

RheoSparse: Exploring Finer Grained Structured Sparsity for Small Language Models

Jianing Zheng, Gang Chen*

School of Computer Science and Engineering, Sun Yat-sen University

Abstract—Small Language Models (SLMs) are designed for efficient on-device deployment, but compressing them without significant accuracy loss remains challenging. Structured sparsity methods like N:M pruning, which removes N parameters out of every M, are widely used to improve hardware efficiency. However, their rigid patterns, such as the commonly adopted 4:8 format, often degrade SLM performance, since these models are more sensitive to parameter removal than larger ones. We observe that finer-grained patterns like N:64 can better preserve accuracy under the same sparsity budget, yet current inference systems do not efficiently support them, especially during token generation. Furthermore, applying such fine-grained sparsity uniformly across all layers is suboptimal, as different layers respond differently to pruning. To address this, we propose RheoSparse. First, we use coarse-to-fine evolutionary search to assign sparsity levels across layers under a global budget. Second, we design a highly-optimized Sparse Matrix-Vector Multiplication (SpMV) kernel that efficiently supports arbitrary structured sparsity patterns during token generation. For example, on Qwen2.5-1.5B, RheoSparse reduces perplexity (PPL) by 33.09% and improves downstream task performance by 9.3% compared to 4:8 sparsity, while maintaining the same parameter count. Furthermore, our SpMV kernel outperforms the state-of-the-art sparse kernel SpInfer by up to 49.6%.

I. INTRODUCTION

Large language models (LLMs) have demonstrated exceptional performance across a wide range of natural language processing (NLP) tasks. However, their large parameter counts and high computational demands make them impractical for deployment in edge devices. In contrast, Small Language Models (SLMs) (e.g., Llama3.2-1B, OPT-1.3B and Qwen2.5-1.5B), offer a promising trade-off between model performance and efficiency, making them well-suited for on-device applications. These SLMs are often derived from larger models via knowledge distillation and task-specific fine-tuning, achieving strong performance in specialized domains. Despite these advances, even state-of-the-art SLMs (1–3B parameters) remain challenging to deploy on edge hardware due to limited memory capacity and bandwidth. Therefore, further optimization is essential to enable efficient inference on such platforms.

Model compression techniques, like quantization and sparsification, have been widely adopted to reduce the computational and memory overhead of LLMs while preserving model accuracy. However, when applied to SLMs, especially through structured sparsification methods, these techniques often lead to substantial performance degradation. As illustrated in Fig. 1(a), higher sparsity levels lead to a sharp increase

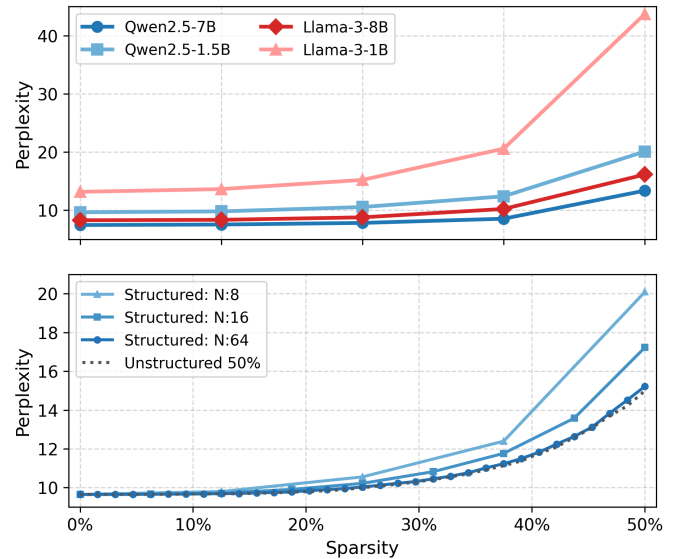


Fig. 1. Perplexity of language models under different structured sparsity patterns (lower is better). (a) Using N:8 structured pruning, smaller models experience higher perplexity. (b) In contrast, finer-grained patterns (e.g., N:64) consistently yield lower perplexity under the same sparsity budget.

in perplexity, especially for SLMs. This suggests that existing sparsification strategies are less effective for smaller models.

This performance degradation stems from both smaller parameter count and suboptimal sparse pattern design. Sparsification methods can be categorized into unstructured and structured pruning. While unstructured pruning typically achieves higher accuracy by removing individual weights based on importance, it introduces inefficiencies memory access patterns, limiting its practicality on modern hardware. Consequently, most recent work adopts structured N:M pruning, where every block of M parameters contains N non-zero elements. This approach aligns with contemporary GPU tensor cores that support fixed ratios such as 4:8. However, as shown in Fig. 1(b), rigid N:M patterns lack flexibility. For example, using a uniform 4:8 configuration yields higher perplexity than relaxed patterns like 8:16 or 32:64. These finer-grained configurations can achieve better accuracy under the same overall sparsity budget, suggesting that finer-grained structured sparsity can approach the performance of unstructured pruning while significantly improving efficiency in SLMs.

