Unified Static–Dynamic Pruning for Efficient LLM Inference

Jinhyeok Kim
Seoul National University
Seoul, South Korea
jhkim00@snu.ac.kr

Yejoon Lee
Seoul National University
Seoul, South Korea
leeyejoon@snu.ac.kr

Jaeyoung Do*
Seoul National University
Seoul, South Korea
jaeyoung.do@snu.ac.kr

ABSTRACT

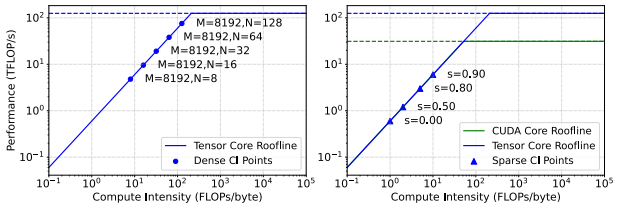
The increasing deployment of large language models (LLMs) has magnified the computational and memory bottlenecks of autoregressive decoding, where low compute intensity and bandwidth-bound kernels dominate inference cost. Weight pruning offers a promising remedy, but existing methods remain confined to either static pruning (SP)—which permanently removes redundant weights but lacks adaptivity—or dynamic pruning (DP)—which adapts to input sparsity but introduces runtime irregularity. This paper presents SPDP, a unified sparse-inference framework that integrates unstructured SP with input-adaptive DP for efficient LLM inference on GPUs. SPDP co-designs a new Tiled-Column-wise Bitmap Compressed (Tiled-CBC) format and two complementary GPU kernels: (1) a CUDA-core spMspV kernel featuring Hybrid Activation-aware Dynamic Shared-Memory Bitmap Decoding (HAD-SMBD) for fine-grained, runtime activation skipping, and (2) a Tensor-Core SpMM kernel optimized for prefill computation. This joint format–kernel design harmonizes static and dynamic sparsity, maintaining bandwidth-efficient memory access and high compute intensity under both phases of LLM inference. Comprehensive evaluations on inference-optimized GPUs demonstrate that SPDP achieves 1.24×–1.37× average speedup (up to 2.51×) over state-of-the-art sparse frameworks such as SpInfer, while matching perplexity with up to 25% higher sparsity. SPDP advances the inference efficiency–quality Pareto frontier, showing that unified static–dynamic pruning can deliver substantial throughput and performance-per-watt improvements in large-scale LLM serving.

PVLDB Reference Format:
Jinhyeok Kim, Yejoon Lee, and Jaeyoung Do. Unified Static–Dynamic Pruning for Efficient LLM Inference. PVLDB, 19(11): 2950 - 2963, 2026.
doi:10.14778/3836663.3836665

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/AIDASLab/SPDP.

1 INTRODUCTION

The rapid evolution of Large Language Models (LLMs) [21, 26, 60, 64] has fundamentally transformed artificial intelligence, enabling state-of-the-art performance in reasoning [21, 26], dialogue [52], summarization [68], and code generation [28]. However, this progress comes at substantial costs: modern LLMs with tens of billions of parameters demand immense memory bandwidth and arithmetic throughput. The problem has intensified with the advent of *test-time scaling* [21, 26]—which intentionally increases computation during inference to enhance reasoning quality—and the rise of *agentic AI* systems [2, 56, 65] that autonomously plan, reason, and interact with tools. These paradigms drastically increase the number of generated tokens per session, shifting workloads toward iterative, decode-heavy inference where memory bandwidth and latency become critical bottlenecks, thereby reducing performance-per-watt due to the power-inefficient decoding phase [17].



(a) Output with scaling in GEMM (b) Sparsity scaling in SpMV

Figure 1: Roofline analysis of matrix multiplication on an A10G GPU. (a) Effect of output width (N) on compute intensity and throughput in dense GEMM. As N decreases, performance shifts from compute- to memory-bound. (b) Effect of sparsity (s) on throughput in sparse SpMV ($M = 8192$, $N = 1$). Higher sparsity reduces memory traffic and increases compute intensity.

To address these growing costs, model compression has re-emerged as a cornerstone for efficient LLM serving. Among various compression strategies, *weight pruning* (or sparsification) has shown strong promise in reducing both memory footprint and arithmetic cost [18, 22, 58, 70, 75]. By removing redundant weights, pruning introduces sparsity that can theoretically reduce the number of required floating-point operations. Pruning approaches differ in granularity: structured pruning eliminates entire neurons or channels, offering hardware regularity but often at the cost of accuracy; semi-structured pruning provides a middle ground that can be exploited by custom kernels; and unstructured pruning offers the finest granularity, selectively removing individual weights while typically preserving accuracy even at high sparsity levels.

While structured and semi-structured pruning achieve practical acceleration on specific hardware, their rigid sparsity patterns make it difficult to preserve model quality and generalize across architectures. As a result, recent research has increasingly focused on *unstructured pruning*—the most flexible yet system-challenging paradigm—where individual weights are selectively removed without

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836665

any structural constraints. Despite its algorithmic advantages, unstructured sparsity introduces substantial *system-level challenges* for large-scale LLMs. Whereas smaller networks can tolerate extreme sparsity (70–90%) with minimal accuracy degradation, large LLMs exhibit much lower redundancy and suffer pronounced quality loss at similar sparsity levels. Furthermore, autoregressive decoding reduces computation to inherently memory-bound skinny matrix multiplications, making the realized benefit of sparsity highly dependent on how the sparse structure is represented and scheduled on GPUs. Consequently, general-purpose sparse kernels [19, 44, 73] often fail to provide meaningful acceleration under moderate sparsity (30–60%)—a regime typical in LLM pruning—due to limited compute intensity and nontrivial metadata overheads.

Recent systems such as Flash-LLM [62], SpInfer [14] address this limitation by introducing *load-as-sparse, compute-as-dense* pipelines that reconstruct dense tiles for Tensor Core execution, effectively mitigating memory bottlenecks. However, these methods rely exclusively on *static sparsity*, where pruning patterns are fixed offline, thus forfeiting adaptivity to input-dependent redundancy. In contrast, *dynamic pruning* [11, 16, 31, 35, 37] adaptively skips computation for unimportant activations at runtime, offering input-aware efficiency. Unfortunately, existing dynamic pruning frameworks typically restrict themselves to coarse-grained or structured activation masks for hardware compatibility, limiting fine-grained control and failing to exploit the potential of unstructured sparsity.

Our key insight is that static and dynamic pruning are inherently complementary. Static pruning (SP) permanently removes redundant *weights*, reducing memory footprint and bandwidth cost, while dynamic pruning (DP) adaptively skips redundant *activations*, reducing token-dependent computation. SP and DP act along orthogonal dimensions: SP prunes weights across rows, whereas DP skips activations across columns. Their sparsity patterns therefore overlap minimally and *multiply* effective sparsity. Our roofline-based analysis (Figure 1) shows that this combined sparsity significantly raises compute intensity and attainable throughput in memory-bound decode workloads, offering a new efficiency frontier for LLM inference (for more details, see Section 3.1).

However, translating this theoretical complementarity into practical acceleration exposes *system-level incompatibilities*. Existing SP and DP frameworks are not directly composable: static sparse kernels are row-major and optimized for sparse matrix–matrix multiplication (SpMM), whereas dynamic pruning induces column-wise sparsity and reduces workloads to sparse matrix–sparse vector multiplication (spMspV). This mismatch disrupts coalesced memory access, breaks warp-level stride regularity, and severely degrades GPU throughput. Furthermore, Tensor Core pipelines—designed for dense, fixed-tile fragments—cannot efficiently process per-token activation sparsity, leading to sharp drops in utilization. Therefore, achieving end-to-end acceleration under joint SP–DP sparsity requires full-stack co-design across layout, scheduling, and execution.

This paper presents **SPDP**, a unified sparse-inference framework that integrates unstructured *static pruning* with input-adaptive *dynamic pruning* for efficient LLM inference. SPDP bridges the static and dynamic paradigms through a joint *format–kernel co-design*. At its core is the *Tiled-Column-wise Bitmap Compressed (Tiled-CBC)* format, which encodes statically pruned weights in column-major micro-tiles aligned with DP's activation sparsity pattern. Built atop this layout, a CUDA-core-based *spMspV kernel* exploits hybrid activation-aware decoding and a fine-grained asynchronous pipeline to skip inactive columns dynamically while preserving coalesced memory access. During the prefill stage, the same format seamlessly supports a Tensor-Core-based *SpMM kernel* through efficient shared-memory layout alignment, maintaining high compute utilization without reformatting. Together, these designs make it possible—*for the first time*—to unify static and dynamic sparsity under a single GPU execution framework that achieves both algorithmic and system-level efficiency.

Comprehensive evaluations on inference-optimized GPUs show that SPDP achieves higher SM utilization—the fraction of active GPU compute units—and fewer shared-memory conflicts than state-of-the-art systems such as SpInfer, while sustaining comparable memory throughput. End-to-end experiments on LLaMA-2–7B confirm that SPDP reduces time per output token (TPOT) at matched perplexity, demonstrating that static–dynamic co-pruning improves the speed–quality trade-off in memory-bound decoding.

This paper makes the following contributions:

- We present a unified analysis of static and dynamic pruning, revealing their orthogonal sparsity dimensions and quantifying their theoretical synergy through roofline modeling.
- We propose SPDP, a co-designed sparse inference framework that harmonizes unstructured static pruning and input-dependent dynamic pruning for efficient LLM inference.
- We introduce the *Tiled-CBC* format and two complementary sparse kernels: (1) a CUDA-core `spMspV` kernel with Hybrid Activation-aware Dynamic Shared-Memory Bitmap Decoding (HAD-SMBD), and (2) a Tensor-Core `SpMM` kernel for prefill.
- We empirically demonstrate that combining SP and DP advances the inference efficiency–quality Pareto frontier. SPDP achieves **1.24× and 1.37× average speedup (up to 1.70× and 2.51×)** over SpInfer, and **1.88× and 2.11× average speedup (up to 3.32× and 3.52×)** over the cuBLAS baseline on A10G and L4 GPUs, respectively. While maintaining comparable perplexity with **up to 25% higher sparsity**, SPDP delivers both lower latency and improved performance-per-watt in large-scale LLM serving.

