Long-Context Modeling with Dynamic Hierarchical Sparse Attention for On-Device LLMs

Siheng Xiong[1,2]* **Joe Zou**[2] **Faramarz Fekri**[1] **Yae Jee Cho**[2]
[1]Georgia Institute of Technology [2]Google
sxiong45@gatech.edu zouj@google.com fekri@ece.gatech.edu yaejeecho@google.com

Abstract

The quadratic cost of attention hinders the scalability of long-context LLMs, especially in resource-constrained settings. Existing static sparse methods such as sliding windows or global tokens utilizes the sparsity of attention to reduce the cost of attention, but poorly adapts to the content-dependent variations in attention due to their staticity. While previous work has proposed several dynamic approaches to improve flexibility, they still depend on predefined templates or heuristic mechanisms. Such strategies reduce generality and prune tokens that remain contextually important, limiting their accuracy across diverse tasks. To tackle these bottlenecks of existing methods for long-context modeling, we introduce **Dynamic Hierarchical Sparse Attention (DHSA)**, a data-driven framework that dynamically predicts attention sparsity online without retraining. Our proposed DHSA adaptively segments sequences into variable-length chunks, then computes chunk representations by aggregating the token embeddings within each chunk. To avoid the bias introduced by varying chunk lengths, we apply length-normalized aggregation that scales the averaged embeddings by the square root of the chunk size. Finally, DHSA upsamples the chunk-level similarity scores to token level similarities to calculate importance scores that determine which token-level interactions should be preserved. Our experiments on Gemma2 with Needle-in-a-Haystack Test and LongBench show that DHSA matches dense attention in accuracy, while reducing prefill latency by 20–60% and peak memory usage by 35%. Compared to other representative baselines such as block sparse attention, DHSA achieves consistently higher accuracy (6–18% relative gains) with comparable or lower cost, offering an efficient and adaptable solution for long-context on-device LLMs[2].

1 Introduction

Long-context modeling is crucial for real-world applications [Yang et al., 2024]. However, the quadratic complexity of attention [Vaswani et al., 2017] makes scaling to long sequences prohibitively expensive, especially in resource-constrained scenarios like on-device applications. While prior work has explored sparsity-based methods to tackle the high cost of implementing long sequences, many rely on static sparse patterns (e.g., Longformer [Beltagy et al., 2020], BigBird [Zaheer et al., 2020]) failing to generalize across tasks or recent dynamic methods (e.g., Minference [Jiang et al., 2024], LM-Infinite [Han et al., 2023], H2o [Zhang et al., 2023], Scissorhands [Liu et al., 2023]) rely on oversimplified templates or heuristic cache-eviction rules, restricting adaptability and accuracy.

In this paper, we propose **Dynamic Hierarchical Sparse Attention (DHSA)**, a **lightweight plug-in** module that dynamically predicts attention sparsity during both prefill and decode stages. Unlike prior

*Work done during internship at Google.
[2]Code and data available at `https://github.com/xiongsiheng/DHSA`.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Efficient Reasoning.

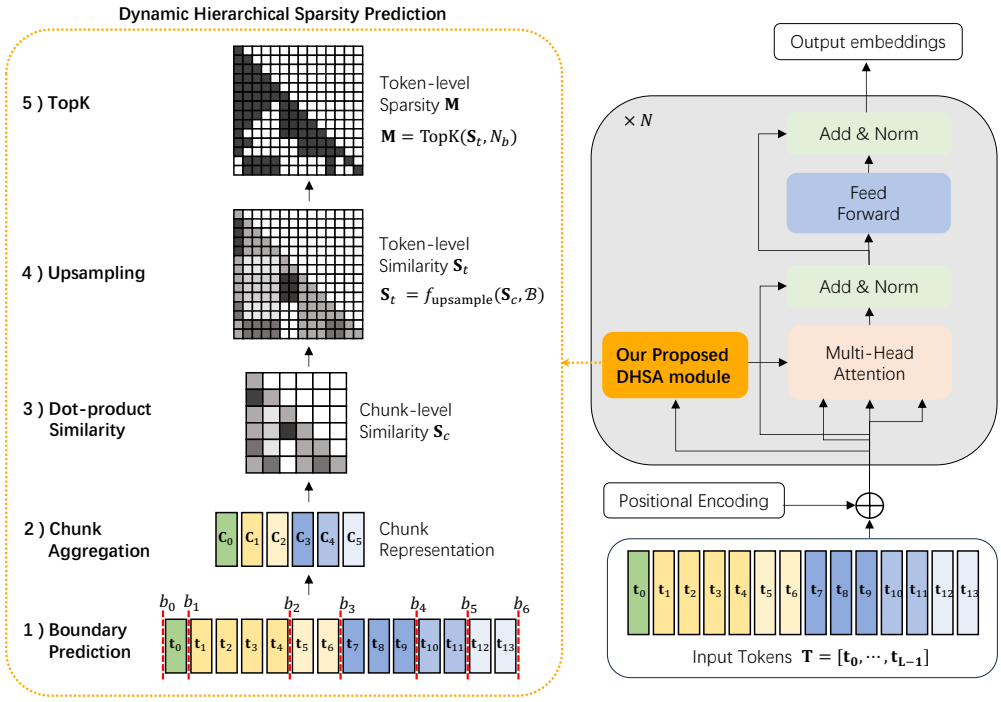


Figure 1: Overview of the proposed Dynamic Hierarchical Sparse Attention (DHSA) framework.

approaches that rely on hand-crafted sparsity patterns, DHSA learns to infer sparsity through chunk-level similarity and adaptive boundary prediction. This makes DHSA fully data-driven, enabling it to adapt across tasks without retraining the base model or manually tuning sparsity patterns.

Our contributions are summarized as follows:

- **Hierarchical sparsity prediction**. We estimate chunk-level similarity and upsample it to token-level importance scores, enabling DHSA to focus on the most impactful attention weights and thereby reduce latency and memory while preserving accuracy (see Section 2.1).
- **Dynamic chunking**. We introduce our proposed boundary-prediction method that adaptively segments input sequences into variable-length chunks, overcoming the limitations of fixed-size chunking and enabling efficient long-sequence processing (see Section 2.2).
- **Robust chunk representation**. We propose a length-normalized aggregation strategy that mitigates the pitfalls of naive average pooling and better captures chunk-level representations (see Section 2.3).

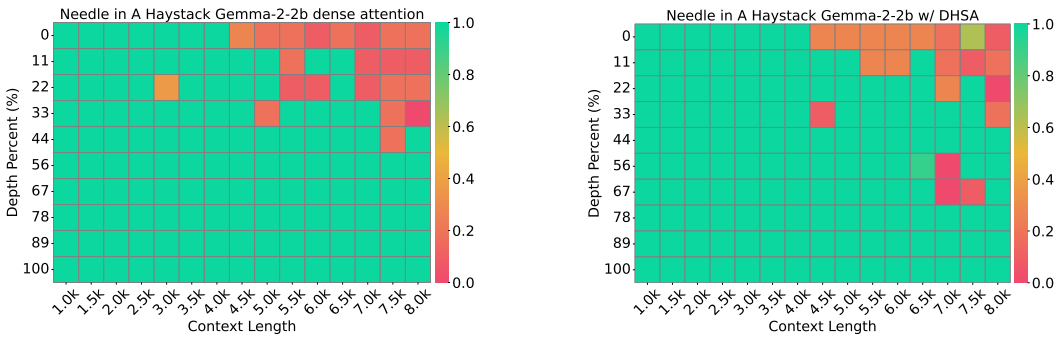
Our experiments on Needle-in-a-Haystack Test and LongBench [Bai et al., 2023] show that DHSA matches dense attention in accuracy while reducing prefill latency by 25–45% and peak memory usage by 30–35%, and consistently outperforms other dynamic sparsity baselines [Han, 2024] with 6–18% relative gains.