Recent studies [1], [2] have shown that different linear layers within a model contribute unequally to overall perfor-

*Corresponding author: Gang Chen, Email: cheng83@mail.sysu.edu.cn

mance. This observation motivates adaptive sparsity allocation, where less critical layers undergo more aggressive pruning while important layers retain more parameters. This globally-aware strategy can improve downstream task accuracy compared to uniform sparsity. However, this approach is often incompatible with standard tensor core operations. Although modern GPU tensor cores support sparse computation, they require a fixed proportion of non-zero elements: high sparsity ($> 50\%$) does not reduce tensor core execution time, while low sparsity ($< 50\%$) fails to activate sparse mode. Consequently, achieving both high accuracy with complex sparse patterns and hardware efficiency remains a challenging trade-off.

To address this, we propose RheoSparse, a framework that enables finer-grained structured pruning in SLMs while maintaining hardware efficiency. Our key insight comes from the asymmetric stages of auto-regressive language model inference. The prefill stage processes prompts in batch and benefits from tensor core acceleration. This is followed by the decode stage, which generates tokens one at a time. With the growing use of language models for long reasoning tasks that require continuous decoding, decode latency becomes a critical factor in overall inference time. With sequential token generation in the decode stage, the bottleneck shifts from compute-bound to memory bandwidth-bound. However, existing sparse GPU prefill kernels do not efficiently handle finer-grained structured sparsity during the decode stage.

Based on these observations, we make the following contributions:

- We present a coarse-to-fine sparse pattern search framework that allocates structured sparsity adaptively across layers under a global budget.
- We design a flexible SpMV kernel for the decode stage, efficiently supporting arbitrary structured sparsity patterns without relying on tensor cores.
- On Qwen2.5-1.5B, RheoSparse delivers a 33.09% perplexity reduction and a 9.3% improvement in downstream task performance over the widely adopted 4:8 sparsity baseline. In addition, our SpMV kernel reduces latency by up to 49.6% compared to the state-of-the-art GPU kernel SpInfer [3].

The code is available at <https://github.com/zjnyly/RheoSparse>.

II. RELATED WORK

A. Pruning Methods

Model pruning compresses neural networks by removing redundant weights or structures. However, directly applying pruning methods designed for LLMs to SLMs often yields suboptimal results. Existing LLM pruning approaches use saliency criteria to remove unimportant parameters, neurons, or channels. For example, magnitude-based pruning [4] enables fast compression but often sacrifices accuracy, while Pruner-Zero [5] and SparseGPT [6] leverage gradient or Hessian-based sensitivity to better preserve critical weights. Beyond weight pruning, recent works explore structured channel or layer level removal to improve hardware efficiency.

Methods such as SliceGPT [7], LLM-Pruner [8], and Týr-the-Pruner [9] perform channel-wise pruning to yield thinner models. However, such coarse-grained approaches often cause significant accuracy loss, especially in SLMs. In contrast, global pruning optimizes sparsity allocation across the entire model. ShortGPT [1] demonstrates that certain layers in LLMs can be safely removed without catastrophic degradation. EvoPress [10] and Týr-the-Pruner adopt evolutionary search to find optimal sparsity configurations across layers. However, EvoPress relies on unstructured pruning, which is inefficient on hardware, while Týr-the-Pruner sacrifices accuracy for throughput due to coarse-grained channel pruning. In contrast, RheoSparse introduces a globally optimized, fine-grained structured pruning framework tailored for SLMs. By preserving hardware-friendly sparsity patterns and enabling adaptive allocation across layers, RheoSparse achieves a superior accuracy-efficiency trade-off compared to existing approaches.

B. Sparse Matrix Multiplication Acceleration

Language model inference consists of two computational phases: the prefill stage, which uses SpMM to process input tokens in parallel, and the decode stage, which performs auto-regressive token generation via SpMV. cuBLAS¹ leverages NVIDIA’s tensor cores to accelerate structured sparsity patterns (e.g., 2:4 or 4:8). While effective for LLMs, these patterns can severely degrade SLM performance. Recent systems such as Flash-LLM [11] and SpInfer [3] support unstructured SpMM by dynamically reconstructing sparse weights into dense blocks during computation. The high computational intensity of the prefill stage allows weight reconstruction to be overlapped with execution. However, during the decode stage, performance becomes memory-bandwidth-bound rather than compute-bound. With the increasing prevalence of long-context reasoning, optimizing SpMV performance during decoding is critical. Unlike prior work that relies on unstructured sparsity, which can achieve higher accuracy, such approaches incur reconstruction and memory access overhead. In contrast, RheoSparse adopts fine-grained structured sparsity (e.g., N:64) to achieve accuracy comparable to unstructured pruning. Its structured memory layout also enables regular memory access patterns and reduces indexing overhead, resulting in improved end-to-end inference efficiency.

