitive in the moderate-loss regime, improving prefill speedup to 1.24x∼1.44x at roughly 5%∼16% loss while remaining more quality-efficient than higher-sparsity static baselines. At larger loss levels, static **NK** becomes the strongest width-pruning frontier method, reaching 1.51x speedup at 16.97% loss and 1.77x at 26.10% loss. By contrast, the two static **K** variants, static **NK** (cross-layer), and dynamic **NK** remain off the main frontier due to weaker realized speedup, poorer quality retention, or both. The attention-only dynamic **NK** preserves quality mainly in long-context settings, such as WikiText2. Overall, these results show that practical acceleration is determined not by sparsity alone, but by pruning taxonomy and how effectively its induced structure translates into realized throughput.

To assess whether the Pareto frontier remains stable across models, we further evaluate Qwen3-4B, focusing on the three pruning taxonomies that reach the frontier on Llama3.1-8B: static **M**, dynamic **M**, and static **NK**. As shown in Figure 5, the prefill frontier transitions from static **M** at 0∼5% performance loss, to dynamic **M** at 6∼10%, and then to static M (11∼13%) and static **NK** (13∼24%), similar to the pattern observed on Llama3.1-8B.

For decoding, static **M** dominates the 0∼18% loss range, whereas static **NK** leads at 19∼24%. Thus, despite several model-dependent shifts, the overall pattern remains consistent across models. Within the models and methods evaluated, each pruning taxonomy therefore occupies a distinct regime in which it offers the best throughput–quality trade-off.

5.3 Prefill and Decode Throughput

> **Finding 2:** Static **M** provides the most stable speedup across prefill and decode, while width and dynamic pruning show substantially different realized gains even under the same sparsity budget.

As shown in Figures 6 and 11, a consistent pattern across both prefill and decode is that static methods provide more stable acceleration than dynamic ones. Among them, static **M** remains the strongest baseline throughout, showing that **M**-axis pruning translates most directly into realized end-to-end gains. At 50% sparsity in the prefill stage, the average speedups of static **K** (low-rank), static **NK**, static **NK** (cross-layer), and dynamic **M** are only comparable to those achieved by static **M** at 34%, 48%, 39%, and 38% sparsity, respectively. The same pattern holds in decoding, where static **K** (low-rank), static **NK**, static **NK** (cross-layer), and dynamic **M** at 50% sparsity match only the decoding gains of static **M** at 34%, 45%, 37%, and 23% sparsity. Width pruning therefore still trails depth pruning by a clear margin even under the same sparsity budget, and this gap is not uniform across width families: static **NK** is consistently the strongest, whereas static **K** and static **NK** (cross-layer) realize substantially smaller gains, especially in attention-dominated long-context scenarios. These results suggest that the main divide is not simply static versus dynamic or depth versus width, but how the pruning pattern is organized and how effectively it removes executable computation in the end-to-end inference pipeline.

5.4 Latency Breakdown

> **Finding 3:** The gap between theoretical and realized acceleration comes from inefficiency in the GEMMs and taxonomy-specific non-GEMM overhead, which becomes especially critical during decoding.

To further analyze how different pruning strategies affect the LLM forward trace, we measure the latency breakdown of core GEMM and remaining non-GEMM operations. Figure 7 shows that the gap between theoretical and realized acceleration cannot be explained by pruned GEMM throughput alone. In particular, dynamic **M** and static **K** both incur substantial auxiliary overheads beyond the main compute kernels. Their non-GEMM overhead increases by 42.4% and 40.8% in prefill, and by 61.5% and 287.2% in decode, respectively, directly offsetting part of their nominal compute reduction[3]. Static **NK** (cross-layer) is similarly affected by its extra residual projection path, while static **K** (low-rank) also pays additional launch overhead from factorizing one GEMM into two smaller ones. These two sources increase decoding non-GEMM overhead by 48.5% and 30.9%, respectively. Meanwhile, the core pruned operators themselves still differ in realized efficiency: dynamic **M** and static **K** remain noticeably below their apparent kernel-level headroom, static **K** (low-rank) reaches only about 1.66× GEMM speedup in decoding, and dynamic **NK** still trails static **NK** in attention throughput under the same sparsity budget (Table 10).

Overall, dynamic **M** and static **K** remain the two methods furthest from their apparent upper bounds, as both are limited by non-GEMM overheads and GEMM optimization. By contrast, static **K** (low-rank) and dynamic **NK** are constrained more by the efficiency of their pruned GEMM operation. These results show that the deployable acceleration of a pruning method depends not only on how much GEMM work it removes, but also on the extra non-GEMM cost it introduces along the pipeline.

5.5 Taxonomy-Level Implications

The above results reveal distinct roles and bottlenecks across pruning taxonomies:

Static M is the current throughput anchor. Its coarse-grained pruning pattern largely explains why it remains closest to the speedup bound and delivers stable gains across serving scenarios. The same rigidity, however, also constrains its quality at higher sparsity, especially beyond 37.5%.

Dynamic M is a promising quality-aware direction. Its token-level routing adaptively maintains model quality well, while this dynamism also introduces challenges for acceleration, especially for decoding's non-GEMM overhead.

Static NK reaches the width-pruning frontier so far. Its finer-grained neuron pruning yields a more favorable quality profile than static **M** at high

[3] Sparse tensor cores only support $\mathbf{AB}^\top$ with $\mathbf{A}$ in 2:4 sparse format, which requires a $(\mathbf{BA}^\top)^\top$ execution form.

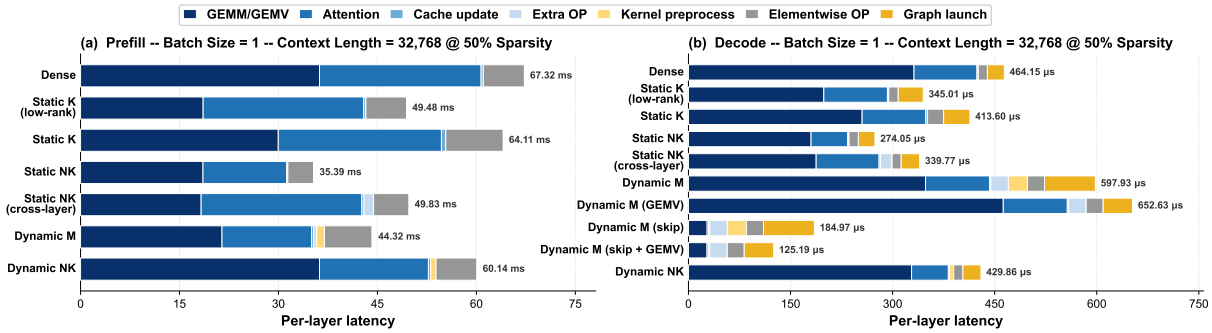

Figure 7: Breakdown of wall-clock time for all operations in one layer. The *extra operation* includes overheads apart from the vanilla GEMM/attention/element-wise kernels (e.g., residual projection and routing), where *kernel preprocess* includes overheads before the core GEMM/attention kernel execution (e.g., mask reordering).

sparsity, while the vast pruning search space prevents current methods from fully exploiting this advantage, especially in the low-sparsity regime.

Other width-pruning methods remain quality-preserving but not frontier-ready. Static **K** generally retain quality well, while static **NK** (cross-layer) further covers the LM head. In practice, however, their limitations in non-GEMM overhead and GEMM-side optimization are non-negligible.

Dynamic NK is currently a long-context-specialized direction. It mainly captures attention-side sparsity and therefore becomes attractive only when the context is sufficiently long.

6 Conclusion

This work presents a GEMM-centric view of LLM pruning that organizes diverse pruning methods through the logical **M**, **N**, and **K** dimensions and their propagation behavior across Transformer computation. Based on this view, we build a unified and reusable benchmarking framework, while evaluating representative pruning taxonomies under fixed sparsity budgets from the perspectives of downstream quality, realized prefill/decode throughput, and kernel-level latency breakdown.

Our results show that nominal sparsity alone is a weak predictor of deployable acceleration. In practice, realized speedup is jointly determined by pruning taxonomy, structural propagation, operator coverage, and system overheads introduced during execution. More broadly, these findings suggest that the next Pareto frontier will likely come not from pushing a single taxonomy in isolation, but from hybrid designs that combine a quality-aware branch with structured and static backbone.

Limitations

Our study focuses on taxonomy-level inference behavior rather than exhaustive deployment coverage. The main experiments are conducted on Llama3.1-8B with RTX Pro6000 (sm120), with additional validation on Qwen3-14B and A800 (sm80) in the appendix. Although the trends are consistent across these settings, several limitations remain.

First, we do not evaluate Mixture-of-Experts (MoE) models or other accelerator platforms such as data-center Blackwell (sm100). These settings may require different execution pipelines, including warp-specialized kernels, and may shift the relative advantages of dynamic **M** and dynamic **NK**. Second, our Triton and TileLang implementations are designed as portable baselines and may underperform hand-written CUDA kernels or hardware-specific implementations using features such as Tensor Memory Accelerator (TMA) and async tensor core instructions (e.g., `tcgen05.mma`). Third, we benchmark representative recent methods for each taxonomy rather than exhaustively covering all pruning variants. Therefore, the reported Pareto frontier should be interpreted as the current frontier under our selected methods, models, and benchmark settings, rather than a definitive upper envelope over all possible pruning methods. Finally, our measurements focus on single-step prefill/decode latency and kernel-level behavior, rather than full serving workloads in production inference frameworks such as SGLang. As a result, our findings provide deployment-relevant evidence at the model-step and kernel levels, while system-level scheduling, distributed serving, and production integration remain important directions for future work.

References

Yongqi An, Xu Zhao, Tao Yu, Ming Tang, and Jinqiao Wang. 2023. Fluctuation-based Adaptive Structured Pruning for Large Language Models. https://arxiv.org/abs/2312.11983v1.

Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. 2023. SliceGPT: Compress Large Language Models by Deleting Rows and Columns. In *The Twelfth International Conference on Learning Representations*.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439.

Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. 2024. A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):10558–10578.

Hung-Yueh Chiang, Bokun Wang, and Diana Marculescu. 2025. ELANA: A Simple Energy and Latency Analyzer for LLMs. http://arxiv.org/abs/2512.09946. *Preprint*, arXiv:2512.09946.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Longwei Ding, Anhao Zhao, Fanghua Ye, Ziyang Chen, and Xiaoyu Shen. 2026. From llms to lrms: Rethinking pruning for reasoning-centric models. *Preprint*, arXiv:2601.18091.

Xuan Ding, Pengyu Tong, Ranjie Duan, Yunjian Zhang, Rui Sun, and Yao Zhu. 2025. Pruning as a Cooperative Game: Surrogate-Assisted Layer Contribution Estimation for Large Language Models. In *The Fourteenth International Conference on Learning Representations*.

Yingqi Fan, Junlong Tong, Anhao Zhao, and Xiaoyu Shen. 2026. What do visual tokens really encode? uncovering sparsity and redundancy in multimodal large language models. *Preprint*, arXiv:2603.00510.

Yingqi Fan, Anhao Zhao, Jinlan Fu, Junlong Tong, Hui Su, Yijie Pan, Wei Zhang, and Xiaoyu Shen. 2025. VisiPruner: Decoding discontinuous cross-modal dynamics for efficient multimodal LLMs. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18885–18902, Suzhou, China. Association for Computational Linguistics.

Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. 2024. MaskLLM: Learnable Semi-Structured Sparsity for Large Language Models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *Proceedings of the 40th International Conference on Machine Learning*, pages 10323–10337. PMLR.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, and 5 others. 2024. The language model evaluation harness.

Yizhao Gao, Shuming Guo, Shijie Cao, Yuqing Xia, Yu Cheng, Lei Wang, Lingxiao Ma, Yutao Sun, Tianzhu Ye, Li Dong, Hayden Kwok-Hay So, Yu Hua, Ting Cao, Fan Yang, and Mao Yang. 2025a. Sparse Attention Adaptation for Long Reasoning. In *The Fourteenth International Conference on Learning Representations*.

