SharePrefill: Accelerating Prefilling for Long-Context LLMs via Sparse Pattern Sharing

Anonymous ACL submission

Abstract

Sparse attention methods exploit the inherent sparsity in attention to speed up the prefilling phase of long-context inference, mitigating the quadratic complexity of full attention computation. While existing sparse attention methods rely on predefined patterns or inaccurate estimations to approximate attention behavior, they often fail to fully capture the true dynamics of attention, resulting in reduced efficiency and compromised accuracy. Instead, we propose a highly accurate sparse attention mechanism that shares similar yet precise attention patterns across heads, enabling a more realistic capture of the dynamic behavior of attention. Our approach is grounded in two key observations: (1) attention patterns demonstrate strong inter-head similarity, and (2) this similarity remains remarkably consistent across diverse inputs. By strategically sharing computed accurate patterns across attention heads, our method effectively captures actual patterns while requiring full attention computation for only a small subset of heads. Comprehensive evaluations demonstrate that our approach achieves superior or comparable speedup relative to state-of-the-art methods while delivering the best overall accuracy. The code will be made available upon publication.

1 Introduction

Long-context inference is essential for real-world applications of large language models (LLMs). Modern models like GPT-4.1 and Gemini 1.5 (Team et al., 2024) now support contexts up to one million tokens, advancing multi-document QA (Wang et al., 2024), code understanding (Bairi et al., 2024; Ziftci et al., 2025), and multi-turn dialogue (Zhang et al., 2025). Nonetheless, the prefilling phase of long-context inference remains time-consuming, as the vanilla attention mechanism entails quadratic computational complexity with respect to sequence length (Fu, 2024).

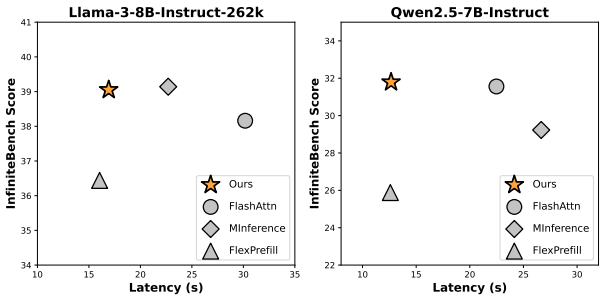


Figure 1: Comparison of our method with baselines across different models between latency (under 128K) and the average score on Infinitebench.

Sparse attention offers a promising solution by computing only significant attention scores, leveraging inherent sparsity in attention mechanisms. Many works have discovered various patterns with distinct characteristics and exploited them to perform sparse attention computations, such as the sink pattern in (Xiao et al., 2024), the A-shape, vertical-slash and block-sparse patterns in MInference (Jiang et al., 2024). However, these static patterns fail to generalize to varied inputs, as attention patterns inherently vary with different inputs as shown in Figure 2. To cope with the fundamental requirements of dynamic patterns, MInference dynamically adjusts the vertical-slash index, and FlexPrefill (Lai et al., 2025) further adapts the vertical-slash sparsity ratio dynamically and uses pooled queries and pooled keys to estimate query-aware block-wise patterns. However, we argue that pooling-based pattern estimation struggles to fully capture critical blocks due to the inaccuracies inherent in its approximations (detailed in Section 3).

Alternatively, we discover two interesting phenomena. Firstly, the sparse pattern of many attention heads tends to be highly similar. More importantly, the similarity relationships among these heads remain largely consistent, even though the sparse patterns themselves vary significantly across different inputs, as shown in Figure 2. Conse-

quently, we propose a highly accurate sparse attention mechanism that shares similar yet precise attention patterns across heads, mitigating reliance on predefined patterns and avoiding inaccurate pattern estimation. By computing dense attention using only a subset of heads, the prefilling is accelerated while preserving its high accuracy.

Our contributions are summarized as follows:

- We empirically demonstrate two fundamental properties of sparse attention patterns: (1) similarity across attention heads and (2) similarity consistency across different inputs.
- We propose SharePrefill, a novel highly accuracy-preserving sparse attention method to accelerate the prefilling phase by dynamically generating accurate sparse patterns and sharing them across heads.
- We conduct extensive experiments on several different models and tasks and show that our proposed method achieves superior or comparable speedup to state-of-the-art methods while achieving the best overall accuracy.

2 Related Work

Existing sparse attention methods for accelerating model inference can be categorized into two types: training-free sparse attention and training-based sparse attention. The former relies on predefined sparse patterns or pattern estimation, while the latter involves training sparse models to dynamically predict sparse patterns during inference.

Training-free Sparse Attention Several methods focus on predefined attention patterns, such as shifted sparse attention (Chen et al., 2024), sink attention (Xiao et al., 2024) and the A-shape, vertical-slash and block-sparse patterns used in MInference (Jiang et al., 2024). However, these patterns, often derived from limited cases, lack the flexibility to effectively adapt to varying input demands. MInference introduced partially dynamic patterns, by adjusting vertical-slash indexes based on inputs. FlexPrefill (Lai et al., 2025) adapts sparsity ratios via cumulative thresholds and incorporates query-aware sparse patterns to enhance flexibility. However, query-aware sparse patterns rely on pooled query and key representations for pattern selection, which may cause information loss and lead to less accurate pattern estimation. Our method aligns with this line of work but further enhances pattern modeling by dynamically providing more precise sparse patterns through pattern sharing, thereby achieving better accuracy preservation.

