WIDE: BOOSTING ADAPTIVE LLM INFERENCE VIA TOKEN-LEVEL DYNAMIC WIDTH PRUNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Pruning is a promising approach for improving the efficiency of large language models (LLMs). Static structured pruning is hardware-friendly, but its input-agnostic computation allocation often degrades model quality at high sparsity. Dynamic pruning adapts computation to individual tokens, but coarse-grained layer-level routing limits allocation flexibility, and finer-grained decisions introduce irregular execution that can offset computational savings. To address these challenges, we present WIDE, the first end-to-end differentiable framework that combines token-wise structured dynamic width pruning with GPU kernel co-design for both prefill and decoding. WIDE uses differentiable routers to select GQA-aligned query-head groups and GPU-tile-aligned FFN-channel groups for each token. To execute these decisions efficiently, it reorders routing masks to cluster active tokens within each group, enabling hardware-agnostic block-level skipping and architecture-aware skipping of inactive loads and tensor-core operations. Experiments on the Llama3 and Qwen3 families demonstrate improved quality retention over representative static and dynamic pruning methods. At 50% target routing sparsity, WIDE improves Llama3.1-8B average zero-shot accuracy by up to 20.26 points over the strongest dynamic-depth baseline under calibration-only settings, while retaining 97.6%/88.2% of dense Qwen3-14B likelihood/generation performance after LoRA tuning. Separate latency benchmarks demonstrate end-to-end speedups of up to $1.68\times$ for prefill and $1.55\times$ for decoding over the dense baseline. These results demonstrate that pruning–kernel co-design pushes the frontier of token-wise dynamic structured pruning, combining improved quality retention with practical acceleration across both prefill and decoding.

1 INTRODUCTION

The rapid advancement of LLMs has enabled their widespread deployment across a broad range of real-world applications (Team et al., 2026; DeepSeek-AI, 2026; Meng et al., 2026; Dai et al., 2026; Chen et al., 2026b). However, as models continue to scale, efficient LLM serving becomes increasingly challenging. Beyond system-level and hardware-specific optimizations (Kwon et al., 2023; Zheng et al., 2024; Lin et al., 2024; Zadouri et al., 2026), model pruning has also evolved rapidly as a model-level approach to improving inference efficiency. By removing redundant structures along the depth and width dimensions, pruning reduces model size and computational cost, thereby offering a complementary path toward efficient LLM/VLM serving (Cheng et al., 2024; Frantar & Alistarh, 2023; Ma et al., 2023; Ashkboos et al., 2024; Men et al., 2025; Wu et al., 2026a;c;b).

Currently, LLM pruning is dominated by static schemes, which remove predefined structures such as entire layers (Kim et al., 2024), attention or feed-forward network (FFN) sublayers (Zhong et al., 2025), rows or columns of weight matrices (Li et al., 2025a), and individual neurons (Fang et al., 2024). Once calibrated, these units are permanently removed, making static pruning simple to deploy and compatible with existing hardware backends. However, applying the same pruning decisions to every input often sacrifices model quality, especially under aggressive sparsity.

In contrast, dynamic pruning adapts computation to individual tokens through trainable routers (Raposo et al., 2024; Zhao et al., 2025; Han et al., 2025) or activation thresholds (Shrestha et al., 2025; Chen et al., 2026a). Representative routing-based approaches make execution decisions at the layer or sublayer level, requiring each token to retain or skip an entire attention or FFN module.

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

This coarse granularity limits their ability to preserve useful computation within a module while removing its redundant parts. Finer-grained width-level routing offers more flexible computation allocation, but introduces a systems challenge: different tokens may activate different subsets of heads or channels, producing irregular workloads that are difficult to execute efficiently on GPUs. The resulting synchronization and data-movement overheads can offset the savings from reduced computation (He et al., 2026; Hu et al., 2026). Meanwhile, activation-based unstructured pruning primarily targets sparse matrix-vector operations during decoding, leaving efficient execution across both prefill and decoding insufficiently addressed. These limitations motivate a central question: *How can token-wise dynamic pruning achieve finer-grained computation allocation while preserving the execution regularity needed for practical end-to-end acceleration?*

Motivated by these challenges, we present **WIDE**. To the best of our knowledge, WIDE is the first framework to combine token-wise dynamic width pruning with GPU kernel co-design for both prefill and decoding. Specifically, WIDE groups attention heads and FFN channels according to GPU execution granularity and treats each group as the atomic pruning unit for each token, providing a trade-off between structured pruning granularity and execution flexibility. Furthermore, to translate these fine-grained decisions into acceleration, instead of explicitly materializing token-specific gathered tensors, WIDE first performs mask reordering to cluster active tokens separately for each group into tile-aligned prefixes, *converting irregular sparsity into CTA (Cooperative Thread Array)-level regularity.* Based on this representation, WIDE kernels progressively eliminate unnecessary computation at multiple granularities: hardware-agnostic predicates remove fully inactive CTAs, while architecture-aware predicates further skip inactive activation loading and tensor-core computation. This design allows fine-grained dynamic sparsity to retain efficient GPU execution patterns and deliver practical acceleration for both prefill and decoding workloads.

Experiments on the Llama3 and Qwen3 families demonstrate improved quality retention over representative static and dynamic pruning methods. At 50% target routing sparsity, WIDE improves Llama3.1-8B average zero-shot accuracy by up to 20.26 points over the strongest dynamic-depth baseline under calibration-only settings, while retaining 97.6%/88.2% of dense Qwen3-14B likelihood/generation performance after LoRA tuning. Separate latency benchmarks show end-to-end speedups of up to 1.68× for prefill and 1.55× for decoding over the dense baseline. Kernel-level comparisons against naive gather–scatter implementations further show up to 200× speedup and a 1,000× reduction in peak memory usage. Our contributions are threefold:

(1) We introduce WIDE, the first end-to-end structured dynamic pruning framework for fine-grained token-wise width allocation. WIDE formulates dynamic width pruning over GQA-aligned query-head groups and GPU-tile-aligned FFN-channel groups, with differentiable routers learning token-specific execution decisions. This extends structured routing beyond layer-level decisions to group-level allocation within attention and FFN modules.

(2) We present the first unified framework that unlocks practical acceleration for fine-grained dynamic sparsity. To address synchronization and data-movement overheads, we combine mask reordering, block-level skipping, and architecture-aware intra-block predication, converting irregular token-wise selections into block-level regularity without materializing token-specific weight subsets. Dedicated fast paths support efficient decoding.

(3) We advance the pruning frontier through pruning–kernel co-design. Experiments demonstrate improved quality retention alongside practical acceleration across both prefill and decoding workloads. Further analysis characterizes how group size, recovery training, and GPU architecture shape the quality–efficiency trade-off, providing guidance for selecting pruning granularity. .

2 Related Works

Static Pruning Among existing pruning methods, static pruning remains dominant. Given a small set of calibration samples, these methods estimate the importance of predefined units from Transformer layers to individual neurons, and permanently remove low-importance units until the target sparsity is reached. According to the pruning unit, existing work can be broadly categorized into depth and width pruning. **Depth** pruning removes computation along the layer dimension by calibration-based metric ranking, e.g., perplexity, Taylor (Kim et al., 2024; Zhong et al., 2025), and surrogate-assisted Shapley values (Ding et al., 2026). **Width** pruning instead removes channels,

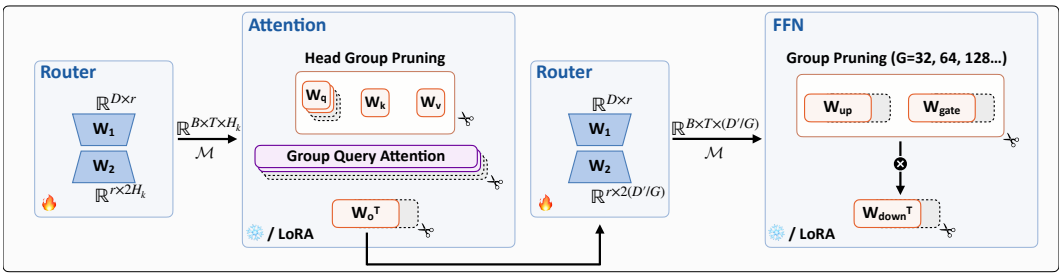


Figure 1: The pruning dimension of WIDE in the attention and FFN layer, with $\mathbf{W}_q$, $\mathbf{W}_{\text{up}}$, and $\mathbf{W}_{\text{gate}}$ pruned over out features (N-axis), and $\mathbf{W}_o$, $\mathbf{W}_{\text{down}}$ over in features (K-axis). Each router produces $2N_G$ logits for N_G binary decisions to align with the Gumbel Softmax.

neurons, or attention heads within layers. Methods include SliceGPT (Ashkboos et al., 2024) and Týr-the-Pruner (Li et al., 2025a) allocation row-wise sparsity under heuristic metrics, similar to depth pruning, while recent Deterministic Differentiable Pruning (DDP) (Huang et al., 2026) further learns differentiable structured masks under an ℓ_0 sparsity constraint via augmented Lagrangian optimization. Beyond structural pruning, methods like Wanda (Sun et al., 2024) and HyperPrune (Sun & Sakuma, 2026) further incorporate semi-structured or unstructured masks to prune individual weights. These static methods are deployment-friendly, but their input-agnostic pruning decisions limit their ability to allocate computation adaptively across tokens.

Dynamic Pruning Dynamic pruning allocates computation adaptively during inference instead of permanently removing a fixed set of parameters. Existing methods mainly focus on token-wise depth or sublayer routing with trainable decision modules, like Mixture-of-Depths (MoD) (Raposo et al., 2024), D-LLM (Jiang et al., 2024), SkipGPT (Zhao et al., 2025), and LFF (Han et al., 2025). PolarSparse and FastForward instead introduce the dynamic width pruning for batch decode / prefill only (Gautam et al., 2026; Shrestha et al., 2025). As for router-free decision, methods such as WINA (Chen et al., 2026a) incorporate dynamic unstructured pruning with activation-based threshold for decoding acceleration. While these approaches improve adaptive computation, they mainly operate at coarse granularity or specific scenario. WIDE instead formulates token-wise structured width pruning over query-head groups and FFN-channel groups, with differentiable routers learning group decisions end to end. It couples this fine-grained allocation with GPU-oriented group-wise skipping for both prefill and decoding.

3 WIDE: Preliminary, Training, and Inference Design

As previously introduced, WIDE is a dynamic width pruning framework that selectively activates head / channel groups under token-wise routing. In this section, we first define the minimal atomic pruning unit of WIDE, then describe the designs for training and GPU-side execution.

3.1 Preliminaries

Consider a standard Transformer layer with input token embeddings $\mathbf{X} \in \mathbb{R}^{B\times T\times D}$ with B for batch size, T for token length, and D for hidden size, WIDE follows the granularity of SkipGPT (Zhao et al., 2025), which decomposes the entire layer into two computation parts: attention and FFN. In attention, we use T_q, T_k to denote the token length of query and key value, H_q, H_k denote the number of query / key value attention heads, and $d = \frac{D}{H_q}$ for the head size. For FFN, D' denotes the intermediate size of FFN's up scaling. The overall pruning granularity of WIDE is shown in Figure 1, where each token allocates its computation budget based on the atomic pruning group size G.

Attention Pruning Unit For the attention layer, considering the compatibility with the standard FlashAttention tiling scheme that launches over B, H_q, T_q dimensions and consumes one head at each step (Dao, 2024), we assume the head as the minimal pruning atom to skip d or multiple-d under each token routing. Such pruning strategy can be equivalently represented with a mask of the

form:

$$[\mathbf{Q}, \mathbf{K}, \mathbf{V}] = [\mathbf{X}\mathbf{W}_q^\top, \mathbf{X}_k\mathbf{W}_k^\top, \mathbf{X}_v\mathbf{W}_v^\top] \tag{1}$$

$$\mathbf{X}' = \mathsf{S}\left(\underbrace{\text{Attn.}(\mathbf{Q}, \mathbf{K}, \mathbf{V})}_{\mathbb{R}^{B\times T_q\times H_q\times d}} \odot_G \mathcal{M}_{\text{attn}}\right)\mathbf{W}_o^\top \quad \text{where } \mathcal{M}_{\text{attn}} \in \{0,1\}^{B\times T_q\times H_k},\ G = \frac{H_q}{H_k}\times d \tag{2}$$

Here $\odot_G$ denotes group-wise gating, S denotes the reshape operation. The outer mask $\mathcal{M}_{\text{attn}}$ also leaves $\mathbf{W}_k$ and $\mathbf{W}_v$ unpruned to preserve the dense KV cache. Additionally, since recent LLMs predominantly adopt GQA (Ainslie et al., 2023), WIDE sets attention's G_{attn} to the head group size $\frac{H_q}{H_k}\times d$ for preserving compatibility with existing GQA decoding optimizations (e.g., XQA (NVIDIA, 2026; Ye et al., 2025)) while minimizing the degradation of pruning flexibility.

FFN Pruning Unit Unlike attention layers, FFN layers comprise only general matrix multiplication (GEMM) and element-wise operations. For a standard GEMM operation with its left-hand-side operand (LHS) $\mathbf{A} \in \mathbb{R}^{M\times K}$ and right-hand-side operand (RHS) $\mathbf{B} \in \mathbb{R}^{N\times K}$, the tiling strategy typically slices all three logical dimensions M, N, and K, with tile sizes $(BM, BN, BK) \in \mathcal{T}^3, \quad \mathcal{T} = \{2^k \mid k \in \mathbb{Z}_{\geqslant 0}\}$. Based on this, WIDE adopts commonly used tiling sizes as the G for FFN to prune the intermediate dimension D' into contiguous groups. This dimension corresponds to the out-feature in $\mathbf{W}_{\text{up}}/\mathbf{W}_{\text{gate}}$ and the in-feature in $\mathbf{W}_{\text{down}}$, ensuring a single mask matrix to represent the entire FFN layer pruning:

$$\mathbf{X}' = \left(\underbrace{(\phi(\mathbf{X}\mathbf{W}_{\text{gate}}^\top) \odot (\mathbf{X}\mathbf{W}_{\text{up}}^\top))}_{\mathbb{R}^{B\times T\times (D'/G, G)}} \odot_G \mathcal{M}_{\text{ffn}}\right)\mathbf{W}_{\text{down}}^\top \quad \text{where } \mathcal{M}_{\text{ffn}} \in \{0,1\}^{B\times T\times \frac{D'}{G}},\ G \in \mathcal{T} \tag{3}$$

where ϕ denotes element-wise activation function. The FFN unit offers more flexibility in choosing G than the attention unit, as its available group sizes are drawn from the tiling-size set $\mathcal{T}$, trading off pruning quality ($G \downarrow$) against potential acceleration ($G \uparrow$). Therefore, unless otherwise specified, G refers to the FFN group size G_{ffn} hereafter.

Routing Components Given the pruning units and masks above, WIDE uses a bottleneck router to map each input activation to N_G binary decisions, supporting varying dimensions and Gumbel Softmax sampling during training:

$$\mathbf{R} = \mathsf{S}\left(\mathbf{X}\mathbf{W}_1\mathbf{W}_2\right) \in \mathbb{R}^{B\times T\times N_G\times 2} \quad \text{where } \mathbf{W}_1 \in \mathbb{R}^{D\times r},\ \mathbf{W}_2 \in \mathbb{R}^{r\times 2N_G} \tag{4}$$

$$\mathcal{M} = \begin{cases} \mathbf{1}\left[\text{argmax}_{c\in\{0,1\}}\mathbf{R}_{\ldots,c} = 0\right], & \text{when inference,} \\ \text{GumbelSoftmax}_\tau^{\text{hard}}(\mathbf{R})_{\ldots,0}, & \text{when training with straight-through estimator (STE),} \end{cases} \tag{5}$$

where τ is the temperature, $r \ll D$ is the bottleneck rank, and N_G is D/G for attention and D'/G for FFN, with class 0 denotes execution and class 1 for skipping.

Two-stage Training To fully unlock the potential of WIDE, a two-stage training pipeline is introduced to restore model's original performance. In the first stage, called **Router training** (S1), all parameters except the router remain frozen, with calibration data employed only to optimize the routing components. In the next optional **LoRA tuning** (S2) stage, LoRA adapters are added into base models' linear modules to further support performance recovery, with routers remain trainable. Both stages utilize not only the standard language modeling loss but also an auxiliary sparsity loss to ensure the pruning meets the target sparsity. For a model with L layers, the overall objective is:

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \alpha\left|SP - \frac{1}{2L}\sum_{i=1}^{L}\left(\left(1 - \frac{1}{|\mathcal{M}_{\text{attn}}^i|}\sum\mathcal{M}_{\text{attn}}^i\right) + \left(1 - \frac{1}{|\mathcal{M}_{\text{ffn}}^i|}\sum\mathcal{M}_{\text{ffn}}^i\right)\right)\right| \tag{6}$$

where α is a hyperparameter controlling the weight of the sparsity loss, $|\mathcal{M}|$ denotes the total element number of the mask. Here, SP denotes the target routing sparsity. Since the total computational proportion of attention varies with T, we follow SkipGPT and weight attention and FFN equally when computing the overall sparsity.