III. DISCOVERING FINE-GRAINED SPARSITY PATTERNS VIA EVOLUTIONARY SEARCH

A. Sparse Pattern Generation

Model compression is typically formulated as a layer-wise optimization problem, with the objective of minimizing the output discrepancy between the original and pruned linear layers. Specifically, we minimize the ℓ^2 -norm of the output difference, as formalized in Equation (1):

$$\arg \min_{M_l, \widehat{W}_l} \left\| W_l X_l - (M_l \circ \widehat{W}_l) X_l \right\|_2^2 \quad (1)$$

¹<https://docs.nvidia.com/cuda/cublas/>

where X_l the input to layer l , W_l the original weights, \widehat{W}_l the pruned weights, and M_l a binary pruning mask.

However, solving this joint optimization, even for SLMs, has very high computational complexity. Therefore, following OBS [12] and SparseGPT [6], we adopt an approximation strategy, where the pruning error ε_m and compensation δ_m for weight w_m are estimated using the inverse Hessian \mathbf{H}^{-1} :

$$\varepsilon_m = \frac{w_m^2}{[\mathbf{H}^{-1}]_{mm}}, \quad \delta_m = -\frac{w_m}{[\mathbf{H}^{-1}]_{mm}} \cdot \mathbf{H}_{:,m}^{-1}, \quad (2)$$

Here, we use $\mathbf{H} \approx X_{M_l}^\top X_{M_l}$, which serves as a Fisher information approximation of the Hessian. Since matrix–vector multiplication computes row-wise dot products, the inputs (activations) associated with each row are sufficient for acquiring ε_m and computing δ_m . To ensure hardware-friendly memory alignment while preserving accuracy, we adopt N:64 fine-grained structured pruning, removing the least important N weights within each 64-element block per row. The compensation term δ_m redistributes the effect of the pruned weight to the remaining ones, thereby mitigating error accumulation while maintaining fine-grained sparsity control.

B. Coarse-to-Fine Evolutionary Search

As observed in ShortGPT [1], different linear layers have varying importance to overall model performance. Applying uniform sparsity across all layers is therefore suboptimal, as some layers are more sensitive to pruning than others. EvoPress [10] demonstrates that evolutionary search significantly outperforms random search in finding optimal sparsity allocations under a global sparsity budget. Similarly, Týr-the-Pruner [9] adopts evolutionary search with a coarse-to-fine pruning strategy. Specifically, it iteratively searches 9 sparsity levels, starting with a large sparse search interval of 12.5% to cover the entire sparsity range. Sparsity levels are randomly exchanged between selected layer pairs while meeting the global budget constraint.

As shown in Fig. 1, perplexity follows a power-law relationship with sparsity: the small gain from decreasing sparsity (retaining more parameters) in one layer is outweighed by the loss from increasing sparsity (removing more parameters) in another. Therefore, starting the search with a large stride and a wide search space is unnecessary and can often be misleading for subsequent gene evolution in finer-grained searches. This phenomenon is particularly pronounced in layer-wise and channel-wise pruning methods, as observed in ShortGPT [1] and Týr-the-Pruner [9], compared to our element-wise pruning. To avoid poor initialization in evolutionary search, we employ a more efficient search space with only three sparsity levels per iteration i , starting from $i = 0$, with a step size of $6.25\%/2^{i-1}$ (corresponding to $[S_{\text{base}} \pm 4/2^{i-1} : 64]$ structured sparsity, where S_{base} denotes the base sparsity). We then encode the sparsity configuration of all linear layers as a gene sequence using the alphabet $\{-1, 0, +1\}$, where -1 decreases sparsity, 0 keeps it at S_{base} , and $+1$ increases sparsity. For example, Qwen2.5-1.5B contains 28 identical transformer blocks, each with 7 linear layers (abbreviated as

Q, K, V, O, Up, Gate, and Down projections). We generate 7 gene segments, each of length 28, representing the sparsity configurations for layers of the same type across transformer blocks. Mutation is performed by randomly selecting a gene pair within each segment and flipping them between $\{-1, +1\}$ and $\{0, 0\}$, preserving the global sparsity budget. This reduces the search space for Qwen2.5-1.5B to approximately 10^{13} , which is substantially smaller than the 10^{27} space explored by Týr-the-Pruner [9] and the 10^{50} full combinatorial space (for N:64 sparsity), enabling efficient evolutionary search while maintaining practically effective coverage.