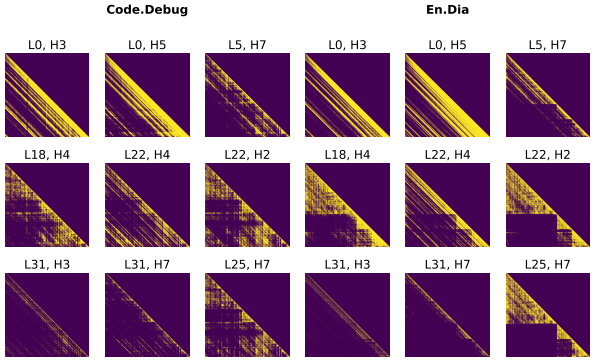
Training-based Sparse Attention Training-based sparse attention methods introduce attention gates, train the gate-associated network, and automatically predict important sequence segments during inference. In this series of works, approaches like MoBA (Lu et al., 2025) and NSA (Yuan et al., 2025) continue training the entire model, while SeerAttention (Gao et al., 2024) employs a linear layer as a learnable gate, training only the attention gate. Even though training-based sparse attention methods show promising acceleration while maintaining accuracy, the cost of resource-intensive and time-consuming training hinders their widespread practical applicability.

3 Static Patterns and Pooling-based Pattern Estimation are Not Enough

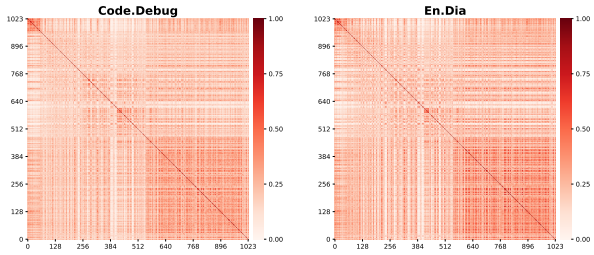
Attention patterns are highly dynamic, showing substantial variation both across different heads and within the same head under different inputs, as shown in Figure 2. In particular, the staircase-like patterns in *En.Dia* and the highly irregular patterns in *Code.Debug* deviate significantly from previously proposed static patterns like the vertical-slash pattern (Jiang et al., 2024). The highly dynamic nature intrinsic to attention mechanisms exposes the limitations of fixed-pattern approaches and underscores the need for adaptive, dynamic attention modeling techniques.

FlexPrefill (Lai et al., 2025) leverages pooled queries ($\boldsymbol{Q}$) and keys ($\boldsymbol{K}$) to estimate the average attention scores within each block for identifying critical regions, thus alleviating the reliance on predefined patterns. However, we highlight that this pooling-based method struggles to fully capture important blocks, and we identify that this challenge is rooted in two critical aspects.

Disregard for Token Alignment: The pooling operation disregards token-level position alignment within the query ($\boldsymbol{Q}$) and key ($\boldsymbol{K}$) segments, while attention mechanisms are inherently sensitive to token-level position alignment. This discrepancy leads to pooled results $pool(\boldsymbol{Q}) \cdot pool(\boldsymbol{K})$ that cannot accurately estimate the average of actual attention scores for the block. For example, consider two 1-dimensional $\boldsymbol{Q}$, $\boldsymbol{K}$ for 3 tokens: $\boldsymbol{Q}$=[0, 0, 1], $\boldsymbol{K}$=[0, 1, 0]. Due to ignoring position alignment, the $pool(\boldsymbol{Q}) \cdot pool(\boldsymbol{K})$ appears slightly significant($\frac{1}{9}$). However, all attention scores within



(a) Visualization of attention patterns for different heads across various tasks. Each group of three columns corresponds to the heads within a specific task.



(b) Similarity matrices show the pattern similarity between each head and other heads across different tasks.

Figure 2: Attention patterns of different heads and their similarity matrices across various tasks.

the block are actually zero, leading to an *overestimation* of the block's importance.

Smoothing of High-/Low- Values: The pooling operation smooths out high and low values within $\boldsymbol{Q}$ and $\boldsymbol{K}$, which often contribute to high and low attention scores, resulting in inaccurate importance estimation. For instance, $\boldsymbol{Q}$=[0, 0, 1], $\boldsymbol{K}$=[0, -1, 1]. During pooling, the the high-value and low-value elements in $\boldsymbol{Q}$ and $\boldsymbol{K}$ are diluted, resulting in $pool(\boldsymbol{Q}) \cdot pool(\boldsymbol{K}) = 0$, which is less than the actual average of attention scores $pool(\boldsymbol{Q} \cdot \boldsymbol{K}) = \frac{1}{9}$, leading to an *underestimation* of the importance of the block.

4 Observation: Dynamic Attention Heads Exhibit Similar Patterns and Static Similarity Relationships

We present the foundational observations motivating our method: different heads exhibit high similarity, and the similarity remains highly consistent across varying inputs and tasks. Specifically, these observations distill into two key properties:

(1) ***Inter-head*** *Pattern Similarity*: We observe many similar sparse patterns across attention heads, both within and between layers, as shown in Figure 2(a). These patterns are derived from the Llama-3-8B-Instruct-262k model using samples from different tasks in InfiniteBench (Zhang et al., 2024), with each group of three columns corresponding to heads from a specific task. For example, heads such as (L18, H4), (L22, H2), and (L25, H7) in the *En.Dia* task exhibit highly consistent staircase-like patterns, where L is the layer index and H is the head index. Additionally, Figure 2(b) shows the statistical similarity matrix based on Jaccard similarity scores (# intersection / # union) between one head and all others. This measure avoids artificially high similarity values that could arise from the presence of many zeros in these sparse patterns. Notably, a large number of similarity scores exceed 0.5, indicating that each head has many similar counterparts among the others.

(2) ***Cross-input*** *Similarity Consistency*: More importantly, the similarity among attention heads remains consistent regardless of the specific input or task, even though the pattern of a given attention head varies across different inputs and tasks. Figure 2(a) shows that (L18, H4), (L22, H2), and (L25, H7) are highly similar in *Code.Debug*, consistent with their previously observed similarity in *En.Dia* (Property 1), though their patterns differ between the two tasks. This highlights the consistent similarity among attention heads across inputs and tasks, suggesting that sparse patterns can potentially transfer to similar heads, regardless of context.