2 BACKGROUND & RELATED WORKS

We review LLM inference and pruning-based acceleration. We contrast prefill and decode, categorize pruning by granularity and adaptivity, and summarize sparse GPU kernels for LLMs.

2.1 LLM Inference

Modern Transformer-based LLM [61] inference comprises two workloads: *prefill* and *decode*. Prefill computes the KV cache of all prompt tokens in a single forward pass, whereas decode generates one token per pass and appends its KV states to the cache. In prefill, the input tensor has shape [B, L, H], where B, L, and H denote batch size, sequence length, and hidden dimension, respectively. Large B and L amortize the cost of loading weights into on-chip memory across both dimensions. Decode instead operates on [B, 1, H], yielding much lower compute intensity than prefill; at $B = 1$, the linear layers reduce to matrix-vector multiplications.

2.2 Pruning

Pruning the weights of neural network can reduce the size of weights loaded to on-chip memory. Based on its granularity, pruning can be divided into two categories: (semi-) structured pruning and unstructured pruning. (Semi-) structured pruning prunes structured elements, such as the neurons of an MLP layer [4, 6, 37, 39] or N elements out of M consecutive elements [15, 75], while unstructured pruning [18, 24, 58] prunes the individual elements. Pruning methods are also divided into two distinct flavors, based on whether pruning is dependent on the input.

Static Pruning. Static pruning (SP) applies a fixed mask $\mathcal{M}$ to the weight matrix W, resulting in a pruned weight W_{sp}. $\odot$ denotes element-wise multiplication.

$$W_{sp} = W \odot \mathcal{M},\ W \in \mathbb{R}^{M\times K}, \mathcal{M} \in \{0,1\}^{M\times K}$$

Structured static pruning is the most common and easiest way to accelerate, however, it degrades the quality of LLM compared to classical deep learning models. Therefore, 2:4 semi-structured pruning emerges with NVIDIA Sparse Tensor Core (SpTC) [42], which enhances the tradeoff between model quality and speedup. Various works aim to mitigate the quality degradation imposed by the rigid 2:4 constraint, including learnable semi-structured mask training [15], adaptive matrix reparameterization for high-performance 2:4 sparsity [36], proximal regularization frameworks for structured mask learning [34], as well as hardware-aware 2:4 pruning and fine-tuning strategies designed to fully exploit SpTC.

In this work, we limit our focus to *unstructured* static pruning. Magnitude pruning [22] prunes individual elements based on their magnitude. SparseGPT [18] formulates the selection problem by sparse regression and proposes an efficient solve. Wanda [58] considers activation feature norms alongside weight magnitudes. Unstructured static pruning better preserves model quality and, owing to its fine granularity, enables continuous sparsity ratios and flexible layer-wise allocation across Transformer blocks [8, 38]. In contrast, structured pruning operates at a coarser granularity and often incurs larger accuracy degradation, while fixed-pattern semi-structured methods (e.g., 2:4 sparsity) restrict the achievable sparsity allocation despite being easier to accelerate on modern GPUs.

Dynamic Pruning. Dynamic pruning (DP), also known as activation sparsity or contextual sparsity [11, 31, 35, 37, 41, 57, 72], prunes the LLM based on the given input. The mask $\mathcal{M}$ is not fixed; it is instead dynamically determined by the input x.

$$W_{dp}(x) = W \odot \mathcal{M}(x),\ W \in \mathbb{R}^{M\times K}, \mathcal{M}(x) \in \{0,1\}^{M\times K}, x \in \mathbb{R}^{K}$$

DejaVu [37] pioneered DP by exploiting the high sparsity of activations, stemming from using ReLU as an activation function.

Recent DP methods [11, 31, 35] extends DP to general models not using ReLU. Existing DP methods typically operate at a structured granularity, which limits opportunities for finer-grained sparsity.

To the best of authors' knowledge, DuoGPT [67] is the only work so far to combine dynamic pruning with static pruning. It achieves better generation quality compared to SparseGPT or Wanda, under the same compression ratio. However, the speedup from sparsity is only provided as a theoretical argument, leaving contribution for GPU kernel implementation and practical speedup.

2.3 Unstructured Pruning Acceleration

Exploiting unstructured pruning for acceleration has long been constrained by the inefficiencies of sparse matrix–matrix multiplication (SpMM) on GPUs. Traditional GPU libraries such as cuSPARSE [44], sparse kernel generator TACO [30], or recent works on GPU SpMM [13, 33, 44, 51, 71] or SpMV [12, 32] generally focus on extremely high sparsity, often targeting scientific workloads. In contrast, LLMs tend to exhibit moderate sparsity, where the overhead of index storage and irregular memory access often outweighs the computational savings.

Sputnik [19] and SparTA [73] introduced DNN-oriented sparse GPU kernels. Sputnik exploited regularities of DNNs, and SparTA adopted a compiler-level design with the Tensor-with-Sparsity-Attribute (TeSA) abstraction, generating specialized kernels for known sparsity patterns. SparseTIR [66] proposed a composable sparse compilation framework that supports flexible sparse formats and schedule transformations. However, its performance remains low compute intensity in LLM workloads.

Flash-LLM [62] advanced this line of work with a hardware-aware *load-as-sparse, compute-as-dense* (LSCD) approach. Instead of performing operations directly in the sparse domain, Flash-LLM loads pruned weights in compressed form from global memory, reconstructs dense tiles on-chip using shared memory, and executes them with dense Tensor Core instructions. This LSCD-based design effectively mitigates the memory-bandwidth bottleneck while exploiting the GPU's high-throughput compute units. Building upon this, SpInfer [14] further introduced a Tensor-Core-Aware Bitmap Encoding (TCA-BME) scheme, which compresses sparse matrices using lightweight bitmaps optimized for warp-level access patterns, achieving state-of-the-art SpMM performance on modern GPUs. At the hardware-software co-design level, Coruscant [29] co-designed a bitmap-based SpMM kernel with a Sparse Tensor Core extension to efficiently support unstructured LLM weight sparsity.

3 MOTIVATION & CHALLENGES

We examine when sparsity benefits LLM inference, focusing on memory-bound decode. A roofline analysis (Section 3.1) shows that reducing data movement via compression and skipping raises attainable throughput. This motivates combining static and dynamic pruning, which act on orthogonal dimensions (Section 3.2). We then outline three GPU challenges: layout mismatch, metadata overhead, and reduced Tensor Core efficiency (Section 3.3).

3.1 Efficacy of Pruning: Roofline Analysis

The decode phase of small-batch LLM inference is inherently *memory-bound*. Since there is limited parallelism across the batch dimension, data transfers between off-chip and on-chip memory dominate execution time. This bottleneck becomes more severe on consumer-grade GPUs—commonly used for low-batch inference—which typically rely on GDDR memory offering much lower bandwidth than HBM-equipped enterprise GPUs. The limited memory bandwidth further exacerbates the memory wall in small-batch LLM decoding. Moreover, the recent rise of test-time scaling and reasoning-oriented models [21, 26] drastically increases the number of generated tokens per session, further amplifying the cost of decode computation.

To quantitatively analyze how sparsity alleviates the memory bottleneck, we adopt the *roofline model*. Two metrics characterize the efficiency of sparse computation, including the *Compression Ratio (CR)* and the *Compute Intensity (CI)*.

For an FP16 weight matrix $W \in \mathbb{R}^{M\times K}$, the CR is defined as

$$CR = \frac{Stor_{dense}}{Stor_{compressed}} = \frac{2B \times M \times K}{Stor_{compressed}}, \tag{1}$$

where B denotes bytes, and each FP16 element occupies $2B$. Here, $Stor_{compressed}$ is the total storage size of the compressed sparse matrix, while $Stor_{dense} = 2B \times M \times K$ is that of the dense FP16 weight matrix. A higher CR indicates stronger compression, reducing memory footprint and data transfer overhead.

For input matrix $X \in \mathbb{R}^{K\times N}$ and a weight matrix $W \in \mathbb{R}^{M\times K}$, the CI is the ratio of computation to memory traffic, given by

$$CI = \frac{\text{FLOPs}}{\text{Bytes transferred}} = \frac{M \times N}{\frac{M}{CR} + N} \tag{2}$$

A higher CI implies more computation is performed per unit of memory access, improving performance in memory-bound regimes.

Based on these definitions, the *roofline model* expresses the achievable kernel performance as $P = \min(P_{\text{peak}},\ CI \times B_{\text{mem}})$, where P_{peak} denotes the peak compute throughput (FLOPs/s) and B_{mem} is the available memory bandwidth (Bytes/s).

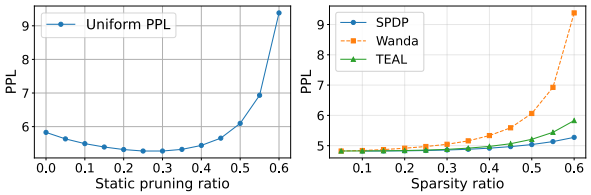
The roofline model thus visualizes the transition between the compute-bound and memory-bound regions: kernels with low CI are limited by memory bandwidth, while those with high CI can approach the peak compute performance (i.e., P_{peak}).

Figure 1 illustrates this effect on an NVIDIA A10G GPU. As shown in Figure 1a, throughput saturates at small N due to memory limitations. In Figure 1b, increasing sparsity effectively reduces data movement, raising CI and thereby achievable throughput. In our analysis, sparsity could improve performance by up to 10× under memory-bound decode workloads. Therefore, pruning emerges as an essential mechanism for accelerating LLM inference, especially in small-batch or interactive settings where memory bandwidth is the primary bottleneck.

3.2 Bring the Best of Both Worlds: Combining Static and Dynamic Pruning

Static and dynamic pruning each provide unique benefits but also suffer from fundamental limitations. *Static pruning (SP)* [18, 22, 58, 70, 75] removes weights offline, permanently reducing model size and memory footprint. However, since SP is input-agnostic, it may remove parameters critical for specific contexts, leading to quality degradation. *Dynamic pruning (DP)* [11, 16, 31, 35, 37], on the other hand, adaptively zeros out activations based on input, enabling context-aware sparsity. Yet, DP typically operates at structured granularity—pruning entire columns of the weight matrix—and thus lacks fine-grained flexibility. Moreover, since the dense weights remain loaded in memory, DP offers little benefit in terms of memory footprint and introduces runtime overhead for mask generation and control flow.