2 Proposed Method: Dynamic Hierarchical Sparse Attention (DHSA)

Our proposed DHSA is a plug-in module integrated into each of the N Transformer layers of an LLM (see Fig. 1). It takes the token embeddings at the current layer as the input and outputs a sparsity mask to prune unimportant token pairs. The core idea is to leverage **chunk-level similarity** to inform **token-level sparsity** prediction. This requires addressing two key challenges: (1) fixed-size chunking is too rigid to capture **content shifts**, and (2) average pooling poorly handles **variable-length chunks**. We resolve these with the solutions detailed in Sections 2.2 and 2.3.

2.1 Hierarchical Sparsity Prediction

Given a sequence of tokens $\mathbf{T} = [\mathbf{t}_0, \mathbf{t}_1, ..., \mathbf{t}_{L-1}]$ of total length L, its token-level sparsity mask is denoted as $\mathbf{M} \in \{0, 1\}^{L \times L}$, where if the i^{th} row and j^{th} column element of $\mathbf{M}$ is equal to 1, i.e.,



(a) Dense Attention (b) DHSA (budget = 1k)

Figure 2: Needle-in-a-haystack results on Gemma2-2b-it (maximum context length = 8k). The budget is specified per query per layer.

	Single-Document QA			Multi-Document QA			Summarization			Few-shot Learning	
Method	NrtvQA	Qasper	Mf-en	HotpotQA	2WikiMQA	Musique	GovReport	QMSum	MultiNews	TriviaQA	SAMSum
Dense	22.37	35.32	37.32	41.63	32.05	19.05	27.08	21.08	25.48	87.00	41.26
Block sparse	16.74	26.15	32.83	35.74	31.93	14.44	26.20	19.54	25.30	86.12	40.38
DHSA	**20.69**	**30.20**	**34.98**	38.78	31.96	**15.90**	**26.75**	**20.74**	25.38	87.03	**41.46**
DHSA (+bs)	19.64	28.67	31.20	**39.50**	**32.97**	14.39	26.72	20.12	**25.57**	**87.74**	40.68

Table 1: Performance comparison of DHSA, dense attention, and block sparse attention on LongBench using Gemma2-2b-it. DHSA (+bs) denotes a variant with boundary sharing across layers. For sparse method, the budget is set as 2k. Block size and query chunk size are both set to 256 for block sparse and DHSA, respectively.

$\mathbf{M}_{i,j} = 1$, it indicates that the interaction between $\mathbf{t}_i$ and $\mathbf{t}_j$, $i, j \in [0, L-1]$ should be preserved. Conversely, $\mathbf{M}_{i,j} = 0$ implies that the interaction between $\mathbf{t}_i$ and $\mathbf{t}_j$ can be skipped.

Predicting the full matrix $\mathbf{M}$ directly would require scoring all $L \times L$ token pairs, which is computationally prohibitive for long contexts. Instead, we adopt a two-step hierarchical approach (see Appendix A for the detailed workflow):

Step 1 **Chunk-level prediction**: We partition the entire token sequence $\mathbf{T}$ into N_c non-overlapping chunks $\{\mathbf{C}_0, \mathbf{C}_1, ..., \mathbf{C}_{N_c-1}\}$, defined by the boundary indices $\mathcal{B} = \{b_0, b_1, ..., b_{N_c}\}$, where $0 = b_0 < b_1 < ... < b_{N_c} = L$. Each chunk $\mathbf{C}_k$, $k \in [0, ..., N_c - 1]$ contains the consecutive tokens indexed from b_k to b_{k+1}. We then construct a chunk-level similarity matrix $\mathbf{S}_c \in \mathbb{R}^{N_c \times N_c}$, where the l^{th} row and k^{th} column element of $\mathbf{S}_c$, i.e., $(\mathbf{S}_c)_{l,k}$ represents the predicted importance of interactions between chunks $\mathbf{C}_l$ and $\mathbf{C}_k$. The procedure for obtaining $\mathbf{S}_c$ will be described in Section 2.3.

Step 2 **Token-level selection**: Starting from $\mathbf{S}_c$, we upsample it to obtain the token-level similarity matrix $\mathbf{S}_t \in \mathbb{R}^{L \times L}$, which encodes the predicted importance of attending from each query token to every key token. Concretely, for each chunk pair $(\mathbf{C}_l, \mathbf{C}_k)$, $\{l, k\} \in [0, N_c - 1]$, the corresponding submatrix $(\mathbf{S}_t)_{[b_l:b_{l+1}],[b_k:b_{k+1}]}$ is assigned the same value $(\mathbf{S}_c)_{l,k}$. We define such mapping function as $f_{\text{upsample}}(\mathbf{S}_c, \mathcal{B}) \coloneqq \mathbf{S}_t$. Given the token-level similarity matrix $\mathbf{S}_t$, we generate the token-level sparsity mask $\mathbf{M}$ by applying a TopK selection with a per-query token budget defined as N_b. Here N_b is a hyperparameter, which may be set dynamically based on available computational or memory resources.

2.2 Dynamic Boundary Detection

To better estimate $\mathbf{S}_t$ from $\mathbf{S}_c$, we propose a dynamic chunking strategy that adaptively determines boundary indices $\mathcal{B}$ based on the input sequence. We formulate chunking as a **boundary detection** problem, where the goal is to decide whether each token position marks the *end of a chunk*. Formally, for each position $i \in [0, L-1]$, we define a boundary indicator function $\delta(i) = 1$ if $i = b_k$ for some k; Otherwise, $\delta(i) = 0$. We estimate this indicator using a neural network with three components:

Encoder. For each candidate position i, we extract two local windows:

$$\mathbf{k}_{\text{left}} = f_{\text{MHA}}([\mathbf{k}_{i-w+1}, \cdots, \mathbf{k}_i]), \quad \mathbf{k}_{\text{right}} = f_{\text{MHA}}([\mathbf{k}_{i+1}, \cdots, \mathbf{k}_{i+w}]) \tag{1}$$

where w is the window size and $\mathbf{k}_j$ denotes the key vector of token j. Each window is processed by a Multi-Head Attention (MHA) module defined as f_{MHA} with pooling.

Attn. Implem.	Method	Acc. (%)	Latency (s)	Peak Mem. (GB)
eager	Dense	21.15	1.65	10.72
	Block	17.04	1.00	9.08
	DHSA	20.12	1.19	6.91
torch.sdpa	Dense	22.37	1.10	6.33
	Block	16.74	0.88	9.88
	DHSA	19.37	0.91	6.99

Table 2: Comparison on NarrativeQA with Gemma2. Both block-sparse and DHSA use a 2k budget (block/query chunk size = 256, max context = 8k).