Unlike Týr-the-Pruner, which enforces uniform sparsity within Attention (Q, K, V, O) and FFN (Up, Gate, and Down) groups to preserve channel alignment for each transformer layer, our fine-grained, element-wise pruning allows each layer to evolve independently. This layer-wise individuality significantly improves the performance of the searched model. After evaluating the fitness of each generation, we select the top-2 individuals as parents for crossover and mutation to produce the next generation. To avoid repeated local pruning during evolution, we pre-prune the weight matrices at sparsity levels $\{S_{\text{base}} + i \mid i \in [-7, 7]\}$. This enables rapid assembly of models according to any gene sequence.

C. Model Calibration via Knowledge Distillation

Since SLMs are deployed in multi-task scenarios, task-specific sparsity search may harm generalization. To ensure behavioral alignment between the pruned and dense models at the probabilistic level, we use Kullback-Leibler (KL) divergence over output logits as the fitness function for evaluation:

$$\hat{\theta} = \arg \min_{\theta} \text{KL} (p_{\text{dense}}(x) \parallel p_{\text{sparse}}(x; \theta)) \quad (3)$$

where $\hat{\theta}$ denotes the optimal sparsity configuration (individual), and $p_{\text{dense}}(x)$ and $p_{\text{sparse}}(x; \theta)$ represent the output probability distributions over the vocabulary for the dense and sparse models, respectively. A lower KL divergence indicates better preservation of the dense model’s behavior. We also experimented with minimizing the ℓ^2 -norm between hidden states across transformer blocks as an additional constraint, but found it reduced search efficiency. This likely arises because different blocks have varying sensitivity to pruning, making a global average score less informative.

IV. DESIGN OF RHEOSPARE SPMV KERNEL

A. Bitmask-Driven Index Recovery

Traditional sparse formats, such as CSR (Compressed Sparse Row), store column indices for each non-zero element. Tiled-CSR further organizes weights into small tiles, reducing the bitwidth for column indices within each tile. These formats are well-suited for scientific computing workloads with extreme sparsity. However, they become suboptimal for pruned neural networks, which typically operate at moderate sparsity levels (e.g., 50%) to preserve accuracy. As shown in Fig. 2, we define the compression ratio as the size of the sparse representation relative to the dense storage. At 50% sparsity,

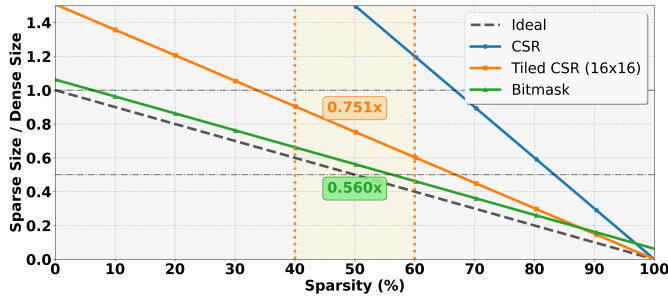


Fig. 2. Compression ratios of sparse formats under different sparsity levels.

CSR-like formats incur significant overhead, sometimes even exceeding the space required to store the dense matrix due to index storage. In contrast, bitmask-based representations achieve a compression ratio of 0.56 \times , closely approaching the ideal 0.5 \times case (ignoring storage cost for sparse metadata).

However, decoding sparse structures directly from bitmasks is non-trivial, as bitmasks do not provide explicit indices. To address this, as illustrated in Fig. 3(a) and Algorithm 1, we assign each GPU thread to compute the dot product between a row of sparse weights and the input vector. During execution, each thread loads a bitmask packed into a `uint64`, representing N:64 structured sparsity, and decodes the positions of non-zero elements using `__ffsll` (find first set) combined with bitwise operations. To mitigate the overhead of irregular global memory access during indexing, we pre-load the input vector into shared memory. This enables efficient broadcasting of values within a warp (a hardware scheduling unit of 32 threads) when threads access the same addresses, while allowing direct low-latency access for other cases.

Compared to existing LLM acceleration frameworks such as Flash-LLM [11] and SpInfer [3], which reconstruct dense tiles from unstructured sparse weights to leverage tensor cores during the prefill stage (dominated by SpMM), our method is specifically optimized for the decode stage. In this phase, SpMV rather than SpMM dominates, and computation is insufficient to hide memory and reconstruction overhead. Our approach eliminates reconstruction entirely by performing on-the-fly index decoding and direct computation, significantly accelerating the decode stage. Moreover, the use of fine-grained structured sparsity preserves downstream task accuracy while delivering substantial efficiency gains.