Despite their differences, SP and DP are inherently complementary. DP's input-dependent adaptivity mitigates SP's input-agnostic nature, while SP's unstructured fine granularity compensates for DP's coarse activation-level pruning and improves overall storage



(a) PPL under combined pruning **(b) PPL across various methods**

Figure 2: Perplexity comparison across pruning strategies. Combining static and dynamic pruning yields the lowest perplexity across all sparsity levels.

efficiency. This complementarity naturally raises the question: *Can static and dynamic pruning be jointly employed to push the Pareto frontier of speed and quality?* To answer this question, we empirically analyze how SP and DP interact using state-of-the-art pruning methods. For SP, we adopt magnitude pruning [22], SparseGPT [18], and Wanda [58]; for DP, we use TEAL [35]. SparseGPT and Wanda require a small calibration set to estimate input statistics. Following prior work [18, 58], we sample 128 sequences, each with the full model context length, from the C4 [54] dataset. To evaluate the text quality of jointly pruned models, we report perplexity on the WikiText [40] dataset using Llama2-7B [60]. Within this setup, we also measure the number of elements pruned by both SP and DP—referred to as the *overlapping sparsity*.

3.2.1 Perplexity Comparison. We compare perplexity across various static and dynamic sparsity ratios. When combining static sparsity sp and dynamic sparsity dp, the total sparsity is defined as $1-(1-sp)(1-dp)$. As shown in Figure 2a, the lowest perplexity is achieved when both sp and dp are moderate (e.g., around 0.3–0.4), rather than extreme on either side. This indicates that jointly applying SP and DP yields a balanced pruning effect—retaining essential weights while dynamically skipping unimportant activations. Figure 2b further shows that this advantage persists across different total sparsity levels, and becomes more pronounced at high sparsity ratios where substantial speedup is practically attainable. Overall, combining SP and DP improves the speed–quality trade-off beyond what either method can achieve alone.

3.2.2 Overlapping Sparsity Analysis. When combining static and dynamic pruning, an important consideration is whether the two methods prune the same weights or act independently. If SP and DP operate independently of each other, their overlapping sparsity ratio should equal the product of the two sparsity ratios, $sp \times dp$. For example, when both sp and dp are 0.6, the theoretical overlap is 0.36, resulting in a total sparsity of $1-(1-sp)(1-dp) = 0.84$. In contrast, if one pruning method fully subsumes the other, the total sparsity would remain 0.6, representing the worst-case overlap.

In the following analysis, we measure the deviation of observed overlap from the theoretical expectation. Figure 3 shows the deviation of overlapping sparsity, as a change from the theoretical baseline $sp \times dp$. As shown in Figure 3, the overlaps between each unstructured SP method and TEAL remain close to or slightly below the theoretical value, indicating that SP and DP make largely independent pruning decisions. Consequently, the total sparsity

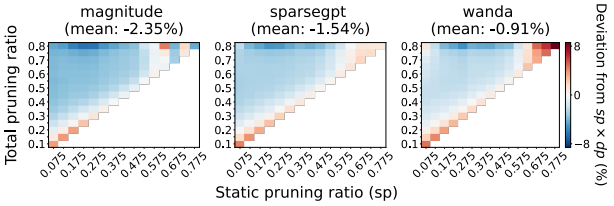


Figure 3: Deviation of measured overlapping sparsity from theoretical level $sp \times dp$. Color represents the deviation in %, while X and Y axes represents the sparsity ratios. Across three SP methods combined with TEAL, overlap remains on par or slightly below the theoretical level.

$1-(1-sp)(1-dp)$ holds in practice, confirming that combining SP and DP achieves the expected level of sparsity—and thus retains the potential for proportional speedup without redundancy.

3.2.3 Roofline Analysis. The theoretical benefit of combining SP and DP can be further analyzed through the roofline model introduced earlier. We extend the formulation to capture their joint impact on CI:

$$CI_{SPDP} = \frac{\text{FLOPs}}{\text{Bytes transferred}} = \frac{M \times N}{\frac{M}{CR} \cdot (1-s) + N} \tag{3}$$

where s denotes the *effective dynamic sparsity ratio*—the fraction of weight matrix columns skipped at runtime due to DP. This reflects all columns removed dynamically, including those already pruned statically (i.e., overlapping sparsity). Accordingly, $(1-s)$ represents the fraction of activations that remain active and thus determine the amount of weight data that must be accessed during inference.

In this formulation, SP increases the compression ratio (CR), while DP further reduces the number of active columns by $(1-s)$. Together, these effects multiplicatively boost CI, leading to higher throughput under memory-bound conditions. In practice, minor overlap between SP and DP may slightly reduce the effective gain, since some columns pruned by DP may already have been removed by SP. However, as shown in Section 3.2.2, the measured overlap remains negligible, validating that s provides a reliable estimate of the runtime sparsity.

This roofline analysis provides a simple yet intuitive perspective on how SP (via CR) and DP (via s) complement each other to enhance computational efficiency, particularly in memory-bound scenarios. While the roofline model assumes ideal integration between static and dynamic sparsity, achieving such synergy in practice remains challenging due to memory layout mismatches and runtime scheduling overheads—issues further discussed in the next section.

3.3 Challenges in Realizing Acceleration

Although static pruning (SP) and dynamic pruning (DP) are complementary in theory, their theoretical sparsity benefit does not directly translate into practical GPU acceleration. Realizing this benefit requires system support across three coupled aspects: a sparse layout that preserves static compression under input-dependent dynamic sparsity, a decode execution path that avoids Tensor-Core underutilization for DP-induced spMspV, and prefill support that preserves Tensor-Core SpMM efficiency without maintaining a separate sparse representation. Existing unstructured SP formats [14, 62] and DP methods [35] do not jointly satisfy these requirements. The resulting system-level challenges are summarized as follows.

(1) Layout mismatch and runtime sparse-fragment access overhead. SP kernel [14, 62] compresses a static sparse weights and optimize data movement for SpMM. DP changes the problem during decode: zeroing token-dependent activation turns it into input-dependent spMspV, where only weight columns associated with nonzero activation entries matter for the current token.

This creates a layout mismatch between SP and DP. A fixed SpMM-oriented format may still decode weight fragments that are inactive under DP. Avoiding this waste requires locating only the active sparse fragments at runtime, but unstructured SP makes those fragments irregular because each column contain different number of nonzeros. Therefore, realizing the theoretical benefit of combined SP and DP requires runtime-aware sparse access that preserves static compression while respecting input-dependent activation sparsity.

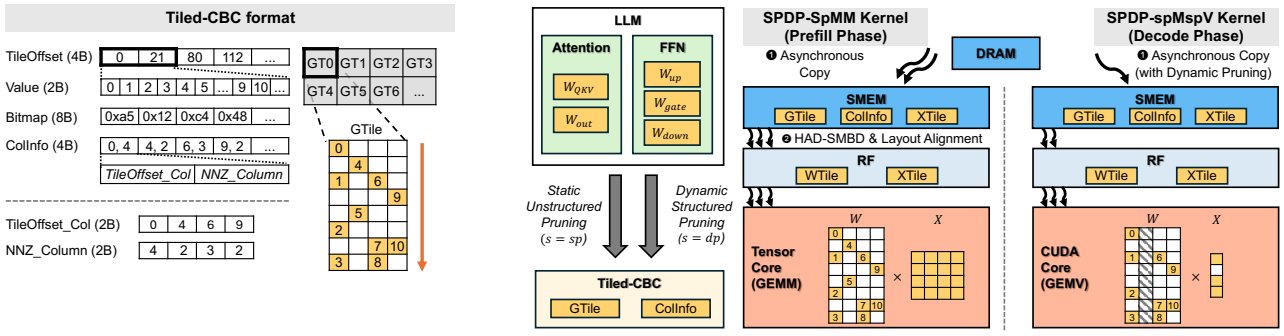
(2) Hardware-level conflict between Tensor-Core utilization and DP. Tensor Cores excel at matrix–matrix computation (GEMM, SpMM), but DP reduces each token's work to *GEMV-like* (or spMspV) execution with no reuse along the N dimension. Under PTX instruction `mma.m16n8k16`, CI can drop by up to 8×, sharply lowering Tensor-Core efficiency. On inference-oriented GPUs [47, 49, 50], achievable Tensor-Core throughput for GEMV is only about one-fourth of the CUDA core's FP32 peak, so utilization can collapse and throughput may fall below dense cuBLAS baselines. Falling back to CUDA cores restores fine-grained, index-aware execution, but requires a redesigned pipeline to maintain instruction-level parallelism (ILP) and overlap metadata, memory, and compute for DP-driven spMspV.

(3) Prefill compatibility under the same format. DP is applied only during decode in SPDP; the prefill phase therefore relies on SP alone and corresponds to SpMM, which is amenable to Tensor-Core execution [14, 62]. A format optimized for decode-time spMspV is not automatically compatible with Tensor-Core SpMM: decode favors selective sparse-fragment access induced by runtime activation sparsity, whereas Tensor-Core SpMM requires regular MMA-friendly operand tiles. A separate prefill format would increase storage and preprocessing overhead, while a single format without careful kernel support can underutilize Tensor Cores.

Together, these issues highlight the gap between theoretical sparsity gains and practical acceleration. Bridging it requires a framework that (i) unifies SP and DP into a bandwidth-efficient layout, (ii) enables efficient CUDA-core spMspV for decode, and (iii) preserves Tensor-Core efficiency for prefill. The next section presents the design of SPDP, which achieves these goals via joint format–kernel co-design.

4 DESIGN OF SPDP: UNIFYING STATIC AND DYNAMIC SPARSITY

The challenges above motivate a joint format-kernel co-design rather than a standalone sparse format or a single sparse kernel.



(a) Overview of the Tiled-CBC format.

(b) Design overview of SPDP.

Figure 4: (a) Tiled-CBC stores nonzeros within each GTile in column-major order using a hierarchical layout. Each GTile has size $TILE\_M \times TILE\_K$, illustrated as 8×4 for simplicity. (b) SPDP combines static pruning (SP) and dynamic pruning (DP) with the Tiled-CBC format and phase-specific inference kernels. The compressed weights are processed by the SpMM kernel during prefill and by the spMspV kernel during decode.