Context Len	Attn. Implem.	Method	Latency (s)	Peak Mem. (GB)
16k	eager	Dense	-	OOM
		DHSA	2.18	9.69
	torch.sdpa	Dense	3.37	8.38
		DHSA	1.98	9.69
32k	eager	Dense	-	OOM
		DHSA	4.51	16.99
	torch.sdpa	Dense	10.97	15.18
		DHSA	4.13	16.99

Table 3: Comparison at varying context lengths with Gemma2 on a single 24 GB GPU. DHSA uses a 2k budget with a query chunk size of 256.

Feature Fusion. Given $\mathbf{k}_{\text{left}}$ and $\mathbf{k}_{\text{right}}$, we construct the feature vector:

$$\mathbf{h}_i = [\mathbf{k}_{\text{left}},\ \mathbf{k}_{\text{right}},\ |\mathbf{k}_{\text{left}} - \mathbf{k}_{\text{right}}|,\ \mathbf{k}_{\text{left}} \odot \mathbf{k}_{\text{right}},\ \text{sim}(\mathbf{k}_{\text{left}}, \mathbf{k}_{\text{right}})] \tag{2}$$

where $\odot$ is element-wise multiplication and $\text{sim}(\cdot,\cdot)$ is cosine similarity. Further rationale and analyses of this fusion choice are provided in Appendix B.

MLP. The fused feature $\mathbf{h}_i$ is passed through two linear layers defined as f_{MLP}:

$$\Pr(\delta(i) = 1) = f_{\text{MLP}}(\mathbf{h}_i) \tag{3}$$

yielding the probability that position i is a boundary. The training and inference details for our boundary predictor are provided in Appendix B.

2.3 Robust Chunk Representation

Now that we have the boundaries, we aggregate token embeddings to form chunk representations. We identify two main challenges in this process: (1) average pooling after padding is problematic, as zero embeddings from padding dilute the average, and (2) average pooling is sensitive to chunk length. To address these issues, we compute the sum of embeddings and divide by the actual (unpadded) chunk length, followed by **length normalization**:

$$\mathbf{q}_c = \sqrt{|\mathbf{C}|} \cdot \bar{\mathbf{q}}, \quad \mathbf{k}_c = \sqrt{|\mathbf{C}|} \cdot \bar{\mathbf{k}} \tag{4}$$

where $|\mathbf{C}|$ denotes the number of tokens in chunk $\mathbf{C}$, and $\bar{\mathbf{q}}$ and $\bar{\mathbf{k}}$ is the average of token queries and keys within the chunk respectively. We then compute the dot product similarity between each chunk query and chunk key using $\mathbf{S}_c = \mathbf{Q}_c\mathbf{K}_c^\top$ where $\mathbf{Q}_c$ and $\mathbf{K}_c$ are the respective query and key matrices.

3 Evaluation

Experiment setup. We implement our method in PyTorch with Hugging Face Transformers and run all experiments on a single NVIDIA RTX 3090 GPU. We use Gemma2-2b-it [Gemma Team et al., 2024] and Gemma3-1b-it [Gemma Team et al., 2025] with `torch.bfloat16` precision and batch size 1. Comparisons are mainly against dense and block sparse attention [Han, 2024], the latter lacking dynamic chunking and improved chunk representations. Further implementation details and additional results are in Appendices C and D.

Results. We evaluate DHSA on Needle-in-a-Haystack and LongBench against dense and block sparse attention. (1) **Needle-in-a-Haystack Test**: On Gemma2-2b-it, retaining the top 1k tokens per layer, DHSA matches dense attention while substantially outperforming baselines (see Fig. 2, Fig. 6). (2) **LongBench**: With a 2k budget, DHSA performs close to dense attention and clearly surpasses block sparse attention (see Table 1). (3) **Latency/Memory Usage**: On NarrativeQA (from LongBench), DHSA reduces latency and memory relative to dense attention. While slightly slower than block sparse, it achieves higher accuracy with lower memory usage (see Table 2 and 3).

4 Conclusion

We presented Dynamic Hierarchical Sparse Attention (DHSA), a method that combines dynamic chunking with hierarchical sparsity prediction to efficiently focus on the most relevant tokens in long-context language modeling. Experiments on Needle-in-a-Haystack Test and LongBench demonstrate that DHSA achieves accuracy on par with dense attention while substantially reducing latency and memory usage. Unlike static sparsity or conventional block-sparse approaches, DHSA adapts to input-dependent attention patterns, enabling more effective use of the attention budget and offering a practical solution for efficient long-context modeling.

Limitations

The primary goal of DHSA is to accelerate LLM inference. Although extending the maximum context length could further enhance its utility, we found no reliable implementation for the Gemma family, and our initial adaptation produced unexpected behavior. Debugging this was beyond the scope of the current work, so we leave context-length extension for future exploration. Moreover, although DHSA is fully data-driven, its performance still depends on hyperparameters such as the chunk budget and chunk size. Developing adaptive or learned strategies for budget allocation remains an important direction for future work.

References

Yuan Yang, Siheng Xiong, Ehsan Shareghi, and Faramarz Fekri. The compressor-retriever architecture for language model os. *arXiv preprint arXiv:2409.01495*, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

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

Chi Han, Qifan Wang, Hao Peng, Wenhan Xiong, Yu Chen, Heng Ji, and Sinong Wang. Lm-infinite: Zero-shot extreme length generalization for large language models. *arXiv preprint arXiv:2308.16137*, 2023.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Lab Han. Block-sparse attention. `https://hanlab.mit.edu/blog/block-sparse-attention`, 2024. Accessed: 2025-08-27.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. triviaqa: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension. *arXiv e-prints*, art. arXiv:1705.03551, 2017.

Peng Xu, Wei Ping, Xianchao Wu, Zihan Liu, Mohammad Shoeybi, and Bryan Catanzaro. Chatqa 2: Bridging the gap to proprietary llms in long context and rag capabilities. *arXiv preprint arXiv:2407.14482*, 2024.

Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. In *18th international conference on pattern recognition (ICPR'06)*, volume 3, pages 850–855. IEEE, 2006.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

A DHSA Workflow

DHSA applies to both the **prefill** and **decode** stages. Specifically,

- **Prefill stage**: When all tokens in the prompt sequence of length L are available, we first predict the boundary indices $\mathcal{B}$ for the entire prompt. We then perform chunk-level prediction to obtain $\mathbf{S}_c$, upsample it to produce the token-level similarity matrix $\mathbf{S}_t$ with f_{upsample}, and apply TopK selection with budget N_b to obtain the token-level sparsity mask $\mathbf{M}$ for all tokens in the prompt.
- **Decode stage**: In autoregressive generation, we adapt the approach to handle the incremental arrival of new tokens. Let L' be the total sequence length at the current decoding step (including both prompt and generated tokens). We extend the existing chunk boundaries $\mathcal{B} = [b_0, ..., b_{N_c}]$, $b_{N_c} = L$ to $\mathcal{B}' = [b_0, ..., b_{N_c}, L' - 1, L']$. Here we have that
  - The total number of chunks becomes $N_c + 2$.
  - The chunk $\mathbf{C}_{N_c} = [\mathbf{t}_L, ..., \mathbf{t}_{L'-2}]$ contains all previously generated tokens in the current decoding session.
  - The chunk $\mathbf{C}_{N_c+1} = [\mathbf{t}_{L'-1}]$ contains only the current query token.