Yizhao Gao, Zhichen Zeng, DaYou Du, Shijie Cao, Peiyuan Zhou, Jiaxing Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. 2025b. SeerAttention: Self-distilled Attention Gating for Efficient Long-context Prefilling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The Llama 3 Herd of Models. http://arxiv.org/abs/2407.21783. *Preprint*, arXiv:2407.21783.

Chao Han, Yijuan Liang, Zihao Xuan, Daokuan Wu, Wei Zhang, and Xiaoyu Shen. 2025. Informed routing in llms: Smarter token-level computation for faster inference. *Preprint*, arXiv:2510.13831.

Shwai He, Guoheng Sun, Zheyu Shen, and Ang Li. 2024. What Matters in Transformers? Not All Attention is Needed. http://arxiv.org/abs/2406.15786. *Preprint*, arXiv:2406.15786.

Zicheng He, Anhao Zhao, Xiaoyu Shen, Chen Wu, and Lei He. 2026. Skipopu: An fpga-based overlay processor for large language models with dynamically allocated computation. *Preprint*, arXiv:2603.14785.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

Haozhe Hu, Hao Wu, Peiran Yin, Chao Han, Yunpu Ma, and Xiaoyu Shen. 2026. WIDE: Boosting Adaptive LLM Inference via Token-level Dynamic Width Pruning. http://arxiv.org/abs/2607.28418. *Preprint*, arXiv:2607.28418.

Ming Hu, Chenglong Ma, Wei Li, Wanghan Xu, Jiamin Wu, Jucheng Hu, Tianbin Li, Guohang Zhuang, Jiaqi Liu, Yingzhou Lu, Ying Chen, Chaoyang Zhang, Cheng Tan, Jie Ying, Guocheng Wu, Shujian Gao, Pengcheng Chen, Jiashi Lin, Haitao Wu, and 101 others. 2025. A Survey of Scientific Large Language Models: From Data Foundations to Agent Frontiers. http://arxiv.org/abs/2508.21148. *Preprint*, arXiv:2508.21148.

Yikun Jiang, Huanyu Wang, Lei Xie, Hanbin Zhao, Chao Zhang, Hui Qian, and John C. S. Lui. 2024. D-LLM: A Token Adaptive Computing Resource Allocation Strategy for Large Language Models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Bo-Kyeong Kim, Geonmin Kim, Tae-Ho Kim, Thibault Castells, Shinkook Choi, Junho Shin, and Hyoung-Kyu Song. 2024. Shortened LLaMA: Depth Pruning for Large Language Models with Comparison of Retraining Methods. http://arxiv.org/abs/2402.02834. *Preprint*, arXiv:2402.02834.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. http://arxiv.org/abs/2309.06180. *Preprint*, arXiv:2309.06180.

Guanchen Li, Yixing Xu, Zeping Li, Ji Liu, Xuanwu Yin, Dong Li, and Emad Barsoum. 2025a. Týr-the-Pruner: Structural Pruning LLMs via Global Sparsity Distribution Optimization. http://arxiv.org/abs/2503.09657. *Preprint*, arXiv:2503.09657.

Guangyan Li, Yongqiang Tang, and Wensheng Zhang. 2024. LoRAP: Transformer Sub-Layers Deserve Differentiated Structured Compression for Large Language Models. In *Forty-First International Conference on Machine Learning*.

Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, Yu Wang, and Guohao Dai. 2025b. Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective. http://arxiv.org/abs/2410.04466. *Preprint*, arXiv:2410.04466.

Wenjie Liu, Hao Wu, Xin Qiu, Yingqi Fan, Yihan Zhang, Anhao Zhao, Yunpu Ma, and Xiaoyu Shen. 2026. Vica: Efficient multimodal llms with vision-only cross-attention. *Preprint*, arXiv:2602.07574.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. In *Thirty-Seventh Conference on Neural Information Processing Systems*.

Xin Men, Mingyu Xu, Qingyu Zhang, Qianhao Yuan, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. 2025. ShortGPT: Layers in Large Language Models are More Redundant Than You Expect. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20192–20204, Vienna, Austria. Association for Computational Linguistics.

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

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*.

Wang Qinsi, Jinghan Ke, Masayoshi Tomizuka, Kurt Keutzer, and Chenfeng Xu. 2025. Dobi-SVD: Differentiable SVD for LLM Compression and Some New Perspectives. In *The Thirteenth International Conference on Learning Representations*.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. 2024. Mixture-of-Depths: Dynamically allocating compute in transformer-based language models. http://arxiv.org/abs/2404.02258. *Preprint*, arXiv:2404.02258.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106.

Dmitriy Shopkhoev, Ammar Ali, Magauiya Zhussip, Valentin Malykh, Stamatios Lefkimmiatis, Nikos Komodakis, and Sergey Zagoruyko. 2025. ReplaceMe: Network Simplification via Depth Pruning and Transformer Block Linearization. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Lu Sun and Jun Sakuma. 2026. Learning Semi-Structured Sparsity for LLMs via Shared and Context-Aware Hypernetwork. In *The Fourteenth International Conference on Learning Representations*.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2023. A Simple and Effective Pruning Approach for Large Language Models. In *The Twelfth International Conference on Learning Representations*.

Yutao Sun, Zhenyu Li, Yike Zhang, Tengyu Pan, Bowen Dong, Yuyi Guo, and Jianyong Wang. 2026. Efficient Attention Mechanisms for Large Language Models: A Survey. http://arxiv.org/abs/2507.19595. *Preprint*, arXiv:2507.19595.

Yang Tan, Junlong Tong, Linan Yue, Hao Wu, Pengfei Fang, and Xiaoyu Shen. 2026. Vicostream: Streaming videollms can run beyond 100 fps with stage-wise coordinated inference. *arXiv preprint arXiv:2606.19849*.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, and 307 others. 2026. Kimi K2.5: Visual Agentic Intelligence. http://arxiv.org/abs/2602.02276. *Preprint*, arXiv:2602.02276.

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 10–19, New York, NY, USA. Association for Computing Machinery.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971.

Lei Wang, Yu Cheng, Yining Shi, Zhiwen Mo, Zhengju Tang, Wenhao Xie, Tong Wu, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2025. TileLang: Bridge Programmability and Performance in Modern Neural Kernels. In *The Fourteenth International Conference on Learning Representations*.

Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. 2024. SVD-LLM: Truncation-aware Singular Value Decomposition for Large Language Model Compression. In *The Thirteenth International Conference on Learning Representations*.

Maurice Weber, Daniel Y. Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. 2024. Redpajama: an open dataset for training large language models. *NeurIPS Datasets and Benchmarks Track*.

Hao Wu, Yingqi Fan, Dai Jinyang, Junlong Tong, Yunpu Ma, and Xiaoyu Shen. 2025. HiDrop: Hierarchical Vision Token Reduction in MLLMs via Late Injection, Concave Pyramid Pruning, and Early Exit. In *The Fourteenth International Conference on Learning Representations*.

Hao Wu, Junlong Tong, Xudong Wang, Yang Tan, Changyu Zeng, Anastasia Antsiferova, and Xiaoyu Shen. 2026a. From data to model: A survey of the compression lifecycle in mllms.

Hao Wu, Xudong Wang, Jialiang Zhang, Junlong Tong, Xinghao Chen, Junyan Lin, Yunpu Ma, and Xiaoyu Shen. 2026b. UTPTrack: Towards Simple and Unified Token Pruning for Visual Tracking. http://arxiv.org/abs/2602.23734. *Preprint*, arXiv:2602.23734.

Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. http://arxiv.org/abs/2309.10285. *Preprint*, arXiv:2309.10285.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 Technical Report. http://arxiv.org/abs/2505.09388. *Preprint*, arXiv:2505.09388.

Jiayi Yuan, Cameron Shinn, Kai Xu, Jingze Cui, George Klimiashvili, Guangxuan Xiao, Perkz Zheng, Bo Li, Yuxin Zhou, Zhouhai Ye, Weijie You, Tian Zheng, Dominic Brown, Pengbo Wang, Richard Cai, Julien Demouth, John D. Owens, Xia Hu, Song Han, and 2 others. 2025. BLASST: Dynamic BLocked Attention Sparsity via Softmax Thresholding. http://arxiv.org/abs/2512.12087. *Preprint*, arXiv:2512.12087.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. 2025. SpargeAttention: Accurate and Training-free Sparse Attention Accelerating Any Model Inference. In *Forty-Second International Conference on Machine Learning*.

Anhao Zhao, Fanghua Ye, Yingqi Fan, Junlong Tong, Jing Xiong, Zhiwei Fei, Hui Su, and Xiaoyu Shen. 2025a. SkipGPT: Each Token is One of a Kind. In *Forty-Second International Conference on Machine Learning*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025b. A Survey of Large Language Models. http://arxiv.org/abs/2303.18223. *Preprint*, arXiv:2303.18223.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. http://arxiv.org/abs/2312.07104. *Preprint*, arXiv:2312.07104.

Longguang Zhong, Fanqi Wan, Ruijun Chen, Xiaojun Quan, and Liangzhi Li. 2025. BlockPruner: Fine-grained Pruning for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 5065–5080, Vienna, Austria. Association for Computational Linguistics.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. 2024. A Survey on Efficient Inference for Large Language Models. http://arxiv.org/abs/2404.14294. *Preprint*, arXiv:2404.14294.

A The Use of Large Language Models

We employed large language models (LLMs) solely as general-purpose writing assistants for language refinement, including improving clarity, grammar, and style. Importantly, no LLM was involved in research ideation, methodological design, analysis, or result interpretation; the role of the LLM was limited to linguistic polishing. All substantive contributions originated from the authors. This ensured that the scientific content remained entirely authored by the researchers, while benefiting from improved academic writing quality.

B Implementation Details

Static K. Semi-structured static **K** pruning applies a 2:4 mask to the weight matrices, compressing them to 50% along the **K** dimension, while obtaining metadata to describe their original coordinates and then invoking sparse tensor core for computation, the bit count of metadata typically equals to the number of elements in the original matrix (2-bit per non-zero element). We followed PyTorch's official cuSPARSELt integration, using SGLang's TVM-FFI[4] JIT framework to directly integrate the cuSPARSELt library and bypass the CPU-side overhead in the pytorch implementation. An explicit cache is applied to automatically tune different matrix shapes and maximize reuse of cuSPARSELt's descriptors, thereby significantly reducing kernel preprocessing overhead and CPU-side costs during decoding. For the prefill phase, we used PyTorch's native interface for SpMM and kernel tuning. Notably, in the A800 tests, we found both implementations failed to properly trigger cuSPARSELt's split-k mode, which affect the performance of semi-structured sparsity during decoding on sm80 architecture.

Static K (low-rank). The low-rank variant is conceptually similar to semi-structured, which directly applies the weight-by-weight pruning without propagation. Given a fixed sparsity budget, we first determine the target rank for each matrix, and then monkey-patch the original weight and GEMM path with a low-rank factorization implemented as two consecutive matrix multiplications. Because this transformation reduces the effective intermediate dimension and further increases the number of GEMM launches, static **K** (low-rank) inevitably

[4] https://github.com/apache/tvm-ffi

falls short of the expected per-GEMM speedup and incurs additional graph launch overhead.

Static M, Static NK. We process propagated static structured pruning in a unified order: layer-wise pruning, attention/FFN sublayer pruning, followed by row-column pruning. For depth pruning, we replace the `forward` function of the corresponding layers with identity mappings. For width pruning, each consecutive module pair is treated as the minimum processing unit, to handle **N**-dimension pruning in the first module and **K**-dimension pruning for the second, while simultaneously managing associated element-wise operations (e.g., normalization and residuals). When propagating **NK** pruning to attention, we enforce head-level granularity: when the number of pruned query heads reaches the head group size, the corresponding key and value heads will also be removed as well.