SPDP addresses them by combining the *Tiled-CBC format* with phase-specific inference kernels, as illustrated in Figure 4b. Tiled-CBC (Section 4.1) organizes statically compressed weights for efficient decode-time spMspV while retaining low indexing overhead. The CUDA-core *spMspV kernel* (Section 4.2) then exploits this format to process input-dependent sparse activations during decode using optimized data movement, Hybrid Activation-aware Dynamic Shared Memory Bitmap Decoding (HAD-SMBD), and a fine-grained asynchronous pipeline. The Tensor-Core *SpMM kernel* (Section 4.3) supports prefill by reusing the same compressed format and arranging decoded tiles into Tensor-Core-friendly operand layouts.

4.1 Tiled-CBC: Unified Compression Format

Motivated by the layout mismatch between static compression and dynamic access, we propose the *Tiled-Column-wise Bitmap Compressed (Tiled-CBC)* format as the core sparse representation of SPDP. Tiled-CBC makes sparse weight fragments directly addressable at column granularity while preserving a tiled representation for GPU execution, enabling decode-time spMspV to skip activation-inactive fragments with low indexing overhead.

Tiling Design. Tiled-CBC adopts a tile-level compression scheme organized in a column-major layout, since modern DP methods apply column-wise pruning along the input dimension; this layout therefore better preserves GPU-friendly memory access under DP. As illustrated in Figure 4a, the weight matrix is hierarchically divided into *GTiles*, each consisting of multiple column tiles. Each column tile, with dimensions $TILE\_M \times 1$ (set to 256×1), serves as the smallest unit for compression and pruning. This granularity is intentionally aligned with CUDA's 64-bit word size so that sparsity in each column can be encoded using four `uint64_t` bitmaps, where each bit indicates whether the corresponding element is nonzero. Such organization ensures efficient column-wise pruning and contiguous memory access during DP, consistent with the principle of activation sparsity emphasized in TEAL [35]. At the outer level, each *GTile* covers $TILE\_M \times TILE\_K$ elements and contains multiple column tiles. Thread blocks are assigned to process GTiles, while warps within each block handle column tiles in parallel. GTiles are stored in row-major order to preserve inter-tile coalescing and balance GPU workload distribution.

Storage Structure. Tiled-CBC represents a sparse matrix with five arrays: `TileOffset`, `Values`, `Bitmap`, `ColInfo`, and optional alignment metadata. `TileOffset` indexes each GTile, and `Values` stores nonzeros in tile-column order. The `Bitmap` array holds four 64-bit integers per column in the tile, indicating the location of nonzero entries. To facilitate DP, SPDP introduces a compact `ColInfo` array that packs two 16-bit fields: the upper bits store the column's starting offset within the tile (`TileOffset_Col`), and the lower bits store the number of nonzeros in that column (`NNZ_Column`). This packing enables low-overhead skipping of inactive columns at runtime, achieving contiguous memory access even under input-dependent pruning.

Let $NGT = (M/TILE\_M) \times (K/TILE\_K)$ denote the number of GTiles, $NBM = (M \times K/64)$ the number of bitmaps, and $NNZ = M \times K \times (1-s)$ the number of non-zero elements with static sparsity s. Then the memory cost of Tiled-CBC is:

$$Stor_{\text{Tiled-CBC}} = 4B \times (NGT + 1) + 4B \times (NGT \times TILE\_K) + 8B \times NBM + 2B \times NNZ.$$

Here, a 4B integer in `TileOffset` and `ColInfo`, an 8B `Bitmap` per `uint64_t`, and an 2B `Values` each contribute to the total storage.

Compared with prior formats optimized primarily for statically pruned weights [14, 62], Tiled-CBC exposes the sparse fragments needed by input-dependent DP without scanning inactive fragments. This structure preserves compression at moderate sparsity levels and provides the column-addressable layout used by the CUDA-core SPDP-`spMspV` kernel.

4.2 Decode Phase (spMspV) Kernel Design

The hardware-level conflict between Tensor-Core utilization and DP motivates a CUDA-core decode path rather than Tensor-Core execution. Building on the column-addressable sparse fragments exposed by Tiled-CBC, we develop the SPDP-`spMspV` kernel

Algorithm 1 SPDP-spMspV kernel pseudocode.

```
Inputs: SparseMatrix W, Column Metadata ColInfo (Tiled-CBC), Vector X, Split_K,
        DP threshold
Output: Vector/Matrix Y in ReductionWorkspace
int BatchID = blockIdx.x;
int TileY = blockIdx.y%(M/TILE_M), TileX = 0;
int NumIter = ComputeIterations(BatchID, Split_K);
__shared__ ValueBuffer[TILE_M][TILE_K];                    ▷ Sparse tile buffer
__shared__ BitmapBuffer[TILE_M][TILE_K];                   ▷ Bitmap buffer
__shared__ ColInfoBuffer[2][TILE_K];                       ▷ Double buffer (metadata)
__shared__ XTileBuffer[2][TILE_K];                         ▷ Double buffer (dense)
// Prologue: Pre-loop initialization.
ColInfoLoading(ColInfoBuffer[0], ColInfo);
XTileLoading(XTileBuffer[0], X + BatchID × TILE_K);        ▷ Load data for DP
cp_async_group_commit();
cp_async_wait_group(0);                                    ▷ Wait for ColInfo, XTile
GTileLoading(BitmapBuffer, ValueBuffer, W);                ▷ Load one GTile
cp_async_group_commit();
ColInfoLoading(ColInfoBuffer[1], ColInfo + TILE_K);
XTileLoading(XTileBuffer[1], X + (BatchID + 1) × TILE_K);
cp_async_group_commit();
cp_async_wait_group(1);                                    ▷ Wait for GTile
W_frag = HAD_SMBD(ValueBuffer, BitmapBuffer);              ▷ Asynchronous decode
__syncthreads();
// Main computation loop. Stages are pipelined as shown in Figure 5.
for int k = 0; k < NumIter − 1; k++ do
    GTileLoading(BitmapBuffer, ValueBuffer, W + offset(k));
    cp_async_group_commit();
    ColInfoLoading(ColInfoBuffer, ColInfo + offset(k));
    cp_async_group_commit();
    Y_frag = GEMVCompute(W_frag, X_frag, Y_frag);          ▷ GEMV on CUDA cores
    cp_async_wait_group(1);                                ▷ Wait for XTile, GTile
    __syncthreads();
    XTileLoading(XTileBuffer, X + offset(k));
    cp_async_group_commit();
    W_frag = HAD_SMBD(ValueBuffer, BitmapBuffer);
    cp_async_wait_group(0);                                ▷ Wait for ColInfo, XTile
    __syncthreads();
end for
// Epilogue: final iteration.
Y_frag = GEMVCompute(W_frag, X_frag, Y_frag);
StoreResults(ReductionWorkspace, Y_frag);
```

for decode-time execution. The kernel prefetches column metadata, loads only activation-relevant sparse fragments, and decodes bitmap-compressed columns with *Hybrid Activation-aware Dynamic Shared-Memory Bitmap Decoding (HAD-SMBD)*, a DP-aware extension of SMBD mechanism [14]. The decoded fragments are then accumulated with the corresponding activation values using CUDA-core GEMV.

During each iteration, a thread block performs five main procedures: (1) *ColInfo loading*. To support DP, the kernel uses `ColInfo` to determine where to prune. All threads in the block cooperatively load a `ColInfo` segment from global memory into shared memory. This metadata contains per-column offsets and nonzero counts. (2) *XTile loading*. Many DP methods require input activations for threshold comparison [16, 31, 35]; therefore, the kernel loads `XTile` before the corresponding weight tile. The dense input-vector tile (`XTile`) of X^T is loaded into shared memory and reused without pruning, since pruning such a small vector provides negligible bandwidth savings. (3) *GTile loading*. The kernel then loads a `GTile` from global memory into a shared-memory buffer (`WTile`). Using `ColInfo` and activation information from `XTile` with a pruning metric (e.g., magnitude), the kernel selectively fetches only relevant columns from global memory to realize DP. (4) *WTile decoding*. The `WTile` in shared memory is decoded into registers using HAD-SMBD, which interprets the bitmap layout with activation-aware column skipping. (5) *CUDA core computation*. Finally, CUDA cores perform matrix–vector multiplication between the decoded `WTile` and the `XTile`, both kept in registers. This load-as-sparse, compute-as-dense (LSCD) design is motivated by prior work [14, 62].

4.2.1 Efficient Decompression Process. Flash-LLM and SpInfer introduced an extract stage and Shared Memory Bitmap Decoding (SMBD) for efficient decoding under static sparsity, but they do not address decoding with input-dependent dynamic sparsity. To support the coexistence of static and dynamic sparsity, we extend SMBD from SpInfer into *Hybrid Activation-aware Dynamic Shared Memory Bitmap Decoding (HAD-SMBD)*, which decodes bitmap-compressed tiles while applying dynamic activation masks.

Register layout. HAD-SMBD materializes each compressed tile into warp-level register fragments using a *column-major* mapping. This column-major layout matches CUDA-core-based GEMV accumulation and aligns the decoded operands with the column-oriented Tiled-CBC format.

Hybrid two-phase decoding. For each column, HAD-SMBD first checks whether the column is active under dynamic pruning; inactive columns are skipped entirely. For an active column, threads interpret its 64-bit bitmap and unpack packed nonzeros from `Value` using the per-column base offset `TileOffset_Col` (from `ColInfo`). In **Phase I**, each thread computes a per-thread prefix rank (the number of preceding 1-bits before its position) via *MaskedPopcount*, and if its bit is set, loads the corresponding element from `Value[base + rank]` into the first FP16 lane of its register (otherwise, it inserts zero). In **Phase II**, the thread decodes the paired FP16 lane within the same register by reusing `base` and the Phase-I rank, updating the index with only a lightweight adjustment instead of recomputing a full *MaskedPopcount*. Both phases operate on the same column, so `ColInfo` is fetched once and reused across the two phases.