B. Vectorized Loading and Memory Coalescing

Fine-grained structured sparsity enables regular storage of packed weights, as shown in Fig. 3(b). However, the compressed layout must still maximize memory transaction sizes to fully utilize available bandwidth. Modern GPUs automatically coalesce memory accesses when threads within a warp request consecutive global memory addresses and merge multiple small transactions into larger ones. This mechanism is known as *memory coalescing*. To fully leverage this mechanism, we reorganize the logical layout of compressed weights, as illustrated in Fig. 3(c). Specifically, based on the 6-bit spar-

Algorithm 1: Bitmask-Guided Sparse Matrix-Vector Multiplication

```

Input : sparsity: 6-bit code selecting the pattern in N:64 structured
         sparsity
         packed_weight: compressed weight array (half)
         bitmask: array of uint64_t indicating non-zero positions
         activation: input vector (half)
Output: psum: partial dot-product result of current thread
1 local_mask  $\leftarrow$  bitmask for current tile;
2 local_weight  $\leftarrow$  packed_weight pointer for current tile;
3 local_act  $\leftarrow$  activation pointer for current tile;
4 for  $k \leftarrow 5$  downto 0 do
5   if (sparsity  $\gg k$ ) & 1 then
6      $N_{\text{half}} \leftarrow 2^{5-k}$  // chunk size: 32, 16, 8, 4, 2, 1
7      $I_{\text{iter}} \leftarrow N_{\text{half}}/L$  // L: elements per vector load
8     for  $i \leftarrow 0$  to  $I_{\text{iter}} - 1$  do
9       Load  $L$  half values into vec[0..L - 1];
10      for  $j \leftarrow 0$  to  $L - 1$  do
11        /* Get next non-zero position */
12         $\text{pos} \leftarrow \text{__ffsll}(\text{local\_mask}) - 1$ 
13        /* Clear the bit we just processed */
14         $\text{local\_mask} \leftarrow \text{local\_mask} \& (\text{local\_mask} - 1)$ 
15         $\text{psum} \leftarrow \text{psum} + \text{local\_act}[\text{pos}] \times \text{vec}[j]$ ;

```

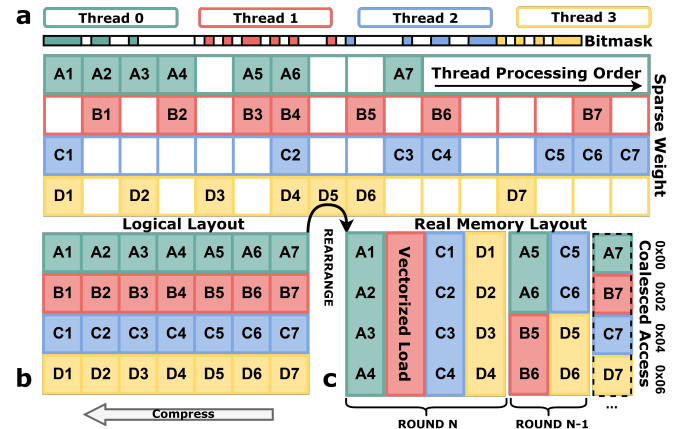


Fig. 3. Data rearrangement strategy for efficient SpMV. (a) Thread assignment and bitmask-guided weight processing. (b) Row-wise compression of non-zero elements after pruning. (c) Multi-granularity packing to enable coalesced memory accesses and maximize bandwidth.

sity pattern mask ($\{0,1\}^6$), we partition compressed weights (stored in `half`) into contiguous chunks of size 32, 16, 8, 4, 2, or 1. Larger chunks (32, 16, 8) are packed using vectorized types such as `float4` and `float2`, enabling efficient *vectorized loads*. The smallest tiles (size 1) are stored contiguously, allowing the hardware to automatically coalesce accesses. This layout ensures that when all threads in a warp access data simultaneously, memory requests are merged efficiently, maximizing bandwidth utilization.