3.2 Inference-time Group Skipping with Unified Mask Reordering

Unlike MoE-style routing, dynamic pruning does not assume a fixed compute budget for each token. A naive implementation therefore typically needs to explicitly convert routing masks into exact active

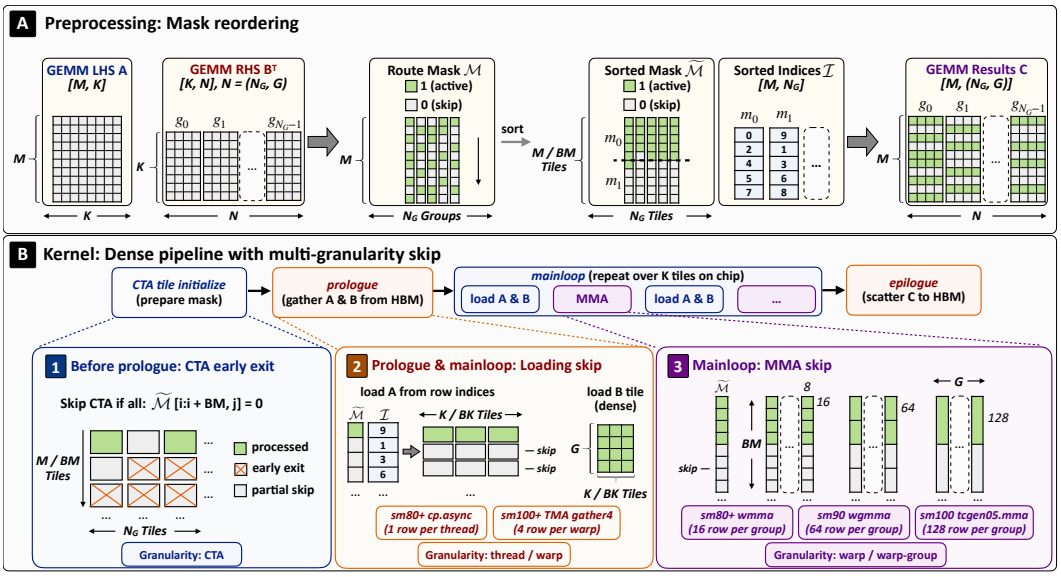


Figure 2: Workflow of WIDE for GEMM N-axis pruning (without split-K). **(a)** Given group size G, WIDE generates a mask matrix of shape $[M, N_G]$ and sorts it along the M dimension, so that active rows of each group are clustered within BM-token tiles. **(b)** Inside the kernel, WIDE first skips fully inactive $[BM, G]$ blocks. For blocks with active entries, it follows the standard GEMM pipeline and applies architecture-dependent intra-block skipping during **A** loading and MMA execution.

indices and then use gather-scatter operations to feed only active parts into kernels. Let $\mathbf{C} = \mathbf{A}\mathbf{B}^\top$, where $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{N\times K}$, and $\mathbf{C} \in \mathbb{R}^{M\times N}$. This strategy introduces two major overheads:

(1). Device-to-host synchronization. The active token layout must be derived from input-dependent masks before constructing the compact gathered matrix, which can trigger device-to-host synchronization and break CUDA Graph execution.

(2). Extra memory movement and storage. For finer-grained dynamic width pruning, each token may select a different subset of $\mathbf{B}$. A naive N-axis implementation therefore incurs traffic of $MK + 3pMNK + 3pMN + MN$, while its K-axis counterpart incurs $3pMK + 3pMNK + MN$, both require intermediate storage of $\mathcal{O}(pMNK)$ for token-specific weight subsets (see Appendix A).

Under this notation, WIDE uses a unified mask-reordering step with fused gather-scatter kernels while retaining the original tile-based pipeline, as illustrated in Fig. 2. Specifically, dynamic routing is represented by a binary mask $\mathcal{M} \in \{0,1\}^{M\times N_G}$, where $N_G = 1$ for M-axis routing (i.e., dynamic depth) and $N_G = D_\mathrm{p}/G$ for width pruning over the pruned feature dimension D_p. For each group g, sorting the mask along M produces a reordered mask $\widetilde{\mathcal{M}}_{:,g}$ and physical row indices $\mathcal{I}_{:,g}$:

$$(\widetilde{\mathcal{M}}_{:,g}, \mathcal{I}_{:,g}) = \mathrm{SortDesc}(\mathcal{M}_{:,g}) \tag{7}$$

This clusters and packs active rows into a contiguous prefix, while $\mathcal{I}_{:,g}$ gathers activation rows and scatters outputs back to their original positions. After sorting, all but the boundary CTA of each group become fully active or inactive. Active rows within each admitted CTA therefore share a group-aligned weight tile $\mathbf{B}_\mathrm{tile} \in \mathbb{R}^{BN\times BK}$, enabling multi-granularity predicated skipping in the dense-style pipeline without token-specific weight materialization. WIDE then applies unified skipping predicates over progressively finer row groups. Let

$$S_0 = BM, \qquad S_1 = S_\mathrm{ld}, \qquad S_2 = S_\mathrm{mma}, \tag{8}$$

where S_0 is the CTA-level block size, while S_ld and S_mma are architecture-dependent granularities for **A** loading and MMA execution. For level $\ell \in \{0,1,2\}$, group g, and row block m, WIDE computes

$$s^{(\ell)}_{m,g} = \bigvee_{i=mS_\ell}^{\min((m+1)S_\ell, M)-1} \widetilde{\mathcal{M}}_{i,g} \tag{9}$$

If $s^{(\ell)}_{m,g} = 0$, the corresponding unit at level ℓ is skipped; otherwise it is executed. This single predicate instantiates three stages: $\ell = 0$ performs hardware-agnostic CTA early exit, $\ell = 1$ skips global memory loading for inactive rows in **A**, and $\ell = 2$ skips inactive MMA fragments. Under this abstraction, the proposed **GEMM-MN**, **GEMM-K**, and **Attention** kernels cover $\mathbf{W}_q$/$\mathbf{W}_\mathrm{up}$/$\mathbf{W}_\mathrm{gate}$

projections, $\mathbf{W}_o$/$\mathbf{W}_{\text{down}}$ projections, and FlashAttention operations with customized routing group, allowing WIDE to exploit the acceleration potential brought by sparsity while preserving its dense-style mainloops. At the kernel level, GEMM-K naturally requires K-axis accumulation to be partitioned according to G. We place multiple K-axis routing groups within a CTA and process them through an outer loop to balance atomic-add pressure. For decoding, reordering-free GEMV/FlashDecoding fast paths have been introduced, which directly perform CTA-level skipping from the original mask. The detailed implementation for each kernel is available at Appendix B.

4 Experiments

4.1 Settings

Training We use Llama3.1-8B and Llama3.2-3B (Grattafiori et al., 2024) as the main backbone models, along with RedPajama-1T (Weber et al., 2024) as the corpus for calibration-only (S1) and LoRA tuning (S2) stages. The weight α in Equation 6 is set to 20, following SkipGPT. As for the Qwen3 family (Yang et al., 2025), Infinity-Instruct (Li et al., 2025b) is introduced as the calibration set to retain the capability of instruction following. More details are available at Appendix C.

Evaluation For evaluating the capability of pruned models, the following metrics are included: perplexity on WikiText2 (Merity et al., 2016), and zero-shot accuracy on ARC-Easy, ARC-Challenge (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). For Qwen3 family, 5-shot MMLU (Hendrycks et al., 2021) is added as an extended likelihood task, with additional IFEval (Zhou et al., 2023), MBPP+ (Austin et al., 2021), and GSM8K (Cobbe et al., 2021) for evaluating the generation capability. The average accuracy and performance retention ratio are computed over the selected likelihood / generation tasks based on the dense model.

Inference Implementation For kernel-level throughput evaluation, we use the Triton benchmark interface (Tillet et al., 2019) and report TFLOPs under CUDA Graph replay and random mask. For end-to-end acceleration, an ELANA-style profiling tool (Chiang et al., 2025) is introduced to measure Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT) under CUDA Graph execution. We implement WIDE kernels in the tile-level Domain Specific Languages (DSLs), e.g., Triton (Tillet et al., 2019) and TileLang (Wang et al., 2026), as well as the lower-level CuTe template library in CUTLASS. All inference experiments are mainly conducted on GPUs with sm120 (NVIDIA RTX 5090) and sm80 (A100-SMX4-40G) architectures.

Baselines We compare WIDE with representative pruning methods, including static depth pruning (Shortened LLaMA (Kim et al., 2024), CoopPruner (Ding et al., 2026)), static width pruning (SliceGPT (Ashkboos et al., 2024), Týr-the-Pruner (Li et al., 2025a), DDP (Huang et al., 2026)), and dynamic pruning (D-LLM (Jiang et al., 2024), SkipGPT (Zhao et al., 2025)). We also compare against

Table 1: Performance on **static depth**, **static width**, and **dynamic** pruning methods under Llama3 and various target sparsity, training stages, and tasks. The S1 denotes calibration-only / router training stage, and S2 denotes LoRA tuning stage. The **WT2** denotes WikiText2 perplexity (↓), **Avg. Acc.** reports average accuracy (↑) on 7 likelihood tasks. G in WIDE (G) denotes the FFN group size.

Methods	Llama3.1-8B								Llama3.2-3B			
	25% sparsity				50% sparsity				50% sparsity			
	S1		S2		S1		S2		S1		S2	
	WT2	Avg. Acc.	WT2	Avg. Acc.	WT2	Avg. Acc.	WT2	Avg. Acc.	WT2	Avg. Acc.	WT2	Avg. Acc.
Dense	7.71	71.55 (100.00%)	7.91	71.36 (99.73%)	7.71	71.55 (100.00%)	7.91	71.36 (99.73%)	9.77	64.91 (100.00%)	9.86	65.00 (100.14%)
Shortened-ppl	25.84	47.65 (66.59%)	12.33	57.25 (80.01%)	473.15	39.62 (55.37%)	22.47	47.26 (66.05%)	676.05	38.23 (58.89%)	29.75	45.83 (70.61%)
Shortened-taylor	30.07	49.30 (68.90%)	12.12	59.83 (83.63%)	4.86e+7	35.99 (50.30%)	22.30	49.97 (69.83%)	2.32e+5	39.35 (60.63%)	31.50	45.84 (70.61%)
CoopPruner	22.96	53.33 (74.54%)	12.03	61.25 (85.61%)	503.86	39.28 (54.90%)	22.66	49.63 (69.36%)	1.79e+5	36.62 (56.41%)	42.69	41.82 (64.43%)
SliceGPT	22.57	53.37 (74.59%)	14.88	60.72 (84.87%)	85.81	38.71 (54.10%)	28.76	46.01 (64.31%)	96.69	37.62 (57.96%)	31.13	42.76 (65.87%)
Týr-the-Pruner	12.89	61.43 (85.86%)	11.49	64.20 (89.73%)	59.88	45.47 (63.55%)	37.99	53.46 (74.72%)	87.37	42.60 (65.63%)	53.11	45.69 (70.38%)
DDP	12.41	64.35 (89.93%)	12.10	64.24 (89.79%)	21.83	53.05 (74.14%)	19.52	54.64 (76.36%)	31.20	48.03 (74.00%)	25.66	48.88 (75.30%)
D-LLM	69.45	38.89 (54.35%)	10.29	66.84 (93.42%)	504.48	36.32 (50.77%)	19.31	56.07 (78.36%)	3877.53	34.99 (53.91%)	193.97	38.66 (59.56%)
SkipGPT	15.63	51.68 (72.23%)	9.87	69.48 (97.11%)	96.04	42.52 (59.42%)	13.90	61.60 (86.10%)	156.58	36.74 (56.60%)	18.91	54.33 (83.70%)
WIDE (32)	**8.49**	**70.06 (97.92%)**	**8.61**	**70.19 (98.09%)**	14.15	**61.84 (86.43%)**	11.99	**64.82 (90.60%)**	**17.28**	**57.00 (87.81%)**	15.28	**58.77 (90.53%)**
WIDE (64)	10.37	64.89 (90.69%)	10.32	69.83 (97.59%)	14.96	61.18 (85.50%)	12.05	64.65 (90.36%)	19.02	56.40 (86.89%)	**15.17**	58.36 (89.90%)
WIDE (128)	10.04	63.65 (88.96%)	12.51	69.53 (97.18%)	**12.57**	61.49 (85.94%)	**11.43**	64.34 (89.92%)	18.11	55.42 (85.38%)	15.26	58.11 (89.52%)

Table 2: Performance on Qwen3-14B with various target sparsity. **Like. Acc.** denotes average accuracy in 8 likelihood tasks, and **Gen. Acc.** denotes average accuracy in 3 generation tasks.

Methods	Qwen3-14B							
	25% sparsity				50% sparsity			
	S1		S2		S1		S2	
	Like. Acc.	Gen. Acc.	Like. Acc.	Gen. Acc.	Like. Acc.	Gen. Acc.	Like. Acc.	Gen. Acc.
Dense	73.57 (100.00%)	84.79 (100.00%)	76.16 (103.52%)	83.86 (98.91%)	73.57 (100.00%)	84.79 (100.00%)	76.16 (103.52%)	83.86 (98.91%)
Shortened-ppl	44.55 (60.55%)	4.33 (5.11%)	59.90 (81.42%)	40.55 (47.82%)	34.39 (46.74%)	3.22 (3.79%)	44.91 (61.05%)	5.68 (6.69%)
Shortened-taylor	36.76 (49.97%)	4.93 (5.81%)	65.26 (88.70%)	67.09 (79.13%)	35.50 (48.26%)	3.79 (4.47%)	46.62 (63.37%)	20.03 (23.63%)
CoopPruner	44.23 (60.12%)	5.92 (6.98%)	62.89 (85.48%)	52.20 (61.56%)	35.66 (48.47%)	4.45 (5.25%)	47.08 (63.99%)	11.44 (13.50%)
DDP	65.26 (88.71%)	73.23 (86.37%)	68.77 (93.48%)	74.62 (88.00%)	53.00 (72.04%)	52.20 (61.56%)	55.74 (75.77%)	54.88 (64.72%)
SkipGPT	61.63 (83.77%)	38.34 (45.21%)	71.96 (97.80%)	76.67 (90.42%)	35.15 (47.77%)	8.09 (9.54%)	57.20 (77.74%)	68.12 (80.34%)
WIDE (32)	72.70 (98.82%)	77.98 (91.96%)	74.01 (100.59%)	80.89 (95.40%)	68.95 (93.71%)	**71.48 (84.31%)**	71.26 (96.86%)	73.11 (86.22%)
WIDE (64)	**73.19 (99.49%)**	77.98 (91.97%)	74.22 (100.88%)	80.80 (95.29%)	**70.15 (95.35%)**	69.82 (82.35%)	71.34 (96.96%)	**75.26 (88.76%)**
WIDE (128)	73.11 (99.37%)	**78.57 (92.67%)**	**74.33 (101.03%)**	**81.46 (96.08%)**	69.24 (94.11%)	70.81 (83.52%)	**71.82 (97.62%)**	74.85 (88.28%)

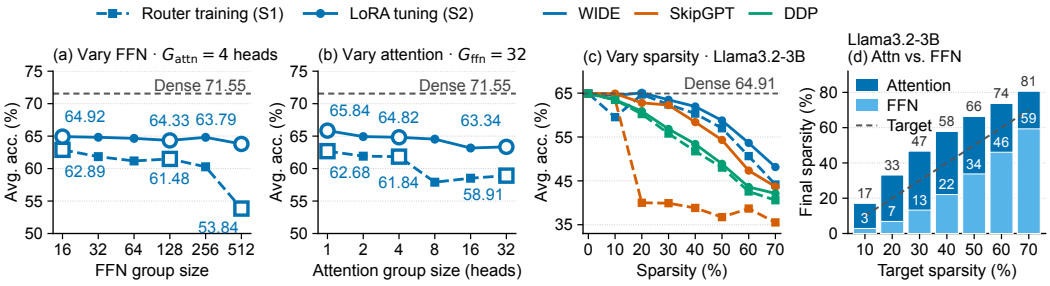


Figure 3: Effects of pruning group size and target sparsity. (a–b) WIDE on Llama3.1-8B at 50% sparsity, varying FFN group size with $G_{\text{attn}} = 4$ heads, or attention group size with $G = 32$. (c) WIDE ($G = 32$), DDP, and SkipGPT's performance on Llama3.2-3B across target sparsity. (d) Final sparsity distribution of WIDE ($G = 32$) during Llama3.2-3B's training.