  We then compute only the interactions between $\mathbf{C}_{N_c+1}$ and all its preceding chunks to obtain last row of the updated chunk-level similarity matrix $\mathbf{s}_{c,\text{new}}$. This row is upsampled to the token level to produce the corresponding row of the updated token-level similarity matrix $\mathbf{s}_{t,\text{new}}$, from which we derive the last row of the sparsity mask $\mathbf{m}_{\text{new}}$ by applying TopK selection with token budget N_b.

We present the pseudocode of our approach for the prefill and decode stages (Algorithms 1 and 2) and analyze the associated computational cost. Dense attention is the most expensive, as each token attends to all L keys, leading to a cost of $O(L^2)$. Block sparse attention is more efficient because each token attends to only N_b keys, reducing the cost to $O(L \cdot N_b)$, while the routing term is typically sub-dominant for realistic sequence lengths. DHSA lies in between: it retains most of the efficiency gains of block sparse attention but introduces an additional $O(L)$ pass for boundary prediction, which identifies chunk boundaries. For sequence lengths where attention is the primary bottleneck (thousands of tokens), the general cost ordering is: **Block sparse attention < DHSA < Dense attention**.

B Boundary Detection

We formulate the chunking task as a **boundary detection** problem, where the objective is to determine whether each token position marks the *end of a chunk*. Formally, for each position $i \in [0, L-1]$, we define a boundary indicator function

$$\delta(i) = \begin{cases} 1 & \text{if } i = b_k \text{ for some } k, \\ 0 & \text{otherwise,} \end{cases}$$

indicating that the token at position i is the last token of a chunk. This end-boundary prediction approach aligns naturally with the way sequences are segmented, as it allows the model to determine when a coherent segment of context has concluded. By framing the task this way, we can leverage binary classification methods to adaptively segment sequences in DHSA.

This probability, denoted as $\Pr(\delta(i) = 1)$, is predicted based on the local key representations surrounding i, using a boundary prediction function that takes as input two windows centered at i:

$$[\mathbf{k}_{i-w+1}, \cdots, \mathbf{k}_i], \quad [\mathbf{k}_{i+1}, \cdots, \mathbf{k}_{i+w}],$$

where w is a hyperparameter that determines the receptive field and $\mathbf{k}_j$ denotes the key vector corresponding to token j.

Local context is sufficient for boundary prediction because chunk boundaries are determined by **local changes in semantic similarity**. Intuitively, if the left window (preceding tokens) and the right window (succeeding tokens) are highly similar, the two regions likely belong to the same chunk, and no boundary should be placed. Conversely, a sharp drop in similarity between these windows indicates a topic or context shift, suggesting the end of a chunk. Moreover, focusing on local context rather than the full sequence significantly reduces computation. Evaluating boundaries requires processing only tokens per position which is essential for **efficiency**. To further validate the

Algorithm 1 DHSA: Prefill Stage

Input: prompt tokens $\{\mathbf{t}_i\}_{i=0}^{L-1}$ of length L; prompt chunk boundaries $\{b_j\}_{j=0}^{N_c}$ of length N_c+1; per-query token budget N_b
Output: token-level sparsity mask $\mathbf{M}$

Define chunks $\mathbf{C}_j \leftarrow [\mathbf{t}_{b_j}, \ldots, \mathbf{t}_{b_{j+1}}]$, for $j = 0$ to $N_c - 1$
Obtain chunk-level queries $\mathbf{Q}_c[j,:]$ and keys $\mathbf{K}_c[j,:]$ for each $\mathbf{C}_j$
Compute chunk-level similarity: $\mathbf{S}_c \leftarrow \mathbf{Q}_c\mathbf{K}_c^\top \in \mathbb{R}^{N_c \times N_c}$

Initialize token-level similarity $\mathbf{S}_t \in \mathbb{R}^{L\times L}$ to zeros
for $j = 0$ to $N_c - 1$ **do**
    for $l = 0$ to $N_c - 1$ **do**
        $\mathbf{S}_t[b_j : b_{j+1}, b_l : b_{l+1}] \leftarrow \mathbf{S}_c[j,l]$
    end for
end for

Initialize mask $\mathbf{M} \in \{0,1\}^{L\times L}$ to zeros
for $i = 1$ to L **do**
    Select Top N_b proceeding keys based on $\mathbf{S}_t[i,:]$
    Update $\mathbf{M}[i,:]$ accordingly
end for
return $\mathbf{M}$

Algorithm 2 DHSA: Decode Stage

Input: prompt tokens $\{\mathbf{t}_i\}_{i=0}^{L-1}$; prompt chunk boundaries $\{b_j\}_{j=0}^{N_c}$ and chunks $\{\mathbf{C}_j\}_{j=0}^{N_c-1}$; current total length L'; previous generated tokens $\{\mathbf{t}_i\}_{i=L}^{L'-2}$; current query token $\mathbf{t}_{L'-1}$; per-query token budget N_b
Output: last row of the updated token-level sparsity mask $\mathbf{m}_{\text{new}}$

Define the updated boundaries $b_{N_c+1} \leftarrow L' - 1$ and $b_{N_c+2} \leftarrow L'$
Define the updated chunk $\mathbf{C}_{N_c} \leftarrow [\mathbf{t}_L, \ldots, \mathbf{t}_{L'-2}]$ and $\mathbf{C}_{N_c+1} \leftarrow [\mathbf{t}_{L'-1}]$
Update chunk-level keys $\mathbf{K}_c'$ for $\mathbf{C}_{N_c}$ and $\mathbf{C}_{N_c+1}$
Compute last row of the updated chunk-level similarity $\mathbf{s}_{c,\text{new}} \leftarrow \mathbf{q}_{L'-1}(\mathbf{K}_c')^\top \in \mathbb{R}^{N_c+2}$

Initialize last row of the updated token-level similarity $\mathbf{s}_{t,\text{new}} \in \mathbb{R}^{L'}$ with zeros
for $j = 0$ to $N_c + 1$ **do**
    $\mathbf{s}_{t,\text{new}}[b_j : b_{j+1}] \leftarrow \mathbf{s}_{c,\text{new}}[j]$
end for

Initialize mask $\mathbf{m}_{\text{new}} \in \{0,1\}^{L'}$ with zeros
Select Top N_b proceeding keys based on $\mathbf{s}_{t,\text{new}}$
Update $\mathbf{m}_{\text{new}}$ accordingly
return $\mathbf{m}_{\text{new}}$

interpretability of these boundaries, we analyzed their correspondence with sentence and paragraph endings and observed a strong alignment, suggesting that DHSA captures meaningful discourse-level transitions.

Architecture. We define the boundary prediction function as a neural network. After exploring various architectures and hyperparameters, we finalize the design shown in Fig. 3. This architecture was chosen because it strikes a balance between **expressiveness**, **efficiency**, and **robustness**. The encoder effectively captures contextualized token embeddings for both the left and right windows. The feature fusion module proved more stable and discriminative than using a single metric in isolation. The final MLP is shallow enough to maintain low latency but still has sufficient capacity. By keeping the receptive field limited to $2w$ tokens, the total prediction cost is linear to token length L.