5 Proposed Approach

5.1 Problem Formulation

Generally, our goal is to replace dense attention with sparse attention in attention layers to reduce computational costs during the pre-filling phase, while minimizing the output loss of each attention layer, thereby accelerating the pre-filling process while preserving accuracy. This can be formulated as a multi-objective optimization problem:

$$\begin{aligned} &\min_{\boldsymbol{M}} |\mathcal{A}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V},\boldsymbol{M}) - \mathcal{A}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V})|, \\ &\min_{\boldsymbol{M}} |\boldsymbol{M}| \end{aligned} \tag{1}$$

where,

$$\mathcal{A}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V},\boldsymbol{M}) = \sigma\left(\frac{1}{\sqrt{d}}\boldsymbol{Q}\boldsymbol{K}^T - c(\boldsymbol{1}-\boldsymbol{M})\right)\boldsymbol{V},$$

$$\mathcal{A}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \sigma\left(\frac{1}{\sqrt{d}}\boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}$$

We define $\mathcal{A}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V},\boldsymbol{M})$ as our sparse attention, where $\boldsymbol{M}$ is a binary mask indicating

the sparse pattern applied in the sparse attention computation, where 1 means the block is computed, and 0 means it is skipped. The output is $\boldsymbol{O} = \mathcal{A}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{M})$ and a sufficiently large constant c ensures that attention score is approximately zero, whenever $\boldsymbol{M}_{ij} = 0$. Here, σ denotes the softmax function. The primary optimization objective is to identify a pattern $\boldsymbol{M}$ that minimizes the variance between the attention matrices of full attention $\mathcal{A}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$ and our proposed sparse attention $\mathcal{A}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{M})$, while also reducing the computational time required for sparse attention computation and sparse pattern generation.

5.2 Accelerating Prefilling via SharePrefill

Our main idea is to compute the full attention for a subset of heads, identify the actual sparse patterns, and share these patterns with other heads that are known to exhibit similar behavior. This approach enables the remaining heads to approximate the actual patterns without computing each one separately, thus maintaining the model's original accuracy while accelerating inference.

As depicted in the overview of SharePrefill in Figure 3, our approach involves two key components: (1) Offline clustering to group heads based on the similarity of their attention score maps. (2) Online inference, where pivotal attention is constructed dynamically and shared with other heads during the inference process. The overall algorithm of SharePrefill is detailed in Algorithm 1.

Offline Clustering of Similar Heads We cluster attention heads across layers into distinct groups based on the similarity of their attention score maps, performed in offline mode. The pre-computed clusters serve as the foundation for constructing and sharing sparse attention patterns within each cluster during inference.

Given the consistent similarity between heads, we perform clustering on their attention score maps using a sample from the *Retr.KV* task in InfiniteBench. We first obtain compressed low-dimensional representations of the attention scores by training an autoencoder network on these attention score maps (the network architecture is illustrated in Appendix C). Next, we normalize the representations and apply hierarchical clustering with a distance threshold to group similar heads into clusters, while isolating dissimilar heads as noise clusters. Notably, we only store the layer and head indices within clusters, rather than the sparse patterns themselves. The actual sparse patterns are dynamically generated during online inference, ensuring adaptability to varying inputs.

Dynamic Pattern Construction and Sharing during Inference During inference, we construct attention patterns dynamically and share adaptive yet accurate patterns among similar heads to facilitate sparse attention computation. This is achieved by computing the attention output for each layer while maintaining an evolving global pattern dictionary, which serves as the basis for sharing patterns across similar heads.

In general, the online inference process—based on dynamic pattern construction and sharing—comprises three key steps, corresponding to the three sub-algorithms outlined in Algorithm 1. The algorithm takes the query matrix $\boldsymbol{Q}$, key matrix $\boldsymbol{K}$, value matrix $\boldsymbol{V}$, similarity threshold τ, sparsity threshold δ and cumulative attention threshold γ as input. For simplicity, the illustration focuses on a single head rather than a layer. However, in practice, we perform sparse attention computation layer-by-layer.

Pivotal Pattern Sharing (See Algorithm 4): Before performing sparse attention computation, we first query the global pivotal pattern dictionary to check if a pivotal pattern is available for reuse. If a pivotal pattern exists, it is shared with the corresponding head; otherwise, the head computes full attention using a dense pattern (i.e., a pattern with all ones).

Sparse Attention Computation (See Algorithm 1): We then perform sparse attention computation to obtain the output for the current layer while simultaneously computing the block-wise average of QK values, denoted as $\tilde{\boldsymbol{A}}$, which captures the average QK scores within each block (line 8 in overall Algorithm 1). The sparse attention kernel is implemented in Triton (Tillet et al., 2019), following the block-wise strategy from FlashAttention 2 (Dao, 2024) and incorporating a block-wise sparse pattern to determine computation regions. Only blocks labeled as 1 in the sparse pattern are computed, while those labeled as 0 are skipped. During the computation for the final output, for each block where the pattern value is one, we compute the average QK value; for blocks where the pattern value is 0, we assign the average QK value as $-\infty$.

Pivotal Pattern Construction (See Algorithm 2): Subsequently, we use the obtained block-