Algorithm 2: Dynamic **M**'s GEMM kernel

Input : input matrix $\mathbf{A} \in \mathbb{R}^{M\times K}$, weight matrix $\mathbf{B} \in \mathbb{R}^{N\times K}$, routing mask $\mathbf{m} \in \{0,1\}^M$
Output : output matrix $\mathbf{D} \in \mathbb{R}^{M\times N}$
$(\tilde{\mathbf{m}}, \mathbf{idx}) \leftarrow \textsc{SortDescending}(\mathbf{m})$;
Initialize $\mathbf{D} \leftarrow \mathbf{0}$;
foreach *tile* (b_m, b_n) *in the output grid* **do**
  $\mathbf{m}^{(b_m)} \leftarrow \tilde{\mathbf{m}}[b_m \cdot B_M : (b_m+1)\cdot B_M]$;
  if $\textsc{All}(\mathbf{m}^{(b_m)}) = 0$ **then**
    continue;
  $\mathbf{r} \leftarrow \mathbf{idx}[b_m \cdot B_M : (b_m+1)\cdot B_M]$;
  $\mathbf{M}_{row} \leftarrow (\mathbf{m}^{(b_m)} = 1)$;
  $\mathbf{C} \leftarrow \mathbf{0}^{B_M\times B_N}$;
  for $k = 0$ **to** $\lceil K/B_K \rceil - 1$ **do**
    $\mathbf{A}_{tile} \leftarrow \textsc{GatherRows}(\mathbf{A}, \mathbf{r}, k)$;
    $\mathbf{B}_{tile} \leftarrow \textsc{LoadTile}(\mathbf{B}, b_n, k)$;
    $\mathbf{C} \leftarrow \mathrm{MMA}(\mathbf{A}_{tile}, \mathbf{B}_{tile}, \mathbf{C})$;
  `// Element-wise operation`
  $\textsc{ApplyEpilogueFuse}(\mathbf{C})$;
  $\textsc{ScatterRows}(\mathbf{D}, \mathbf{r}, \mathbf{C}, \mathbf{M}_{row})$;
return $\mathbf{D}$;

Dynamic M. For dynamic **M**, we implement the corresponding Flash Attention and GEMM kernels under Triton with autotune enabled. Following the approach in Figure 4, we reorder the routing mask to ensure tile-level processing for both skipped and executed blocks, thereby preserving the design of the original dense kernel as much as possible. Specifically, before the kernel enters the pipeline, it will perform a block-level reduction on routing mask to detect if there are any active tokens in the current block. Additionally, for decoding-stage optimization, the flash decoding, split-k GEMM, and GEMV dispatch are also integrated into the attention and GEMM kernels, which can be automatically triggered when the number of blocks is less than GPU's streaming multiprocessors (SMs), with a heuristic algorithm to minimize the waste of SMs. The online skipping schedule can be summarized as Algorithm 2. Additionally, since the key-value projection only occupies a small fraction of the total computation, applying a dynamic skip kernel to these two operations would negligible reduce the overall throughput, so we kept the original dense GEMM for these two operations.

Dynamic NK. For block-sparse sparse attention in dynamic **NK**, the overall implementation is similar to the attention part of dynamic **M**, with the main difference lying in kernel preprocessing, which requires to compute an approximate attention map for $\mathbf{QK}^\top$ and derives a fixed skipping matrix. The vanilla attention pipeline only needs to read the mask for the current tile at the first step to determine whether the tile should be skipped. Following the original SeerAttention (Gao et al., 2025b,a), we also implement the kernel with Tilelang, combined with flash decoding and GQA packing for better decoding performance. Meanwhile, since the original SeerAttention updates the compressed KV cache block-by-block (e.g., every 64 generated tokens), the averaged update overhead per step is less than 1%, so only the gating computation is considered during profiling.

The Selection of DSL. As discussed above, we use Triton and Tilelang to implement dynamic **M** and dynamic **NK**, respectively, both are tile-based Python DSLs. Triton is built entirely around block-level pointer programming, with the block as its smallest control granularity, while providing a relatively unified abstraction over load, store, and compute operations through an LLVM-based backend. In contrast, Tilelang uses TVM as its backend and supports both block-level execution and finer-grained thread-level operations. It also exposes hardware-specific interfaces, such as shared memory, and makes different load, store, and compute primitives more explicit. Overall, Tilelang outperforms Triton in both CPU-side overhead and kernel throughput, which is especially evident in the

SeerAttention implementation (523.64 TFLOPS vs. 501.74 TFLOPS). However, it still has limitations in predicate operations and pipeline fusion. In particular, during the decoding stage of SeerAttention, enabling flash decoding does not support multi-stage pipelining, where the dynamic **M** kernels can only be implemented with Triton due to the incomplete optimization for bulk gather and scatter memory-access patterns.

Training Settings. For static **M**, static **K** (low-rank), static **NK**, static **NK** (cross-layer), and dynamic **M**, we train representative methods within each taxonomy from scratch and report the best-performing one as the taxonomy representative in downstream evaluation. All methods follow the original calibration strategy and training dataset. Because the LoRA fine-tuning dataset can substantially affect downstream performance, while several methods do not include an additional LoRA stage, we standardize LoRA fine-tuning on `RedPajama-Data-1T-Sample-subset850000`[5], using learning rate 2e-4, LoRA dropout 0.1, rank 16, and alpha 32. For Dobi-SVD in the static **K** (low-rank) category, we follow SVD-LLM (Wang et al., 2024) to train **U** first and **V** second, allocating half of the total budget (2,500 steps) to each stage. For static **K** and dynamic **NK** with high training overhead, we directly adopt the released Llama3.1-8B checkpoints. In particular, SeerAttention (Gao et al., 2025b) only releases a Llama3.1-8B-Instruct checkpoint. Since the dense instruct and base models perform nearly identically on our evaluation tasks, we directly reuse its trained attention gate for reference.

Evaluation Settings. We use the lm-evaluation-harness (Gao et al., 2024)[6] to conduct all downstream evaluations under a unified setup. The evaluation suite includes the commonly used benchmarks: WikiText2 (Merity et al., 2016), ARC-e, ARC-c (Clark et al., 2018), BoolQ (Clark et al., 2019), WinoGrande (Sakaguchi et al., 2021), PIQA (Bisk et al., 2020), OpenBookQA (Mihaylov et al., 2018), and HellaSwag (Zellers et al., 2019). All tasks are evaluated in the zero-shot setting with a maximum context length of 4,096. For WikiText2, we report word perplexity (PPL); for BoolQ and WinoGrande, we report accuracy (acc.); and for the remaining tasks, we report normalized accuracy (acc. norm). The overall quality loss and average accuracy metrics for each model only include the aforementioned 7 classification tasks, excluding WikiText2.

As for the baseline in each taxonomy, since exhaustively evaluating every pruning strategy is infeasible, we thus introduce a three-stage screening procedure:

1. Collect recent methods and their cited baselines that release code or checkpoints and can be mapped to our wrapper.
2. Evaluate all candidates at 25% and 50% sparsity under the same pruning and LoRA adaptation protocol.
3. Retain the best candidate in each taxonomy for the remaining sparsity levels.

Following this procedure, we evaluate Shortened-ppl, Shortened-taylor (Kim et al., 2024), Coop-Pruner (Ding et al., 2025), BlockPruner (Zhong et al., 2025), and ReplaceMe (Shopkhoev et al., 2025) for static **M**; MaskLLM (Fang et al., 2024), HyperPrune (Sun and Sakuma, 2026), and Dobi-SVD (Qinsi et al., 2025) for static **K**; FLAP (An et al., 2023), Týr-the-Pruner (Li et al., 2025a), LoRAP (Li et al., 2024), and SliceGPT (Ashkboos et al., 2023) for static **NK**; MoD (Raposo et al., 2024), DLLM (Jiang et al., 2024), and SkipGPT (Zhao et al., 2025a) for dynamic **M**; and SeerAttention (Gao et al., 2025b) for dynamic **NK**. MaskLLM provides only a Llama3.1-8B checkpoint, so we use HyperPrune for Qwen3-4B. SeerAttention likewise provides only a Llama3.1-8B checkpoint, while training Dobi-SVD incurs substantial computational cost; we therefore evaluate both methods only on Llama3.1-8B.

Profiling Settings. We build our unified benchmark framework for pruning methods following the overall design of ELANA (Chiang et al., 2025), the main different is that it will call a unified pruning interface for all models before profiling, then relies on Triton's built-in benchmarking utilities to flush the L2 cache, run repeated measurements, collect wall-clock times, and generate final reports. The CUDA graph interface is based on native PyTorch APIs, we record an additional forward trace under the given CUDA stream after warmup, and then replay the trace to record the wall-clock time in each step. To support CUDA graph execution and

[5] https://huggingface.co/datasets/konwoo/RedPajama-Data-1T-Sample-subset850000

[6] https://github.com/EleutherAI/lm-evaluation-harness

emulate the paged KV-cache update cost, a static-length dummy KV cache is allocated to replace the vanilla dynamic KV cache, thus the KV cache updating in each step is a directly in-place writing operation, avoiding the overhead of online memory allocation.

During the model pruning, the mask and indices are strictly generated by sorting and top-k selection to ensure a consistent distribution across pruning modules. For end-to-end throughput evaluation, we reinitialize each evaluated model three times to sample different pruning distributions and report the mean value over all results (i.e., 3 times resample with 50 times repeating per run). We perform 10 warm-up iterations before each set of 50 sample iterations. For kernel-level latency, we use the PyTorch profiler to sample 10 iterations and report the GPU-side execution time in middle layer (e.g., layer 16) per iteration. For the result in Table 5, we additionally perturb the target pruning ratio assigned to each module, producing non-uniform pruning ratios and unaligned dimensions. In all other experiments, we prune the attention and FFN modules with a unified sparsity ratio, avoiding confounding effects from different sparsity definitions, e.g., the fraction of removed dimensions for static **NK** and the fraction of skipped tokens for dynamic **M**.

All the profiling results are run with CUDA 12.8, PyTorch 2.9.1, and Transformers 4.57.1. Semi-structured sparsity uses cuSPARSELt 0.9.0, integrated through the sglang's JIT framework (0.5.8.post1). Custom kernels are implemented with Triton 3.6.0 and Tilelang 0.1.8 for full optimization. All the experiments on RTX Pro6000 are under bf16 precision, where the A800 runs in fp16 precision because sm80 do not support `atomic add` with bf16 precision. The evaluated models are Llama3.1-8B[7] and Qwen3-14B[8].

C Additional Experiments

Influence of Sparsity Distribution. For layer pruning, the distribution of sparsity across attention and FFN modules can affect the overall speedup at a given sparsity level, for Llama3.1-8B, the theoretical balance where attention and FFN have equal computational cost occurs at $T = 4D$ (sequence length $T = 16384$, hidden size $D = 4096$). When non-GEMM overheads are taken into account, Figure 8 and Figure 7 show that this balance point shifts to around 32k, where attention and FFN consume comparable wall-clock time. Since current pruning methods typically apply higher sparsity to attention than FFN layers (Zhao et al., 2025a), a long-context scenario can achieve better acceleration effects under the provided sparsity.

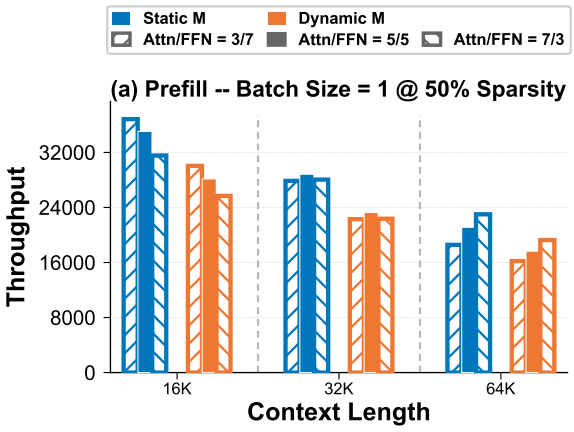

Figure 8: Prefill throughput with 50% sparsity and different pruning component distribution (Attention/FFN).