Dense-to-MoE (Pei et al., 2026), semi-structured pruning (Sun & Sakuma, 2026), and activation-based dynamic pruning (Shrestha et al., 2025; Chen et al., 2026a) in Appendix D. Baseline training details are provided in Appendix C.

4.2 Main Results

Benchmarks on Llama3-family and Qwen3 Table 1 and 2 compare pruning methods on the Llama3 / Qwen3 families at 25% and 50% sparsity. WIDE achieves the strongest performance across both budgets, with the largest gains under 50% sparsity and calibration-only settings. On Llama3, WIDE outperforms the strongest static width and dynamic depth baselines by up to 18.68% and 55.14%, respectively. After LoRA tuning, the gains remain 20.23% and 8.17%. Broader comparisons with Dense-to-MoE, semi-structured pruning, and activation-based unstructured pruning (Table 7) further show that WIDE remains competitive against methods with even finer pruning granularity. On Qwen3-14B, WIDE further retains nearly lossless likelihood-task performance after tuning, together with 96% and 89% of dense generation-task performance at 25% and 50% sparsity, respectively.

Benefits from Finer-grained Routing Beyond the end-to-end task-level advantage, WIDE also benefits from its finer-grained decisions in reducing layer-wise error accumulation. As shown in Table 3 and Appendix D, with the dense model as the reference, WIDE consistently yields lower errors than SkipGPT both at individual layers and at the final-layer output, which also helps explain its stronger performance over SkipGPT.

Table 3: Pruned models' final-layer error and similarity to dense Qwen3-14B at 50% sparsity. WIDE is closer in both stages.

Methods	Relative $\ell_2 \downarrow$	Cos. Sim.$\uparrow$
S1		
SkipGPT	1.3090	0.7337
WIDE (128)	**0.5778**	**0.8556**
S2		
SkipGPT	0.6642	0.8447
WIDE (128)	**0.5510**	**0.8506**

Performance across Group Size and Sparsity For WIDE, group size G and target sparsity govern the quality–speed trade-off. G_{ffn} spans $[1, D']$, from row-wise to full-FFN skipping, while G_{attn} typically uses $\frac{H_q}{H_k} \times d$ but can vary over $[1, H_q] \times d$, up to SkipGPT's

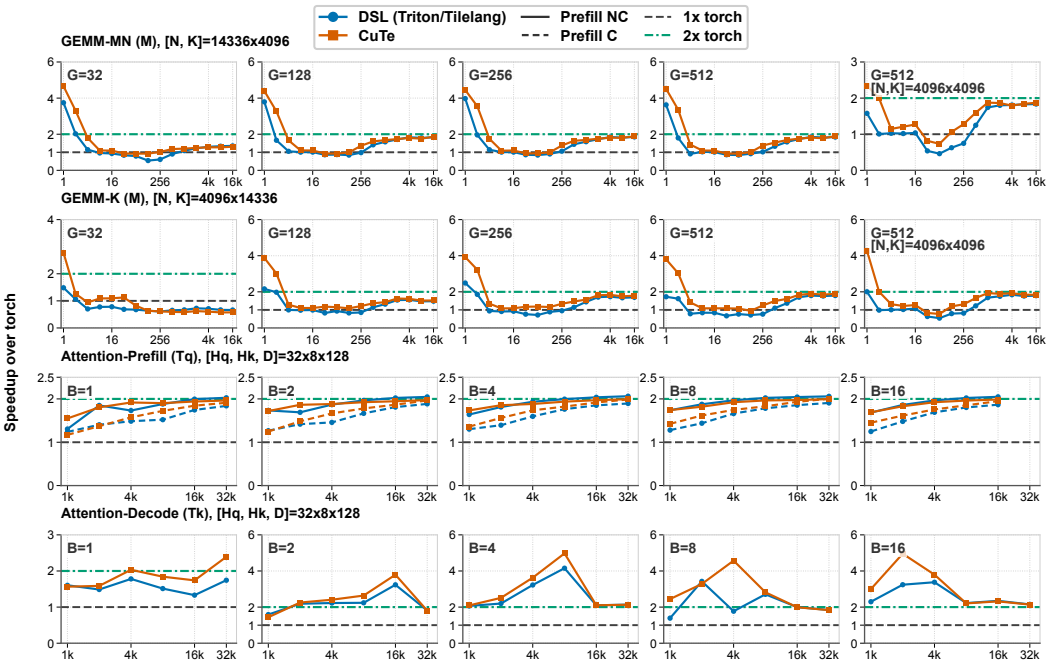


Figure 4: Kernel-level speedups of the four types of WIDE kernels on Llama3.1-8B shapes, with a random 50% sparsity mask. NC and C denote non-causal and causal attention kernels. The dense baseline is set to PyTorch cuBLAS for GEMM, and SDPA (FlashAttention/FlashDecoding backend) for Attention. Overhead from workspace initialization is also taken into account.

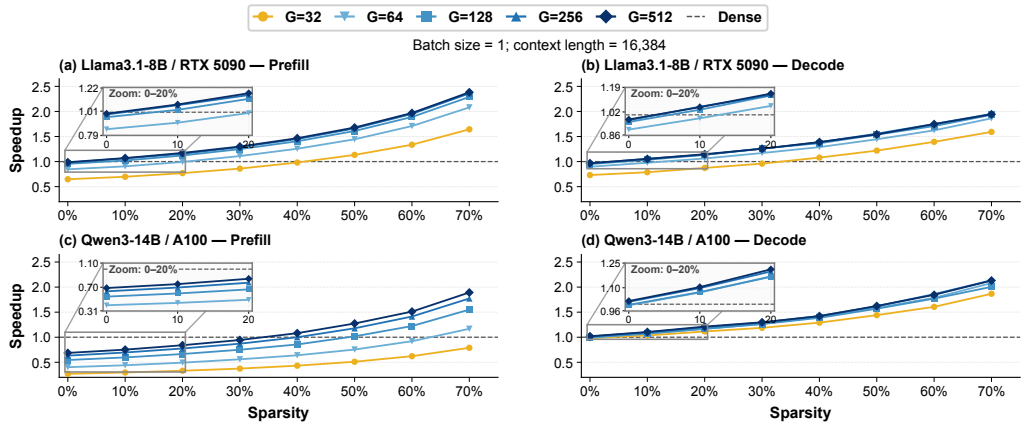


Figure 5: WIDE's end-to-end speedup over the dense baseline across models, sparsity, and G.

granularity. As shown in Fig. 3 (a–b), router training favors smaller FFN groups: increasing the group size to 512 reduces accuracy by 9.05 points, also helping explain SkipGPT's weak stage-1 results. After LoRA tuning, sensitivity to group size drops substantially, with only 2.50 and 1.13-point accuracy reductions for attention and FFN from the finest to coarsest granularity, respectively. For target sparsity (Fig. 3 (c–d)), WIDE's accuracy declines steadily but remains above SkipGPT and DDP, with router-trained WIDE outperforming LoRA-tuned SkipGPT at most sparsity levels. In terms of sparsity allocation, WIDE follows a trend similar to SkipGPT by assigning more sparsity budget to attention, while the attention-to-FFN sparsity ratio gradually decreases from 5.9:1 at 10% target sparsity to 1.3:1 at 70%.

Kernel-level Speedup As shown above, WIDE's quality is insensitive to group size after LoRA tuning, favoring larger G that provides more tiling options and improves kernel pipeline efficiency. The kernel benchmarks in Fig. 4, using random 50% routing masks, show that GEMM-K approaches the ideal speedup over dense kernels only when $G > 128$, as smaller K-axis groups provide insufficient mainloop work to amortize pipeline overhead. GEMM-MN likewise requires $G \geqslant 128$ to enable flexible tiling. Head-wise attention kernels are less sensitive to G, although GQA-packing decoding still requires GQA-aligned groups (Appendix D, Fig. 10). In addition, pruned GEMM provides limited acceleration when $BM \geqslant M$ under fully random masks, since the probability of a tile being fully inactive is close to zero in such scenario, leaving limited opportunities for CTA-level skipping.

End-to-end Speedup Building on the kernel-level results, we further evaluate end-to-end acceleration across different hardware and models under varying sparsity levels and FFN group sizes, with input shape fixed at $B = 1$, $T = 16{,}384$. As shown in Fig. 5, the best setting on sm120 with $G \leqslant 128$ retains 0.98x of dense throughput, while 50% sparsity increases the speedup to 1.68x for prefill and 1.55x for decoding. On sm80, which lacks support for vectorized atomic operations, WIDE cannot sustain the same prefill gains, as the pruned GEMM, especially GEMM-K, rely heavily on split-K parallelism and atomic bandwidth. Nevertheless, WIDE still delivers up to a 1.62× speedup in decoding.

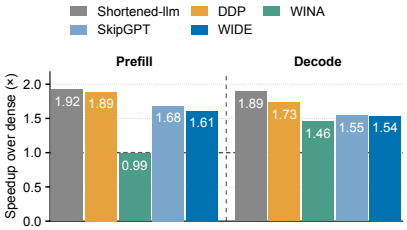


Figure 6: 50% sparsity's end-to-end speedup over the Llama3.1-8B dense baseline across pruning methods.

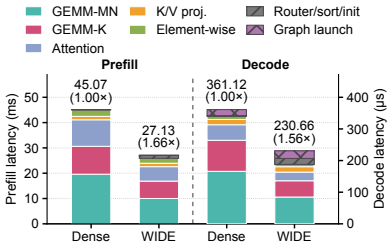


Figure 7: Layer-wise latency breakdown on Llama3.1-8B at 50% sparsity.

Beyond the acceleration landscape of WIDE itself, Fig. 6 further compares the end-to-end speedups of different pruning schemes under the same input shapes. WINA uses the sparse GEMV kernel from TEAL (Liu et al., 2025), while SkipGPT uses the same kernels as WIDE. Throughout the evaluation, WIDE achieves acceleration comparable to SkipGPT. WINA delivers practical speedup for single-request decoding, but is not compatible with prefill workloads. Overall, WIDE provides a better balance between practical acceleration and routing flexibility.

After evaluating the end-to-end acceleration potential of WIDE , we further analyze how close it comes to the theoretical speedup bound. In the averaged forward trace of a Llama3.1-8B layer at 50% sparsity with $G = 128$ (Fig. 7), the pruned path reaches $1.82\times$ speedup for prefill and $1.92\times$ for decoding, close to the ideal bound, while the remaining work accounts for about 16.7% and 29.1% of prefill and decoding time, respectively. These overheads come from the additional computation of routing-based pruning, workspace initialization in the optimized kernels, and fixed costs from unpruned operations, e.g., element-wise operators, explaining why the overall speedup remains below the theoretical bound.

5 Conclusion

In this work, we identify the potential limitation for current naive token-wise dynamic pruning pipeline under real-world inference system, and present a finer-grained group-based structured dynamic width pruning method with unified acceleration design for routing-based dynamic structural pruning. The presented WIDE uses lightweight bottleneck routers and differentiable masks to select GQA head groups and FFN channel groups for each token, pushing structured pruning below the granularity of layers and sublayers. On the system side, WIDE formulates mask-based dynamic pruning as two steps: mask reordering and progressive intra-kernel skipping. Across the evaluations, WIDE consistently beats existing static and dynamic structured pruning methods in overall quality retention, while effectively translating sparsity into practical speedups under both prefill and decode workloads. These results demonstrate the potential of WIDE for adaptation and optimization across broader model families and inference systems.

AI USE STATEMENT

In this work, we used generative AI tools for translation, language editing, draft pseudocode (Appendix B), memory analysis (Appendix A), and figure preparation using author-provided code and data. We have reviewed and polished all AI-assisted work by manually checking and reviewing the contents from generative AI tools. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, December 2023.

Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. SliceGPT: Compress Large Language Models by Deleting Rows and Columns. In *The Twelfth International Conference on Learning Representations*, May 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. https://arxiv.org/abs/2108.07732v1, August 2021.

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

Sihan Chen, Dan Zhao, Jongwoo Ko, Colby Banbury, Huiping Zhuang, Luming Liang, Pashmina Cameron, and Tianyi Chen. WINA: Weight Informed Neuron Activation for Accelerating Large Language Model Inference. In *The Fourteenth International Conference on Learning Representations*, April 2026a.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. AVO: Agentic Variation Operators for Autonomous Evolutionary Search. http://arxiv.org/abs/2603.24517, March 2026b.

Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations. http://arxiv.org/abs/2308.06767, August 2024.

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

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

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. https://arxiv.org/abs/2110.14168v2, October 2021.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA Agent: Large-Scale Agentic RL for High-Performance CUDA Kernel Generation. http://arxiv.org/abs/2602.24286, February 2026.

Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*, May 2024.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro, 2026.

Xuan Ding, Pengyu Tong, Ranjie Duan, Yunjian Zhang, Rui Sun, and Yao Zhu. Pruning as a Cooperative Game: Surrogate-Assisted Layer Contribution Estimation for Large Language Models. In *The Fourteenth International Conference on Learning Representations*, April 2026.

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

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

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, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL `https://zenodo.org/records/12608602`.

Aayush Gautam, Mukul Gagrani, Junyoung Park, Mingu Lee, Chiris Lott, and Narasimha Reddy. Fast Forward: Accelerating LLM Prefill with Predictive FFN Sparsity. https://arxiv.org/abs/2602.00397v1, January 2026.

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, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane

Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu

Yang, Zhiwei Zhao, and Zhiyu Ma. The Llama 3 Herd of Models. http://arxiv.org/abs/2407.21783, November 2024.

Chao Han, Yijuan Liang, Zihao Xuan, Daokuan Wu, Wei Zhang, and Xiaoyu Shen. Informed Routing in LLMs: Smarter Token-Level Computation for Faster Inference. http://arxiv.org/abs/2510.13831, October 2025.

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

Zicheng He, Anhao Zhao, Xiaoyu Shen, Chen Wu, and Lei He. SkipOPU: An FPGA-based Overlay Processor for Large Language Models with Dynamically Allocated Computation. https://arxiv.org/abs/2603.14785v1, March 2026.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations*, May 2021.

Haozhe Hu, Hao Wu, Anhao Zhao, Longwei Ding, Peiran Yin, Yunpu Ma, and Xiaoyu Shen. Beyond FLOPs: Benchmarking Real Inference Acceleration of LLM Pruning under a GEMM-Centric Taxonomy. http://arxiv.org/abs/2606.09080, June 2026.

Weiyu Huang, Pengle Zhang, Xiaolu Zhang, Jun Zhou, Jun Zhu, and Jianfei Chen. Deterministic Differentiable Structured Pruning for Large Language Models. http://arxiv.org/abs/2603.08065, May 2026.

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

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

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

Guanchen Li, Yixing Xu, Zeping Li, Ji Liu, Xuanwu Yin, Dong Li, and Emad Barsoum. Týr-the-Pruner: Structural Pruning LLMs via Global Sparsity Distribution Optimization. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, October 2025a.

Jijie Li, Li Du, Hanyu Zhao, Bo-wen Zhang, Liangdong Wang, Boyan Gao, Guang Liu, and Yonghua Lin. Infinity Instruct: Scaling Instruction Selection and Synthesis to Enhance Language Models. https://arxiv.org/abs/2506.11116v1, June 2025b.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, May 2024.

James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. Training-Free Activation Sparsity in Large Language Models. In *The Thirteenth International Conference on Learning Representations*, April 2025.

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

Xin Men, Mingyu Xu, Qingyu Zhang, Qianhao Yuan, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. ShortGPT: Layers in Large Language Models are More Redundant Than You Expect. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 20192–20204, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1035.

Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. 2026. doi: 10.20944/preprints202604.0428.v3.

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

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

NVIDIA. Tensorrt-llm. `https://github.com/NVIDIA/TensorRT-LLM`, 2026.

Zehua Pei, Hui-Ling Zhen, Lancheng Zou, Xianzhi Yu, Wulong Liu, Sinno Jialin Pan, Mingxuan Yuan, and Bei Yu. Analytical FFN-to-MoE Restructuring via Activation Pattern Analysis. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4777–4789, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.218.

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

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

Susav Shrestha, Bradley Settlemyer, Nikoli Dryden, and A. L. Narasimha Reddy. Polar Sparsity: High Throughput Batched LLM Inferencing with Scalable Contextual Sparsity. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, October 2025.

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

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A Simple and Effective Pruning Approach for Large Language Models. In *The Twelfth International Conference on Learning Representations*, May 2024.