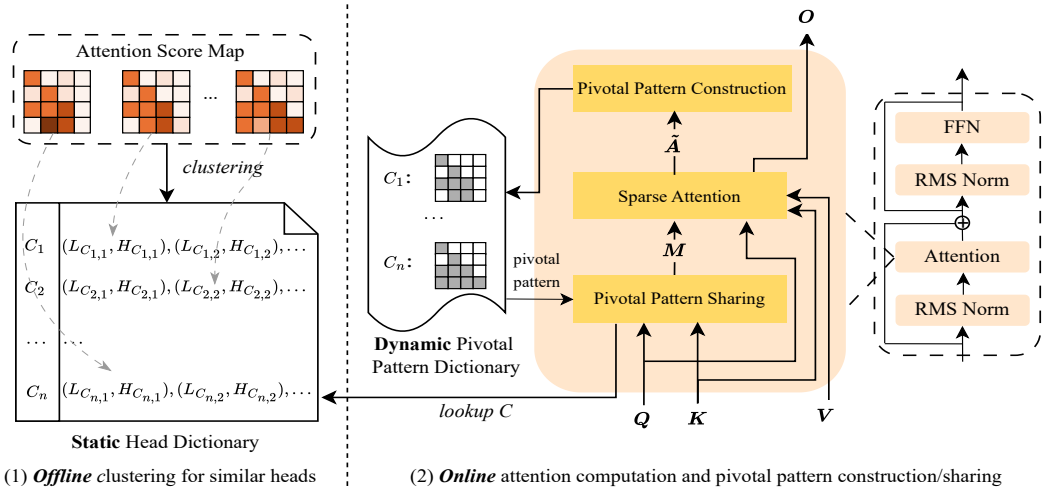


Figure 3: Overview of proposed SharePrefill. Attention heads are clustered offline based on the similarity of their attention score maps to create a static head dictionary. During inference, each head retrieves its cluster index C_i. Pivotal Patterns are shared if available; otherwise, a dense pattern is assigned. The sparse attention output $\boldsymbol{O}$ is computed using $\boldsymbol{M}$, and $\tilde{\boldsymbol{A}}$ updates the dynamic pivotal pattern dictionary.

wise average QK values to compute the block-wise average attention scores after applying softmax. These scores are then used to construct new pivotal patterns by applying a cumulative score threshold γ, which selects the minimal number of blocks required to cover the target cumulative attention score, as detailed in Algorithm 2. The resulting patterns are then updated into the pivotal pattern dictionary.

To ensure safe dynamic pattern sharing, we verify similarity before sharing patterns to prevent incorrect sharing that could adversely impact accuracy. Specifically, we compute the Jensen-Shannon (JS) distance between the block-wise average attention score of the last row block of the current head $\hat{\boldsymbol{a}}$ and the corresponding pivotal block-wise average attention score of the last row $\tilde{\boldsymbol{a}}$, which is also stored in the pivotal pattern dictionary (line 6 in Algorithm 3). This distance serves as a measure to predict the similarity between the current head and its corresponding pivotal head. If the JS distance is below a given similarity threshold τ, we share the pivotal pattern with the current head. Otherwise, we fall back to a conservative vertical-slash pattern (lines 7-11 in Algorithm 3) using a cumulative threshold-based vertical-slash pattern search algorithm (outlined in Algorithm 5), as proposed in FlexPrefill (Lai et al., 2025). Additionally, noisy clusters, which include dissimilar patterns, also revert to the vertical-slash pattern.

To enhance efficiency, we exclude highly sparse heads from the pivotal pattern construction and sharing process, as we consider that computing full attention for these heads to derive pivotal patterns is not cost-effective in terms of acceleration. For highly sparse heads, we instead fall back to searching for a vertical-slash pattern for each head (see line 10 in Algorithm 3), as the pattern often serves as a suitable approximation for highly sparse heads (Jiang et al., 2024). To identify these highly sparse heads, we compute the Jensen-Shannon (JS) distance between the block-wise average attention score of the last row block of the current head $\hat{\boldsymbol{a}}$ and a uniform distribution $\boldsymbol{u}$ (see line 6 in Algorithm 3). We then compare this distance to a predefined sparsity threshold δ. If the JS distance is not less than the threshold, we classify the head as a highly sparse head.

6 Experiments

6.1 Settings

This section outlines the models, datasets, baselines, and implementation details of our method in comparison with baseline methods. Additional information is provided in Appendix A.

Models, Datasets, and Baselines We employ two cutting-edge, renowned long-context LLMs: (i) Llama-3-8B-Instruct-262k (Pekelis et al., 2024), (ii) Qwen2.5-7B-Instruct (Team, 2024). The models are evaluated on InfiniteBench (Zhang et al., 2024), a state-of-the-art public benchmark de-

Algorithm 1 Sparse Attention

Input: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d_h}$; δ, τ, γ; l, h

 # Decide the pattern type based on $\boldsymbol{Q}$, $\boldsymbol{K}$, sparsity threshold δ and similarity threshold τ

$pattern \leftarrow$ Determine Sparse Pattern ($\boldsymbol{Q}$, $\boldsymbol{K}$, δ, τ)

 # Decide the sparse pattern $\boldsymbol{M}$ based on pattern, $\boldsymbol{Q}$, $\boldsymbol{K}$ and pattern threshold γ

if $pattern$ == shared_pivot **then**

  $\boldsymbol{M} \leftarrow$ Share Pivotal Pattern(l, h)

else if $pattern$ == vertical_slash **then**

  $\boldsymbol{M} \leftarrow$ Search Vertical Slash Pattern($\boldsymbol{Q}$, $\boldsymbol{K}$, γ)

end if

 # Compute the output $\boldsymbol{O}$ and block-averaged QK values $\tilde{\boldsymbol{A}}$ by applying sparse pattern $\boldsymbol{M}$

$\boldsymbol{O}, \tilde{\boldsymbol{A}} \leftarrow \mathcal{A}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{M})$

 # Construct and update global dynamic pivotal patterns via the newest block-averaged QK values $\tilde{\boldsymbol{A}}$ and pattern threshold γ

Construct Pivotal Pattern($\tilde{\boldsymbol{A}}$, γ, l, h)

return $\boldsymbol{O}$

Algorithm 2 Construct Pivotal Pattern

Input: $\tilde{\boldsymbol{A}}$; γ; l, h

if $\tilde{\boldsymbol{A}}$ is fully attention computed **then**

 # Compute block-averaged attention score $\tilde{\boldsymbol{A}}$ by applying softmax on block-averaged QK values

$\tilde{\boldsymbol{A}}$ = softmax($\tilde{\boldsymbol{A}}$)

 # Take the last row of $\tilde{\boldsymbol{A}}$ as pivotal representative

$\tilde{\boldsymbol{a}} \leftarrow \tilde{\boldsymbol{A}}_{[-1:]}$

 # Flatten and normalize attention score map

$\tilde{\boldsymbol{A}} \leftarrow$ flatten($\tilde{\boldsymbol{A}} / \sum_{i,j} \tilde{\boldsymbol{A}}[I, j]$)