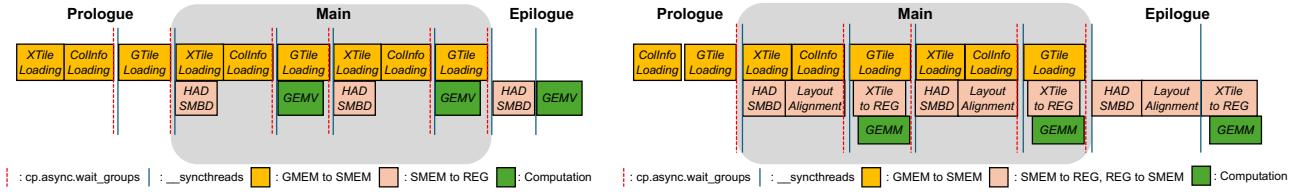
MaskedPopcount. To compute the per-thread rank within the packed `Value` buffer, HAD-SMBD uses *MaskedPopcount*, i.e., a prefix count of set bits in the bitmap (implemented with `__popcll()`).

Differences from SMBD. Compared to SMBD, HAD-SMBD (1) adopts column-major decoding for CUDA-core GEMV (instead of Tensor-Core-oriented row-major fragments), (2) leverages `ColInfo` to obtain column offsets required by the Tiled-CBC layout, and (3) keeps this shared-memory metadata access efficient (and skippable for DP-inactive columns), enabling DP-compatible decoding without disrupting the static bitmap pipeline.

Overall, HAD-SMBD bridges static bitmap decoding (for SP) with activation-dependent column skipping (for DP) in a unified decompression path.

4.2.2 Asynchronous Pipeline Design. We further optimized a fine-grained asynchronous pipeline from SpInfer, to eliminate the overhead while benefiting from memory access enabled by DP. Figure 5a shows the asynchronous pipeline design, which improves the utilization of compute by maximizing the overlap between memory transfers and computation operation.

Fine-Grained Asynchronous Group Management. SPDP adopts double buffering with two shared-memory buffers for GTiles and XTiles, allowing the next tiles to be prefetched while the current ones are being processed. Unlike SpInfer, which uses two asynchronous copy groups, we introduce a third independent group for *ColInfo* loading, enabling three concurrent operations: (1) metadata prefetching for DP decisions (*ColInfo loading*), (2) *XTile loading* for upcoming computations, and (3) *WTile decoding* via Hybrid



(a) SPDP-spMspV kernel Asynchronous Pipeline Design

(b) SPDP-SpMM kernel Asynchronous Pipeline Design

Figure 5: Asynchronous pipeline design for SPDP-spMspV and SPDP-SpMM kernels. Both kernels use HAD-SMBD for efficient decompression and fine-grained asynchronous group management to fully overlap stages without stalls. The SPDP-SpMM kernel adds a Layout Alignment stage for Tensor Core execution, carefully scheduled to minimize overhead.

Activation-aware Shared Memory Bitmap Decoding (HAD-SMBD). Since these tasks are independent, their concurrent execution fully hides the latency of HAD-SMBD and maintains high instruction-level parallelism with minimal synchronization.

Prefetching by Metadata Prefetching. Beyond overlapping weight and activation loading, SPDP adds a preemptive *metadata prefetching* stage that retrieves column activation patterns ahead of data loading. The prefetched metadata identifies which columns of GTile should be fetched in the next iteration, forming a two-step prefetch pipeline—metadata first, data second—so that only activation-relevant tiles are transferred. This metadata-driven prefetching ensures continuous overlap between decoding and memory access phases, eliminating unnecessary memory transactions and maximizing memory–compute concurrency. Although the added prefetching and double buffering slightly increase shared memory usage and extend the pipeline's prologue and epilogue, their effect is negligible. This is because occupancy is already bounded by register usage, and the main loop typically runs for many iterations, making the relative cost of the prologue and epilogue latency insignificant. Consequently, SPDP executes dynamic pruning with near-zero additional latency while preserving the lightweight, static-pipeline structure of SpInfer.

4.2.3 Implementation details. We further optimize our kernel to effectively utilize the underlying hardware, as detailed below.

Throughout steps (1)–(3), SPDP employs `cp.async` (compiled to `LDGSTS.128`) for high-efficiency global-to-shared memory transfer. Introduced in the Ampere architecture [48], this mechanism streams data directly from global to shared memory, reducing latency and register pressure. These transfers can be committed and synchronized using their intrinsic instructions, as illustrated in Algorithm 1. Each thread transfers a 128-bit vector (eight FP16s), and arrays are padded to satisfy alignment requirements. In particular, each column's values are padded to 16-byte boundaries (e.g., if 7 FP16 values exist, 2 bytes are padded), which is negligible with the large $TILE\_M = 256$. Given the large $TILE\_M$, we set $TILE\_K = 16$ to balance per-iteration granularity and overhead.

In step (3), the kernel streams a `GTile` into shared memory with an activation-aware selection pipeline. Using magnitude-based DP (e.g., TEAL), threads determine active columns and access `ColInfo` metadata from shared memory to compute offsets and nonzero counts without extra global reads. Predicate registers guard inactive columns to avoid warp divergence, while active ones proceed with coalesced `LDGSTS.128` transactions. The column loop is unrolled to enhance ILP and reduce loop overhead. We further adopted a tile-based scheme combined with Split-K parallelization to evenly distribute workloads, where each thread block processes an independent segment of the K dimension.

4.3 Prefill Phase (SpMM) Kernel Design

While SPDP is highly efficient for spMspV in decode, prefill must also be supported to cover the entire pipeline of LLM inference. Prefill, where only SP is applied, involves *sparse matrix-matrix multiplication (SpMM)*. However, this conflicts with our Tiled-CBC format, primarily designed for spMspV.

Therefore, we designed the SPDP-SpMM kernel for prefill, as shown in Figure 5b, while maintaining minimal overhead from Tiled-CBC format. The overall design of the SPDP-SpMM kernel is similar to that of SPDP-spMspV kernel, but differs in several stages.

First, prefill corresponds to matrix–matrix multiplication and is therefore amenable to Tensor-Core execution; we leverage *Tensor Cores*, following prior work [14, 62]. Second, we introduce an additional *layout alignment stage* to align the register layout between the Tiled-CBC format and the Tensor Core format. Third, we eliminate the DP stage to reduce overhead of it, mainly from *WTile, GTile loading*. Finally, several stages, including *XTile loading* and *XTile-to-register transfer*, are modified or newly added to accommodate the transition from vector to matrix input.

4.3.1 Layout Alignment. The Layout Alignment stage converts the sparse weight tile (`WTile`) stored in the SPDP-spMspV layout (column-major) into the Tensor-Core-compatible operand layout (row-major) required for SpMM. The conversion is performed entirely in shared memory, which serves as a low-latency staging buffer. However, a naive transpose-like reordering can cause severe shared-memory bank conflicts because threads within a warp access non-contiguous addresses.

To avoid this penalty, we apply a padding-based scheme. Each thread stores its elements into shared memory with a `TILE_K_PAD` stride (i.e., with an extra padded column per bank segment), which offsets addresses across lanes so that the threads in a warp map to distinct banks. The data is then reloaded into registers following the Tensor Core operand mapping, resulting in a bank-conflict-free transpose and improved alignment with `ldmatrix.x4` loads.

This two-phase reordering introduces only a single `__syncthreads()` barrier and some additional shared-memory

traffic, but the overhead is negligible compared to the latency avoided by eliminating bank conflicts. The padding cost is also minimal because occupancy is constrained by register usage, and the stage reuses the shared-memory buffer from the previous stage without requiring additional shared-memory allocation.

4.3.2 Asynchronous pipeline design. Similar to the SPDP-`spMspV` kernel, the SPDP-`SpMM` kernel uses an asynchronous pipeline, but it additionally accounts for the layout-alignment stage. Compared with decode-time `spMspV`, prefill removes DP-specific column selection in GTile loading stage and reduces the overhead of HAD-SMBD, while introducing layout alignment for Tensor-Core SpMM. In prefill, XTile spans hundreds to thousands of prompt tokens, increasing the latency of the XTile loading stage compared with decode-time SPDP-`spMspV`. This longer stage provides a larger window to overlap activation loading with sparse-tile decoding, layout alignment, and Tensor-Core computation.

Although SPDP-`SpMM` primarily targets prefill, it can be extended to support batched DP in both prefill and decode; doing so would require kernel modifications and additional algorithmic support.

5 EVALUATION

To evaluate SPDP, we examine both model quality and system performance. We first describe the experimental setup, including models, pruning procedures, GPU platforms, baselines, and evaluation metrics such as kernel throughput and time per output token (TPOT). We then present kernel-level and end-to-end results, together with profiling analyses that explain them.

5.1 Setup

We evaluate the model quality of SPDP on the following benchmarks: WikiText [40] (perplexity; lower is better), HumanEval [7] (`pass@1`) for code generation, GSM8K [9] (exact-match accuracy) for math reasoning, CoQA [55] (token-level F1) for conversational QA, and MMLU [23] (multiple-choice accuracy) for general knowledge. We compare against TEAL as a DP baseline and Wanda as a SP baseline. Unless stated otherwise, all results are reported on `Llama-2-7B-hf`. Since `Llama-2-7B-hf` [60] has limited code-generation capability, we evaluate all methods on HumanEval using `Qwen3-32B` [64]. While evaluating quantization compatibility, we adopt 4-bit round-to-nearest (RTN) weight-only quantization and follow a fixed pipeline of SP → quantization → DP; DP is enabled at inference time on top of the (pruned and quantized) weights, and our current pipeline does not support other orderings.

We evaluate the performance of SPDP at two levels: (1) the SPDP-`spMspV` kernel level and (2) the end-to-end framework level. Experiments are conducted on three NVIDIA GPUs: (1) A10G, (2) L4, and (3) L40S. For kernel-level analysis, we use *NVIDIA Nsight Compute* [45] to collect detailed hardware metrics, including instruction throughput, memory efficiency, and shared-memory conflicts. For end-to-end evaluation, each inference is executed 100 times after 10 warm-up iterations, and the mean wall-clock latency is reported. We compare SPDP against state-of-the-art sparse inference frameworks, including `SpInfer`, integrated into the Hugging Face Transformers library [25]. These systems primarily accelerate the decoding phase. We therefore report *Time Per Output Token* (TPOT) as our main latency metric, isolating GPU-side latency using *NVIDIA Nsight Systems* [46] to exclude CPU overhead.