Attention-side acceleration for static K and NK (cross-layer). As shown in Figure 7, the vanilla static **K** (low-rank), static **K**, and static **NK** (cross-layer) only prune the main linear modules per layer, which causes their suboptimal speedup as sequence length increases. Meanwhile, based on Table 1, these methods naturally eliminate the propagation to attention computation, making them compatible with existing sparse attention techniques. Therefore, we can conveniently combine these approaches to obtain a hybrid width pruning method that also provide benefits under long-context scenarios. As demonstrated in Figure 9, during prefill and decoding stages at 32k sequence length, both hybrid methods achieve approximately 10%∼20% average speedup compared to their original version, respectively, with these gains continuing to increase for longer sequences. While dynamic **NK** enables static **NK** (cross-layer) to outperform dynamic **M** in long-context scenarios, it provides only marginal improvement for static **K** pruning, which is bounded by its acceleration potential on linear layers.

[7] https://huggingface.co/meta-llama/Llama-3.1-8B

[8] https://huggingface.co/Qwen/Qwen3-14B

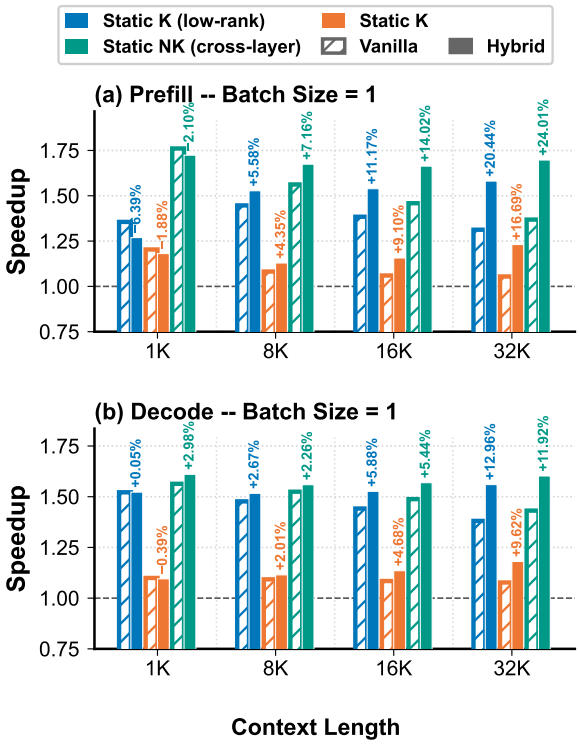

Figure 9: Prefill & decode speedup on vanilla static **K** (low-rank)/static **K**/ static **NK** (cross-layer) and their variants combined with SeerAttention-style sparse attention.

Influence of Dimension Alignment. For width pruning, the modification of parameter matrices shape also impacts the execution efficiency of GEMM operation. To quantify this effect, we evaluate different alignment strategies for the same target sparsity. As results in Table 5 and 6, it is clear that an unaligned layout can cause up to 35% loss of speedup. Since only 2/7 of GEMM in static **NK** prunes the leading dimension, it retains a more robust performance than static **NK** (cross-layer) with 5/7 of pruned GEMM in leading dimension, and static **K** (low-rank), with 7/14 of the low-rank GEMM expose their rank size dimension r to the logical **K** dimension.

To further analyze the sensitivity of structured width pruning to dimension alignment, we benchmark the kernel-level GEMM performance over different alignment strategies. As results in Table 6, the trend of standalone GEMM is broadly consistent with Table 5. The non-leading **N** dimension imposes relatively weak alignment constraints, whereas the **K** dimension becomes increasingly sensitive to alignment at lower precision. In fp8, when the alignment is smaller than 16 (16×1 bytes), throughput reaches only 11% of the baselines, but recovers to 70% once the alignment is a multiple of 16. Since the alignment requirements vary across the thread layout of memory access and compute instructions such as `mma.sync` under different precision, these results should be treated only as a rough trend. In practical width pruning, the pruning granularity should be at least aligned to (vectorized memory access width / data width) under the target precision, in order to maximize speedup.

Pruning over N vs. Pruning over K. To compare the differences between two width-pruning axis at the kernel-level, we further measure their speedups over the dense baseline across different sparsity levels and weight shapes, with results in Table 7 and 8. During prefill, pruning along the **N** axis leads to uneven speedups because reducing **N** also changes the total number of tiles, with the **M** tiling fixed, speedup appears only when the total number of GPU execution waves decreases. By contrast, pruning along **K** directly reduces the loop count within each block, lowering per-block runtime and yielding more stable speedups. However, although **N**-axis pruning in the prefill stage can only beat **K**-axis pruning in the up and gate projections with larger **N** dimensions, standard static **NK** still retains an advantage in linear projection operations due to the 53.84% proportion of these two GEMM's operation counts in each layer of Llama3.1-8B.

When turns to the batch-size-1 decoding, the **M** tiling is fixed to 1, forcing both schemes to activate split-k to increase the block count to match the number of SMs, therefore have the similar wave counts, and pruning along **N** or **K** can both change the split-k strategy and the runtime of each wave, resulting in a smoother speedups for normal static **NK** pruning than static **NK** (cross-layer). In this setting, pruning along **K** exposes the prologue and epilogue latency of the per-block pipeline under split-k, whereas pruning along **N** preserves the original loop count along **K**. As a result, **N**-dimension pruning achieves better decoding speedups.

Customized Kernel Latency. We further measure the throughput of the custom kernels for dynamic **M**, dynamic **NK**, and static **K** under different inputs, reporting TFLOPS in Tables 9, 10, and 11. For GEMM, static **K** generally needs **M**=128 to match the dense baseline, although it surpasses dense at **M**=8 when **N** is large. Dynamic **M** with online skipping starts to outperform dense only

Table 3: Ranking of averaged speedup in Llama3.1-8B across different hardware (RTX Pro6000 / A800 / H20), with S1–S4 corresponded to settings in Figure 6, i.e., prefill vary context length / batch size, and decode vary context length / batch size.

Taxonomic	S1	S2	S3	S4
Static M	1/1/1	1/1/1	1/1/1	1/1/1
Static K	6/6/6	6/6/6	6/7/6	5/5/6
Static K (low-rank)	5/4/5	4/4/4	4/4/5	3/4/4
Static NK	2/2/2	2/2/2	2/2/2	2/2/3
Static NK (cross-layer)	3/3/4	3/3/3	3/3/3	4/3/2
Dynamic M	4/5/3	5/5/5	5/5/4	7/7/5
Dynamic NK	7/7/-	7/7/-	7/6/-	6/6/-

Table 4: Ranking of averaged speedup in RTX Pro6000 across different model (Llama3.1-8B / Qwen3-14B / Qwen3-4B). Dynamic NK (Tilelang) encounters a compilation error on H20.

Taxonomic	S1	S2	S3	S4
Static M	1/1/1	1/1/1	1/1/1	1/1/1
Static K	6/7/6	6/6/6	6/6/6	5/5/5
Static K (low-rank)	5/4/5	4/4/4	4/4/5	3/4/4
Static NK	2/2/2	2/2/2	2/2/2	2/2/2
Static NK (cross-layer)	3/3/3	3/3/3	3/3/3	4/3/3
Dynamic M	4/5/4	5/5/5	5/5/4	7/7/6
Dynamic NK	7/6/-	7/7/-	7/7/-	6/6/-

when **M** is above 512. Once **M** exceeds 8192, the two sparse GEMM variants reach peak speedups of 1.35x and 2.20x, respectively. Attention shows a similar pattern in prefill: both sparse operators need about 32k query tokens to reach their best speedups, while sparse attention's block-wise $\mathbf{QK}^\top$ preprocessing results in a 21% gap relative to dynamic **M**. In decoding, dynamic **M** remains close to a 2x speedup under all settings because the online skipping can be processed token by token, similar to the situation in GEMV.

Task-wise Sensitive Analysis. We further examine task-specific sensitivity to sparsity, as shown in Figure 10. ARC-C, ARC-E, and HellaSwag are the most sensitive tasks, with their average relative performance loss approaching or exceeding 10% for each 12.5% increase in sparsity. By contrast, PIQA is less sensitive, with several pruning taxonomies retaining over 90% performance at four sparsity levels. Across taxonomies, dynamic **M** performs best in most settings when excluding static **K**, whose results are concentrated at 50% sparsity, where static **M** can remain competitive at low sparsity for tasks like ARC-C, Winogrande, and OpenbookQA.

Experiments on Different Sparsity, Models, and Hardware. We also evaluate Llama3.1-8B on RTX Pro6000 at 12.5%/25%/37.5% sparsity (Figure 12 to 14), Qwen3-4B on RTX Pro6000 at 12.5%/25%/37.5%/50% sparsity (Figure 15 to 18), Qwen3-14B at 50% sparsity (Figure 19), and Llama3.1-8B on A800-80G (Figure 20) and H20-96G (Figure 21). These results follow the same overall trend as the experiments in main text (Table 3 and Table 4).

Range	Align	Static K (low-rank)		Static NK		Static NK (cross-layer)	
		Speed	Param. Ratio	Speed	Param. Ratio	Speed	Param. Ratio
± 0%	–	19259.84 ± 28	56.73%	28334.09 ± 41	56.55%	20892.26 ± 13	53.35%
± 20%	odd	0.7034 ± 0.0038	56.11%	0.7069 ± 0.0323	55.42%	0.6552 ± 0.0088	50.72%
± 20%	even	0.9019 ± 0.0061	56.14%	0.9614 ± 0.0421	54.67%	0.8533 ± 0.0020	53.94%
± 20%	4	0.9279 ± 0.0057	56.14%	0.9961 ± 0.0238	54.84%	0.8762 ± 0.0057	53.41%
± 20%	8	0.9878 ± 0.0090	56.24%	1.0572 ± 0.0240	55.21%	0.9664 ± 0.0250	53.14%
± 20%	16	1.0048 ± 0.0080	56.37%	1.0669 ± 0.0264	55.79%	0.9794 ± 0.0243	53.42%
± 20%	128	0.9946 ± 0.0117	58.21%	1.0654 ± 0.0277	55.83%	0.9852 ± 0.0106	54.49%

Table 5: Prefill speed and retained parameters under different pruning and alignment strategies at batch size 1, context length 32,768, and 50% sparsity, where the 0% row reports absolute throughput in token/s and the remaining rows report speed normalized by the corresponding 0% baseline. Here, a 20% range denotes the pruning probability of each unit randomly fluctuates by 20% around the target sparsity, and align = 4 denotes alignment pruning dimensions to the nearest multiple of 4.