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, Yanru Chen, Yanxu Chen, Yicun Chen, Yimin Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Ziwei Chen, Dazhi Cheng, Minghan Chu, Jialei Cui, Jiaqi Deng, Muxi Diao, Hao Ding, Mengfan Dong, Mengnan Dong, Yuxin Dong, Yuhao Dong, Angang Du, Chenzhuang Du, Dikang Du, Lingxiao Du, Yulun Du, Yu Fan, Shengjun Fang, Qiulin Feng, Yichen Feng, Garimugai Fu, Kelin Fu, Hongcheng Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Chengyang Gong, Xiaochen Gong, Zhuoma Gongque, Qizheng Gu, Xinran Gu, Yicheng Gu, Longyu Guan, Yuanying Guo, Xiaoru Hao, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Jiaxi Hu, Yangyang Hu, Zhenxing Hu, Ke Huang, Ruiyuan Huang, Weixiao Huang, Zhiqi Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yu Jing, Guokun Lai, Aidi Li, C. Li, Cheng Li, Fang Li, Guanghe Li, Guanyu Li, Haitao Li, Haoyang Li, Jia Li, Jingwei Li, Junxiong Li, Lincan Li, Mo Li, Weihong Li, Wentao Li, Xinhang Li, Xinhao Li, Yang Li, Yanhao Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zheming Li, Weilong Liao, Jiawei Lin, Xiaohan Lin, Zhishan Lin, Zichao Lin, Cheng Liu, Chenyu Liu, Hongzhang Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Tianyu Liu, Weizhou Liu, Xiangyan Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yuanxin Liu, Yue Liu, Zhengying Liu, Zhongnuo Liu, Enzhe Lu, Haoyu Lu, Zhiyuan Lu, Junyu Luo, Tongxu Luo, Yashuo Luo, Long Ma, Yingwei Ma, Shaoguang Mao, Yuan Mei, Xin Men, Fanqing Meng, Zhiyong Meng, Yibo Miao, Minqing Ni, Kun Ouyang, Siyuan Pan, Bo Pang, Yuchao Qian, Ruoyu Qin, Zeyu Qin, Jiezhong Qiu, Bowen Qu, Zeyu Shang, Youbo Shao, Tianxiao Shen, Zhennan Shen, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Feifan Song, Pengwei Song, Tianhui Song, Xiaoxi Song, Hongjin Su, Jianlin Su, Zhaochen Su, Lin Sui, Jinsong Sun, Junyao Sun, Tongyu Sun, Flood Sung, Yunpeng Tai, Chuning Tang, Heyi

Tang, Xiaojuan Tang, Zhengyang Tang, Jiawen Tao, Shiyuan Teng, Chaoran Tian, Pengfei Tian, Ao Wang, Bowen Wang, Chensi Wang, Chuang Wang, Congcong Wang, Dingkun Wang, Dinglu Wang, Dongliang Wang, Feng Wang, Hailong Wang, Haiming Wang, Hengzhi Wang, Huaqing Wang, Hui Wang, Jiahao Wang, Jinhong Wang, Jiuzheng Wang, Kaixin Wang, Linian Wang, Qibin Wang, Shengjie Wang, Shuyi Wang, Si Wang, Wei Wang, Xiaochen Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yipu Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhexu Wang, Zihan Wang, Zizhe Wang, Chu Wei, Ming Wei, Chuan Wen, Zichen Wen, Chengjie Wu, Haoning Wu, Junyan Wu, Rucong Wu, Wenhao Wu, Yuefeng Wu, Yuhao Wu, Yuxin Wu, Zijian Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Yuchong Xie, Yifei Xin, Bowei Xing, Boyu Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinbo Xu, Xinran Xu, Yangchuan Xu, Yichang Xu, Yuemeng Xu, Zelai Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Guangyao Yang, Hao Yang, Junwei Yang, Kai Yang, Ningyuan Yang, Ruihan Yang, Xiaofei Yang, Xinlong Yang, Ying Yang, Yi Yang, Yi Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Dan Ye, Wenjie Ye, Zhuorui Ye, Bohong Yin, Chengzhen Yu, Longhui Yu, Tao Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Xiaokun Yuan, Yang Yue, Weihao Zeng, Dunyuan Zha, Haobing Zhan, Dehao Zhang, Hao Zhang, Jin Zhang, Puqi Zhang, Qiao Zhang, Rui Zhang, Xiaobin Zhang, Y. Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yushun Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Chenguang Zhao, Feifan Zhao, Jinxiang Zhao, Shuai Zhao, Xiangyu Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Junfeng Zhong, Longguang Zhong, Weiming Zhong, M. Zhou, Runjie Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yuxuan Zhu, Zhen Zhu, Jingze Zhuang, Weiyu Zhuang, Ying Zou, and Xinxing Zu. Kimi K2.5: Visual Agentic Intelligence. http://arxiv.org/abs/2602.02276, February 2026.

Philippe Tillet, H. T. Kung, and David Cox. 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, pp. 10–19, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6719-6. doi: 10.1145/3315508.3329973.

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. TileLang: Bridge Programmability and Performance in Modern Neural Kernels. In *The Fourteenth International Conference on Learning Representations*, April 2026.

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. Redpajama: an open dataset for training large language models. *NeurIPS Datasets and Benchmarks Track*, 2024.

Hao Wu, Yingqi Fan, Dai Jinyang, Junlong Tong, Yunpu Ma, and Xiaoyu Shen. HiDrop: Hierarchical Vision Token Reduction in MLLMs via Late Injection, Concave Pyramid Pruning, and Early Exit. In *The Fourteenth International Conference on Learning Representations*, April 2026a.

Hao Wu, Junlong Tong, Xudong Wang, Yang Tan, Changyu Zeng, Anastasia Antsiferova, and Xiaoyu Shen. From data to model: A survey of the compression lifecycle in mllms. February 2026b. doi: 10.36227/techrxiv.177220375.55495124/v1. URL `http://dx.doi.org/10.36227/techrxiv.177220375.55495124/v1`.

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

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, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger

Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. http://arxiv.org/abs/2505.09388, May 2025.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. In *Eighth Conference on Machine Learning and Systems*, May 2025.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and Kernel Pipelining Co-Design for Asymmetric Hardware Scaling. http://arxiv.org/abs/2603.05451, March 2026.

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

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

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. SGLang: Efficient Execution of Structured Language Model Programs. http://arxiv.org/abs/2312.07104, June 2024.

Longguang Zhong, Fanqi Wan, Ruijun Chen, Xiaojun Quan, and Liangzhi Li. BlockPruner: Fine-grained Pruning for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 5065–5080, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.262.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-Following Evaluation for Large Language Models. https://arxiv.org/abs/2311.07911v1, November 2023.

A THE COST FOR NAIVE GATHER-SCATTER IMPLEMENTATIONS

In this section, we first provide a narrow theoretical memory accounting for naive gather-scatter implementations of M-axis, N-axis, and K-axis dynamic pruning. We count the logical data-element transfers required to materialize and consume the explicit tensors in the PyTorch-style implementations below. Routing masks, indices, offsets, and their construction, storage, and traffic are excluded, as are hardware-specific cache effects and framework workspace. We assume that $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ use the same data type, so byte traffic is obtained by multiplying T_M, T_N, or T_K by the data-type size. Consider a dense GEMM

$$\mathbf{C} = \mathbf{A}\mathbf{B}^\top, \qquad \mathbf{A} \in \mathbb{R}^{M\times K}, \quad \mathbf{B} \in \mathbb{R}^{N\times K}, \quad \mathbf{C} \in \mathbb{R}^{M\times N} \tag{10}$$

The ideal dense traffic is

$$T_{\text{dense}} \approx MK + NK + MN \tag{11}$$

which corresponds to reading $\mathbf{A}$, reading $\mathbf{B}$, and writing $\mathbf{C}$.

Naive M-axis pruning M-axis pruning (i.e., dynamic depth pruning) selects a subset of token rows for the current operator. Let $M' = pM$ be the number of active rows, where p is the active ratio, and let $I_M \in \mathbb{Z}^{M'}$ denote the active row indices. A naive PyTorch-style implementation gathers active rows, computes the compact GEMM, and scatters the result back like:

```
# A: [M, K]
# B: [N, K]
# I_M: [M']

A_sub = A[I_M, :]                  # [M', K]
C_sub = A_sub @ B.T                # [M', N]

C = torch.zeros(M, N, device=A.device, dtype=A.dtype)
C[I_M, :] = C_sub                  # scatter back to dense output
```

When ignoring index traffic, the total memory traffic is

$$\begin{aligned} T_M &\approx 3M'K + NK + 3M'N + MN \\ &= 3pMK + NK + 3pMN + MN \end{aligned} \tag{12}$$

The terms $3M'K$ and $3M'N$ come from materializing and rereading $\mathbf{A}_{\text{sub}}$ and $\mathbf{C}_{\text{sub}}$, respectively, while MN comes from initializing the dense output tensor. The additional storage, excluding the final dense output $\mathbf{C}$, is

$$\begin{aligned} S_M &\approx M'K + M'N \\ &= pMK + pMN \end{aligned} \tag{13}$$

Naive N-axis pruning. For N-axis pruning, different N-axis groups are selected for each token. We divide the N-axis into groups of size G and assume that each token selects R groups in this baseline. The active N-axis extent is

$$N' = RG = pN \tag{14}$$

where p denotes the active ratio along the N-axis. Let $I_N \in \mathbb{Z}^{M\times R}$ denote the selected group indices for each token. The fully vectorized PyTorch-style implementation below require to expand the group indices and materializes token-specific subsets of $\mathbf{B}$:

```
# A: [M, K]
# B: [N, K]
# I_N: [M, R], group indices
# G: group size
# N_prime = R * G

offset = torch.arange(G, device=A.device)              # [G]
```

```
J_N = I_N[..., None] * G + offset                          # [M, R, G]
J_N = J_N.reshape(M, R * G)                                # [M, N']

B_sub = B[J_N, :]                                          # [M, N', K]

C_sub = torch.einsum("mk,mnk->mn", A, B_sub)               # [M, N']

C = torch.zeros(M, N, device=A.device, dtype=A.dtype)
C.scatter_(dim=1, index=J_N, src=C_sub)                    # [M, N]
```

Ignoring index traffic and the temporary offset data, the overall memory traffic is

$$\begin{aligned} T_N &\approx MK + 3MN'K + 3MN' + MN \\ &= MK + 3pMNK + 3pMN + MN \end{aligned} \tag{15}$$

The dominant term $3MN'K$ comes from reading selected rows of $\mathbf{B}$, writing the gathered tensor $\mathbf{B}_{\text{sub}}$, and reading $\mathbf{B}_{\text{sub}}$ again during token-wise dot products. Unlike dense GEMM, where $\mathbf{B}$ is shared across all M rows and read once as NK, the naive N-axis implementation replicates selected weight groups for each token, with additional storage in

$$\begin{aligned} S_N &\approx MN'K + MN' \\ &= pMNK + pMN \end{aligned} \tag{16}$$

which is dominated by the per-token gathered weight tensor $\mathbf{B}_{\text{sub}} \in \mathbb{R}^{M\times N'\times K}$, where the remaining MN' entries belong to $\mathbf{C}_{\text{sub}}$.

Naive K-axis pruning. For attention output and FFN down projections, K-axis pruning is applied along the reduction dimension K. We divide the K-axis into groups of size G and assume that each token selects R groups. The active (K)-axis extent is

$$K' = RG = pK \tag{17}$$

where p denotes the active ratio along the K-axis. Let $I_K \in \mathbb{Z}^{M\times R}$ denote the selected K-group indices for each token. A token-specific PyTorch-style implementation can be written as:

```
# A: [M, K]
# B: [N, K]
# I_K: [M, R], group indices
# G: group size
# K_prime = R * G

offset = torch.arange(G, device=A.device)                  # [G]

J_K = I_K[..., None] * G + offset                          # [M, R, G]
J_K = J_K.reshape(M, R * G)                                # [M, K']

A_sub = torch.gather(A, dim=1, index=J_K)                  # [M, K']
B_sub = B[:, J_K].permute(1, 0, 2)                         # [M, N, K']

C = torch.einsum("mk,mnk->mn", A_sub, B_sub)               # [M, N]
```

Ignoring index traffic and the small offset tensor, the total memory traffic is

$$\begin{aligned} T_K &\approx 3MK' + 3MNK' + MN \\ &= 3pMK + 3pMNK + MN \end{aligned} \tag{18}$$

The term $3MK'$ comes from reading selected entries of $\mathbf{A}$, writing $\mathbf{A}_{\text{sub}}$, and reading $\mathbf{A}_{\text{sub}}$ again during the token-wise dot product. The dominant term $3MNK'$ comes from reading selected entries of $\mathbf{B}$, writing the token-specific tensor $\mathbf{B}_{\text{sub}}$, and reading $\mathbf{B}_{\text{sub}}$ again during the einsum. The additional materialized data-tensor storage is

$$\begin{aligned} S_K &\approx MNK' + MK' \\ &= pMNK + pMK \end{aligned} \tag{19}$$

which is dominated by the token-specific gathered weight tensor $\mathbf{B}_{\text{sub}} \in \mathbb{R}^{M\times N\times K'}$, where the remaining MK' entries belong to $\mathbf{A}_{\text{sub}}$.

These estimates highlight why the fully vectorized N-axis and K-axis implementations above are especially inefficient: they materialize token-specific subsets of $\mathbf{B}$, incurring $\mathcal{O}(pMNK)$ intermediate storage and destroying the regular weight reuse of dense GEMM, which highlights the role of customized fused kernels. The results are summarized in Table 4.

Pattern	Naive data-tensor traffic	Extra intermediate storage
Dense GEMM	$MK + NK + MN$	–
Dynamic pruning (M-axis)	$3pMK + NK + 3pMN + MN$	$pMK + pMN$
Dynamic pruning (N-axis)	$MK + 3pMNK + 3pMN + MN$	$pMNK + pMN$
Dynamic pruning (K-axis)	$3pMK + 3pMNK + MN$	$pMNK + pMK$

Table 4: Theoretical data-tensor traffic and additional materialized intermediate storage of naive gather-scatter implementations. Routing masks, indices, and offsets are excluded, and the storage calculation also excludes the final dense output $\mathbf{C}$.

Building on the theoretical analysis, we further benchmark the throughput and peak memory usage of the dense GEMM baseline, naive gather-scatter kernel, and optimized CuTe GEMM kernel for pruning along the M, N, and K axes. Since the naive implementation can be adapted to grouped-GEMM by mask-reordering preprocessing, we also introduce it as a stronger baseline. The results are shown in Figure 8. For the coarsest-grained M-axis pruning setting, the explicit gather-scatter kernel still delivers a meaningful speedup but incurs an additional memory overhead of roughly 2x. For the finer-grained GEMM-N and GEMM-K settings, however, the naive implementation is constrained by the massive additional memory traffic, achieving only about 1.0 TFLOPS and requiring roughly 1,000x the memory footprint compared to the optimized kernels, while the grouped-GEMM kernel still requires up to 34x the incremental memory of dense GEMM and only achieves 6$\sim$102 TFLOPS, failing to reach the dense GEMM baseline. In contrast, the fused CuTe kernel incurs virtually no additional memory-storage overhead and can already deliver speedups for matrices of size 2048^3.