For the model used during evaluation, we use `Llama-2-7B-hf` with Wanda [58] as the unstructured SP method and TEAL [35] as DP method. The SP method is calibrated on 128 samples of C4 dataset [54], while the DP method is calibrated on 300 samples of Alpaca dataset [59] to build a layer-wise empirical cumulative distribution function (ECDF) of activation magnitudes. Given a target dynamic sparsity ratio, we set a layer-specific threshold by taking the corresponding ECDF quantile, following TEAL's uniform sparsity setting. The calibration is performed offline once per model; therefore, the threshold selection process does not introduce any runtime overhead during inference. Note that although our experiments are performed on NVIDIA GPUs, the design principles of SPDP generalize to other accelerators such as Intel CPUs and NPUs with custom matrix-vector units, providing insights for future sparsity-aware kernel co-design across heterogeneous hardware.

5.2 Model Quality under Downstream Tasks

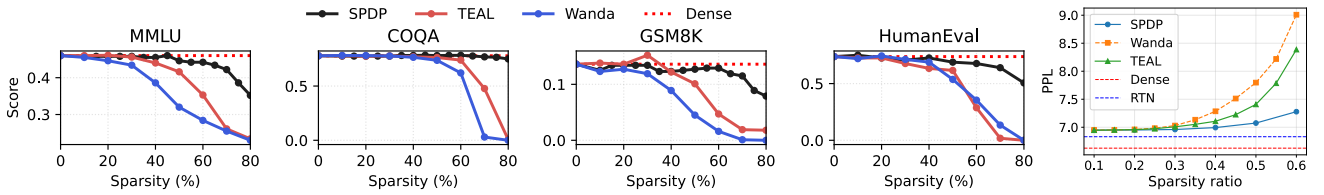
Model quality under downstream tasks. Figure 6a summarizes downstream-task accuracy across diverse workloads. At a moderate sparsity level (e.g., 40%), SPDP matches or closely tracks the baselines across tasks. As sparsity increases to a practically relevant regime for acceleration, SPDP consistently degrades less than DP-only (TEAL) and SP-only (Wanda), indicating that unified static-dynamic pruning preserves model quality more robustly across task types while enabling higher effective sparsity.

Compatibility with Quantization. Figure 6b reports perplexity under 4-bit RTN quantization on `Qwen3-32B`. We observe that the perplexity trend remains stable across sparsity levels, suggesting that SPDP is compatible with weight-only quantization and can potentially stack compression gains without a significant additional perplexity increase. Moreover, combining sparsity with 4-bit weights further reduces model footprint and weight-bandwidth demand, creating additional opportunities for decode-phase TPOT speedups when paired with quantized inference kernels.

5.3 Kernel Performance Analysis

Sparsity-regime characterization. Figure 7 characterizes the decode-kernel performance envelope by sweeping the SP ratio from 0% to 90%, while fixing the DP ratio to 30%. At low static sparsity, the benefit of static compression is limited, although SPDP can still exploit the DP by skipping activation-inactive columns. As static sparsity increases into the moderate regime targeted by LLM pruning, static weight compression and DP jointly reduce memory traffic, while the CUDA-core GEMV path preserves regular register-level accumulation. At very high static sparsity, some sparse-domain baselines become more competitive because they operate directly on very small nonzero counts, whereas Tiled-CBC still pays bitmap, `ColInfo`, and tile-level decoding overheads. Such high sparsity is less relevant to LLM inference, as model quality constraints limit aggressive unstructured pruning.

We next evaluate practical LLM operating points across weight matrix shapes, where static sparsity lies in the moderate range and DP provides additional input-dependent column skipping.



(a) Accuracy on downstream tasks

(b) Compatibility with quantization

Figure 6: Accuracy on downstream tasks and compatibility with quantization.

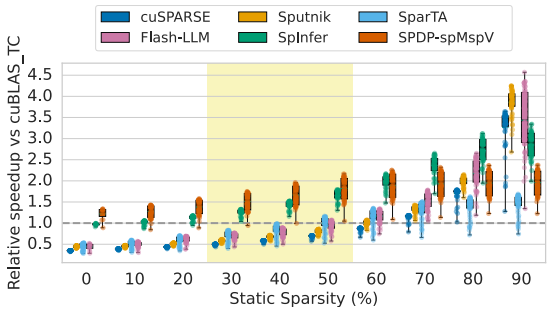
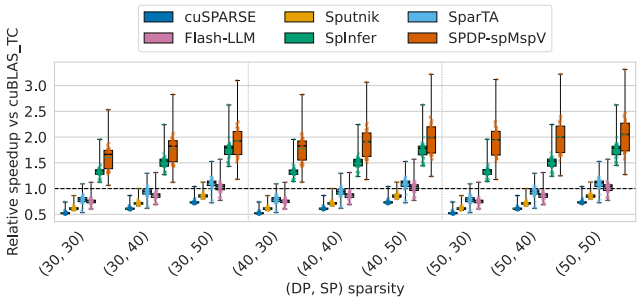
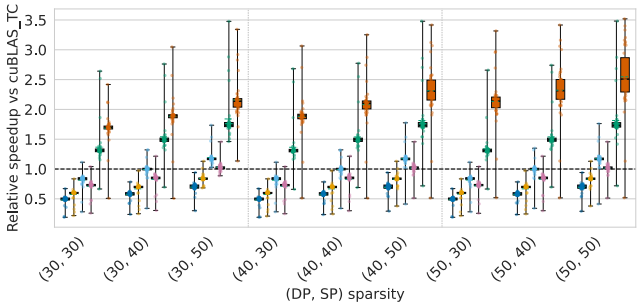


Figure 7: Kernel speedup on A10G GPU over cuBLAS_TC under a static sparsity sweep from 0% to 90%, with DP fixed at 30%. The shaded region marks the moderate-sparsity regime targeted by LLM inference.



(a) Kernel performance on A10G GPU



(b) Kernel performance on a L4 GPU

Figure 8: Kernel-level performance comparison of the SPDP-spMspV kernel on A10G and L4 GPUs. All speedups are normalized to cuBLAS_TC.

LLM projection matrix. We compare the SPDP-spMspV kernel against six baselines: (1) Tensor-Core-based cuBLAS [43], which serves as the dense baseline, (2) cuSPARSE [44], general sparse kernel for diverse sparse application, (3) Sputnik [19], (4) SparTA [73], which targets general deep learning sparse kernel, (5) Flash-LLM [62] (6) SpInfer [14], a state-of-the-art Tensor-Core-based sparse kernel optimized for moderate sparsity. Following prior work [14, 62], we focus on the practical sparsity range of 30–50% for both unstructured SP and DP, which represents the regime most applicable to large-scale LLMs.

Note that the kernel is evaluated using the same set of weight matrix shapes adopted in SpInfer [14], derived from major LLM architectures including OPT (13B–175B) [69], LLaMA-2 (7B–70B) [60], LLaMA-3 (8B, 70B) [20], Qwen (7B, 14B) [5], Qwen2 (7B, 72B) [63], and Mixtral-8×7B MoE [27], Phi-2, 3 (Mini, Medium) [1], gpt-oss (20B, 120B) [3]. These matrices capture representative dimensions of projection and feed-forward layers in modern Transformer blocks.

Figure 8 reports the kernel throughput (TFLOP/s) normalized to the Tensor-Core-based cuBLAS baseline, indicated by the blue dashed line. Across all GPU platforms, SPDP-spMspV consistently outperforms both dense and sparse baselines. On the A10G GPU, it achieves an average speedup of 1.88× over cuBLAS and 1.24× over SpInfer, with a maximum of 3.32× and 1.70×, respectively. Similar trends are observed on the L4 GPU, where average speedups reach 2.11× and 1.37×, and maximum improvements reach 3.52× and 2.52× over cuBLAS and SpInfer, respectively. The L40S results, used for end-to-end evaluation (Section 5.5), show the same relative ordering, confirming the consistency of the kernel's scalability.

These results highlight the effectiveness of SPDP's format–kernel co-design. The spMspV kernel efficiently exploits both static and dynamic sparsity through the *Tiled-CBC* layout, achieving high compute intensity and minimizing index overhead. By aligning memory access with column-wise activation sparsity, it maintains coalesced data movement and reduces metadata traffic, which are dominant factors in GPU-bound workloads.

Notably, DP provides an additional efficiency gain by skipping inactive neurons at runtime, improving both computational density and memory locality. This adaptivity enables SPDP-spMspV to outperform SpInfer, which is limited to static sparsity. Consequently, SPDP bridges the gap between static compression and activation-dependent sparsity, achieving high throughput and scalability across diverse GPU architectures.

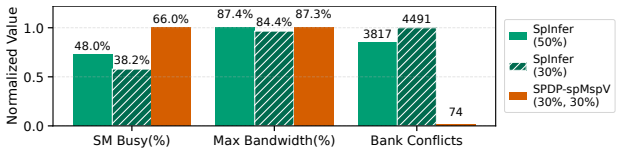


Figure 9: Kernel profiling results on the A10G GPU. SPDP-spMspV uses (DP, SP) = (30%, 30%), compared with SpInfer (30%, same SP) and SpInfer (50%, same total sparsity). All metrics are normalized to the highest value.

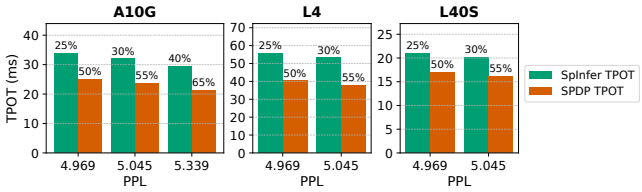


Figure 10: Perplexity vs. TPOT on A10G, L4, and L40S GPUs. SPDP attains lower TPOT with a 25% sparsity ratio difference, suggesting improved memory efficiency.

5.4 Kernel Hardware Analysis

Figure 9 profiles GPU utilization (*SM busy*, *Max bandwidth*, and *Bank conflicts*) for SPDP-spMspV with (DP, SP) = (30%, 30%) and SpInfer with SP = 30% and 50%. *SM busy* is the fraction of cycles where any SM unit is active (preferred over Instruction Per Cycle (IPC) since Tensor Core mma instructions naturally yield lower IPC), *Max bandwidth* is achieved DRAM bandwidth normalized to peak, and *Bank conflicts* captures serialized shared-memory transactions.