In the **feature fusion** module, we combined the raw context vectors $\mathbf{k}_{\text{left}}, \mathbf{k}_{\text{right}}$, absolute differences $|\mathbf{k}_{\text{left}} - \mathbf{k}_{\text{right}}|$, multiplicative interactions $\mathbf{k}_{\text{left}} \odot \mathbf{k}_{\text{right}}$, and cosine similarity $\text{sim}(\mathbf{k}_{\text{left}}, \mathbf{k}_{\text{right}})$ because each signal captures complementary aspects of boundary semantics. The raw vectors preserve local context information, absolute differences highlight directional changes between left and right spans, multiplicative interactions emphasize co-activation patterns, and cosine similarity provides a normalized measure of alignment. Together, these features make the boundary predictor more robust to scale, length, and semantic variation. In ablations, we found that using only a single similarity measure (e.g., cosine similarity) was less stable, whereas combining multiple signals yielded consistently better boundary detection.

We selected the optimal hyperparameters using the Long Data Collections dataset[3]. The selection process involved monitoring training loss, training metrics, and validation metrics, including precision, recall, F1 score, and topK overlap. In addition, we evaluated the end-to-end performance of the trained model on the validation set. The chosen configuration includes a context window size of $w = 4$ (i.e., 8 tokens per position), 8 attention heads, average pooling, a hidden size of 256 in the MLP, and a total model size of 20 MB shared across layers and datasets.

[3] `https://huggingface.co/datasets/togethercomputer/Long-Data-Collections`

Explicit modeling incorporates multiple similarity signals beyond raw encodings, enhancing both interpretability and performance. Adaptability is achieved by using a single model shared across layers and datasets, which promotes generalization and efficiency. The model is also lightweight: with a size of only 20 MB, it is well-suited for deployment in resource-constrained environments.

Automatic labelling. To train the boundary predictor, we automatically derive ground-truth labels from attention scores, avoiding the need for manual annotation. We adopt this intermediate labelling strategy instead of end-to-end training from final task performance. End-to-end optimization is computationally intractable, as it would require differentiating through boundary indices with only sparse, delayed supervision from downstream metrics. In contrast, derived labels provide dense, local supervision, turning boundary detection into a well-defined, efficient classification problem that still captures the structural cues implicit in the model's own attention behavior.

Specifically, we analyze accumulated attention mass patterns. Tokens within a coherent span typically exhibit consistent accumulated attention profiles, meaning the distribution of attention mass over preceding tokens remains relatively stable across positions within the span. In contrast, a boundary is often marked by a sudden change in this profile, such as when the subsequent token's accumulated attention shifts sharply toward a different subset of preceding tokens.

Fig. 4 illustrates our automatic labeling strategy. For each candidate position, we examine the accumulated attention mass patterns in its left window and right window, where each row in the heatmap corresponds to a token and color intensity indicates the magnitude of accumulated attention mass. In the left example, the left and right windows (both outlined in blue) exhibit highly similar attention profiles, indicating that the tokens around this position belong to the same coherent span; thus, the position is not labeled as a boundary. In the right example, the attention profiles of the left window (blue) and right window (green) differ markedly, signaling a sharp change in attention behavior. This difference suggests that the position lies at the end of a chunk and should be labeled as a boundary.

Formally, let the token sequence be

$$\mathbf{T} = [\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{L-1}]$$

with length L, and let $A \in \mathbb{R}^{L \times L}$ denote its attention matrix, where $A_{u,v}$ is the attention weight from token u to token v.

For each position $i \in [0, L-1]$, we consider whether it marks the end of a chunk. To do so, we examine two local windows of size w on either side of i, i.e., the past window: tokens $\{\mathbf{t}_{i-w+1}, \ldots, \mathbf{t}_i\}$ and the future window: tokens $\{\mathbf{t}_{i+1}, \ldots, \mathbf{t}_{i+w}\}$.

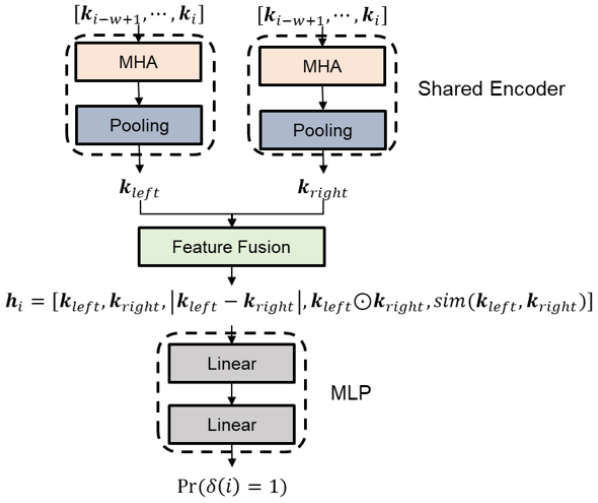


Figure 3: Architecture of the boundary predictor, consisting of a shared encoder, a feature fusion module, and an MLP classifier.

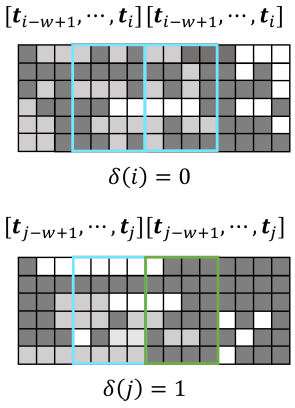


Figure 4: Illustration of our automatic labeling strategy. The core intuition is that tokens within a coherent span tend to receive similar attention distributions.

We then compute the past cumulative attention mass:

$$a_{\text{past}}(i) = \frac{1}{L-1-i-w} \sum_{u=i+w+1}^{L-1} \sum_{v=i-w+1}^{i} A_{u,v}, \tag{5}$$

and the future cumulative attention mass:

$$a_{\text{fut}}(i) = \frac{1}{L-1-i-w} \sum_{u=i+w+1}^{L-1} \sum_{v=i+1}^{i+w} A_{u,v}, \tag{6}$$

where $w = 4$ is the local window size (receptive field), and the outer sum index u iterates over future tokens (beyond $i + w$) while the inner sum index v iterates over the tokens in the corresponding window.

We define the attention ratio:

$$r_i = \frac{\max\big(a_{\text{fut}}(i), a_{\text{past}}(i)\big) + \varepsilon}{\min\big(a_{\text{fut}}(i), a_{\text{past}}(i)\big) + \varepsilon}, \tag{7}$$

where $\varepsilon = 0.001$ is added for numerical stability.

Given the maximum allowed number of chunks N_c and a threshold $\theta_r = 1.1$, we select the top $N_c - 1$ positions i whose r_i exceeds θ_r (with positions 0 and L always included as boundaries), with N_c serving as the primary constraint in practice.

Visualization. We show token-level (normalized) similarity matrix with static and dynamic chunking in Fig. 5. Static chunking divides the sequence into fixed-size non-overlapping chunks for attention computation, whereas dynamic chunking adapts chunk boundaries based on our labelling strategy. These examples are taken from Gemma2-2b-it, which alternates local and global attention layers. Layer 0 is a local attention layer and Layer 1 is a global attention layer, and we treat them separately in the visualization to highlight the differences.