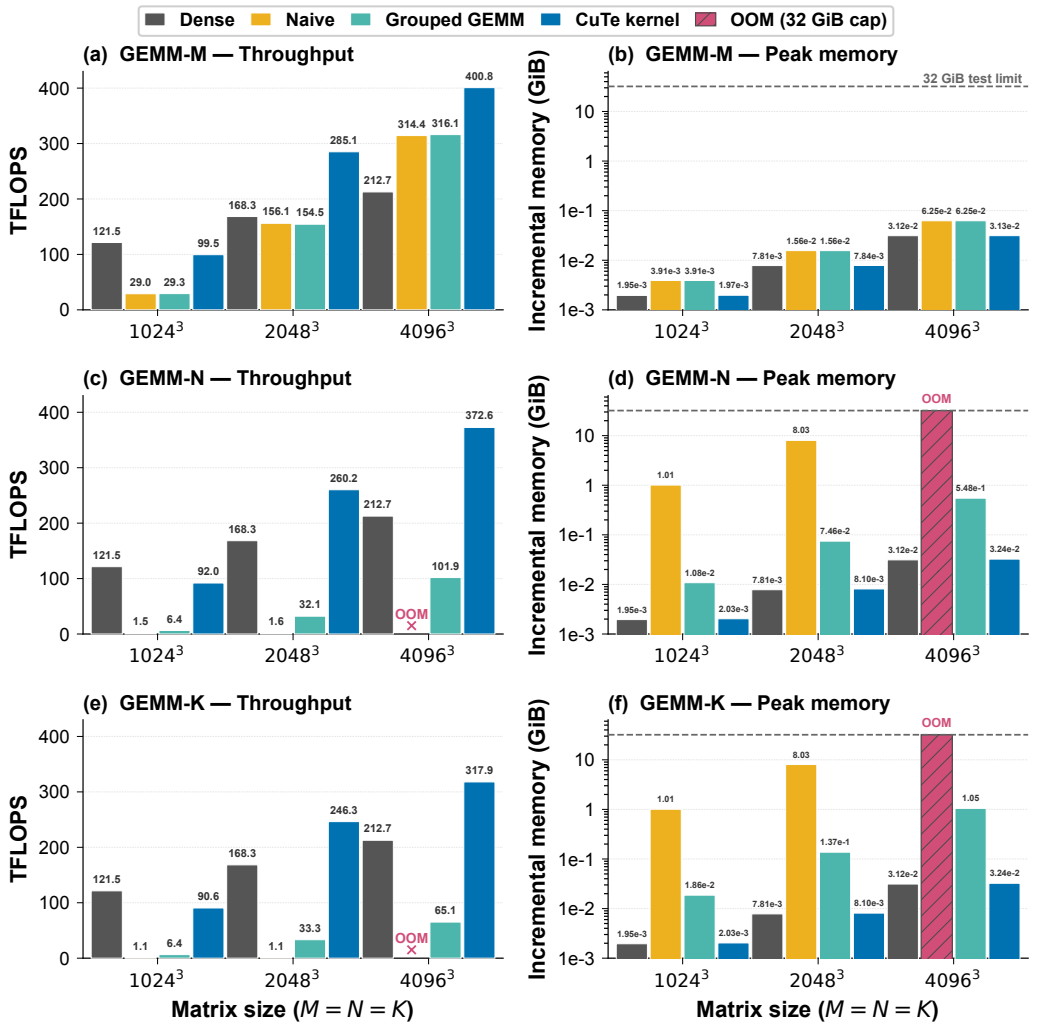


Figure 8: Throughput and memory overhead benchmarks for M, N, and K-axis GEMM pruning based on dense GEMM baseline, naive gather-scatter kernel, grouped-GEMM kernel, and fused CuTe kernel at 50% sparsity, with $G = 128$ for width pruning.

B CuTe Kernel Details

At a high level, all three CuTe kernels follow the same execution path as the framework in the main text: (1) standalone mask-reordering preprocessing, (2) hardware-agnostic block-level skipping at stage $\ell = 0$, (3) a dense-style tiled pipeline with optional hardware-dependent intra-block skipping during operand gathering ($\ell = 1$, corresponding to $\mathbf{A}$ loading in GEMM and $\mathbf{Q}$ loading in attention) and MMA execution ($\ell = 2$), and (4) a scatter epilogue. The $\ell = 1$ and $\ell = 2$ predicates are enabled only when specified by the architecture-specific kernel configuration.

Herein, we first describe the metadata shared by all three kernels. A *routing column* is $\mathcal{M}_{:,g}$ after flattening the batch and token dimensions for GEMM, and $\mathcal{M}_{b,:,g}$ for FlashAttention. As shown in Algorithm 1, each routing column is sorted independently on the GPU. The resulting index $\mathcal{I}_{j,g}$ maps logical row j to its physical row. Activation tensors remain in their original layout and are gathered/scattered directly through the reordered indices. Consecutive operators sharing the same routing decision can reuse this metadata, while zero initialization still remains necessary for skipped blocks and atomic-add epilogues.

Algorithm 1 Unified mask reordering

Require: Binary routing mask $\mathcal{M}$
Ensure: Reordered mask $\widetilde{\mathcal{M}}$ and original row indices $\mathcal{I}$
1: **for all** groups g (and batch elements b for attention) in parallel **do**
2: $\quad(\widetilde{\mathcal{M}}_{:,g}, \mathcal{I}_{:,g}) \leftarrow \mathrm{SortDesc}(\mathcal{M}_{:,g})$ ▷ batch index omitted
3: **end for**
4: **return** $(\widetilde{\mathcal{M}}, \mathcal{I})$

For a sorted CTA row tile beginning at m_0, the block-level predicate is simply $s^{(0)} = \widetilde{\mathcal{M}}_{m_0,g}$. Since the column is monotone, $s^{(0)} = 0$ can prove that the whole CTA and every later CTA for this group are inactive. Conversely, if a full tile's last entry is active, the kernel takes an all-active fast path and avoids loading per-row masks. For contiguous row blocks, the predicates simplify to

$$s^{(\ell)}_{m,g} = \widetilde{\mathcal{M}}_{mS_\ell,g}, \qquad mS_\ell < M \tag{20}$$

The $\ell = 1$ predicate is instantiated per thread or per warp according to the CuTe copy atom. The $\ell = 2$ predicate is reduced with a warp vote instruction and therefore remains uniform across all lanes issuing the same tensor-core instruction. The two finer skipping levels are independently configurable, disabling either level causes execute all its in-bounds units.

In the pseudocode below, $s^{(1)}$ and $s^{(2)}$ collect the predicates within the admitted tile, and $\mathcal{I}$ directly supplies the original row indices. We omit CuTe layout construction, register-fragment permutations, and architecture-specific pipeline stages and barriers, with four primitives shared by all pseudo kernels below: GATHER loads reordered activation rows admitted by $s^{(1)}$; LOAD stages a dense weight or KV tile; MASKEDMMA accumulates row fragments admitted by $s^{(2)}$; and SCATTER writes only rows with $\widetilde{\mathcal{M}} = 1$, with configurable behavior between direct stores (`store`) and atomic addition (`add`). Additionally, the MASKEDMMA in sm80 and sm120 also denote to skip the explicit shared-to-register transfer required by their tensor core.

Implementation of GEMM-MN GEMM-MN is the shared kernel for M-axis and N-axis pruning. Its N-axis path applies routing to the attention $\mathbf{W}_q$ and FFN $\mathbf{W}_{\text{up}}$/$\mathbf{W}_{\text{gate}}$, while M-axis dynamic depth pruning is the $N_G = 1$ special case of the same execution path. Let $\mathbf{C} = \mathbf{A}\mathbf{B}^\top$, with $\mathbf{A} \in \mathbb{R}^{M\times K}$ and $\mathbf{B} \in \mathbb{R}^{N\times K}$, and, for N-axis pruning, partition N into groups of width G, with $BN \mid G$. Each CTA owns one (BM, BN) output tile, whose N coordinate selects a routing group g, where M-axis pruning uses $g = 0$ for every output tile. Algorithm 2 makes the CTA admission, predicated computation, and scatter epilogue explicit.

Algorithm 2 GEMM-MN for output tile (m_0, n_0)

Require: $\mathbf{A}, \mathbf{B}$, zero-initialized $\mathbf{C}$, $(\widetilde{\mathcal{M}}, \mathcal{I})$, tile sizes (BM, BN, BK), assigned K range
$g \leftarrow 0$ for M-axis pruning; otherwise $g \leftarrow \lfloor n_0/G \rfloor$
if $\widetilde{\mathcal{M}}_{m_0,g} = 0$ **then**
  return ▷ $\ell = 0$: skip the output tile
end if
Read $s^{(1)}, s^{(2)}$ and $\mathcal{I}_{m_0:m_0+BM,g}$
$\mathbf{C}_{\text{acc}} \leftarrow \mathbf{0}$
for each BK-tile beginning at k in the assigned K range **do**
  $\mathbf{A}_{\text{tile}} \leftarrow \textsc{Gather}(\mathbf{A}[\mathcal{I}_{m_0:m_0+BM,g}, k:k+BK], s^{(1)})$
  $\mathbf{B}_{\text{tile}} \leftarrow \textsc{Load}(\mathbf{B}[n_0:n_0+BN, k:k+BK])$
  $\mathbf{C}_{\text{acc}} \leftarrow \textsc{MaskedMMA}(\mathbf{A}_{\text{tile}}, \mathbf{B}_{\text{tile}}^{\top}, \mathbf{C}_{\text{acc}}, s^{(2)})$
end for
`mode` $\leftarrow$ `add` for split-K; otherwise `mode` $\leftarrow$ `store`
$\textsc{Scatter}(\mathbf{C}_{\text{acc}}, \mathbf{C}[\mathcal{I}_{m_0:m_0+BM,g}, n_0:n_0+BN], \texttt{mode})$

It is clear that the standard dense GEMM pipeline still remaining, since each logical K tile still commits exactly one asynchronous-copy group, the sparse predicates suppress only $\mathbf{A}$ load packets, with dense $\mathbf{B}$ copy keeps the producer stage well-defined. Without split-K, the assigned range is $[0, K)$ and each output entry has one writer. Split-K variants partition this range across CTAs and atomically reduce the scattered partial outputs into the zero-initialized buffer. The GEMV variant can be dispatched when $M \leqslant 4$.

Implementation of GEMM-K GEMM-K implements the K-axis path for the attention $\mathbf{W}_{\text{o}}$ and FFN $\mathbf{W}_{\text{down}}$. Here the routed dimension is the reduction dimension, with $N_G = K/G$:

$$\mathbf{C}_{m,n} = \sum_{g=0}^{N_G-1} \mathcal{M}_{m,g} \sum_{k=gG}^{(g+1)G-1} \mathbf{A}_{m,k}\mathbf{B}_{n,k} \tag{21}$$

Since different groups have different row permutations, each active group computes a partial output and scatters it with an atomic reduction, as shown in Algorithm 3. For each (m_0, n_0), reduction groups are partitioned among CTAs, with $BK \mid G$.

Algorithm 3 GEMM-K for row/output tile (m_0, n_0) and assigned reduction groups

Require: $\mathbf{A}, \mathbf{B}$, zero-initialized $\mathbf{C}$, $(\widetilde{\mathcal{M}}, \mathcal{I})$, tile sizes (BM, BN, BK)
for each reduction group g assigned to this CTA **do**
  if $\widetilde{\mathcal{M}}_{m_0,g} = 1$ **then** ▷ $\ell = 0$: admit this group
    Read $s^{(1)}, s^{(2)}$ and $\mathcal{I}_{m_0:m_0+BM,g}$
    $\mathbf{C}_{\text{acc}} \leftarrow \mathbf{0}$
    for each BK-tile beginning at k in $[gG, (g+1)G)$ **do**
      $\mathbf{A}_{\text{tile}} \leftarrow \textsc{Gather}(\mathbf{A}[\mathcal{I}_{m_0:m_0+BM,g}, k:k+BK], s^{(1)})$
      $\mathbf{B}_{\text{tile}} \leftarrow \textsc{Load}(\mathbf{B}[n_0:n_0+BN, k:k+BK])$
      $\mathbf{C}_{\text{acc}} \leftarrow \textsc{MaskedMMA}(\mathbf{A}_{\text{tile}}, \mathbf{B}_{\text{tile}}^{\top}, \mathbf{C}_{\text{acc}}, s^{(2)})$
    end for
    $\textsc{Scatter}(\mathbf{C}_{\text{acc}}, \mathbf{C}[\mathcal{I}_{m_0:m_0+BM,g}, n_0:n_0+BN], \texttt{add})$
  end if
end for

Unlike GEMM-MN, GEMM-K scopes its pipeline to one active K-group. The producer–consumer ring and accumulator reset at each group boundary, while an inactive group skips its copies, synchronizations, MMA, and epilogue work entirely. This group-local design sacrifices prefetching across adjacent K-groups, but allows $\ell = 0$ block-level skipping to bypass the entire inner pipeline. Advancing a shared ring over skipped groups would otherwise leave unmatched wait/commit operations and increase barrier-management overhead. It also creates substantial atomic-reduction

contention because around $\lceil K/G \rceil$ groups may scatter partial outputs to the same entries of $\mathbf{C}$. We mitigate this contention with an outer loop that packs multiple K-groups into one CTA and executes them sequentially, thus reducing concurrent writebacks without changing the number of group contributions. For small M, the GEMV variant is dispatched according to both M and G.

On sm90+, TMA also supports asynchronous tensor reduction from shared to global memory through `cp.reduce.async.bulk.tensor`. This enables atomic writeback from shared memory, allowing a warp-specialized pipeline to overlap the atomic epilogue with the mainloop and potentially improve GEMM-K performance for small G. This optimization is less suitable for sm120: persistently storing $\mathbf{C}$ in shared memory consumes too much of the limited shared-memory capacity and substantially reduces occupancy. We therefore leave it as future work.

Implementation of Attention As for prefill attention, a CTA owns a query tile at batch b and head h_q. The query head determines both its KV head for GQA and routing group g. Algorithm 4 follows the tiled online-softmax organization of FlashAttention (Dao, 2024), while fusing the same three predicates into the query-side work. Its path is query-tile admission, $\mathbf{Q}$ prologue, dense $\mathbf{K}/\mathbf{V}$ pipelined mainloop, and scatter epilogue. Here OnlineSoftmax updates the running maximum $\mathbf{m}$ and normalization sum $\mathbf{z}$, rescales $\mathbf{O}_{\text{acc}}$, and returns unnormalized weights $\mathbf{P}$ for active rows. A row with no valid keys in a tile uses $\mathbf{P} = \mathbf{0}$ and leaves its running state unchanged.

Using the head size d and group size G from the main text, the KV head and routing group are

$$h_k = \left\lfloor \frac{h_q}{H_q/H_k} \right\rfloor, \qquad g = \left\lfloor \frac{h_q}{G/d} \right\rfloor \tag{22}$$

For the broader causal attention scenario, since tokens in a gathered query tile have different real position, the causal masking admits key position j only if $j \leqslant \mathcal{I}_{b,m,g}$. Therefore, Algorithm 4 uses a CTA-wide ReduceMax over active query positions to allocate the maximum position in this query tile.

Algorithm 4 Routed causal prefill attention for query tile (b, h_q, m_0)

Require: $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, zero-initialized $\mathbf{O}$, $(\widetilde{\mathcal{M}}, \mathcal{I})$, tile sizes (BM, BN)
1: Map h_q to KV head h_k and routing group g
2: **if** $\widetilde{\mathcal{M}}_{b,m_0,g} = 0$ **then**
3: **return** ▷ $\ell = 0$: skip the query tile
4: **end if**
5: Read $s^{(1)}, s^{(2)}$ and $\mathcal{I}_{b,m_0:m_0+BM,g}$
6: $j_{\max} \leftarrow \min(T_k, 1 + \text{ReduceMax}(\mathcal{I}_{b,m_0:m_0+BM,g}))$
7: $\mathbf{Q}_{\text{tile}} \leftarrow \text{Gather}(\mathbf{Q}[b, \mathcal{I}_{b,m_0:m_0+BM,g}, h_q, :], s^{(1)})$
8: $(\mathbf{m}, \mathbf{z}, \mathbf{O}_{\text{acc}}) \leftarrow (-\infty, \mathbf{0}, \mathbf{0})$
9: **for** each aligned key tile $[j, j + BN)$ intersecting $[0, j_{\max})$ **do**
10: $\mathbf{K}_{\text{tile}} \leftarrow \text{Load}(\mathbf{K}[b, j : j + BN, h_k, :])$
11: $\mathbf{V}_{\text{tile}} \leftarrow \text{Load}(\mathbf{V}[b, j : j + BN, h_k, :])$
12: $\mathbf{S} \leftarrow \text{MaskedMMA}(\mathbf{Q}_{\text{tile}}, \mathbf{K}_{\text{tile}}^{\top}, \mathbf{0}, s^{(2)})/\sqrt{d}$
13: Apply causal masking using $\mathcal{I}$; set inactive rows and invalid keys of $\mathbf{S}$ to $-\infty$
14: $(\mathbf{m}, \mathbf{z}, \mathbf{O}_{\text{acc}}, \mathbf{P}) \leftarrow \text{OnlineSoftmax}(\mathbf{S}, \mathbf{m}, \mathbf{z}, \mathbf{O}_{\text{acc}})$
15: $\mathbf{O}_{\text{acc}} \leftarrow \text{MaskedMMA}(\mathbf{P}, \mathbf{V}_{\text{tile}}, \mathbf{O}_{\text{acc}}, s^{(2)})$
16: **end for**
17: Normalize active rows: $\mathbf{O}_{\text{acc}} \leftarrow \mathbf{O}_{\text{acc}}/\mathbf{z}$
18: $\text{Scatter}(\mathbf{O}_{\text{acc}}, \mathbf{O}[b, \mathcal{I}_{b,m_0:m_0+BM,g}, h_q, :], \texttt{store})$

Once a CTA is admitted, K and V remain dense and their two producer–consumer pipelines advance once per logical key tile. Row-level predicates never suppress a barrier arrival or stage transition. They only suppress the gathered Q packets and the QK/PV MMA fragments, preserving barrier phase agreement across the CTA. For decoding ($T_q = 1$) phase, routing can be checked directly without mask reordering, while the split online-softmax pipeline follows the same abstraction.