SPDP-spMspV attains the highest *SM busy*, indicating that combining SP and DP better sustains compute–memory overlap by skipping redundant activation reads; SpInfer(30%) is lower without DP, and SpInfer(50%) improves only modestly, highlighting the benefit of runtime adaptivity. All three kernels achieve comparable Max bandwidth, with SPDP-spMspV slightly lower than both SpInfer(50%) and SpInfer(30%). This phenomenon indicates that decompression and DP introduce negligible memory-stall overhead while preserving effective computation–memory overlap. Finally, SPDP-spMspV achieves the lowest *Bank conflicts* thanks to the Tiled-CBC column-aligned tiling, which yields more regular shared-memory accesses than SpInfer's SMBD under mixed sparsity. Overall, these hardware-level efficiencies explain the throughput advantage of SPDP over dense and static-sparse baselines.

5.5 End-to-end LLM Inference Evaluation

Sparsity Ratio vs. TPOT Speedup. Figure 11 shows the end-to-end *Time Per Output Token (TPOT)* speedup of the Llama-2-7B-hf model across varying sparsity ratios and GPUs. For each configuration, SPDP determines the optimal combination of SP and DP that minimizes perplexity. For instance, the best configuration at 65% total sparsity uses 30% SP and 50% DP.

As sparsity increases, both frameworks show diminishing TPOT speedups accompanied by rising perplexity. Perplexity escalates sharply beyond 70% static sparsity, where essential model weights are excessively pruned. Unlike kernel-level evaluations, this experiment measures end-to-end decode latency—including attention, normalization, and other operations. Many of these components are not sparsity-sensitive, so overall acceleration follows an Amdahl-style upper bound: even large speedups in sparse kernels translate to modest end-to-end gains. Decoding workloads are also primarily memory-bound; latency is dominated by data movement rather than computation. Thus, increasing sparsity improves arithmetic efficiency but only moderately reduces latency, as performance saturates under limited memory bandwidth. Despite these constraints, SPDP achieves higher TPOT speedup (average ×1.34) than SpInfer at comparable and moderate sparsity, while maintaining lower perplexity (under 5.4). This demonstrates that integrating DP enables finer-grained adaptivity to input-dependent redundancy, yielding a better speed–quality trade-off than static sparsity alone.

Perplexity vs. TPOT Speedup. While Figure 11 captures the aggregate impact of sparsity on performance and quality, it does not isolate the effect of DP under matched model quality. To ensure a fair comparison, we analyze TPOT speedup as a function of perplexity at equivalent accuracy levels.

As shown in Figure 10, at the same perplexity, SPDP attains substantially higher overall sparsity—up to 25% more—than SpInfer (Wanda-based SP), while achieving comparable or lower TPOT. This indicates that DP effectively exploits activation-dependent sparsity without compromising model fidelity. Although additional sparsity does not drastically reduce latency due to the memory-bound nature of decoding, it meaningfully decreases data movement, thereby improving *energy efficiency*. By transferring and processing fewer bytes per token, SPDP delivers higher performance-per-watt and better scalability in large-scale serving scenarios.

Overall, these results demonstrate that the synergy of static and dynamic pruning in SPDP not only reduces token-generation latency but also enhances energy efficiency and quality preservation under realistic, memory-limited inference conditions.

6 DISCUSSION AND FUTURE WORK

System-level generalization. SPDP is built around an execution model that combines unstructured SP with threshold-based DP in memory-bound decode. Many DP methods rely on threshold-based selection [16, 31], and can reuse SPDP's column-wise skipping with minor interface changes (e.g., how thresholds or per-token scores are produced). DP variants that materialize explicit masks/bitmaps can also be accommodated by mapping them to the same column-selection interface.

Prefill typically has a large output width ($N = B \times L$), which increases CI and often shifts the workload toward a compute-bound regime; as a result, LSCD-style sparse prefill does not consistently outperform dense cuBLAS_TC, and decoding/layout-alignment overheads become more visible, as shown in Figure 12. Accordingly, SPDP's prefill path mainly prioritizes *unified-format compatibility* (reusing a single compressed representation across prefill and decode without reformatting) and *memory-footprint reduction*, while the primary latency benefit comes from the bandwidth-bound decode stage. In a prefill–decode disaggregated deployment [53, 74], a practical option is to run dense GEMM for prefill and use SPDP-compressed weights only for decode.

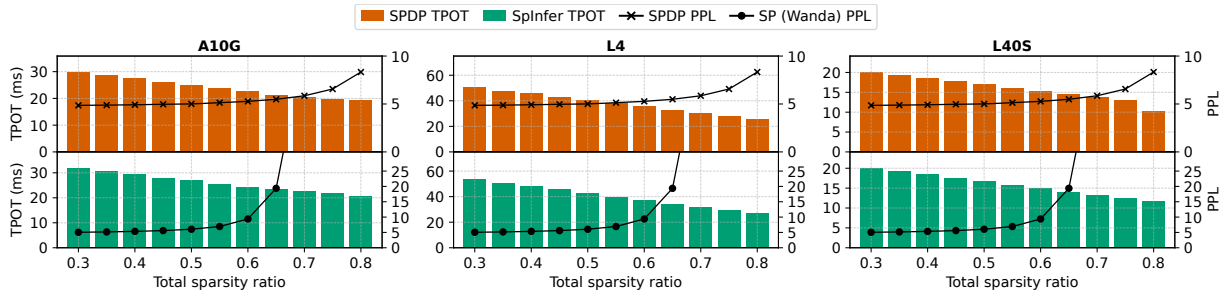


Figure 11: Sparsity ratio vs. TPOT speedup across different GPU architectures. SPDP reduces TPOT while maintaining reasonable perplexity, whereas SpInfer diverges due to the inherent limitation of SP. Beyond 65% sparsity, Wanda's perplexity diverges drastically, rendering SpInfer impractical in this regime.

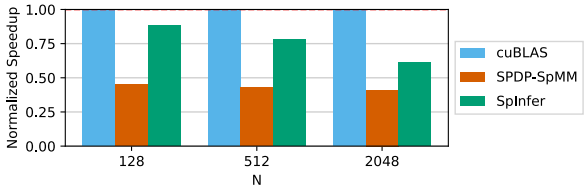


Figure 12: Prefill kernel comparison on A10G under SP=30%. With large N, existing works fail to beat the cuBLAS baseline.

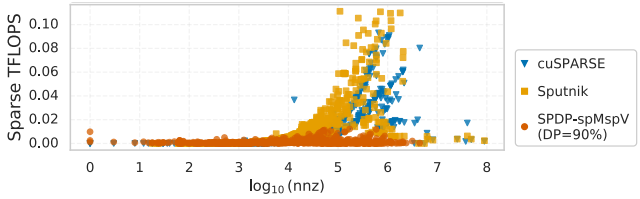


Figure 13: SuiteSparse evaluation. SPDP targets moderate-sparsity LLM decode workloads, whereas many SuiteSparse matrices are extremely sparse.

Integrating SPDP into modern LLM serving engines (e.g., continuous batching or fused prefill–decode) is largely orthogonal to attention execution, since SPDP mainly replaces the projection matmul kernels while leaving attention kernels unchanged.

SPDP kernels and the Tiled-CBC format are compatible with quantization, since `Values` are stored separately from sparsity metadata. However, quantization and unstructured sparsity do not provide multiplicative compression in practice: low-bit precision reduces only `Values`, while `Bitmap`, `ColInfo`, `TileOffset`, and padding remain. As a result, metadata can occupy a large fraction of storage at low precision, making such stacking beneficial mainly at very high sparsity. Exploring coarser-grained static sparsity, such as block or semi-structured patterns, to better amortize metadata under quantization remains future work.

Workload scope and sparsity regime. To clarify whether SPDP is a general-purpose sparse linear algebra kernel, we additionally evaluate non-LLM sparse matrices from the SuiteSparse Collection [10]. Figure 13 compares SPDP-`spMspV` against cuSPARSE and Sputnik on sparse matrix-vector workloads. SPDP is not consistently faster on these matrices. This is expected: many SuiteSparse matrices operate in an extreme-sparsity regime, with roughly 85% of evaluated matrices exceeding 95% sparsity, where CSR/CSC-style sparse-domain execution avoids reconstructing dense tiles and better amortizes index overhead. In contrast, SPDP targets the moderate-sparsity regime typical of LLM pruning, where model quality constraints prevent extreme sparsity and regular tile-level execution remains beneficial. Thus, SPDP should be viewed as a specialized LLM-decode kernel rather than a replacement for general sparse linear algebra libraries.

7 CONCLUSION

We presented SPDP, a unified sparse-inference framework that exploits unstructured static pruning (SP) and input-dependent dynamic pruning (DP). At the core of SPDP is the Tiled-CBC format, which enables runtime sparse-fragment access for decode while remaining reusable for prefill. Built on this representation, SPDP provides a CUDA-core `spMspV` kernel for decode and a Tensor-Core `SpMM` kernel for prefill, allowing a single compressed format to support both phases of LLM inference. Our evaluations show that SPDP improves kernel throughput, TPOT, and power efficiency over prior sparse baselines while maintaining model quality. Future work includes extending SPDP to batched dynamic pruning, production serving integration, and multi-GPU sparsity-aware scheduling.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF; RS-2025-00560762 and RS-2024-00414981), the Institute of Information & Communications Technology Planning & Evaluation (IITP; RS-2024-00454666, RS-2025-25442338, RS-2021-II211343, RS-2026-25551943, RS-2026-25518317, and RS-2024-00397085), the Ministry of Health and Welfare under the ARPA-H BAYS Project (RS-2025-25455095), the Ministry of Science and ICT under the Advanced GPU Utilization Support Program, and Samsung Electronics Co., Ltd. (IO251215-14535-01). The authors thank Samsung MAX Lab for providing research infrastructure. J. Do is affiliated with the Automation and Systems Research Institute (ASRI), Seoul National University (SNU).