V. EVALUATION

A. Experimental Settings

Models and Searching Strategies. We evaluate three widely adopted SLMs: LLaMA 3.2-1B-Instruct², OPT-

²<https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct>

TABLE I

PERFORMANCE OF DIFFERENT PRUNING STRATEGIES ACROSS MULTIPLE EVALUATION TASKS. AVG DENOTES THE AVERAGE PERFORMANCE ACROSS THE DOWNSTREAM TASKS (ARC-C, ARC-E, BOOLQ, HELLA SWAG, OBQA, PIQA, WINOGRANDE, MMLU)

Model	Method	PPL ↓	Arc-C	Arc-E	BoolQ	HellaSwag	OBQA	PIQA	Winogrande	MMLU	AVG ↑
llama-3 (1B)	dense	13.16	38.14	63.38	69.51	60.74	34.40	74.10	60.06	45.88	55.78
	unstructured (50%)	25.72	31.66	54.42	63.18	49.75	32.20	68.17	55.56	33.80	48.59
	4:8 prune	33.53	27.65	49.03	62.66	43.73	28.00	65.94	53.99	29.69	45.09
	32:64 prune	26.94	30.55	52.69	62.26	47.99	28.60	68.01	55.01	33.68	47.35
	RheoSparse	25.67	30.46	52.74	62.45	48.77	30.20	69.15	54.78	34.16	47.84
OPT (1.3B)	dense	14.62	29.52	51.01	57.74	53.70	33.40	72.36	59.51	24.94	47.77
	unstructured (50%)	18.08	26.88	44.19	62.20	46.92	28.20	67.95	56.90	24.12	44.67
	4:8 prune	19.94	25.60	43.39	61.71	44.68	30.20	67.03	58.25	24.16	44.38
	32:64 prune	17.78	25.77	43.22	62.51	46.76	29.00	68.01	57.30	23.47	44.51
	RheoSparse	18.51	26.54	44.44	62.60	47.54	29.20	68.82	57.62	24.53	45.16
Qwen2.5 (1.5B)	dense	9.65	46.67	76.09	78.10	68.20	40.80	75.45	63.22	60.18	63.59
	unstructured (50%)	14.99	38.73	67.47	74.98	59.27	35.00	72.03	62.98	48.14	57.33
	4:8 prune	20.03	32.17	61.74	68.71	52.52	31.80	70.78	59.35	35.18	51.53
	32:64 prune	15.22	37.71	65.78	74.22	57.46	35.20	71.49	61.25	45.12	56.03
	RheoSparse	15.05	38.05	65.57	73.12	57.72	35.40	71.82	62.43	46.55	56.33

TABLE II

COMPARISON OF LOCAL PRUNING BASELINES AND RHEOSPARSE.

Method	llama-3 (1B)		OPT (1.3B)		Qwen2.5 (1.5B)	
	PPL ↓	AVG ↑	PPL ↓	AVG ↑	PPL ↓	AVG ↑
Magnitude	250.86	35.7	125.43	39.05	126.35	38.11
Wanda	36.74	45.36	18.72	45.16	16.18	53.8
RheoSparse	25.67	47.84	18.51	45.16	15.05	56.33

1.3B³, and Qwen2.5-1.5B-Instruct⁴. Our method builds upon SparseGPT [6], leveraging its local pruning algorithms to generate multiple candidate sparsity patterns. We then assemble and search over these candidates to identify a near-optimal global sparsity configuration. Parallel mutation is applied within each gene segment, where each gene encodes the sparsity distribution of a particular type of linear layer.

Calibration Strategies. We adopt the UltraChat_200k dataset⁵ for both local pruning and alignment evaluation. In local pruning, we use real chat data to identify the sensitivity of parameters to the input. For alignment evaluation, we measure the similarity between sparse and dense models. For this purpose, we randomly select 2K samples and fix them throughout the experiments. Our evolutionary search follows a three-stage coarse-to-fine strategy, with each stage evaluating 128 individuals across two generations. In each generation, the top two individuals are selected for crossover and mutation to produce the next generation.

Evaluation Tasks. Following prior work [5], [6], [8], [9], we first evaluate perplexity on WikiText2 [13] to assess language modeling capability. We then report zero-shot accuracy on seven widely used downstream benchmarks: ARC (Easy/Hard) [14], BoolQ [15], HellaSwag [16], OpenBookQA [17], PIQA [18], WinoGrande [19], and MMLU [20]. Lower perplexity indicates better language modeling, while

higher accuracy reflects stronger downstream performance. We report downstream task accuracy using percentage values. Following the aforementioned works, we compute the average across all seven tasks to reflect overall performance.

GPU Kernel Baselines. We use cuBLAS, Nvidia’s officially optimized dense linear algebra library, as a strong baseline. For sparse matrix multiplication, we evaluate Flash-LLM [11] and SpInfer [3], which are optimized for unstructured sparsity during the prefill stage of LLM inference. While these methods already perform well in that stage, our RheoSparse SpMV kernel is specifically designed to accelerate the decoding stage, which is increasingly critical for long-sequence generation. All results are reported in half precision on an Nvidia RTX 3090, with each measurement including 10 warm-up runs and averaged over 200 executions.