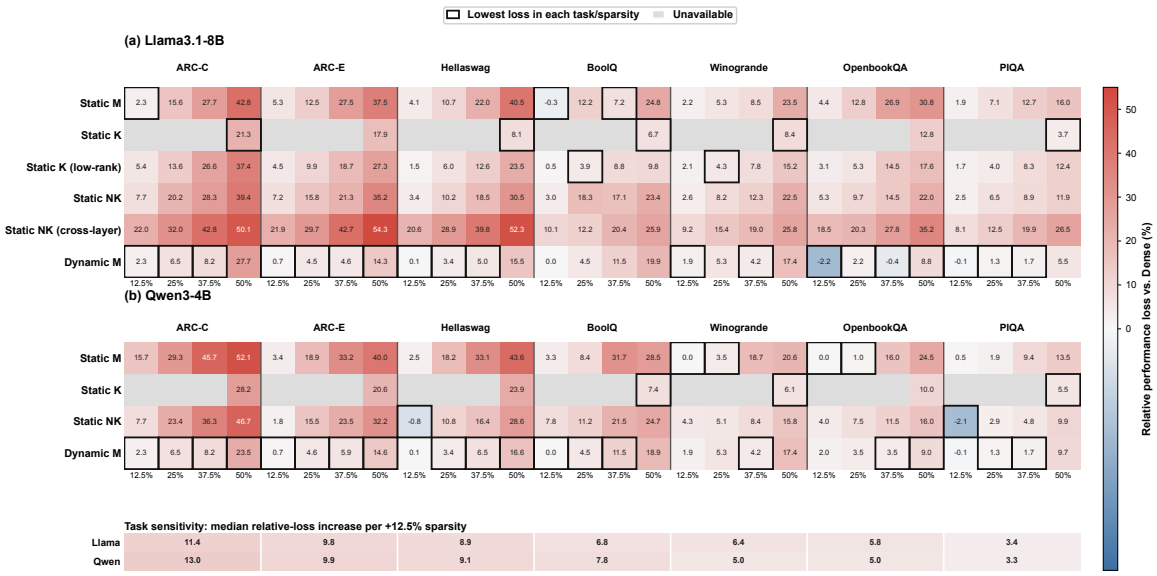

Figure 10: Relative performance loss across tasks, sparsity levels, and pruning taxonomies on Llama3.1-8B and Qwen3-4B. Tasks are ordered by their average sensitivity to sparsity.

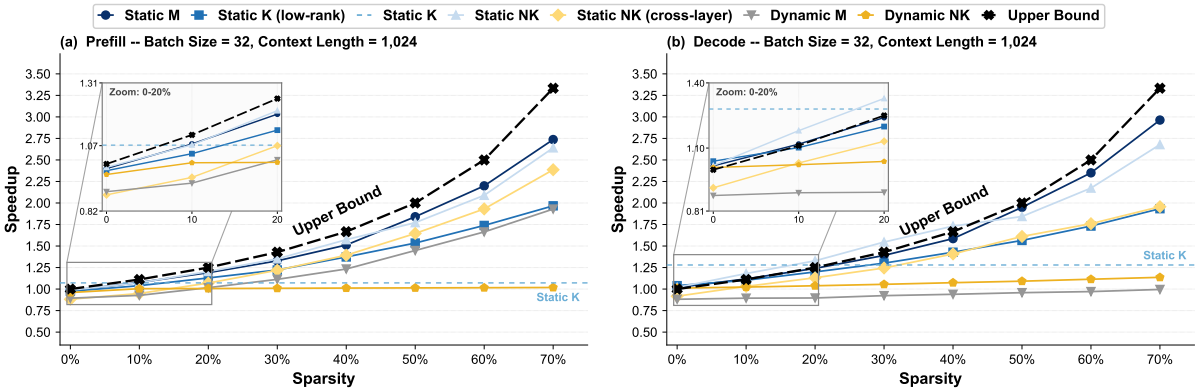

Figure 11: Speedup over different sparsity, and their gap with the theoretical upper bound. Context length is set to 1,024.

Pruning Dimension	Align							
	–	odd	even	4	8	16	32	64
				bf16				
N	345.26 ± 0.49	0.9280 ± 0.0028	0.9177 ± 0.0017	0.9758 ± 0.0012	0.9761 ± 0.0011	0.9867 ± 0.0011	0.9857 ± 0.0015	0.9910 ± 0.0021
K	345.26 ± 0.49	0.3316 ± 0.0023	0.3342 ± 0.0007	0.3359 ± 0.0021	0.3371 ± 0.0027	1.0386 ± 0.0219	1.0112 ± 0.0026	1.0063 ± 0.0023
				fp8_e5m2				
N	491.92 ± 1.35	0.9923 ± 0.0029	0.9925 ± 0.0034	0.9921 ± 0.0027	0.9932 ± 0.0025	0.9999 ± 0.0031	0.9978 ± 0.0029	0.9936 ± 0.0027
K	491.92 ± 1.35	0.1181 ± 0.0003	0.1176 ± 0.0003	0.1182 ± 0.0003	0.1177 ± 0.0003	0.7043 ± 0.0018	0.9742 ± 0.0026	1.0460 ± 0.0026

Table 6: Throughput of Triton's GEMM kernel under different alignment strategies, the base layout is set to [8192,8192]@[8192,8192], with **N**-axis align to the nearest odd denotes to apply [8192,8192]@[8192-1,8192] shape's GEMM.

Sparsity	[4096,4096]@[4096,4096]		[4096,4096]@[14336,4096]		[4096,14336]@[4096,14336]	
	N	**K**	**N**	**K**	**N**	**K**
10%	1.0032	1.0963	1.1056	1.0900	1.0169	1.1009
20%	1.0054	1.2189	1.2450	1.2265	1.0202	1.2406
30%	1.0111	1.3613	1.4189	1.4001	1.4686	1.4306
40%	1.4717	1.5739	1.6569	1.6232	1.5201	1.6674
50%	1.4832	1.8469	1.9802	1.9223	1.9186	1.9752
60%	1.4831	2.2075	2.4407	2.3059	2.2795	2.4348
70%	2.4461	2.7465	3.2071	2.9513	2.9141	3.1823

Table 7: The speedup provided by two different pruning dimensions on the query, up, and down projections of Llama3.1-8B under different sparsity. The shape of each GEMM is [**M**, **K**] @ [**N**, **K**].

Sparsity	[1,4096]@[4096,4096]		[1,4096]@[14336,4096]		[1,14336]@[4096,14336]	
	N	**K**	**N**	**K**	**N**	**K**
10%	1.0119	0.9598	1.2674	0.9504	1.2619	1.1638
20%	1.1205	1.0834	1.4695	1.1414	1.5728	1.5016
30%	1.3939	1.1993	1.8296	1.3321	2.3106	1.9335
40%	1.5471	1.2900	2.0408	1.6442	1.6629	2.0508
50%	1.6532	1.7195	2.5662	2.3590	2.1222	2.9063
60%	1.6396	1.6520	2.9301	2.6541	2.5912	3.2214
70%	1.9354	1.8561	3.3894	3.1360	3.0271	3.0392

Table 8: The speedup provided by two types of width pruning on the query, up, and down projections of Llama3.1-8B under different sparsity. The number of tokens is fixed to 1.

M	[T,4096]@[4096,4096]			[T,4096]@[14336,4096]			[T,14336]@[4096,14336]		
	dense	static **K**	dynamic **M**	dense	static **K**	dynamic **M**	dense	static **K**	dynamic **M**
1	1.58	1.38	2.67 / 15.79	1.49	4.75	5.31 / 52.85	1.45	1.60	3.72 / 55.33
2	4.75	2.77	2.88	10.15	9.56	6.48	8.79	3.19	6.15
4	9.48	5.53	5.77	20.96	19.60	12.98	16.04	6.39	12.21
8	18.37	12.03	11.54	39.74	43.45	26.48	19.36	13.14	24.26
16	34.31	24.06	22.88	76.98	87.11	53.17	34.85	26.32	48.00
32	97.80	48.06	43.73	111.45	174.00	79.32	140.79	52.70	87.71
64	164.53	95.27	70.66	227.04	344.09	126.22	229.31	104.84	115.38
128	195.04	194.42	127.54	260.73	278.14	223.23	234.64	218.25	201.35
256	259.69	361.05	180.41	266.97	368.52	322.42	266.89	356.99	266.41
512	289.43	323.17	309.29	288.08	418.94	383.31	269.66	296.05	436.72
1024	292.02	407.81	231.15	325.65	427.26	513.34	278.75	394.01	261.52
2048	337.75	413.44	447.75	329.33	428.45	626.02	287.21	395.99	503.85
4096	303.37	432.04	599.81	300.49	431.03	640.65	289.59	412.07	640.92
8192	344.71	423.56	626.52	342.33	432.99	638.68	349.08	417.00	649.07
16384	324.97	426.68	652.30	292.01	433.49	649.14	292.42	420.96	644.12
32768	318.38	428.42	621.40	319.03	433.10	624.63	292.71	422.48	627.79

Table 9: TFLOPS of dense, static **K**, and dynamic **M**'s GEMM kernels under different input sizes **M**, with sparsity fixed to 50%. Static **K** reports the sglang JIT cuSPARSELt result, where dynamic M reports both no-skip and skip results when the batch size equals 1.

T	[B,T,32,128]@[B,T,8,128]		
	dense	dynamic **M**	dynamic **NK**
1024	150.08	97.74	95.47
2048	207.57	197.44	186.35
4096	276.94	355.93	345.53
8192	320.49	483.24	433.20
16384	343.88	578.50	485.75
32768	351.15	634.31	523.63
65536	354.87	653.86	517.45

Table 10: Prefill TFLOPS of dense attention, dynamic **M**, and dynamic **NK** under different query lengths, with sparsity fixed to 50%. The batch size is fixed to **1** where the shapes of the query, key, and value tensors are [batch size, context length, head counts, head size].

B	[B,1,32,128]@[B,T,8,128]		
	dense	dynamic **M**	dynamic **NK**
1	7.89	8.16 / 52.92	8.47
2	2.55	10.61	7.58
4	2.63	5.75	4.93
8	2.66	5.94	5.26
16	2.69	6.04	5.00
32	3.05	6.09	6.00

Table 11: Decode TFLOPS of dense attention, dynamic **M**, and dynamic **NK** under different batch sizes, with sparsity fixed to 50%. The KV cache length **T** is fixed to 32768, where dynamic **M** reports both no skip and skip results in batch size equals 1.