C Implementation Details for Training and Evaluation

Since different model families have their own training data distributions, we use separate calibration datasets for the Llama3 and Qwen3 families[1][2]. Specifically, we apply a uniform sampling strategy to each full dataset for constructing a calibration set of around 200,000 untruncated samples with tokenized lengths less than the maximum T_q. For Infinity-Instruct, we also apply an assistant-only mask during end-to-end training, including S1 in DDP, SkipGPT, and WIDE, as well as LoRA fine-tuning for all methods in S2. The learning rate for WIDE's S1 is selected by grid search over {2e-3, 1e-3, 5e-4, 2e-4}, where SkipGPT fails to converge on Qwen3-14B at its default learning rate of 2e-3, so 5e-4 is also applied to SkipGPT in Qwen3-14B training. Both training stages run on 4 NVIDIA A100-SXM4-40G GPUs with PyTorch FSDP2. We further apply gradient checkpointing separately to the attention and FFN modules to reduce training memory usage without replaying the router. During training, to ensure fair comparison, all methods use the same subset for calibration and LoRA tuning. The calibration stage follows each method's original training budget, while the LoRA stage uses the same settings as WIDE. For SkipGPT, we additionally increase the router rank r to match the router capacity of WIDE. More detailed settings are listed in Table 5.

During evaluation, we test all tasks using lm-evaluation-harness (Gao et al., 2024), with a maximum sequence length of 4,096 and a generation limit of 1,024 tokens, except for IFEval, which retains its default limit of 1,280 tokens. The inference engine is set to native Transformers, enabling a static cache and `torch.compile` and disabling sampling throughout. For the attention backend, we use torch SDPA to automatically select among the FlashAttention, memory efficient, and math backends, as in the training stages. All tasks except MMLU are evaluated in the zero-shot setting. In addition, for Qwen3-14B, we find that all pruned models achieve better quality in likelihood tasks when using the chat template, while the dense model does not. We therefore apply the chat template to all methods except the dense baseline for Qwen3-14B.

For the experimental environments, all experiments are conducted with CUDA 13.0, PyTorch 2.11, Transformers 5.12.1, and lm-eval 0.4.12. We use Lightning Fabric 2.6.5 for training and SGLang 0.5.16, Triton 3.7.1, and TileLang 0.1.11 for inference kernel implementation and validation. The CuTe kernel libraries are wrapped with the TVM-FFI Just-In-Time (JIT) interface used by SGLang (Zheng et al., 2024), combined with an autotuning system to easily switch to the optimal tiling and intra-kernel skipping configuration. Among these DSL implementations, CuTe provides the most fine-grained control and enables intra-block skipping during both $\mathbf{A}$ loading and MMA execution, while the tile-level DSL implementations only support CTA-level skipping due to their coarser control granularity.

During throughput benchmarks, we use random masks with top-k selection to maintain the target sparsity for each kernel, pruned module, and method. Kernel-level benchmarks report the latency of the complete function replay, including device-side kernel execution, host-side operations before and after kernel launches, and other associated latency. The Triton benchmark interface is introduced as the kernel profiling tool, with a repetition window of 200ms for repeated measurements. For end-to-end evaluation, we use a fixed-length static cache to avoid runtime KV-cache allocation, and further exclude the final sampling and detokenization latency. Before measurement, we perform one warmup iteration to complete kernel tuning, then capture a CUDA Graph and replay it 20 times to report the average latency from CUDA event.

[1] https://huggingface.co/datasets/ZengXiangyu/RedPajama-Data-1T-Sample

[2] https://huggingface.co/datasets/BAAI/Infinity-Instruct

Table 5: Detailed hyperparameter settings for WIDE's training stages.

Params.	Llama3.1-8B	Llama3.2-3B	Qwen3-14B
Calibration Data	RedPajama-1T		Infinity-Instruct (7M-Core)
Maximum LR (S1)	2e-3		5e-4
H_q/H_k	32/8	24/8	40/8
G_{attn}/d (heads)	4	3	5
G_{ffn}	{16, 32, 64, 128, 256, 512}		
Router r	32		
B	16		
Maximum T_q	4,096		
Initial τ	5		
Final τ	0.5		
Warmup Steps (S1)	1,000		
Training Steps (S1)	10,000		
Warmup Steps (S2)	1,000		
Training Steps (S2)	10,000		
Maximum LR (S2)	2e-4		
LoRA α	32		
LoRA r	16		
LoRA dropout	0.1		
Optimizer	AdamW (weight decay = 0.01, [β_1, β_2] = [0.9, 0.999])		

D Additional Experimental Results

Benchmark over Uniform Sparsity We further evaluate WIDE with a uniform sparsity constraint that allocates the pruning budget equally between the attention and FFN modules. Specifically, we decompose the sparsity loss into separate attention and FFN terms:

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \frac{\alpha}{2}\left(\left|SP - \frac{1}{L}\sum_{i=1}^{L}\left(1 - \frac{\sum \mathcal{M}^i_{\text{attn}}}{|\mathcal{M}^i_{\text{attn}}|}\right)\right| + \left|SP - \frac{1}{L}\sum_{i=1}^{L}\left(1 - \frac{\sum \mathcal{M}^i_{\text{ffn}}}{|\mathcal{M}^i_{\text{ffn}}|}\right)\right|\right) \tag{23}$$

This design constrains the sparsity of the two modules independently rather than matching only their average, keeping their module-wise sparsity levels close to a 1:1 ratio. As shown in Table 6, the uniform constraint causes a larger accuracy drop at larger group sizes, but LoRA recovery restores most of the lost accuracy. After recovery, the uniform variant is within roughly 4% of the unconstrained WIDE variant and still significantly outperforms static structured pruning at the same budget. The layer-wise sparsity profiles in Fig. 9 support the same interpretation: enforcing uniformity between attention and FFN does not alter how sparsity is distributed across layers, it only shifts the profile without changing its shape. Thus, the allocation learned by WIDE appears to capture a stable layer-importance pattern induced by the router.

Benchmark over Broader Baselines Beyond the static-depth, static-width, and dynamic-depth baselines considered in the main experiments, we broaden our comparison to three additional pruning paradigms: Dense-to-MoE conversion with similar granularity in FFN compared to WIDE, semi-structured pruning, and activation-based contextual structured / unstructured pruning at the finest granularity. Specifically, we evaluate CMoE (Pei et al., 2026), HyperPrune (Sun & Sakuma, 2026), PolarSparsity (Shrestha et al., 2025), and WINA (Chen et al., 2026a) on Llama3.1-8B under a 50% sparsity budget. We follow the training budget specified in each original paper for calibration, followed by a unified 10k-step LoRA schedule for stage 2. Specifically, PolarSparsity only prunes the attention module due to the non-ReLU activation in Llama3.1-8B, CMoE only prunes the FFN, leaving the full attention module dense, whereas HyperPrune and WINA prune only the linear projections in each layer, while keeping the attention computation itself dense. As shown in Table 7,

Table 6: Downstream performance and the observed average sparsity on Llama3.1-8B with DDP, SkipGPT, and WIDE at 50% target sparsity, where *uniform* denotes imposing a 50% sparsity target separately on attention and FFN. Sparsity pairs report the sparsity in attention/FFN module.

Methods	Real Sparsity	WikiText2	ARC-E	ARC-C	BoolQ	Winogrande	PIQA	OpenbookQA	Hellaswag	Avg. Acc.
DDP	41.6%/47.9%	21.83	55.13	33.36	65.69	60.77	69.31	30.80	56.28	53.05 (74.14%)
– *LoRA*	41.6%/47.9%	19.52	55.77	34.56	68.26	61.80	71.00	33.60	57.46	54.64 (76.36%)
SkipGPT	56.7%/38.7%	96.04	35.69	24.57	57.65	52.49	58.71	26.60	41.92	42.52 (59.42%)
– *uniform*	49.1%/47.9%	104.91	33.16	24.91	48.72	50.59	60.23	27.60	31.90	39.58 (55.32%)
– *LoRA*	62.7%/31.6%	13.90	72.01	42.24	65.81	64.80	75.68	42.80	67.87	61.60 (86.10%)
– *uniform + LoRA*	47.6%/46.5%	16.17	66.20	37.54	64.40	63.85	74.32	38.00	64.45	58.39 (81.61%)
WIDE(32)	63.1%/30.7%	14.15	69.74	41.55	70.31	63.69	76.22	42.20	69.17	61.84 (86.43%)
– *uniform*	47.8%/47.1%	14.97	61.87	37.29	63.82	64.96	73.56	39.60	64.76	57.98 (81.03%)
– *LoRA*	66.1%/27.9%	11.99	72.90	44.20	77.22	65.98	78.40	43.40	71.66	64.82 (90.60%)
– *uniform + LoRA*	47.6%/46.3%	13.62	68.39	40.78	73.43	66.85	75.90	42.40	68.83	62.36 (87.16%)
WIDE(128)	63.0%/30.0%	12.57	66.37	39.33	69.51	67.32	76.55	42.20	69.14	61.49 (85.94%)
– *uniform*	48.8%/48.5%	21.58	51.64	30.89	57.13	55.01	71.55	32.60	58.17	50.99 (71.27%)
– *LoRA*	66.2%/28.5%	11.43	71.04	41.64	75.38	67.72	78.94	43.80	71.85	64.34 (89.92%)
– *uniform + LoRA*	48.5%/46.8%	13.03	68.73	39.08	73.06	66.61	76.22	40.40	67.30	61.62 (86.13%)

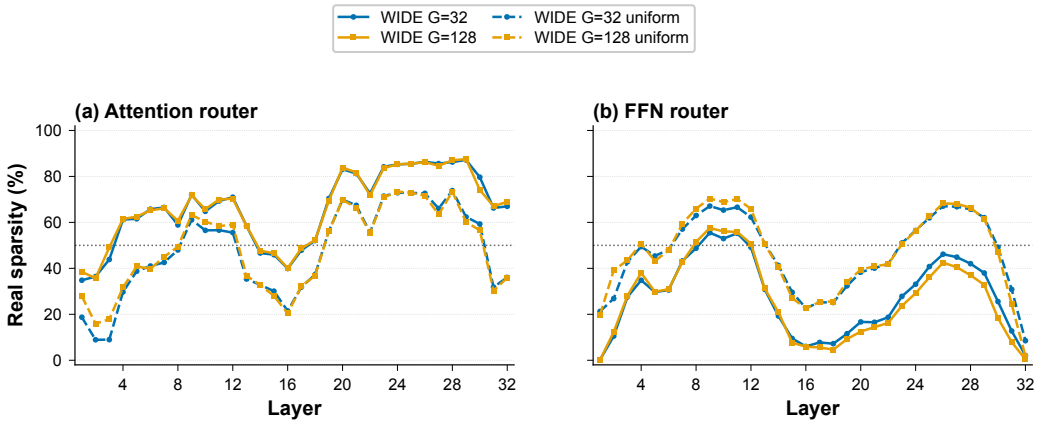


Figure 9: WIDE's layer-wise sparsity with different G and sparsity allocation.

Table 7: Additional baseline performance under Llama3.1-8B with target sparsity set to 50%. "S2A2E8" for CMoE denotes splitting the dense FFN into 8 experts, with 2 shared experts and top-2 expert routing. **A / F** denotes pruning only the attention / FFN module.

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.↑
Calibration-only (S1)									
CMoE (S2A2E8)^{F}	40.35	45.03	27.90	58.62	55.80	60.39	30.20	41.43	45.62 (63.76%)
CMoE (S4A4E16)^{F}	40.72	44.99	28.41	63.06	55.41	62.30	30.20	41.45	46.54 (65.05%)
HyperPrune	20.30	58.67	35.40	69.57	66.69	70.72	33.80	56.69	55.93 (78.17%)
PolarSparsity^{A}	14.03	72.52	42.41	61.04	63.85	77.58	39.80	66.64	60.54 (84.62%)
WINA	9.12	77.69	51.11	**81.74**	**70.96**	78.35	**43.20**	76.63	68.52 (95.77%)
WIDE (32)	14.15	69.74	41.55	70.31	63.69	76.22	42.20	69.17	61.84 (86.43%)
WIDE (128)	12.57	66.37	39.33	69.51	67.32	76.55	42.20	69.14	61.49 (85.94%)
WIDE (128)^{F}	15.32	63.30	38.99	66.24	64.40	73.56	39.60	64.48	58.65 (81.97%)
WIDE (128)^{A}	**8.68**	**78.57**	**53.32**	80.97	70.87	**80.08**	43.00	**78.57**	**69.34 (96.91%)**
LoRA Tuning (S2)									
CMoE (S4A4E16)^{F}	17.26	57.95	31.48	70.21	59.12	67.14	31.60	56.44	53.42 (74.66%)
WINA	9.09	77.82	51.11	79.72	70.96	79.71	43.40	77.09	68.54 (95.79%)
PolarSparsity^{A}	8.94	78.32	50.68	77.77	67.96	**79.98**	43.80	76.72	67.89 (94.88%)
WIDE (32)	11.99	72.90	44.20	77.22	65.98	78.40	43.40	71.66	64.82 (90.60%)
WIDE (128)	11.43	71.04	41.64	75.38	67.72	78.94	43.80	71.85	64.34 (89.92%)
WIDE (128)^{F}	12.15	71.63	41.64	78.35	64.80	76.17	41.20	70.09	63.41 (88.62%)
WIDE (128)^{A}	**8.62**	**79.46**	**52.81**	**81.37**	**71.58**	79.97	**44.80**	**78.67**	**69.80 (97.56%)**

WIDE clearly outperforms CMoE and the semi-structured HyperPrune while jointly pruning attention and the FFN, but still trails WINA with its finer pruning granularity.

Decoding Speedup for Different Attention Group Sizes Figure 10 shows how the attention group size affects decoding. For $G_{\text{attn}} \leqslant \frac{H_q}{H_k} \times d$, the kernel progressively transitions from vanilla multi-head attention to GQA-packing attention, where a single CTA partition can read and compute at most $\frac{H_q}{H_k}$ heads in parallel. Increasing G_{attn} beyond this threshold does not affect the tiling strategy used for attention decoding.

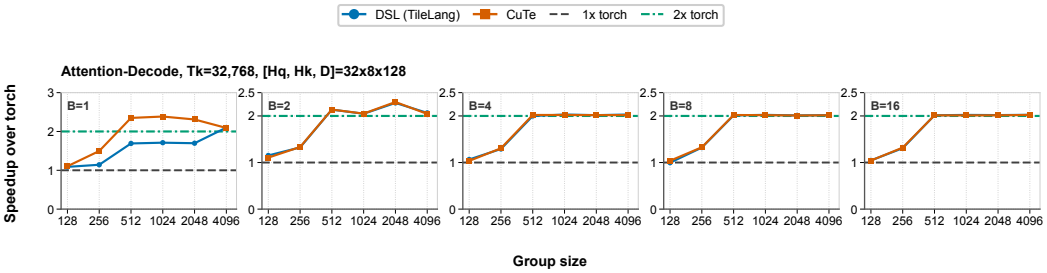


Figure 10: Attention decoding speedup for different group sizes on Llama3.1-8B shapes, with a random 50% sparsity mask.

Analysis of Layer-wise Error Accumulation Beyond downstream task performance, we also examine how errors accumulate across layers in SkipGPT and WIDE at 50% sparsity compared to the dense baseline. Figures 11 and 12 report errors and similarity in the attention and FFN outputs, together with the accumulated error across each layer, with relative ℓ_2 error defined as $\frac{\|\mathbf{X}_{\text{dense}} - \mathbf{X}_{\text{pruned}}\|_{\text{F}}}{\|\mathbf{X}_{\text{dense}}\|_{\text{F}}}$, where $\mathbf{X}$ denotes a layer's output tensor, and the subscripts denote the dense and pruned models. All evaluation uses 32 randomly sampled reference answers from GSM8K. It is evident that WIDE consistently yields lower layer-wise accumulated errors than the coarser-grained SkipGPT across the two training stages.