In the static chunking plots (left), chunk boundaries are fixed at uniform intervals, producing rigid, evenly spaced attention bands that may split semantically related tokens across chunks. In contrast, the dynamic chunking plots (right) show irregularly sized chunks whose boundaries shift with the attention patterns, allowing both local and global attention layers to capture longer, more coherent spans within a single chunk. This alignment to content can reduce boundary-induced information loss and improve performance.

Training of the boundary predictor. We describe below the key implementation details of our training pipeline that are non-trivial and contribute meaningfully to performance.

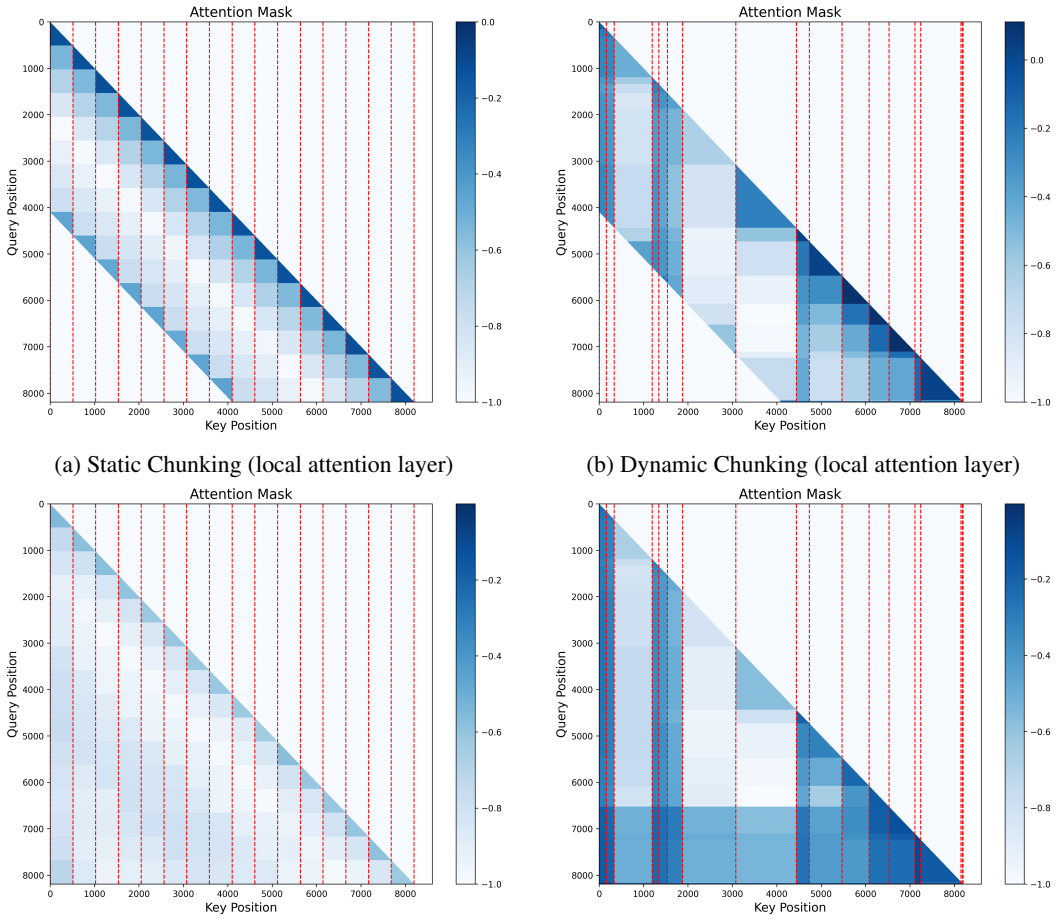
Instead of using hard labels, which are obtained by selecting the top $N_c - 1$ positions with the highest attention ratios, we adopt a soft labeling strategy. Hard labels are inherently sensitive to the chunk number constraint, i.e., the fixed number $N_c - 1$ of chunk boundaries to be selected per sequence. Under this scheme, a position with a given ratio r might receive a label of 1 for one choice of N_c but a label of 0 for another, even though its underlying ratio has not changed. This sensitivity can confuse the model and discard useful information about positions near the cutoff.

To address this, we convert the attention ratio r into a continuous probability value using the following transformation:

$$p = \sigma\big(\alpha \cdot (\log(r + \zeta) - \beta)\big) \tag{8}$$

where r is the ratio, $\zeta = 10^{-6}$ is a small constant for numerical stability, $\alpha = 2.0$ and $\beta = \lg(2.0)$ are scalar parameters controlling the slope and offset, with base $e \approx 2.71828$, and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

This maps the ratio to a probability in the $[0, 1]$ range, preserving the relative ordering of positions and capturing confidence without enforcing a hard cutoff. Positions with higher ratios produce probabilities closer to 1, but those with moderately high ratios still receive meaningful supervision. This approach is in line with knowledge distillation, where soft targets have been shown to improve generalization by providing richer learning signals than binary labels.



(a) Static Chunking (local attention layer)

(b) Dynamic Chunking (local attention layer)

(c) Static Chunking (global attention layer)

(d) Dynamic Chunking (global attention layer)

Figure 5: Token-level similarity matrices with static (left) and dynamic (right) chunking in Gemma2-2b-it. Static chunking uses fixed-size chunks, while dynamic chunking adapts boundaries to content, shown for a local attention layer (layer 0) and a global attention layer (layer 1).

Using the conventional Binary Cross-Entropy (BCE) loss for training our boundary predictor in dynamic chunking, we identify two key issues: (1) *class imbalance*, where boundary tokens are much fewer than non-boundary tokens, and (2) *varying sample difficulty*, where some boundaries are easier to detect than others. To address these, we adopt the **focal BCE loss**, which down-weights the contribution of easy examples and amplifies the focus on harder, misclassified cases. The weighting term $(1-p_i)^\gamma$ automatically reduces the loss for well-classified positions (large p_i for positives, small p_i for negatives), while the fixed positive-class weight w offsets the imbalance between boundary and non-boundary positions.

$$\mathcal{L}_i = (1-p_i)^\gamma \Big[-wy_i \log p_i - (1-y_i)\log(1-p_i) \Big] \tag{9}$$

where $y_i \in [0,1]$ is the ground-truth soft label for position i, $z_i \in \mathbb{R}$ is the logit produced by the boundary predictor, $p_i = \sigma(z_i) = \frac{1}{1+e^{-z_i}}$ is the predicted probability, $w = 1.3$ is the fixed positive-class weight for class imbalance, and $\gamma = 2.0$ is the focal parameter controlling emphasis on hard examples.

We further accelerate training through the following strategies. (1) **Storing only boundary labels**: Instead of storing both input embeddings and boundary labels, which would consume excessive memory, we store only the labels. This allows the model to perform dynamic chunking directly from the loaded labels without recalculating them. Additionally, the labels can be generated in parallel for all training samples, significantly improving efficiency. (2) **Training all layers simultaneously**: The boundary predictor needs to be trained for all layers. Rather than performing multiple forward

passes, one for each layer, we perform a single forward pass of the language model per sample to obtain input embeddings for all layers. These embeddings are then used to train the predictor for all layers simultaneously.