 # Sort attention scores

$\boldsymbol{I_a} \leftarrow argsort(\tilde{\boldsymbol{A}})$

 # Obtain the minimum computational budget making the sum of the scores exceeds γ

$K \leftarrow$ min {k: $\sum_{i \in \boldsymbol{I_a}[1:k]} \tilde{\boldsymbol{A}}[i] \geq \gamma$ }

 # Select index set

$\boldsymbol{S} \leftarrow \boldsymbol{I_a}[1 : K]$

 # Convert index set $\boldsymbol{S}$ to mask pattern $\boldsymbol{M}$

$\boldsymbol{M} \leftarrow$ index_to_mask($\boldsymbol{S}$)

 # Lookup cluster index c in `head_dict`

c $\leftarrow$ lookup(l, h; `head_dict`)

 # Update $\boldsymbol{M}$ and $\tilde{\boldsymbol{a}}$ into `pivotal_pattern_dict`

`pivotal_pattern_dict`.update({c: ($\tilde{\boldsymbol{a}}$, $\boldsymbol{M}$)})

end if

Algorithm 3 Determine Sparse Pattern

Input: $\boldsymbol{Q}$, $\boldsymbol{K}$; δ, τ

 # Take a representative query subset

select $\hat{\boldsymbol{Q}} = \boldsymbol{Q}_{[-block\_size:]}$

 # Compute estimated block-averaged average attention $\hat{\boldsymbol{a}}$ and pivotal block-averaged attention $\tilde{\boldsymbol{a}}$

$\hat{\boldsymbol{a}} \leftarrow$ softmax(pool($\hat{\boldsymbol{Q}}\boldsymbol{K}^T$) $/\sqrt{d}$)

 # Retrieve cluster index c in `head_dict`,

c $\leftarrow$ lookup(l, h; `head_dict`)

 # Fetch the pivotal representative $\tilde{a}$

$\tilde{\boldsymbol{a}} \leftarrow$ lookup(c; `pivotal_pattern_dict`)

 # Compute sparsity and similarity divergence

$d_{sparse} \leftarrow \sqrt{JSD(\hat{\boldsymbol{a}} || \boldsymbol{u})}$, $d_{sim} \leftarrow \sqrt{JSD(\hat{\boldsymbol{a}} || \tilde{\boldsymbol{a}})}$

 # Determine whether to use pattern sharing strategy

if $d_{sparse} < \delta$ and $d_{sim} < \tau$ **then**

  $pattern \leftarrow$ shared_pivot

else

  $pattern \leftarrow$ vertical_slash

end if

return $pattern$

Algorithm 4 Share Pivotal Pattern

Input: l, h

 # Retrieve cluster index c in `head_dict`

c $\leftarrow$ lookup(l, h; `head_dict`)

 # Fetch the pivotal sparse pattern from the dynamic `pivotal_pattern_dict` $\boldsymbol{M}$

$\boldsymbol{M} \leftarrow$ lookup(c; `pivotal_pattern_dict`)

if $\boldsymbol{M}$ not exist **then**

 # Assign a dense pattern to the first head within the cluster c for subsequent full attention computation

$\boldsymbol{M} \leftarrow \boldsymbol{ones}$

else

 # Share existing pivotal pattern $\boldsymbol{M}$

end if

return $\boldsymbol{M}$

Models	Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Number	Retr.KV	Avg
Llama-3-8B-Instruct-262k	FlashAttn	25.88	8.63	67.69	5.00	12.66	20.81	26.57	100.00	100.00	14.40	38.16
	FlexPrefill	19.91	**12.60**	57.21	5.50	11.63	22.84	20.86	100.00	100.00	13.80	36.44
	MInference	**25.51**	8.50	**65.94**	8.00	**12.14**	22.08	**32.86**	100.00	100.00	16.40	39.14
	Ours	20.24	8.00	63.32	**11.84**	11.94	24.11	30.00	**100.00**	**100.00**	21.00	39.05
	Ours (δ = 1.01)	19.35	11.74	64.63	5.50	11.96	**28.17**	29.14	**100.00**	**100.00**	**23.00**	**39.35**
Qwen2.5-7B-Instruct	FlashAttn	15.53	3.18	35.81	10.50	3.95	14.47	38.57	100.00	93.56	0.00	31.56
	FlexPrefill	14.20	**3.09**	31.88	8.00	3.54	15.99	9.43	97.29	75.42	0.00	25.88
	MInference	14.83	2.86	34.93	**9.00**	3.81	14.97	38.29	96.78	76.78	0.00	29.23
	Ours	**15.31**	2.88	**38.43**	8.50	**3.99**	**17.26**	**44.57**	**99.49**	**87.46**	**0.00**	**31.79**

Table 1: Performance comparison of different methods on various models and tasks on InfiniteBench. The best and second-best results are highlighted in bold and underlined, respectively.

Methods	En.Sum	En.QA	En.MC	En.Dia	Zh.QA	Code.Debug	Math.Find	Retr.PassKey	Retr.Number	Retr.KV	Avg	128K Latency (s)
Our w/o Sharing (τ=0)	19.68	11.86	63.76	9.00	11.65	23.86	25.14	22.00	100.00	22.00	38.70	17.01
Our w/o Exclusion (δ=1.01)	19.35	11.74	64.63	5.50	11.96	28.17	29.14	100.00	100.00	23.00	**39.35**	20.02
Ours	20.24	8.00	63.32	11.84	11.94	24.11	30.00	100.00	100.00	21.00	39.05	**16.92**

Table 2: Performance of ablation methods evaluated using LLaMA-3-8B-Instruct-262K on InfiniteBench.