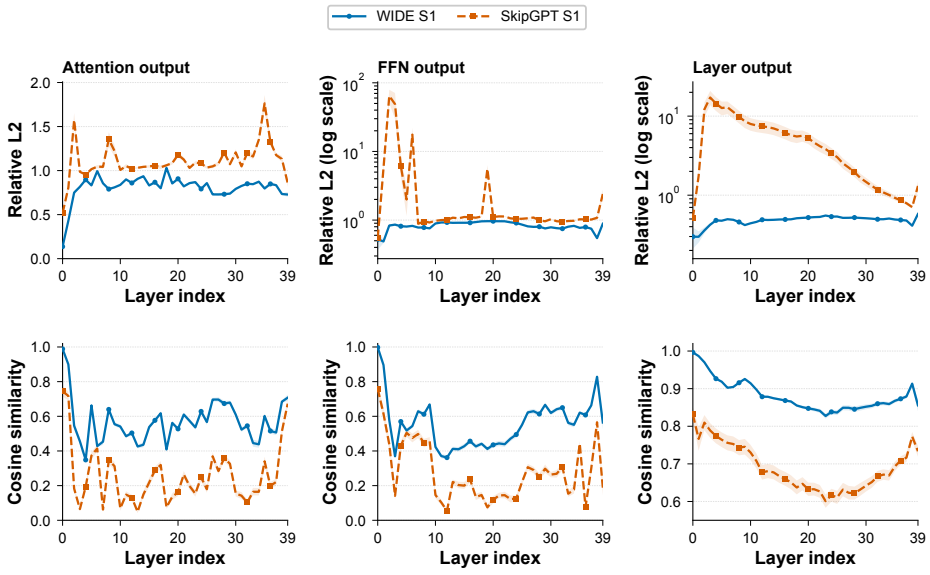


Figure 11: Layer-wise error accumulation in SkipGPT and WIDE on Qwen3-14B after router training, measured by relative ℓ_2 error and cosine similarity against the dense model.

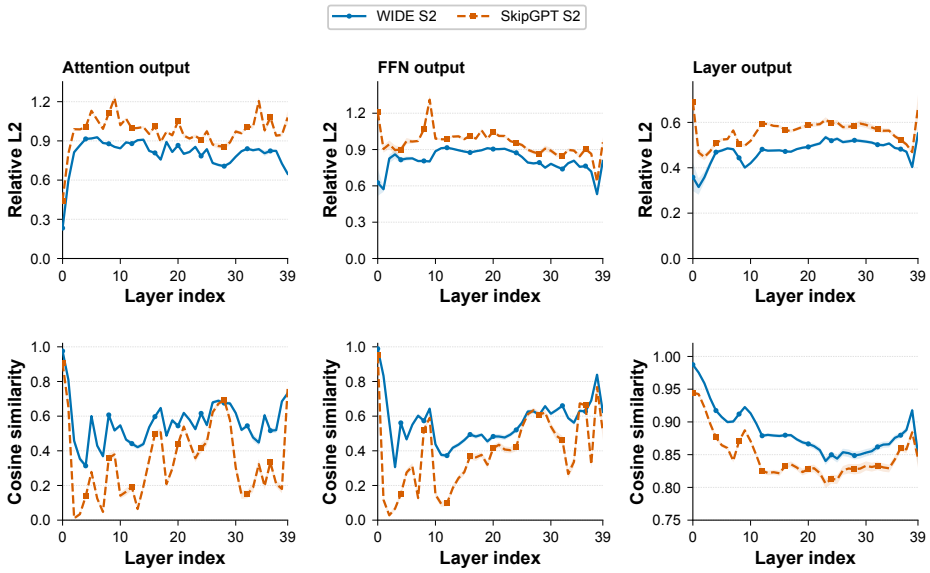


Figure 12: Layer-wise error accumulation in SkipGPT and WIDE on Qwen3-14B after LoRA tuning.

Routing Behavior and Visualizations To understand how WIDE allocates its compute budget across models and sparsity levels, we inspect the learned routing patterns at both the layer and token levels. All analyses use $G_{\text{ffn}} = 128$, while the attention-side group size is aligned with the KV group of each model. For the layer-wise allocation, we compare WIDE with the layer-wise strategy learned by DDP, using LoRA-tuned checkpoints for both methods. As shown in Fig. 13, the learned allocation is highly non-uniform across modules and layers. Attention carries most of the skipping, reaching 66.2% sparsity on WikiText2, while the FFN remains more conservative at 28.5%. Across layers, both curves peak around layers 4–12 and 24–28, and dip near the model boundaries and layers 15–18. These profiles echo the layer sensitivity observed in static pruning studies (He et al., 2024; Huang et al., 2026), but are produced here by token-conditioned group routing. The GSM8K analysis of Qwen3-14B further shows that WIDE adopts model-dependent allocation patterns. As shown in Fig. 14, Qwen3-14B instead applies higher FFN sparsity in layers 1–6, with the allocation also shifting between prefill and decoding.

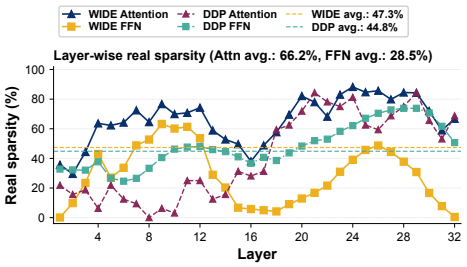


Figure 13: WIDE and DDP's layer-wise sparsity distribution of Llama3.1-8B on WikiText2, with target sparsity set to 50%.

For the token-wise routing behavior, Figures 15 to 20 show several representative case studies on evaluation tasks. Here, the FFN routers keep semantic content tokens such as `boy` and `track`, especially in layers 9–11 where the average FFN sparsity is high, while function words like `that` and `the` are often skipped. This analysis highlights the ability of WIDE to dynamically allocate its budget across layers and tokens based on semantic content.

As for the layer-wise group routing preferences, Figure 21 further compares the probability of skipping each routing group across low and high sparsity layers in the attention and FFN modules, where the patterns differ across layers.

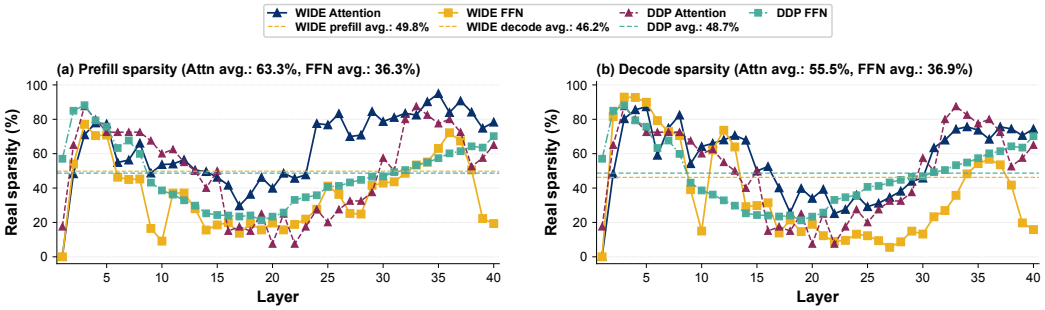


Figure 14: WIDE and DDP's layer-wise sparsity distribution of Qwen3-14B on GSM8K's prefill and decoding stages, with target sparsity set to 50%.

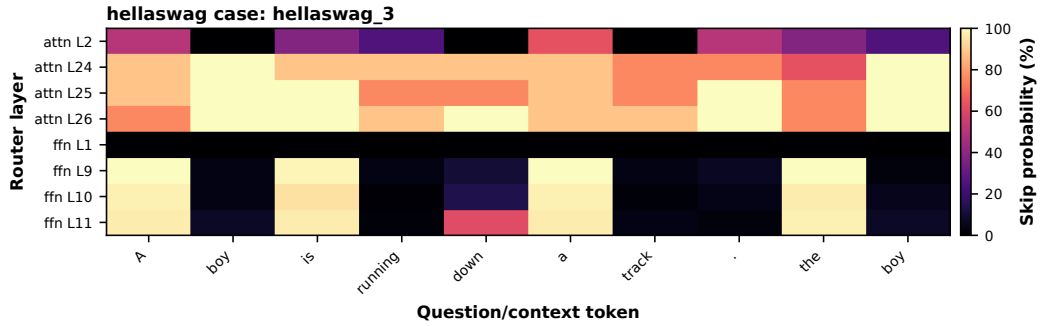


Figure 15: Token-wise routing probability for Llama3.1-8B in a selected HellaSwag sample.

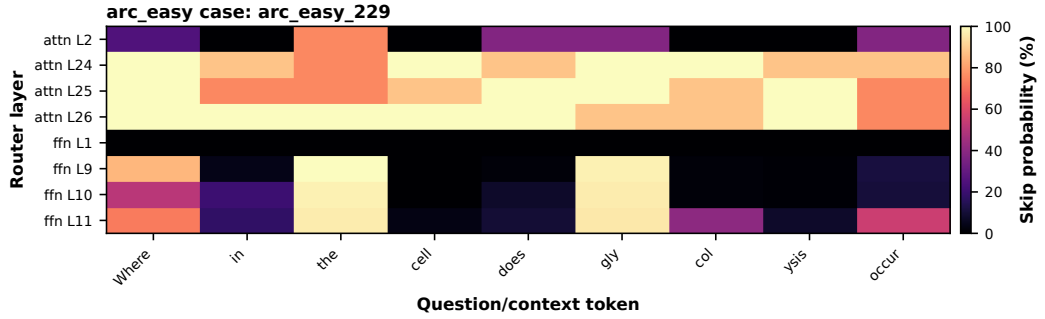


Figure 16: Token-wise routing probability for Llama3.1-8B in a selected ARC-Easy sample.

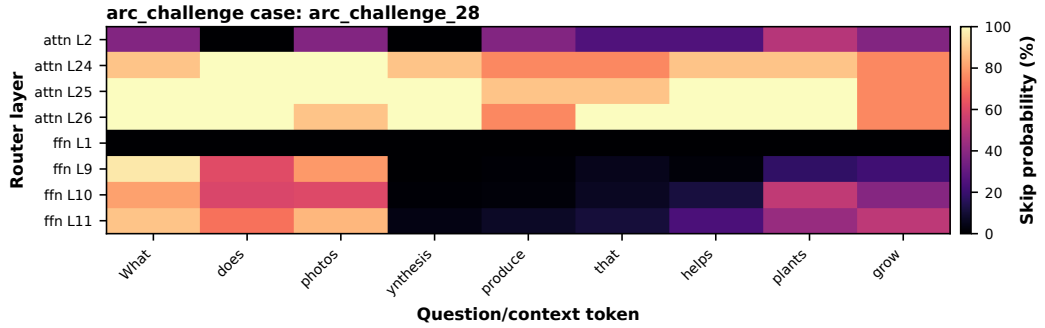


Figure 17: Token-wise routing probability for Llama3.1-8B in a selected ARC-Challenge sample.

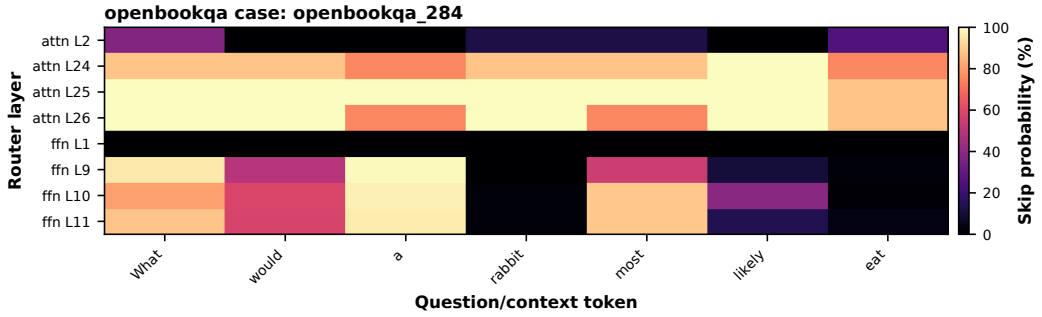


Figure 18: Token-wise routing probability for Llama3.1-8B in a selected OpenBookQA sample.

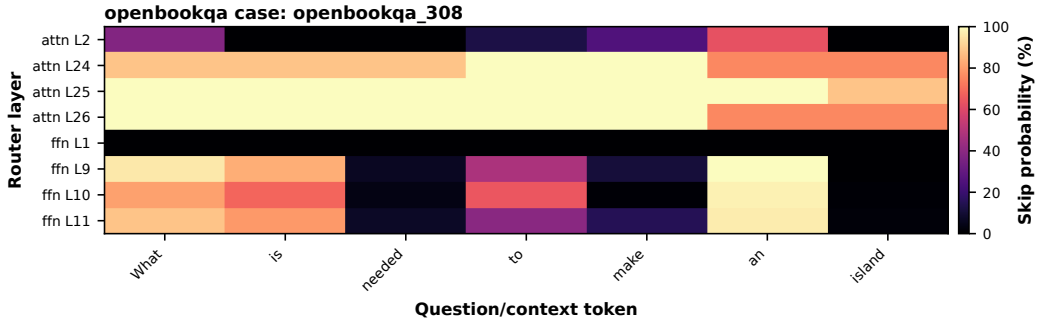


Figure 19: Token-wise routing probability for Llama3.1-8B in a selected OpenBookQA sample.

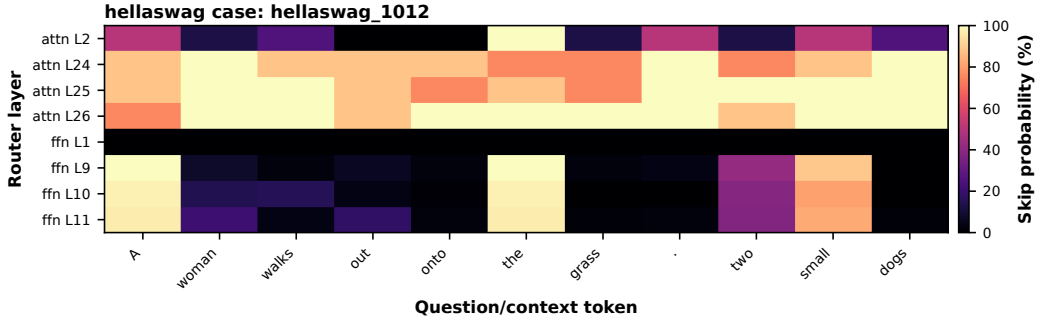


Figure 20: Token-wise routing probability for Llama3.1-8B in a selected HellaSwag sample.

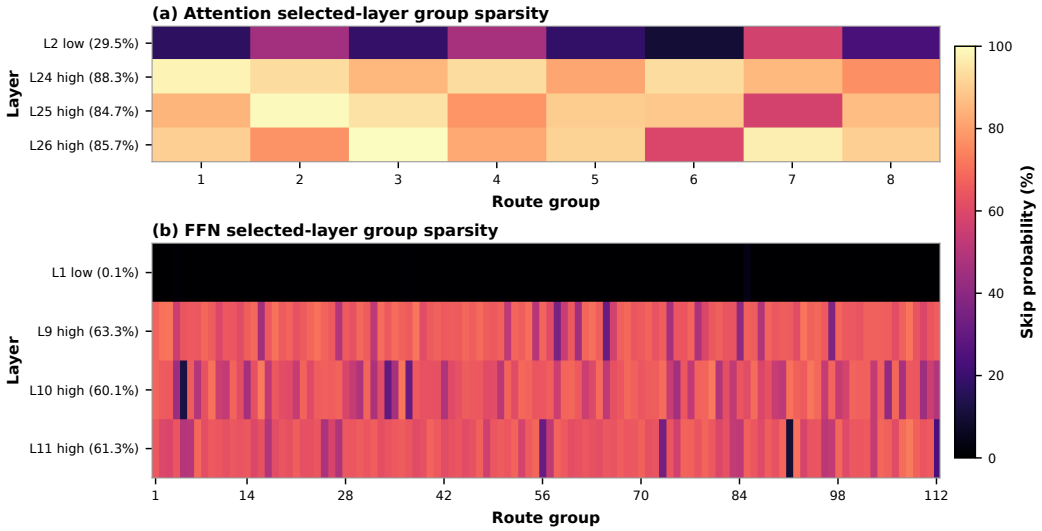


Figure 21: Group-wise skipping probabilities in different layers of Llama3.1-8B on WikiText2.

E Full Downstream Benchmark Results

Table 8, 9, 10, 11, and 12 report the full results for Llama3.1-8B, Llama3.2-3B, and Qwen3-14B at the task level.

Table 8: Llama3.1-8B performance with target sparsity 25%.