REFERENCES

[1] Marah Abdin et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[2] Deepak Bhaskar Acharya, Karthigeyan Kuppan, and B. Divya. 2025. Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey. *IEEE Access* 13 (2025), 18912–18936. https://doi.org/10.1109/ACCESS.2025.3532853

[3] Sandhini Agarwal et al. 2025. gpt-oss-120b & gpt-oss-20b Model Card. arXiv:2508.10925 [cs.CL] https://arxiv.org/abs/2508.10925

[4] Yongqi An, Xu Zhao, Tao Yu, Ming Tang, and Jinqiao Wang. 2024. Fluctuation-based adaptive structured pruning for large language models. In *AAAI (AAAI'24/IAAI'24/EAAI'24)*. AAAI Press, Article 1212, 9 pages. https://doi.org/10.1609/aaai.v38i10.28960

[5] Jinze Bai et al. 2023. Qwen Technical Report. arXiv:2309.16609 [cs.CL] https://arxiv.org/abs/2309.16609

[6] Hritik Bansal, Karthik Gopalakrishnan, Saket Dingliwal, Sravan Bodapati, Katrin Kirchhoff, and Dan Roth. 2023. Rethinking the Role of Scale for In-Context Learning: An Interpretability-based Case Study at 66 Billion Scale. In *ACL*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 11833–11856. https://doi.org/10.18653/v1/2023.acl-long.660

[7] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *ArXiv* abs/2107.03374 (2021).

[8] Yuli Chen, Bo Cheng, Jiale Han, Yingying Zhang, Yingting Li, and Shuhao Zhang. 2025. DLP: Dynamic Layerwise Pruning in Large Language Models. In *ICML*.

[9] Karl Cobbe, Vineet Kosaraju, Mo Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *ArXiv* abs/2110.14168 (2021).

[10] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[11] Harry Dong, Beidi Chen, and Yuejie Chi. 2024. Prompt-prompted Adaptive Structured Pruning for Efficient LLM Generation. In *CoLM*.

[12] Zhen Du, Ying Liu, Ninghui Sun, Huimin Cui, Xiaobing Feng, and Jiajia Li. 2025. SRSparse: Generating Codes for High-Performance Sparse Matrix-Vector Semiring Computations. *ACM Trans. Archit. Code Optim.* 22, 2, Article 69 (July 2025), 26 pages. https://doi.org/10.1145/3722114

[13] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *ASPLOS* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/3620666.3651378

[14] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *EuroSys* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 243–260. https://doi.org/10.1145/3689031.3717481

[15] Gongfan Fang, Hongxu Yin, Saurav Muralidharan, Greg Heinrich, Jeff Pool, Jan Kautz, Pavlo Molchanov, and Xinchao Wang. 2025. MaskLLM: learnable semi-structured sparsity for large language models. In *NeurIPS* (Vancouver, BC, Canada) *(NeurIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 248, 23 pages.

[16] Marco Federici, Davide Belli, Mart Van Baalen, Amir Jalalirad, Andrii Skliar, Bence Major, Markus Nagel, and Paul Whatmough. 2025. Efficient LLM Inference using Dynamic Input Pruning and Cache-Aware Masking. In *MLSys*.

[17] Jared Fernandez, Clara Na, Vashisth Tiwari, Yonatan Bisk, Sasha Luccioni, and Emma Strubell. 2025. Energy Considerations of Large Language Model Inference and Efficiency Optimizations. arXiv:2504.17674 [cs.CL] https://arxiv.org/abs/2504.17674

[18] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *ICML (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 10323–10337. https://proceedings.mlr.press/v202/frantar23a.html

[19] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC*.

[20] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[21] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[22] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both weights and connections for efficient neural networks. In *NeurIPS* (Montreal, Canada) *(NeurIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[23] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. In *ICLR*.

[24] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph (Seffi) Naor, and Daniel Soudry. 2021. Accelerated sparse neural training: a provable and efficient method to find N:M transposable masks. In *NeurIPS (NeurIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 1614, 13 pages.

[25] Hugging Face. n.d.. Huggingface Transformers. Retrieved 2025-10-29 from https://huggingface.co/docs/transformers/index

[26] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[27] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[28] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). https://doi.org/10.1145/3747588 Just Accepted.

[29] Donghyeon Joo, Helya Hosseini, Ramyad Hadidi, and Bahar Asgari. 2025. Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core to Advocate Unstructured Sparsity in Efficient LLM Inference. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 232–245. https://doi.org/10.1145/3725843.3756065

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[31] Donghyun Lee, Jaeyong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. 2024. CATS: Context-Aware Thresholding for Sparsity in Large Language Models. In *CoLM*.

[32] Min Li, Yulong Ao, and Chao Yang. 2020. Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1842–1853.

[33] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Yunjun Gao, and Yaofeng Tu. 2025. HC-SpMM: Accelerating Sparse Matrix-Matrix Multiplication for Graphs with Hybrid GPU Cores . In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 501–514. https://doi.org/10.1109/ICDE65448.2025.00044

[34] Hongyi Liu, Rajarshi Saha, Zhen Jia, Youngsuk Park, Jiaji Huang, Shoham Sabach, Yu-Xiang Wang, and George Karypis. 2025. PROXSPARSE: REGULARIZED LEARNING OF SEMI-STRUCTURED SPARSITY MASKS FOR PRETRAINED LLMS. In *ICML*.

[35] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. 2025. Training-Free Activation Sparsity in Large Language Models. In *ICLR*.

[36] Lawrence Liu, Alexander Liu, Mengdi Wang, Tuo Zhao, and Lin F. Yang. 2026. ARMOR: High-Performance Semi-Structured Pruning via Adaptive Matrix Factorization. In *ICLR*.

[37] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In *ICML (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 22137–22176. https://proceedings.mlr.press/v202/liu23am.html

[38] Haiquan Lu, Yefan Zhou, Shiwei Liu, Zhangyang Wang, Michael W. Mahoney, and Yaoqing Yang. 2024. AlphaPruning: Using Heavy-Tailed Self Regularization Theory for Improved Layer-wise Pruning of Large Language Models. In *NeurIPS (NeurIPS '24)*.

[39] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-pruner: on the structural pruning of large language models. In *NeurIPS* (New Orleans, LA, USA) *(NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 950, 19 pages.

[40] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *ICLR*.

[41] Seyed Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. In *ICLR*.

[42] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).

[43] NVIDIA. 2023. *cuBLAS Documentation*. Retrieved 2025-10-29 from https://docs.nvidia.com/cuda/cublas/index.html

[44] NVIDIA. 2023. cuSPARSE Library. Retrieved 2025-10-29 from https://docs.nvidia.com/cuda/cusparse/index.html

[45] NVIDIA. 2024. Nsight Compute. Retrieved 2025-10-29 from https://developer.nvidia.com/nsight-compute

[46] NVIDIA. 2024. Nsight Systems. Retrieved 2025-10-29 from https://developer.nvidia.com/nsight-systems

[47] NVIDIA Corporation. 2021. *NVIDIA A10 Tensor Core GPU.* Technical Report. NVIDIA Corporation. Retrieved 2025-10-29 from https://www.nvidia.com/en-zz/data-center/products/a10-gpu/

[48] NVIDIA Corporation. 2022. *NVIDIA A100 Tensor Core GPU Architecture.* Technical Report. NVIDIA Corporation. Retrieved 2025-10-29 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[49] NVIDIA Corporation. 2023. *NVIDIA L4 Tensor Core GPU.* Technical Report. NVIDIA Corporation. Retrieved 2025-10-29 from https://www.nvidia.com/en-zz/data-center/l4/

[50] NVIDIA Corporation. 2023. *NVIDIA L40S.* Technical Report. NVIDIA Corporation. Retrieved 2025-10-29 from https://www.nvidia.com/en-zz/data-center/l40s/

[51] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High performance unstructured spmm computation using tensor cores. In *SC*. IEEE, 1–14.

[52] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *NeurIPS* (New Orleans, LA, USA) *(NeurIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2011, 15 pages.

[53] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *ISCA*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[54] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[55] Siva Reddy, Danqi Chen, and Christopher D. Manning. 2019. CoQA: A Conversational Question Answering Challenge. *Transactions of the Association for Computational Linguistics* 7 (2019), 249–266. https://doi.org/10.1162/tacl_a_00266

[56] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS (NeurIPS '23)*.

[57] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, and Maosong Sun. 2025. ProSparse: Introducing and Enhancing Intrinsic Activation Sparsity within Large Language Models. In *COLING*, Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 2626–2644. https://aclanthology.org/2025.coling-main.180/

[58] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. In *ICLR*.

[59] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. Retrieved 2025-10-29 from https://github.com/tatsu-lab/stanford_alpaca

[60] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS* (Long Beach, California, USA) *(NeurIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[62] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 211–224. https://doi.org/10.14778/3626292.3626303

[63] An Yang et al. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[64] An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[65] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*.

[66] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *ASPLOS*. 660–678.

[67] Ruokai Yin, Yuhang Li, Donghyun Lee, and Priyadarshini Panda. 2025. DuoGPT: Training-free Dual Sparsity through Activation-aware Pruning in LLMs. arXiv:2506.20194 [cs.LG] https://arxiv.org/abs/2506.20194

[68] Haopeng Zhang, Philip S. Yu, and Jiawei Zhang. 2025. A Systematic Survey of Text Summarization: From Statistical Methods to Large Language Models. *ACM Comput. Surv.* 57, 11, Article 277 (June 2025), 41 pages. https://doi.org/10.1145/3731445

[69] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL] https://arxiv.org/abs/2205.01068

[70] Yingtao Zhang, Haoli Bai, Haokun Lin, Jialin Zhao, Lu Hou, and Carlo Vittorio Cannistraci. 2024. Plug-and-Play: An Efficient Post-training Pruning Method for Large Language Models. In *ICLR*.

[71] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, Yan Zeng, Yangang Wang, and Xuebin Chi. 2025. Acc-SpMM: Accelerating General-purpose Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. In *PPoPP* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 326–338. https://doi.org/10.1145/3710848.3710888

[72] Haizhong Zheng, Xiaoyan Bai, Xueshen Liu, Zhuoqing Mao, Beidi Chen, Fan Lai, and Atul Prakash. 2024. Learn To be Efficient: Build Structured Sparsity in Large Language Models. In *NeurIPS (NeurIPS '24)*.

[73] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *OSDI*. USENIX Association, Carlsbad, CA, 213–232. https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin

[74] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *OSDI*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[75] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch. In *ICLR*.