Categories	Methods	WikiText2 (ppl↓)	ARC-E (Acc. Norm.↑)	ARC-C (Acc. Norm.↑)	BoolQ (Acc. ↑)	Winogrande (Acc. ↑)	PIQA (Acc. Norm.↑)	OpenbookQA (Acc. Norm.↑)	Hellaswag (Acc. Norm.↑)	Avg. Acc.↑	Avg. Gap↓
	Dense	7.54	82.44	55.20	82.96	74.34	80.84	45.40	79.34	71.50	0.00%
	Dense (LoRA)	7.74	81.35	54.18	83.48	72.69	81.22	46.60	79.94	71.35	0.21%
12.5% Sparsity											
Static **M**	Shortened-taylor	10.54	78.07	**53.92**	**83.21**	72.69	79.33	43.40	76.08	69.52	2.76%
	CoopPruner	10.90	72.47	47.10	74.59	66.46	78.89	43.40	75.16	65.43	8.48%
Static **K**	Dobi-SVD†	8.72	78.75	52.22	82.54	72.77	79.43	44.00	78.13	69.69	2.53%
Static **NK**	Týr-the-Pruner	8.81	76.52	50.94	80.46	72.38	78.78	43.00	76.62	68.38	4.35%
	SliceGPT‡	35.05	64.35	43.08	74.55	67.48	74.31	37.00	62.96	60.53	15.34%
Dynamic **M**	SkipGPT	**8.42**	**81.86**	**53.92**	82.96	**72.92**	**80.90**	**46.40**	**79.26**	**71.17**	**0.45%**
25% Sparsity											
Static **M**	Shortened-ppl	13.61	68.73	42.75	64.77	63.54	76.99	42.40	70.38	61.36	14.17%
	Shortened-taylor	15.52	72.10	46.59	72.81	70.40	75.14	39.60	70.84	63.92	10.59%
	CoopPruner	13.59	68.51	41.80	65.04	63.06	76.27	40.20	70.41	60.75	15.03%
	BlockPruner	16.53	66.20	40.78	66.06	65.35	74.59	39.60	65.10	59.66	16.55%
Static **K**	Dobi-SVD†	10.14	74.24	47.70	**79.69**	**71.11**	77.58	43.00	74.56	66.84	6.52%
Static **NK**	FLAP	22.69	52.31	32.08	62.45	53.99	69.80	33.40	53.57	51.08	28.55%
	Týr-the-Pruner	12.46	69.44	44.03	67.74	68.27	75.57	41.00	71.21	62.46	12.63%
	SliceGPT‡	38.96	57.95	37.54	72.87	62.90	70.73	36.20	56.40	56.37	21.16%
Dynamic **M**	D-LLM	9.58	76.85	48.75	75.44	69.21	77.85	43.20	74.67	66.56	6.90%
	SkipGPT	9.25	**78.70**	**51.62**	79.26	70.40	**79.76**	**44.40**	**76.62**	**68.68**	**3.94%**
Dynamic **NK**	SeerAttention†	**7.72**	49.62	34.81	53.67	54.46	60.94	38.40	41.41	47.61	33.40%
37.5% Sparsity											
Static **M**	Shortened-taylor	26.52	59.76	39.93	**76.97**	68.03	70.57	33.20	61.88	58.62	18.01%
	CoopPruner	23.03	59.43	34.22	60.55	60.14	72.03	34.80	58.21	54.19	24.20%
Static **K**	Dobi-SVD†	12.34	67.05	40.53	75.66	68.51	74.16	38.80	69.37	62.01	13.27%
Static **NK**	Týr-the-Pruner	26.18	64.90	39.59	68.78	65.19	73.61	38.80	64.68	59.36	16.97%
	SliceGPT‡	57.51	47.26	31.56	66.05	60.22	64.79	32.80	47.77	50.06	29.98%
Dynamic **M**	SkipGPT	**10.64**	**78.66**	**50.68**	73.39	**71.19**	**79.43**	**45.60**	**75.37**	**67.76**	**5.23%**
50% Sparsity											
Static **M**	Shortened-ppl	36.43	48.74	28.75	61.50	52.57	65.29	29.60	44.25	47.24	33.92%
	Shortened-taylor	52.92	47.18	31.48	64.40	59.27	65.61	28.80	48.80	49.36	30.96%
	CoopPruner	33.93	51.55	31.56	62.41	56.90	67.89	31.40	47.24	49.85	30.28%
	BlockPruner	71.96	40.82	25.51	55.05	53.12	61.53	27.60	37.38	43.00	39.86%
Static **K**	Dobi-SVD†	15.44	59.93	34.56	74.83	63.06	70.78	37.40	60.66	57.31	19.83%
	MaskLLM	11.45	67.72	**43.43**	**77.43**	**68.11**	**77.86**	39.60	**72.95**	**63.87**	**10.67%**
Static **NK**	FLAP	88.93	38.34	24.82	58.07	51.85	59.73	26.40	33.55	41.82	41.50%
	Týr-the-Pruner	19.80	53.45	33.45	63.58	57.62	71.22	35.40	55.13	52.83	26.10%
	SliceGPT‡	68.71	37.67	27.56	61.44	55.17	59.41	29.40	37.86	44.07	38.36%
Dynamic **M**	D-LLM	16.83	62.58	34.30	62.44	57.69	71.87	36.80	59.84	55.07	22.97%
	SkipGPT	13.14	**70.62**	39.93	66.42	61.40	76.38	**41.40**	67.05	60.45	15.44%
Dynamic **NK**	SeerAttention*	**7.76**	49.62	34.81	48.62	54.46	60.77	38.40	41.41	46.87	34.45%

Table 12: Llama3.1-8B pruning performance under the LoRA configuration, where the static semi-structured **K** methods are reported without LoRA fine-tuning due to the lack of unified LoRA implementation strategies. $*$ for dynamic **NK** denotes applying the attention gate checkpoint for Llama3.1-8B-Instruct to the base Llama3.1-8B due to high retraining overhead.

Categories	Methods	WikiText2 (ppl↓)	ARC-E (Acc. Norm.↑)	ARC-C (Acc. Norm.↑)	BoolQ (Acc. ↑)	Winogrande (Acc. ↑)	PIQA (Acc. Norm.↑)	OpenbookQA (Acc. Norm.↑)	Hellaswag (Acc. Norm.↑)	Avg. Acc.↑	Avg. Gap↓
	Dense	17.31	78.41	54.10	85.17	65.98	74.86	40.00	68.37	66.69	0.00%
12.5% Sparsity											
Static **M**	Shortened-ppl	14.42	72.31	44.03	66.45	59.19	75.35	**41.00**	63.20	60.21	9.71%
	Shortened-taylor	15.77	75.78	45.60	82.39	**65.98**	74.51	40.00	66.63	64.41	3.42%
	CoopPruner	14.47	72.69	44.62	65.87	61.72	74.92	40.00	63.25	60.43	9.38%
	ReplaceMe	19.56	67.80	43.60	78.50	65.75	70.73	37.40	59.80	60.51	9.27%
Static **NK**	LoRAP	14.31	76.30	49.74	75.66	64.09	**76.55**	40.20	65.57	64.01	4.02%
	Týr-the-Pruner	12.72	76.98	49.91	78.50	63.11	76.44	38.40	**68.91**	64.60	3.13%
Dynamic **M**	MoD	12.82	76.02	52.59	84.47	64.38	74.58	40.80	68.04	65.84	1.28%
	SkipGPT	**10.13**	**77.85**	**52.84**	**85.17**	64.71	74.91	39.20	68.30	**66.14**	**0.83%**
25% Sparsity											
Static **M**	Shortened-ppl	16.94	61.20	36.43	59.42	55.49	73.39	39.60	55.79	54.47	18.32%
	Shortened-taylor	17.05	63.59	38.26	78.01	**63.69**	73.46	39.60	55.93	58.93	11.64%
	CoopPruner	17.29	63.13	37.54	51.01	54.93	73.12	**40.40**	55.25	53.62	19.59%
	ReplaceMe	30.82	57.58	35.32	76.57	62.43	66.59	31.20	51.69	54.48	18.31%
Static **NK**	LoRAP	18.03	65.64	40.62	73.91	61.25	73.23	34.20	58.23	58.15	12.81%
	Týr-the-Pruner	14.83	66.25	41.45	75.63	62.61	72.67	37.00	60.96	59.51	10.77%
Dynamic **M**	MoD	13.68	73.70	48.50	80.71	62.61	73.61	38.20	64.50	63.11	5.36%
	SkipGPT	**12.52**	**74.83**	**50.59**	**81.37**	62.49	**73.85**	38.59	**66.02**	**63.96**	**4.09%**
37.5% Sparsity											
Static **M**	Shortened-ppl	26.32	51.73	29.52	58.99	52.25	67.68	34.00	44.83	48.42	27.39%
	Shortened-taylor	45.34	46.00	31.74	66.73	58.48	63.66	30.20	43.08	48.55	27.20%
	CoopPruner	26.03	52.36	29.35	58.20	53.67	67.85	33.60	45.71	48.67	27.01%
	ReplaceMe	53.73	44.28	27.13	57.09	57.06	62.08	30.20	39.42	45.32	32.04%
Static **NK**	LoRAP	25.18	60.14	35.15	68.04	57.85	68.99	34.40	48.76	53.33	20.03%
	Týr-the-Pruner	17.50	60.02	34.47	66.85	60.46	71.27	35.40	57.16	55.09	17.40%
Dynamic **M**	MoD	18.75	72.25	46.18	74.78	60.27	72.93	36.20	61.57	60.59	9.14%
	SkipGPT	**15.89**	**73.76**	**49.67**	**75.34**	**63.18**	**73.55**	**38.60**	**63.94**	**62.58**	**6.17%**
50% Sparsity											
Static **M**	Shortened-ppl	37.89	47.05	25.94	60.86	52.41	64.74	30.20	38.53	45.67	31.51%
	Shortened-taylor	61.50	43.10	25.77	55.20	54.06	61.59	30.40	36.26	43.76	34.37%
	CoopPruner	36.03	47.14	27.22	52.54	51.85	64.04	31.40	39.13	44.76	32.89%
	ReplaceMe	66.89	40.40	25.85	45.63	49.72	58.71	28.80	34.17	40.46	39.32%
Static **K**	HyperPrune†	28.85	62.25	38.82	**78.87**	**61.96**	**70.78**	36.00	52.05	**57.24**	**14.17%**
Static **NK**	LoRAP	36.40	51.14	29.61	64.46	53.51	62.51	32.00	38.95	47.45	28.85%
	Týr-the-Pruner	21.44	53.20	28.84	64.10	55.56	67.46	33.60	48.83	50.22	24.69%
Dynamic **M**	MoD	20.03	64.01	32.79	66.50	54.93	63.20	32.60	**57.51**	53.07	20.42%
	SkipGPT	**19.17**	**66.98**	**41.37**	69.04	54.49	67.59	**36.40**	57.02	56.13	15.84%

Table 13: Qwen3-4B pruning performance under the LoRA configuration. Static **K** methods marked with † are reported without LoRA fine-tuning due to the lack of unified LoRA implementation strategies.

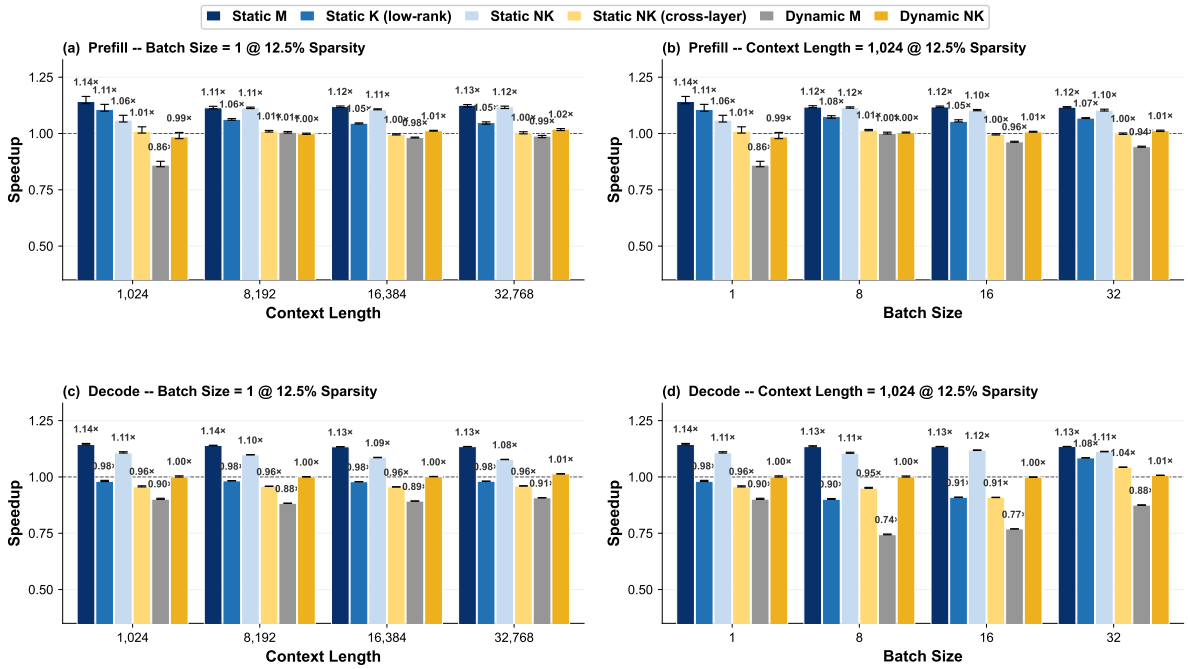

Figure 12: Inference speedup on Llama3.1-8B with different pruning strategies, with sparsity set to 12.5%.