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.↑
Calibration-only (S1)									
Dense	7.71	82.70	55.03	83.06	74.27	81.07	45.40	79.32	71.55 (100.00%)
Shortened-ppl	25.84	52.19	29.95	41.41	54.62	69.80	33.00	52.55	47.65 (66.59%)
Shortened-taylor	30.07	54.42	32.42	40.28	55.49	70.62	35.20	56.68	49.30 (68.90%)
CoopPruner	22.96	59.81	33.79	61.56	53.99	72.52	33.60	58.06	53.33 (74.54%)
SliceGPT	22.57	46.42	29.52	74.92	67.48	66.53	31.80	56.93	53.37 (74.59%)
Týr-the-Pruner	12.89	68.22	38.73	78.59	66.53	74.21	37.40	66.33	61.43 (85.86%)
DDP	12.41	70.01	43.26	77.25	69.85	75.63	42.00	72.42	64.35 (89.93%)
D-LLM	69.45	29.63	22.53	58.78	49.96	54.19	25.80	31.32	38.89 (54.35%)
SkipGPT	15.63	50.38	30.89	69.94	51.30	68.77	31.40	59.06	51.68 (72.23%)
WIDE (32)	**8.49**	**79.84**	**52.82**	**82.17**	**71.67**	**79.76**	**45.60**	**78.58**	**70.06 (97.92%)**
WIDE (64)	10.37	71.34	44.97	80.46	62.15	79.11	40.40	75.78	64.89 (90.69%)
WIDE (128)	10.04	68.77	44.97	77.95	60.77	79.16	38.20	75.73	63.65 (88.96%)
LoRA Tuning (S2)									
Dense	7.91	81.52	54.01	83.30	73.32	81.23	46.20	79.94	71.36 (99.73%)
Shortened-ppl	12.33	67.63	39.08	54.25	58.48	76.61	39.00	65.69	57.25 (80.01%)
Shortened-taylor	12.12	70.58	40.78	64.04	59.83	76.88	38.20	68.53	59.83 (83.63%)
CoopPruner	12.03	73.53	42.41	63.52	62.51	77.04	40.00	69.76	61.25 (85.61%)
SliceGPT	14.88	62.92	36.86	76.17	68.82	73.77	36.80	69.71	60.72 (84.87%)
Týr-the-Pruner	11.49	71.25	43.17	78.50	68.74	76.27	39.80	71.68	64.20 (89.73%)
DDP	12.10	70.16	43.69	78.78	68.35	76.01	40.60	72.12	64.24 (89.79%)
D-LLM	10.29	77.36	48.38	76.39	68.27	78.29	43.80	75.38	66.84 (93.42%)
SkipGPT	9.87	**80.68**	51.96	81.47	70.88	79.65	44.00	77.73	69.48 (97.11%)
WIDE (32)	**8.61**	80.30	**53.50**	**82.66**	70.88	80.30	45.20	**78.46**	**70.19 (98.09%)**
WIDE (64)	10.32	79.50	52.56	81.47	**71.19**	**80.52**	45.20	78.34	69.83 (97.59%)
WIDE (128)	12.51	79.29	51.45	80.58	70.88	80.36	**45.80**	78.35	69.53 (97.18%)

Table 9: Llama3.1-8B performance with target sparsity 50%.

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.↑
Calibration-only (S1)									
Dense	7.71	82.70	55.03	83.06	74.27	81.07	45.40	79.32	71.55 (100.00%)
Shortened-ppl	473.15	32.20	24.40	53.00	49.49	57.40	25.80	35.03	39.62 (55.37%)
Shortened-taylor	4.86e+7	29.17	25.94	38.20	49.88	53.70	27.40	27.63	35.99 (50.30%)
CoopPruner	503.86	32.87	23.21	56.09	49.33	56.80	26.20	30.45	39.28 (54.90%)
SliceGPT	85.81	30.30	21.84	55.47	51.53	53.97	25.80	32.05	38.71 (54.10%)
Týr-the-Pruner	59.88	44.65	24.91	58.44	54.61	64.68	29.40	41.62	45.47 (63.55%)
DDP	21.83	55.13	33.36	65.69	60.77	69.31	30.80	56.28	53.05 (74.14%)
D-LLM	504.48	27.02	23.12	50.52	47.20	52.72	25.20	28.48	36.32 (50.77%)
SkipGPT	96.04	35.69	24.57	57.65	52.49	58.71	26.60	41.92	42.52 (59.42%)
WIDE (32)	14.15	**69.74**	**41.55**	**70.31**	63.69	76.22	**42.20**	**69.17**	**61.84 (86.43%)**
WIDE (64)	14.96	69.28	38.99	67.43	66.69	75.95	41.40	68.50	61.18 (85.50%)
WIDE (128)	**12.57**	66.37	39.33	69.51	**67.32**	**76.55**	**42.20**	69.14	61.49 (85.94%)
LoRA Tuning (S2)									
Dense	7.91	81.52	54.01	83.30	73.32	81.23	46.20	79.94	71.36 (99.73%)
Shortened-ppl	22.47	51.56	27.56	50.37	53.75	68.12	32.00	47.44	47.26 (66.05%)
Shortened-taylor	22.30	52.57	29.69	61.01	54.62	69.75	33.20	48.92	49.97 (69.83%)
CoopPruner	22.66	52.82	29.18	63.15	54.30	67.95	30.20	49.80	49.63 (69.36%)
SliceGPT	28.76	39.64	24.06	64.12	58.24	60.60	28.00	47.43	46.01 (64.31%)
Týr-the-Pruner	37.99	54.75	31.74	68.56	58.72	70.56	33.80	56.09	53.46 (74.72%)
DDP	19.52	55.77	34.56	68.26	61.80	71.00	33.60	57.46	54.64 (76.36%)
D-LLM	19.31	63.05	35.75	62.45	60.14	72.36	37.40	61.31	56.07 (78.36%)
SkipGPT	13.90	72.01	42.24	65.81	64.80	75.68	42.80	67.87	61.60 (86.10%)
WIDE (32)	11.99	72.90	**44.20**	77.22	65.98	78.40	43.40	71.66	**64.82 (90.60%)**
WIDE (64)	12.05	**73.91**	41.72	**77.74**	66.30	78.07	42.80	**72.03**	64.65 (90.36%)
WIDE (128)	**11.43**	71.04	41.64	75.38	**67.72**	**78.94**	**43.80**	71.85	64.34 (89.92%)

Table 10: Llama3.2-3B performance with target sparsity 50%.

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.↑
Calibration-only (S1)									
Dense	9.77	72.01	46.67	73.88	69.22	78.02	40.60	73.98	64.91 (100.00%)
Shortened-ppl	676.05	33.88	23.55	44.19	52.75	56.58	25.20	31.44	38.23 (58.89%)
Shortened-taylor	2.32e+5	34.01	25.60	50.92	51.93	55.71	25.60	31.70	39.35 (60.63%)
CoopPruner	1.79e+5	25.55	24.57	50.67	50.04	52.23	27.00	26.25	36.62 (56.41%)
SliceGPT	96.69	29.12	21.67	54.58	48.61	53.21	25.80	30.38	37.62 (57.96%)
Týr-the-Pruner	87.37	38.38	23.20	62.17	53.35	59.63	27.40	34.06	42.60 (65.63%)
DDP	31.20	47.14	29.78	62.42	56.51	65.34	30.00	45.04	48.03 (74.00%)
D-LLM	3877.53	27.19	24.32	40.86	49.57	50.65	26.00	26.35	34.99 (53.91%)
SkipGPT	156.58	30.89	20.99	43.30	50.04	55.77	25.80	30.38	36.74 (56.60%)
WIDE (32)	**17.28**	**62.84**	**35.41**	65.20	**62.51**	73.01	38.40	**61.60**	**57.00 (87.81%)**
WIDE (64)	19.02	61.74	34.64	**66.67**	59.98	72.36	38.60	60.84	56.40 (86.89%)
WIDE (128)	18.11	57.66	35.32	62.02	60.77	**73.23**	**39.20**	59.76	55.42 (85.38%)
LoRA Tuning (S2)									
Dense	9.86	72.64	46.42	72.87	68.98	77.91	42.20	74.00	65.00 (100.14%)
Shortened-ppl	29.75	46.51	27.47	61.28	51.22	64.85	29.60	39.91	45.83 (70.61%)
Shortened-taylor	31.50	46.13	25.94	61.88	51.70	64.85	28.20	42.16	45.84 (70.61%)
CoopPruner	42.69	39.23	23.12	49.24	52.96	63.11	29.20	35.89	41.82 (64.43%)
SliceGPT	31.13	36.48	22.44	59.32	54.61	59.46	26.60	40.40	42.76 (65.87%)
Týr-the-Pruner	53.11	45.62	25.25	57.58	52.56	65.34	30.00	43.46	45.69 (70.38%)
DDP	25.66	48.36	30.20	63.30	56.12	67.14	30.20	46.84	48.88 (75.30%)
D-LLM	193.97	31.40	21.25	57.09	51.22	57.18	24.40	28.07	38.66 (59.56%)
SkipGPT	18.91	58.92	33.45	62.81	58.64	72.31	36.60	57.59	54.33 (83.70%)
WIDE (32)	15.28	**67.72**	**37.97**	**66.94**	59.27	74.54	**40.60**	64.32	**58.77 (90.53%)**
WIDE (64)	**15.17**	65.78	37.29	65.23	61.01	**75.14**	39.80	64.26	58.36 (89.90%)
WIDE (128)	15.26	65.74	36.43	66.12	**61.88**	74.92	37.00	**64.69**	58.11 (89.52%)

Table 11: Qwen3-14B performance with target sparsity 25%.

Methods	WikiText2	ARC-E	ARC-C	BoolQ	Winogrande	PIQA	OpenbookQA	Hellaswag	MMLU (5-shot Acc.↑)	IFEval (prompt-strict ↑)	MBPP+ (pass@1↑)	GSM8K (Acc.↑)	Like. Acc.	Gen. Acc.
Calibration-only (S1)														
Dense	11.12	82.79	59.98	89.33	72.69	79.82	46.40	78.78	78.79	85.39	89.68	79.30	73.57 (100.00%)	84.79 (100.00%)
Shortened-ppl	30.71	43.56	28.67	61.80	50.99	67.46	33.40	47.35	23.19	10.35	1.06	1.59	44.55 (60.55%)	4.33 (5.11%)
Shortened-taylor	9970.21	28.37	23.63	60.58	50.36	51.47	30.60	25.91	23.20	14.79	0.00	0.00	36.76 (49.97%)	4.93 (5.81%)
CoopPruner	29.32	46.30	29.95	57.89	52.09	66.70	33.20	44.81	22.95	13.12	1.32	3.34	44.23 (60.12%)	5.92 (6.98%)
DDP	16.87	73.95	52.47	88.01	66.06	75.03	40.40	63.59	62.64	**70.79**	80.69	68.23	65.26 (88.71%)	73.23 (86.37%)
SkipGPT	20.86	75.92	52.81	84.03	58.48	75.29	37.80	63.96	44.79	48.24	1.05	65.73	61.63 (83.77%)	38.34 (45.21%)
WIDE (32)	**11.85**	85.69	63.39	88.34	**70.63**	80.08	45.80	74.23	73.50	69.87	81.74	82.33	72.70 (98.82%)	77.98 (91.96%)
WIDE (64)	15.63	**86.82**	**66.21**	87.40	68.66	**80.63**	**48.20**	74.78	72.89	70.60	82.01	81.34	**73.19 (99.49%)**	77.98 (91.97%)
WIDE (128)	12.15	86.53	62.96	**88.65**	69.61	80.35	48.00	**74.88**	**73.94**	70.42	**82.53**	**82.78**	73.11 (99.37%)	**78.57 (92.67%)**
LoRA Tuning (S2)														
Dense	9.58	87.84	68.09	88.99	74.59	82.21	48.60	80.32	78.69	78.18	84.65	88.77	76.16 (103.52%)	83.86 (98.91%)
Shortened-ppl	14.65	77.57	49.83	72.48	60.38	77.20	43.00	62.73	36.08	29.57	46.83	45.26	59.90 (81.42%)	40.55 (47.82%)
Shortened-taylor	88.33	75.59	52.90	85.35	66.22	73.23	40.20	65.78	62.83	58.78	69.05	73.46	65.26 (88.70%)	67.09 (79.13%)
CoopPruner	14.21	78.54	51.54	77.71	64.80	76.71	42.00	64.86	46.97	43.81	53.97	58.83	62.89 (85.48%)	52.20 (61.56%)
DDP	15.54	80.18	58.36	87.43	69.06	76.17	42.40	68.97	67.66	66.36	76.98	80.52	68.77 (93.48%)	74.62 (88.00%)
SkipGPT	12.30	86.19	63.56	87.64	64.95	**81.17**	49.00	74.67	68.50	64.69	79.36	85.97	71.96 (97.80%)	76.67 (90.42%)
WIDE (32)	**11.10**	86.86	65.52	88.37	**71.66**	80.57	48.80	76.14	74.17	**71.16**	83.59	87.94	74.01 (100.59%)	80.89 (95.40%)
WIDE (64)	14.99	87.20	65.27	**89.11**	70.95	80.90	49.00	76.20	75.13	68.94	**83.86**	**89.61**	74.22 (100.88%)	80.80 (95.29%)
WIDE (128)	11.15	**87.45**	**65.69**	89.08	69.29	80.79	**50.40**	**76.73**	**75.24**	**71.16**	**83.86**	89.38	**74.33 (101.03%)**	**81.46 (96.08%)**

Table 12: Qwen3-14B performance with target sparsity 50%.

Methods	WikiText2	ARC-E	ARC-C	BoolQ	Winogrande	PIQA	OpenbookQA	Hellaswag	MMLU (5-shot Acc.↑)	IFEval (prompt-strict ↑)	MBPP+ (pass@1↑)	GSM8K (Acc.↑)	Like. Acc.	Gen. Acc.
Calibration-only (S1)														
Dense	11.12	82.79	59.98	89.33	72.69	79.82	46.40	78.78	78.79	85.39	89.68	79.30	73.57 (100.00%)	84.79 (100.00%)
Shortened-ppl	5.41e+5	26.38	26.36	42.23	49.96	51.25	29.80	25.26	23.91	8.68	0.00	0.98	34.39 (46.74%)	3.22 (3.79%)
Shortened-taylor	2.27e+7	27.23	26.53	47.46	49.25	51.08	30.60	25.54	26.38	11.09	0.00	0.30	35.50 (48.26%)	3.79 (4.47%)
CoopPruner	21058.12	28.91	24.82	40.88	51.30	53.42	32.40	26.93	26.63	13.30	0.00	0.07	35.66 (48.47%)	4.45 (5.25%)
DDP	48.39	55.55	36.51	81.59	57.06	66.70	34.40	49.61	42.60	46.76	50.79	59.05	53.00 (72.04%)	52.20 (61.56%)
SkipGPT	546.29	32.19	26.45	39.02	51.14	51.79	27.20	28.54	24.87	23.65	0.26	0.37	35.15 (47.77%)	8.09 (9.54%)
WIDE (32)	**19.18**	83.62	60.15	**87.58**	65.82	76.27	46.40	67.50	64.27	**58.41**	**76.98**	79.07	68.95 (93.71%)	**71.48 (84.31%)**
WIDE (64)	19.79	**85.10**	**63.13**	87.43	**67.08**	**78.29**	**47.20**	68.11	64.90	54.52	75.66	**79.30**	**70.15 (95.35%)**	69.82 (82.35%)
WIDE (128)	19.85	84.55	60.49	87.55	65.11	78.12	45.00	**68.15**	**65.00**	58.22	75.92	78.31	69.24 (94.11%)	70.81 (83.52%)
LoRA Tuning (S2)														
Dense	9.58	87.84	68.09	88.99	74.59	82.21	48.60	80.32	78.69	78.18	84.65	88.77	76.16 (103.52%)	83.86 (98.91%)
Shortened-ppl	36.18	56.27	28.41	61.68	50.43	65.99	33.20	39.33	24.04	10.16	4.76	2.12	44.91 (61.05%)	5.68 (6.69%)
Shortened-taylor	1137.25	50.92	33.95	66.17	55.48	63.60	29.80	43.17	29.91	23.29	18.25	18.57	46.62 (63.37%)	20.03 (23.63%)
CoopPruner	27.52	61.36	33.36	61.22	50.03	69.04	33.00	43.79	24.88	14.41	13.49	6.44	47.08 (63.99%)	11.44 (13.50%)
DDP	36.47	62.66	39.84	81.65	57.61	68.00	34.40	52.39	49.44	43.62	53.70	67.32	55.74 (75.77%)	54.88 (64.72%)
SkipGPT	21.38	70.15	48.72	75.25	49.80	68.93	32.60	54.47	57.69	49.16	74.86	80.36	57.20 (77.74%)	68.12 (80.34%)
WIDE (32)	15.65	85.31	63.22	87.67	**68.50**	78.56	47.59	71.77	67.51	57.67	78.57	83.09	71.26 (96.86%)	73.11 (86.22%)
WIDE (64)	15.20	85.35	62.88	**88.31**	66.69	**79.48**	48.40	71.35	68.28	**60.44**	**80.68**	84.68	71.34 (96.96%)	**75.26 (88.76%)**
WIDE (128)	**15.11**	**86.36**	**64.07**	88.25	67.56	78.34	**48.80**	**72.56**	**68.66**	60.25	79.10	**85.21**	**71.82 (97.62%)**	74.85 (88.28%)