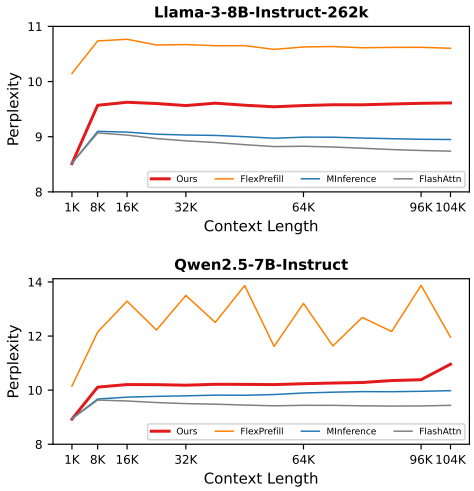


Figure 4: Perplexity results on PG-19 (Rae et al., 2020) using different models and methods.

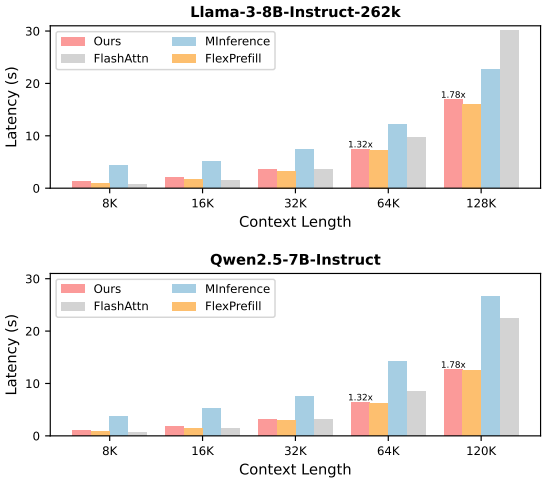


Figure 5: Latency comparison of different approaches across various context lengths using different models.

signed to assess comprehensive long-context understanding. This benchmark comprises synthetic and realistic tasks across various domains, in both English and Chinese, with an average token count of 214K, allowing us to assess SharePrefill's effectiveness across a wide range of long-context scenarios. Additionally, we include the long-context language modeling task on the PG19 dataset (Rae et al., 2020) to evaluate the models' language modeling capability. For efficiency evaluation, we conduct latency benchmarks using the length-adjustable prompts provided in MInference (Jiang et al., 2024). We compare our method with two state-of-the-art sparse attention methods (Jiang et al., 2024; Lai et al., 2025) and the efficient full attention FlashAttention 2 (Dao, 2024) to underscore its effectiveness and efficiency in long-context tasks.

Implementation Details All our experiments were conducted on a single NVIDIA A100 GPU with 80GB of memory. For baseline implementations, we use the official FlashAttention 2 package 6, and adopt the official MInference repository 5 for both MInference and FlexPrefill, which includes the FlexPrefill implementation. For MInference, we employ the default vertical-slash pattern configuration available in its code repository. For FlexPrefill, we use the default parameters with the sparse pattern threshold $\tau = 0.1$ and the cumulative pattern threshold $\gamma = 0.9$ consistently for all models. For a fair comparison, we also set the cumulative pattern threshold $\gamma = 0.9$ in our method; The similarity threshold τ is set to 0.2 and the sparsity threshold δ to 0.3, unless otherwise specified. Additionally, all the baseline methods employ sparse computation during prefilling and transition to dense computation during the decoding phase.

6.2 Main Results

We compare our method with baselines and present the main results on the aforementioned benchmarks and models. The results demonstrate that our approach achieves superior or comparable speedup while delivering the overall best accuracy.

InfiniteBench Table 1 shows that our method preserves most of the model's performance, achieving overall best accuracy maintenance. While our method with default parameters shows slightly lower accuracy compared to MInference, it achieves significantly lower latency, as shown in Figure 5 and Figure 1. However, our method outperforms MInference in both accuracy and efficiency by sharing all similar attention heads, including highly sparse ones.

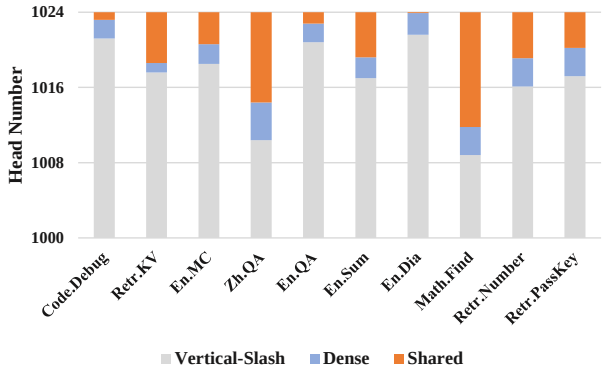


Figure 6: Distribution of three sparse attention head patterns in LLaMA-3-8B-Instruct-262K.

Language Modeling We evaluate our method against baselines on the language modeling task based on the PG-19 dataset (Rae et al., 2020). As shown in Figure 4, the perplexity of our method closely approaches the performance of MInference and FlashAttention 2, with the gap between them being within about 1.0. Moreover, the perplexity score of our method is significantly lower than that of FlexPrefill, with reductions of approximately 1.0~4.0 in Qwen2.5-7B-Instruct and over 1.0 in Llama-3-8B-Instruct-262k. These results demonstrate the strong language modeling capabilities of our approach.

Performance vs. Latency Figure 5 shows the latency across different context windows under Llama-3-8B-Instruct-262k and Qwen2.5-7B-Instruct on a single A100. The results demonstrate that our method achieves better or comparable speedup compared to the baselines. Moreover, Figure 1 compares our method with baselines under different models in terms of model performance on InfiniteBench and average latency under 128K on the latency benchmark. The results indicate that our method achieves a favorable tradeoff between accuracy preservation and inference speedup.