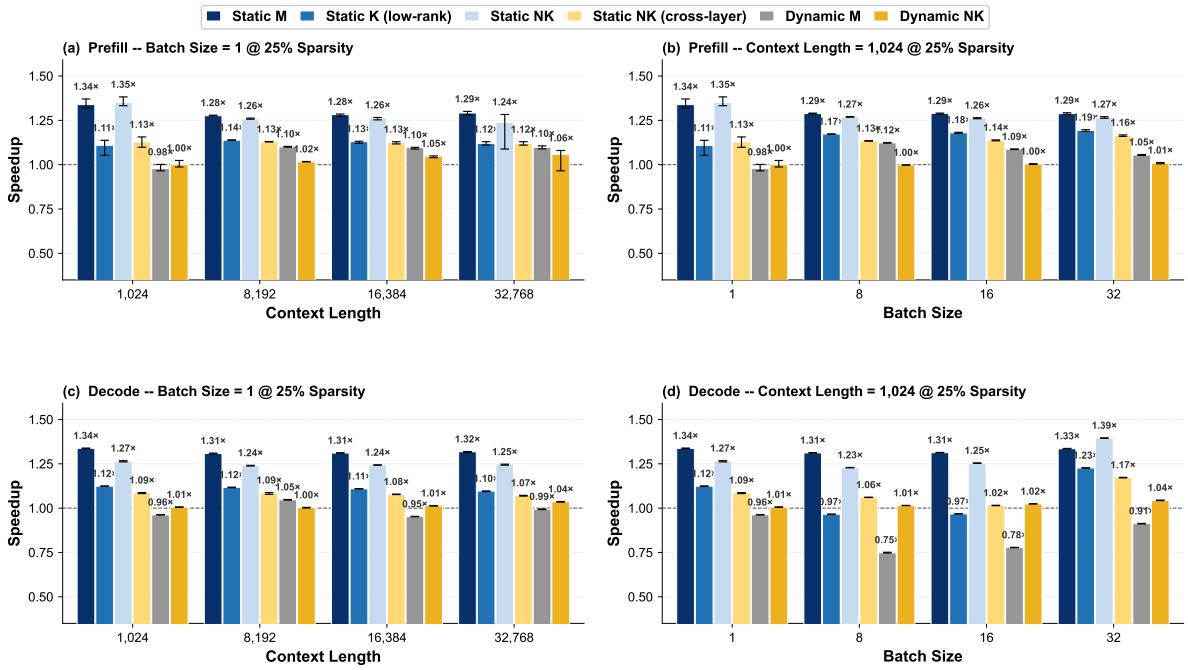

Figure 13: Inference speedup on Llama3.1-8B with different pruning strategies, with sparsity set to 25%.

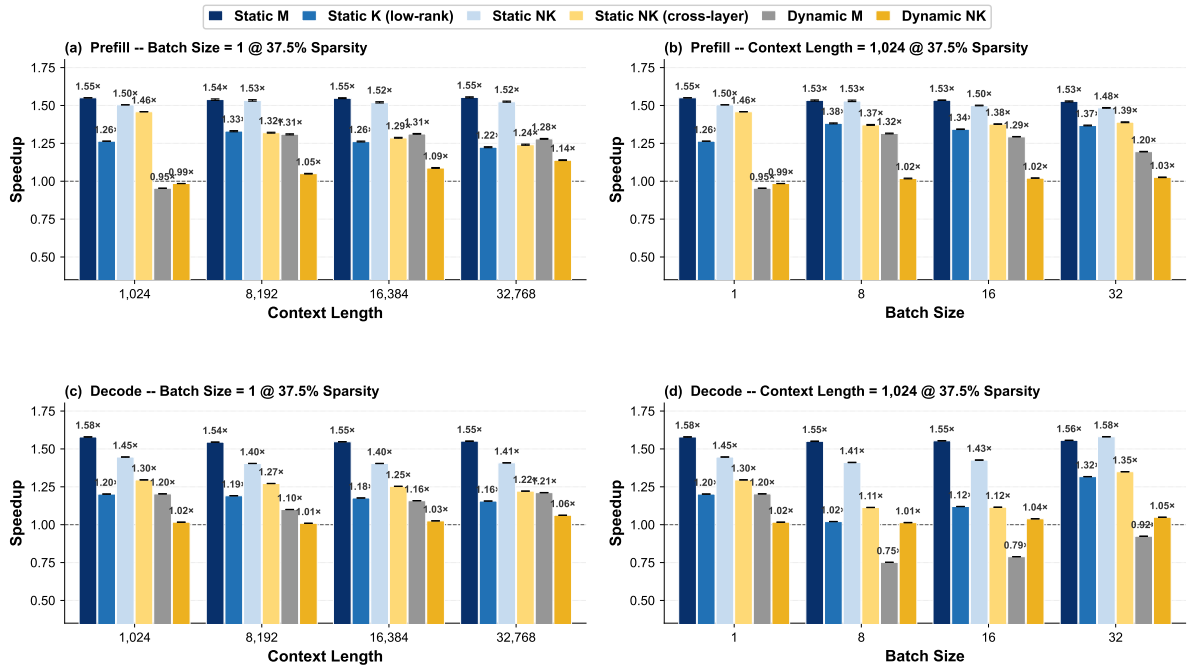

Figure 14: Inference speedup on Llama3.1-8B with different pruning strategies, with sparsity set to 37.5%.

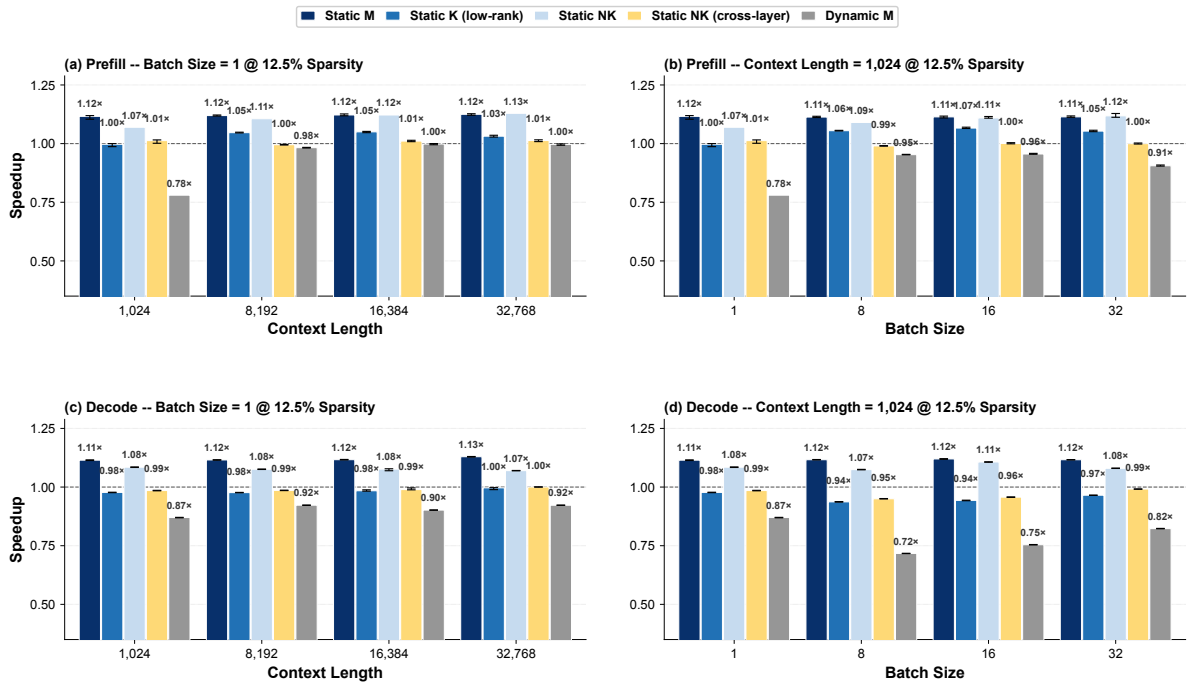

Figure 15: Inference speedup on Qwen3-4B with different pruning strategies, with sparsity set to 12.5%.

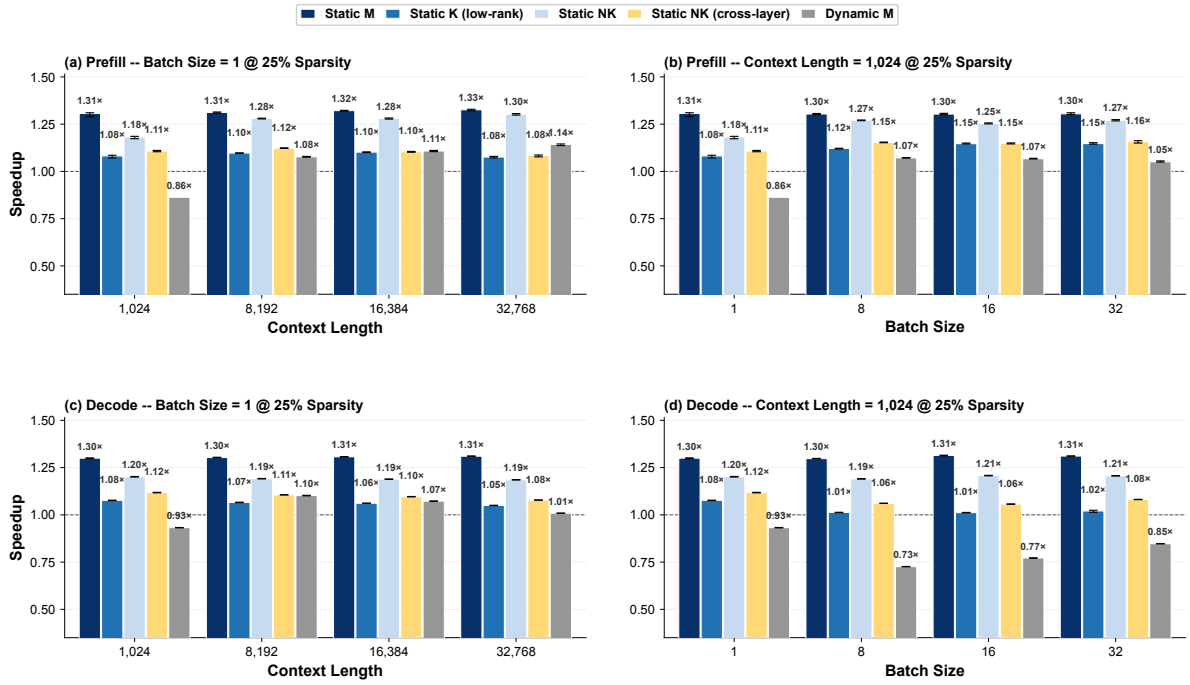

Figure 16: Inference speedup on Qwen3-4B with different pruning strategies, with sparsity set to 25%.

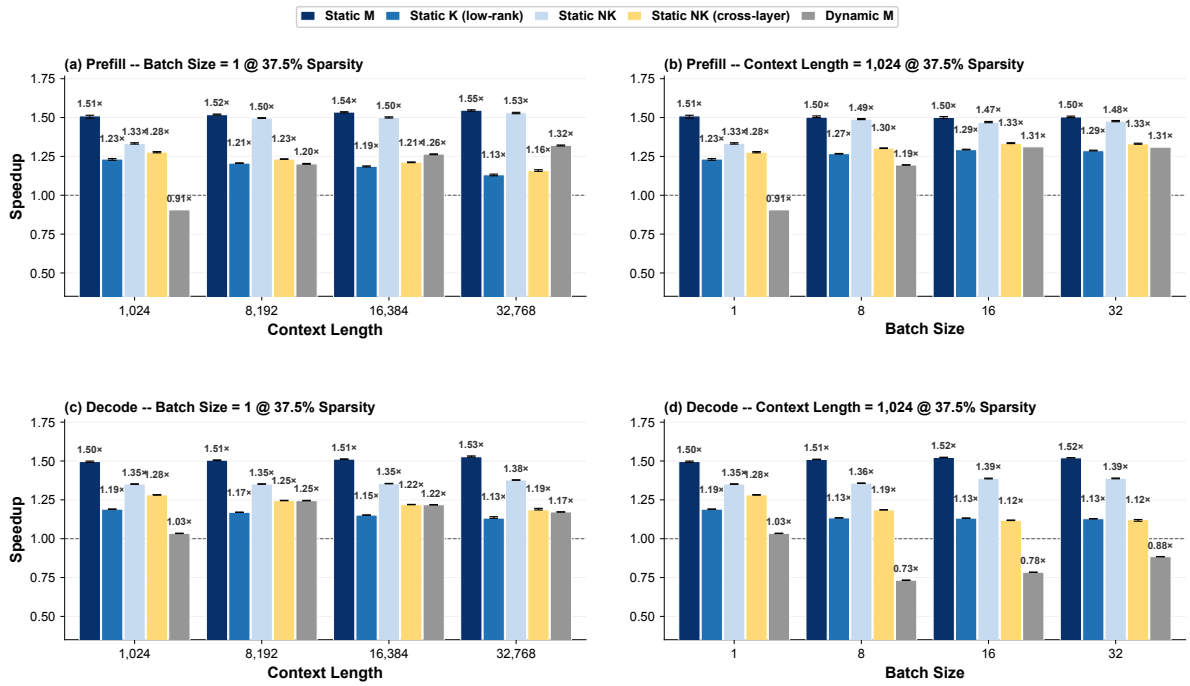

Figure 17: Inference speedup on Qwen3-4B with different pruning strategies, with sparsity set to 37.5%.