B. Language Modeling and Downstream Task Performance

We evaluate SLMs using five methods. First, we apply SparseGPT [6] to perform pruning with three strategies: 50% unstructured pruning, 4:8 structured pruning, and 32:64 structured pruning. In addition, we include the dense model as a reference to illustrate the actual performance loss for each method. Since our focus is on structured pruning, we highlight these methods with colored backgrounds and bold text. As shown in Tab. I, RheoSparse consistently achieves higher average downstream accuracy across tasks. On LLaMA-3 (1B) and Qwen2.5 (1.5B), it also attains lower perplexity (PPL) compared to other pruning strategies. However, on OPT-1.3B, RheoSparse slightly underperforms 32:64 structured pruning in PPL. This discrepancy likely arises because our search evaluates candidate models based on KL-divergence alignment rather than task-specific metrics. Nonetheless, when considering both PPL and downstream accuracy, RheoSparse demonstrates stable and high overall performance. Moreover, it reaches the accuracy of 50% unstructured pruning while providing a structured format that offers significant latency benefits. For local pruning methods, we use SparseGPT as the

³<https://huggingface.co/facebook/opt-1.3b>

⁴<https://huggingface.co/Qwen/Qwen2.5-1.5B-Instruct>

⁵https://huggingface.co/datasets/HuggingFaceH4/ultrachat_200k

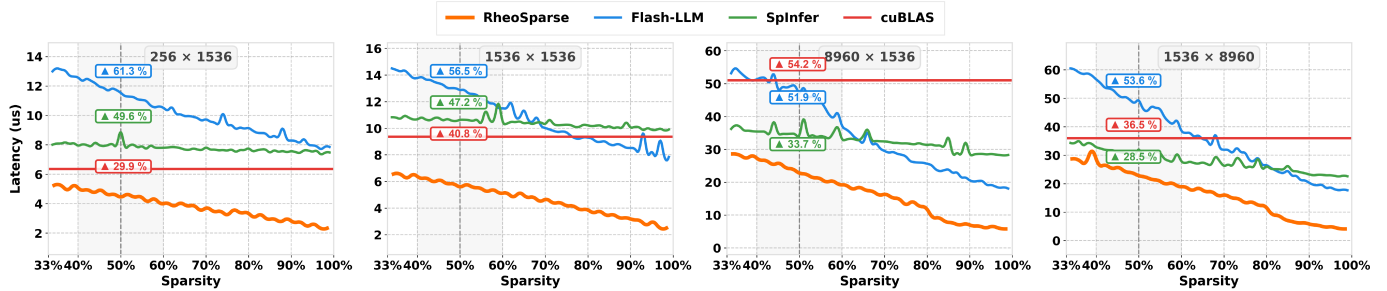


Fig. 4. Latency comparison of cuBLAS, Flash-LLM, SpInfer, and RheoSparse across different matrix dimensions and sparsity levels.

backbone. As shown in Tab. II, combining SparseGPT with RheoSparse search outperforms Wanda [21] and Magnitude pruning [4]. However, we also emphasize that although local pruning methods achieve near-lossless accuracy on LLMs they remain suboptimal on SLMs and need further improvement.

C. RheoSparse SpMV kernel Performance

General Kernel Performance. We benchmark RheoSparse against cuBLAS, Flash-LLM, and SpInfer using matrix dimensions from Qwen2.5 (1.5B): $\{256 \times 1536\}$ (Q and O layers), $\{1536 \times 1536\}$ (K and V), $\{1536 \times 8960\}$ (Up and Gate), and $\{8960 \times 1536\}$ (Down). In Fig. 4, the light-gray shading indicates the practical sparsity range that preserves SLM accuracy. cuBLAS (red) performs dense multiplication and serves as a fixed-latency baseline. Flash-LLM (blue) and SpInfer (green) perform well for large matrices but fall behind dense multiplication for smaller ones. RheoSparse (orange) maintains low latency across all matrix sizes and sparsities. The key difference is that Flash-LLM and SpInfer leverage tensor cores for matrix-matrix multiplication, requiring on-the-fly reconstruction of unstructured weights into structured tiles and balancing load-reconstruct-compute pipelines. In contrast, the decode stage is memory-bandwidth bound, so reconstruction overhead dominates latency. RheoSparse mitigates this bottleneck with fine-grained structured sparsity and an optimized data layout. While LLM serving systems such as vLLM [22] and SGLang [23] rely on chunked prefill to maximize tensor core throughput by merging decode requests into prefill, SLM scenarios such as personal assistants and long reasoning tasks prioritize low latency, which RheoSparse is specifically designed to achieve.