7 Ablation Study

Analysis of Different Components To evaluate the contributions of different components in SharePrefill, we introduce two variants for the ablation study: (1) Ours w/o sharing, which uses only the vertical-slash pattern without pivotal pattern sharing mechanism, corresponding to a similarity threshold $\tau = 0$; (2) Ours w/o exclusion, where removing highly sparse heads strategy and all similar heads participate in the pattern sharing mechanism, corresponding to a sparsity threshold $\delta = 1.01$ (selected to account for boundary conditions, ensuring that patterns with $\delta = 1$ meet the sharing-consideration criterion in line 7 of Algorithm 3). Table 2 presents the ablation results on Llama-3-8B-Instruct-262k. It first demonstrates that removing the pattern sharing mechanism leads to performance degradation, confirming the necessity of our pattern sharing strategy in preserving accuracy. Additionally, removing the strategy of excluding highly sparse heads—where all similar heads, including highly sparse ones, are allowed to share patterns—results in reduced speedup but improved performance. This demonstrates that the strategy of excluding highly sparse heads enhances efficiency while potentially degrading the model's accuracy maintenance potential. The observed accuracy improvement when removing the exclusion strategy can be attributed to more similar heads participating in pattern sharing, rather than being forced into predefined vertical-slash patterns. This further validates the effectiveness of our pattern sharing mechanism in maintaining accuracy.

Pattern Distribution Figure 6 shows the distribution of dense, shared, and conservative vertical-slash patterns used in Llama-3-8B-Instruct-262k. The majority of attention heads adopt the vertical-slash pattern, while only a small number require the full-attention dense pattern—typically just 1 to 4 heads in total. Although the number of shared patterns is limited, they play a significant role in maintaining model accuracy as shown in Table 2.

8 Conclusion and Future Work

In this paper, we observe that attention heads exhibit similarity, and this similarity remains consistent across different inputs. Built on these observations, we propose a novel sparse attention method that highly preserves accuracy while accelerating the prefilling phase. Our method achieves this by dynamically generating accurate patterns and sharing them with other similar heads, thereby capturing more realistic attention dynamics. We conduct extensive experiments across several models and tasks, demonstrating that our proposed method achieves superior or comparable speedups to state-of-the-art approaches while delivering the highest accuracy maintenance. The principle of similarity between heads and the proposed pattern-sharing mechanism holds the potential for accelerating the decoding phase and extending to multi-modular systems, which will be explored in future work.

Limitations

Although we provide observational and statistical evidence on the similarity properties between attention heads, the underlying explanation for the highly consistent similarity relationships among heads across different inputs remains unclear. This open question requires further investigation. Additionally, while our approach demonstrates effectiveness in LLM prefilling on single devices, its scalability to larger-scale scenarios requires further study. Future work will focus on evaluating and further enhancing the scalability of the proposed approach. This includes exploring efficient pattern-sharing mechanisms in scaled scenarios, such as allowing each device to maintain a local partial pivotal pattern dictionary or enabling a global dictionary to be shared across devices through inter-device communication.

References

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2024. Longlora: Efficient fine-tuning of long-context large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Yao Fu. 2024. Challenges in deploying long-context transformers: A theoretical peak performance analysis. *Preprint*, arXiv:2405.08944.

Yao Fu, Litu Ou, Mingyu Chen, Yuhao Wan, Hao Peng, and Tushar Khot. 2023. Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance. *arXiv preprint arXiv:2305.17306*.

Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Peiyuan Zhou, Jiaxing Qi, Junjie Lai, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and 1 others. 2024. Seerattention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276*.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir Abdi, Dongsheng Li, Chin-Yew Lin, and 1 others. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, and 1 others. 2025. Moba: Mixture of block attention for long-context llms. *arXiv preprint arXiv:2502.13189*.

Leonid Pekelis, Michael Feil, Forrest Moret, Mark Huang, and Tiffany Peng. 2024. Llama 3 gradient: A series of long context models.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. 2020. Compressive transformers for long-range sequence modelling. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Yu Wang, Nedim Lipka, Ryan A Rossi, Alexa Siu, Ruiyi Zhang, and Tyler Derr. 2024. Knowledge graph prompting for multi-document question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19206–19214.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, and 1 others. 2025. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. 2025. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and 1 others. 2024. ∞ bench: Extending long context evaluation beyond 100k tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15262–15277.

Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating code at scale with llms at google. *arXiv preprint arXiv:2504.09691*.

A More Details on Experimental Settings

A.1 Models

We employ two state-of-the-art long-context language models: Llama-3-8B-Instruct-262k [1](released under the Meta Llama License) and Qwen2.5-7B-Instruct [2] (released under the Apache 2.0 License). These models were selected due to their strong capabilities in handling long-context understanding tasks, with Llama-3-8B-Instruct-262k supporting contexts of up to 262K tokens and Qwen2.5-7B-Instruct supporting contexts of up to 128K tokens. Both models support multiple languages, primarily English, with Qwen2.5-7B-Instruct also demonstrating excellent performance in Chinese. Additionally, Qwen2.5-7B-Instruct supports up to 128K tokens and demonstrates excellent multilingual performance, with particular strength in Chinese. For further details, refer to the model repositories, as listed in 1 and 2

A.2 Datasets & Benchmarks

- **InfiniteBench** InfiniteBench [3] (Zhang et al., 2024) is publicly released under the Apache-2.0 License. It is a state-of-the-art benchmark designed to evaluate long-context language models with context lengths exceeding 100K tokens. The benchmark consists of 12 unique tasks, each carefully crafted to assess different aspects of language processing and comprehension in extended contexts. These tasks encompass a mix of real-world scenarios and synthetic constructs, including novels, dialogues, code, and math, ensuring a comprehensive evaluation of model capabilities. In our experiments, we compare our method's long-context performance against baselines across 10 tasks, using all available samples. Consistent with MInference and FlexPrefill, we excluded *Code.Run* and *Math.Calc* because they are highly challenging, with full-attention models often scoring near 0.

- **PG-19 Language Modeling Benchmark** (Rae et al., 2020) proposed a long-context language modeling benchmark [4] that evaluates perplexity on the PG-19 dataset and whose repository is released under the Apache 2.0 license. Perplexity quantifies how well a model predicts the next token in a sequence and is commonly used to assess the language modeling performance of long-context LLMs on extended texts. PG-19 contains books with lengths of up to 500K tokens, making it well-suited for long-context evaluation. To assess language modeling performance across different context lengths, we conduct experiments using 100 randomly selected samples from the PG-19 dataset, truncating them to various lengths ranging from 1K to 104K tokens. We then report the average perplexity based on these truncated samples. Due to high memory usage during perplexity computation, we evaluate contexts with lengths of up to 104K tokens.