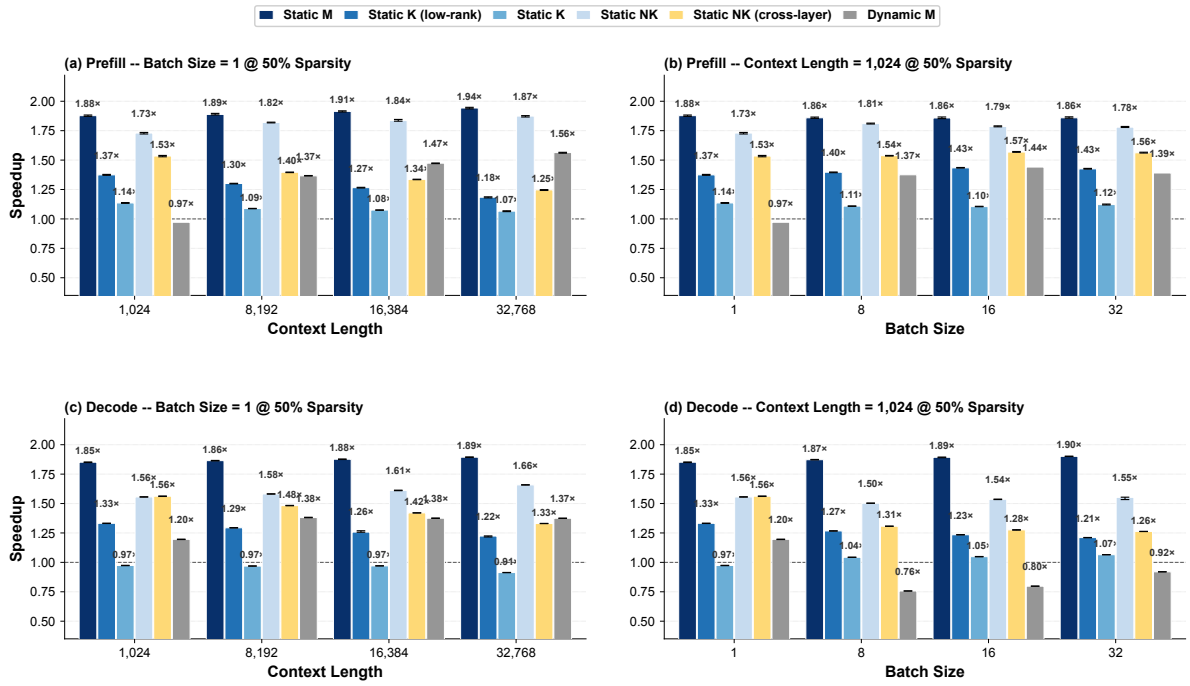

Figure 18: Inference speedup on Qwen3-4B with different pruning strategies, with sparsity set to 50%.

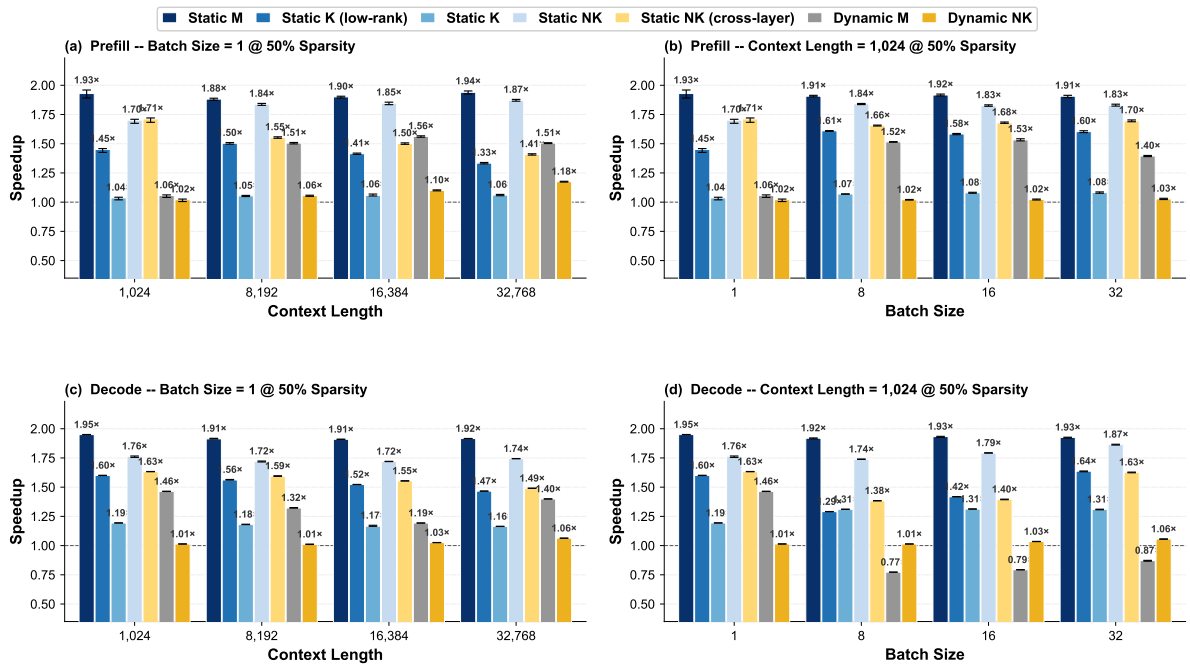

Figure 19: Inference speedup on Qwen3-14B with different pruning strategies, with sparsity set to 50%.

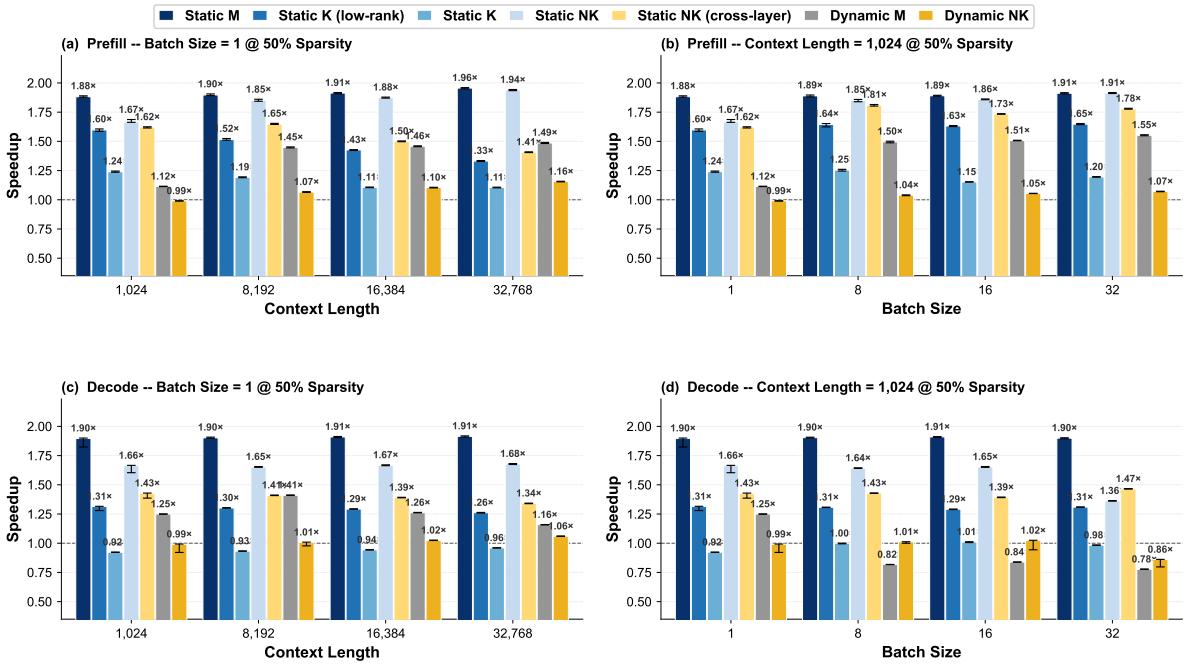

Figure 20: Inference speedup on Llama3.1-8B with different pruning strategies, with sparsity set to A800-80G.

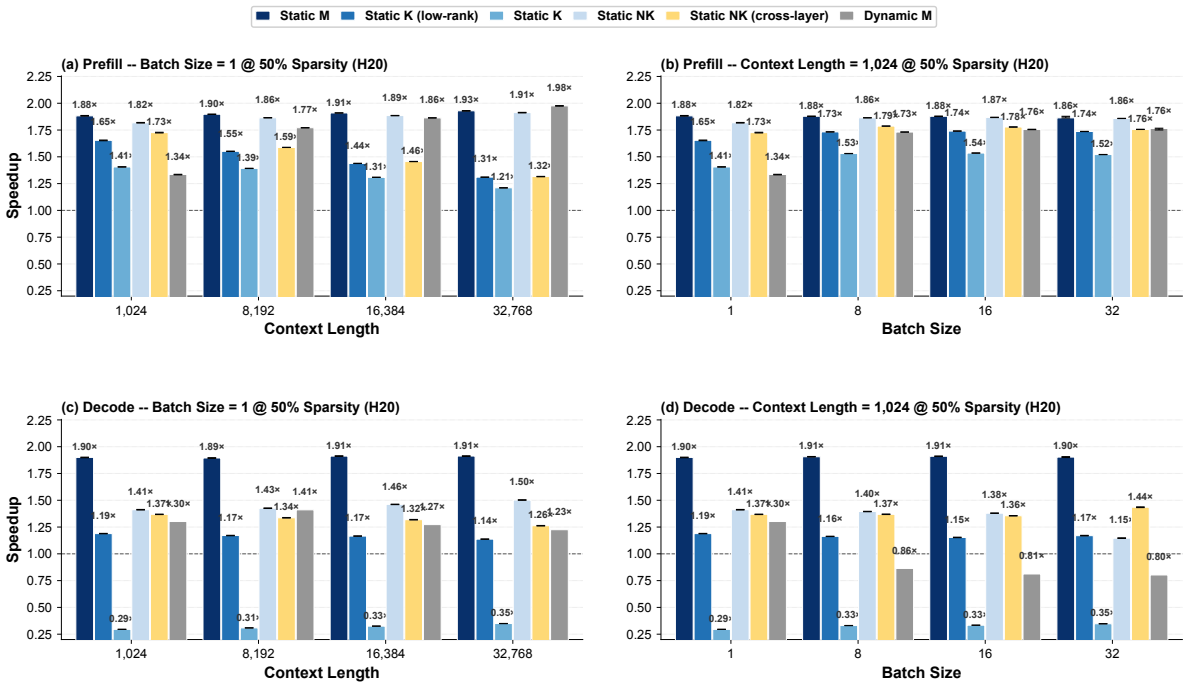

Figure 21: Inference speedup on Llama3.1-8B with different pruning strategies, with sparsity set to H20-96G.