Latency Improvements. As shown in Fig. 5, we evaluate kernel performance under four optimization settings: (1) vanilla (yellow), without memory optimizations; (2) memory coalescing only (cyan); (3) RheoSparse kernel with memory coalescing and vectorized loading (red); and (4) the ideal case (gray), representing the latency of loading weights at the GPU’s peak memory bandwidth (Nvidia RTX 3090). Within the practical sparsity range (light blue shading), memory coalescing alone provides significant latency reduction. With vectorized loading in the RheoSparse kernel, weights are packed into vectorized datatypes. This further improves performance, especially at lower sparsity levels. Overall, RheoSparse closely approaches

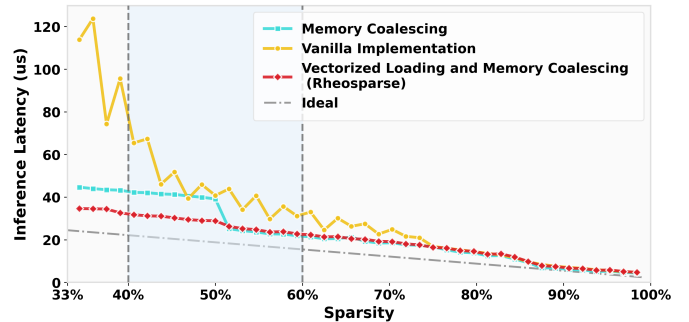


Fig. 5. Latency breakdown of RheoSparse kernel under different optimization strategies.

the ideal case, demonstrating its effectiveness in mitigating memory-bandwidth bottlenecks during the decoding stage.

VI. CONCLUSION

In this paper, we introduce RheoSparse, focusing on fine-grained structured sparsity in small language models. Experiments on Qwen2.5-1.5B show that, through a global budget-constrained coarse-to-fine evolutionary search, RheoSparse increases average downstream task accuracy by 9.3% compared to the widely adopted 4:8 sparsity baseline. Furthermore, the high-performance SpMV kernel we design reduces latency by up to 49.6% compared to the state-of-the-art fine-grained GPU sparse kernel SpInfer [3]. This work demonstrates the potential of fine-grained structured sparsity for practical on-device deployment of small language models.

ACKNOWLEDGMENT

This research was supported by the Guangzhou Basic and Applied Basic Research Foundation under Grant SL2024A04J0183, the Guangxi Key Research and Development Project under Grant GuikeAB25069495, the National Natural Science Foundation of China under Grant 92470202, and the Fund of National Key Laboratory of Multispectral Information Intelligent Processing Technology (No. 202410487201).

REFERENCES

- [1] Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853*, 2024.
- [2] Lujun Li et al. Discovering sparsity allocation for layer-wise pruning of large language models. *Advances in Neural Information Processing Systems, NeurIPS*, 37, 2024.
- [3] Ruibo Fan et al. Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus. In *Proceedings of the European Conference on Computer Systems, EuroSys*, 2025.
- [4] Song Han et al. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.
- [5] Peijie Dong et al. Pruner-zero: Evolving symbolic pruning metric from scratch for large language models. In *International Conference on Machine Learning, ICML*, 2024.
- [6] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning, ICML*, 2023.
- [7] Saleh Ashkboos et al. Slicegpt: Compress large language models by deleting rows and columns. In *The International Conference on Learning Representations, ICLR*. OpenReview.net, 2024.
- [8] Xinyin Ma et al. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems, NeurIPS*, 36:21702–21720, 2023.
- [9] Guanchen Li et al. Týr-the-pruner: Unlocking accurate 50% structural pruning for llms via global sparsity distribution optimization. *arXiv preprint arXiv:2503.09657*, 2025.
- [10] Oliver Sieberling et al. Evopress: Towards optimal dynamic model compression via evolutionary search. *arXiv preprint arXiv:2410.14649*, 2024.
- [11] Haojun Xia et al. Flash-llm: Enabling low-cost and highly-efficient large generative model inference with unstructured sparsity. *Proc. VLDB Endow.*, 17(2):211–224, 2023.
- [12] Babak Hassibi et al. Optimal brain surgeon and general network pruning. In *Proceedings of International Conference on Neural Networks, ICNN*, pages 293–299. IEEE, 1993.
- [13] Stephen Merity et al. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- [14] Peter Clark et al. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- [15] Christopher Clark et al. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.
- [16] Rowan Zellers et al. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- [17] Todor Mihaylov et al. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.
- [18] Yonatan Bisk et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, 2020.
- [19] 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.
- [20] Dan Hendrycks et al. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.
- [21] Mingjie Sun, Zhuang Liu, et al. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.
- [22] Woosuk Kwon et al. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the symposium on operating systems principles, SOSP*, pages 611–626, 2023.
- [23] Lianmin Zheng et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems, NeurIPS*, 2024.