Data preparation. We analyze existing long-context datasets for both training and inference. For training, we select Long Data Collections[4], trivia QA [Joshi et al., 2017][5], ChatQA2 [Xu et al., 2024][6] that mainly focus on high-quality question answering and summarization tasks. We inspected the datasets and found that the samples are quite similar. To accelerate training, we apply sampling by selecting the first 10,000 samples from each dataset for training. For evaluation, we use the first 100 samples from the validation set of each dataset.

Metrics. We monitor the following metrics during training to facilitate debugging and to compare different hyperparameter configurations. We track the training loss across all layers (results shown for Gemma2-2b-it, which has 26 layers in total). For training, we evaluate precision, recall, and F1 score for the positive class (where soft labels ≥ 0.5 are treated as positive), as well as top-K overlap with $K = 500$. Top-K overlap is defined as the number of overlapping positions divided by K, and we monitor this metric because, during inference, the top positions are selected as boundaries. For validation, we compute precision, recall, F1 score, and top-K overlap ($K = 500$) on the validation set.

Inference of the boundary predictor. We observe that selecting positions purely based on the topK scores can lead to suboptimal boundaries due to noise and closely spaced high-score peaks. To mitigate this, we adopt Non-Maximum Suppression (NMS) [Neubeck and Van Gool, 2006], a technique widely used for eliminating redundant detections in computer vision. The core idea is to keep only the highest-scoring candidate within a local neighborhood, thereby producing well-separated, reliable boundaries.

We first obtain the boundary scores by computing the boundary predictor's output scores for all positions in the sequence, which represent the model's confidence that a given position marks a chunk end. Next, we identify candidate boundary positions by selecting all positions whose scores exceed a minimal confidence threshold, thereby pruning out low-probability positions while retaining multiple plausible candidates. The remaining candidates are then sorted in descending order of their scores, ensuring that higher-confidence positions are considered first during suppression. We subsequently apply NMS: starting with the highest-scoring candidate, we mark it as a boundary and remove any other candidates within a specified window size (e.g., 8 or 64 tokens) of this position, as they are considered overlapping or too close to be separate boundaries. After suppression, the final set consists of well-spaced local maxima that are more robust to noise and score fluctuations.

Applying NMS with a small window size (e.g., 8) yields better results than using no NMS. For tasks requiring broader context segmentation, a larger NMS window size (e.g., 64) further improves performance. In some cases, we also augment the output with explicit boundaries such as `\n`, `\n\n`, or prior information from structured prompts to better align with semantic structure.

C Implementation Details

We primarily implement the method using PyTorch and Hugging Face Transformers. All experiments were conducted on a single NVIDIA RTX 3090 GPU (24 GB) running Ubuntu 22.04.4 LTS. The software environment includes Python 3.12, CUDA 12.4, PyTorch 2.5.1+cu124, and Transformers 4.52.3. We used the Gemma2-2b-it[7] and Gemma3-1b-it[8] with `torch.bfloat16` precision and a batch size of 1. In Gemma2-2b-it, global attention is applied in every other layer with a sliding window of 4,096 tokens, and the model supports up to 8,192 position embeddings. In Gemma3-1b-it, global attention is applied in every sixth layer with a sliding window of 512 tokens, and the model supports up to 32,768 position embeddings.

[4] `https://huggingface.co/datasets/togethercomputer/Long-Data-Collections`

[5] `https://huggingface.co/datasets/mandarjoshi/trivia_qa`

[6] `https://huggingface.co/datasets/nvidia/ChatQA2-Long-SFT-data`

[7] `https://huggingface.co/google/gemma-2-2b-it`

[8] `https://huggingface.co/google/gemma-3-1b-it`

Needle-in-a-haystack test. We evaluated different baselines using Gemma2-2b-it and Gemma3-1b-it, starting with an assessment of the models' long-context processing capabilities through the needle-in-a-haystack test. This benchmark evaluates a model's ability to locate a target sentence (the needle) within a long context and is widely used for long-context language modeling. Our setup used context lengths ranging from 1,000 to 8,000 tokens (interval of 100) and depth ranges from 0% to 100% (interval of 10%). The prompt format was: `<|im_start|> This is a very long story book: <book> {context} </book>. Based on the content of the book, Question: {retrieval_question} Answer:` with the needle sentence "The best thing to do in San Francisco is eat a sandwich and sit in Dolores Park on a sunny day." and the corresponding retrieval question "The best thing to do in San Francisco is:". We used ROUGE as the evaluation metric, and visualized results with green indicating correct and red indicating incorrect predictions.

D Additional Results

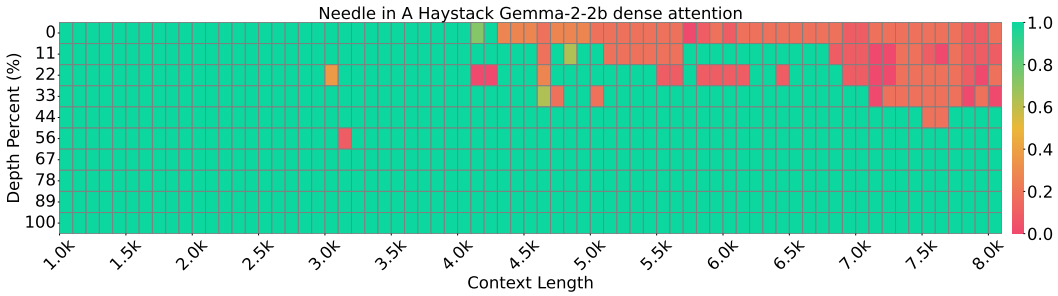
Needle-in-a-haystack test. The results are shown in Fig. 6. Among sparse attention methods, sliding window attention [Beltagy et al., 2020] relies on static patterns that perform poorly when the actual attention deviates from the predefined one. On Gemma2-2b-it, retaining the top 1k tokens per layer, DHSA matches dense attention while substantially outperforming block sparse attention. Even with a budget of 512, DHSA maintains strong performance, highlighting its efficiency. We also observe that Gemma3-1b-it, despite having more local attention layers, exhibits stronger long-range contextual understanding. This improvement likely stems from architectural and training advances, including an extended positional encoding range (32k vs. 8k), optimized placement of global layers, refined local attention designs, and greater exposure to long-form data during pretraining.

Latency/memory usage comparison. We further evaluated the latency and memory usage of different methods: block sparse attention, sliding window attention, StreamingLLM [Xiao et al., 2023], H2O [Zhang et al., 2023], and PyramidKV [Cai et al., 2024]. Note that StreamingLLM, H2O, and PyramidKV focus on KV cache compression, which does not reduce prefill latency or prefill peak memory usage. Block sparse attention, similar to our method, can affect prefill latency and peak memory usage.