- **Latency Benchmark** We follow the latency benchmarks provided in MInference [5] (Jiang et al., 2024), which is released under the MIT License. The prompts, sourced from the Chain-of-Thought Hub (Fu et al., 2023) (also released under the MIT License), were trimmed to varying token lengths to measure the prefilling stage latency. To ensure reliable measurements, we conduct ten repeated experiments after a warm-up phase and report the average latency.

[1] https://huggingface.co/gradientai/Llama-3-8B-Instruct-Gradient-262k

[2] https://huggingface.co/Qwen/Qwen2.5-7B-Instruct

[3] https://huggingface.co/datasets/xinrongzhang2022/InfiniteBench

[4] https://github.com/google-deepmind/pg19

[5] https://github.com/microsoft/MInference.git

A.3 Baselines

1. **FlashAttention 2** [6] (Dao, 2024): Flash Attention 2 is an I/O-aware exact attention algorithm designed to improve the efficiency of dense attention computation. It leverages tiling techniques to minimize the number of memory read and write operations between GPU high-bandwidth memory (HBM) and on-chip SRAM, thereby significantly reducing memory overhead and improving computational throughput. As an optimized implementation of dense attention, Flash Attention 2 enables faster and more scalable transformer inference and training, especially in long-sequence scenarios.

2. **MInference** (Jiang et al., 2024): MInference is a state-of-the-art sparse attention mechanism that exploits the static patterns observed in the attention mechanisms of LLMs, aiming to accelerate the prefilling phase for long-context inputs. It first determines offline which sparse pattern each attention head belongs to. During inference, it approximates the sparse indices online and dynamically computes attention using optimized custom kernels. This design enables significant speedup while maintaining strong accuracy.

3. **FlexPrefill** (Lai et al., 2025): FlexPrefill is another state-of-the-art sparse attention mechanism that enhances flexibility by incorporating cumulative-attention-based index selection and query-aware sparse patterns, enabling more adaptive sparse attention during the prefilling phase of LLM inference.

A.4 Implementation Details

For offline clustering, we train an autoencoder on the attention score map with a latent dimension of 64. The model is trained for 1000 epochs with early stopping and a learning rate of 1e-3. We then apply the hierarchy clustering method fcluster from scipy [7] package on the normalized compressed representation using a distance threshold of 10, assigning clusters with fewer than 5 samples to a noise cluster.

B Detailed Algorithms

Algorithm 5 Search Vertical Slash Pattern

Input: $\boldsymbol{Q}$, $\boldsymbol{K}$; γ

 # Compute a subset of the full attention map
$\hat{\boldsymbol{A}} \leftarrow \text{softmax}(\hat{\boldsymbol{Q}}\boldsymbol{K}^T / \sqrt{d})$, where $\hat{\boldsymbol{Q}} \subset \boldsymbol{Q}$

 # Sum and normalize attention scores along the vertical and slash directions
$\boldsymbol{a_v} \leftarrow \text{sum\_vertical}(\hat{\boldsymbol{A}}) / \sum_{i,j} \hat{\boldsymbol{A}}[I, j])$
$\boldsymbol{a_s} \leftarrow \text{sum\_slash}(\hat{\boldsymbol{A}}) / \sum_{i,j} \hat{\boldsymbol{A}}[I, j])$

 # Sort vertical and slash attention scores
$\boldsymbol{I_v} \leftarrow argsort(\boldsymbol{a_v})$
$\boldsymbol{I_s} \leftarrow argsort(\boldsymbol{a_s})$

 # Obtain the minimum computational budget making the sum of the scores exceeds γ
$K_v \leftarrow \min \{\text{k: } \sum_{i \in \boldsymbol{I_v}[1:k]} \boldsymbol{a_v}[i] \geq \gamma \}$
$K_s \leftarrow \min \{\text{k: } \sum_{i \in \boldsymbol{I_s}[1:k]} \boldsymbol{a_s}[i] \geq \gamma \}$

 # Select vertical and slash index
$\boldsymbol{S_v} \leftarrow \boldsymbol{I_v}[1 : K_v]$, $\boldsymbol{S_s} \leftarrow \boldsymbol{I_s}[1 : K_s]$
$\boldsymbol{S} \leftarrow \boldsymbol{S_v} \cup \boldsymbol{S_s}$
$\boldsymbol{M} \leftarrow \text{index\_to\_mask } \boldsymbol{S}$
return $\boldsymbol{M}$

C Autoencoder Architecture

Autoencoder architecture is shown in Table 3.

[6] https://pypi.org/project/flash-attn
[7] https://scipy.org

Encoder		
Layer	**Type**	**Parameters**
Conv2d	Conv2D	out: 16, kernel: 3×3, padding: 1
ReLU	Activation	
MaxPool2d	Pooling	kernel: 4×4, stride: 4
Conv2d	Conv2D	in: 16, out: 32, kernel: 3×3, padding: 1
ReLU	Activation	
MaxPool2d	Pooling	kernel: 4×4, stride: 4
Flatten	Transformation	start_dim=1, end_dim=-1
Linear	Fully Connected	in: 468512, out: 64
Decoder		
Layer	**Type**	**Parameters**
Linear	Fully Connected	in: 64, out: 468512
ReLU	Activation	
Unflatten	Transformation	shape=(32, 121, 121)
ConvTranspose2d	Transposed Convolution	in: 32, out: 16, kernel: 4×4, stride: 2, padding: 1
ReLU	Activation	
ConvTranspose2d	Transposed Convolution	in: 16, out: 8, kernel: 4×4, stride: 2, padding: 1
ReLU	Activation	
ConvTranspose2d	Transposed Convolution	in: 8, out: 1, kernel: 4×4, stride: 4
Sigmoid	Activation	

Table 3: Network Architecture of the Autoencoder