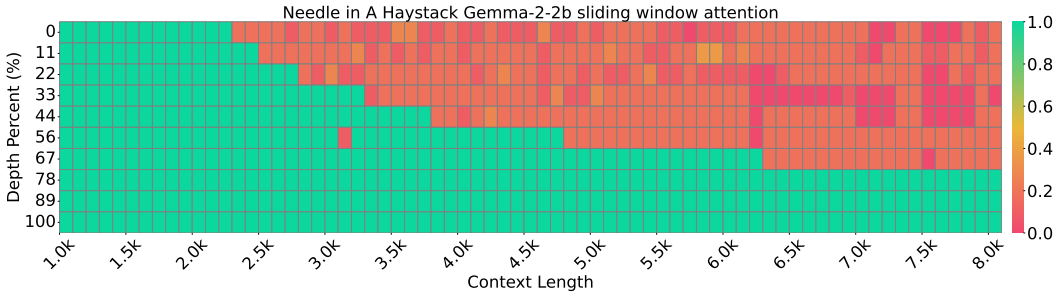
For the latency test on Gemma2-2b-it (Fig. 7), block sparse attention has a prefill budget of 512, sliding window attention has a maximum KV cache capacity of 2048, while StreamingLLM, H2O, and PyramidKV are limited to 128. Block sparse attention with larger block sizes can also reduce prefill latency. On the other hand, for all decode-stage methods, prefill time increases with context length since KV cache compression only applies during decoding. Decoding time remains constant for KV cache compression methods regardless of context length, whereas dense attention decoding time increases as the context grows because more KV states are cached. Reducing the maximum KV cache capacity, such as using 2048 for sliding window attention compared to 128 for compression methods, has only a minor effect on wall-clock time, since the attention forward pass during decoding accounts for only a small portion of the total computation.

For the memory usage test on Gemma2-2b-it (Fig. 8), larger block sizes in block-sparse attention reduce prefill memory usage. In contrast, all decode-stage methods show increasing prefill memory with context length, since KV compression only applies during decoding. During decoding, KV cache compression methods keep peak allocated memory constant, while dense attention memory usage grows with context length. Sliding window attention exhibits higher peak memory usage because of its larger cache capacity.

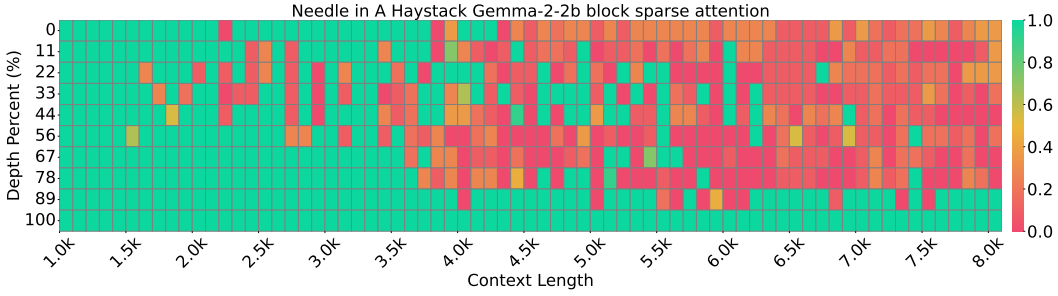
For the latency test on Gemma3-1b-it (Fig. 9), decoding time remains constant regardless of context length because Gemma3 applies global attention only every six layers. In the local attention layers, the KV cache size does not grow with context length. For the memory usage test on Gemma3-1b-it, we noticed that sliding window attention with a large KV cache capacity (2048) shows higher memory usage at the first decoding step because new key/value states for a single token are concatenated to the existing cache. Since `torch.cat()` creates a new tensor rather than modifying in place, both the old and new tensors are temporarily stored in memory. For models with many layers, this can double KV cache memory usage temporarily. After the first decoding step, memory usage stabilizes, and reported values correspond to this stabilized state.



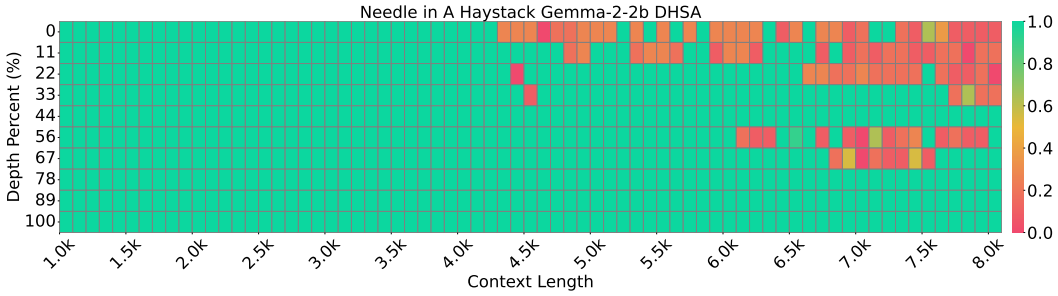
(a) Dense Attention



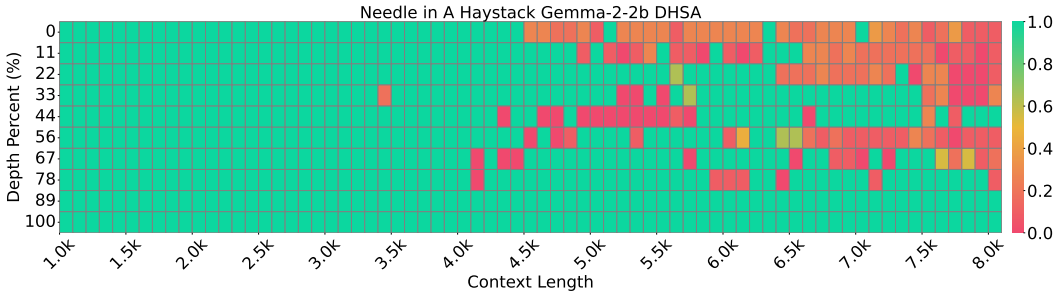
(b) Sliding Window Attention (budget = 2k)



(c) Block Sparse Attention (budget = 1k)

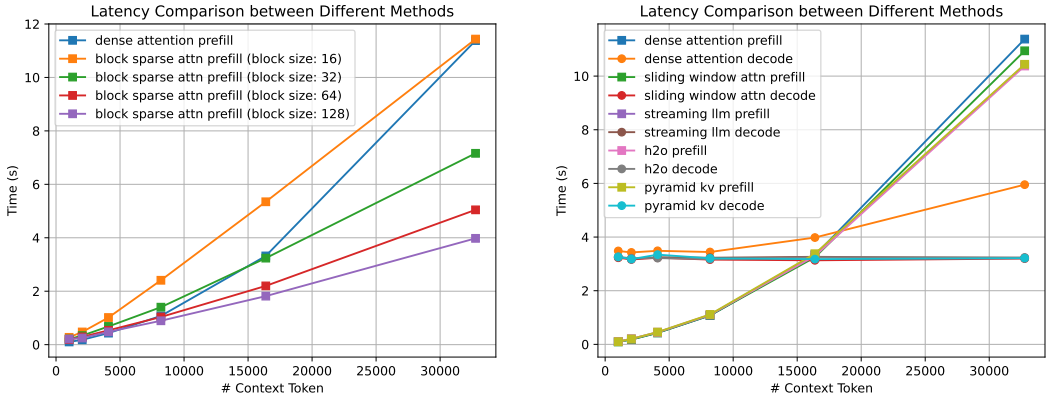


(d) DHSA (budget = 1k)



(e) DHSA (budget = 512)

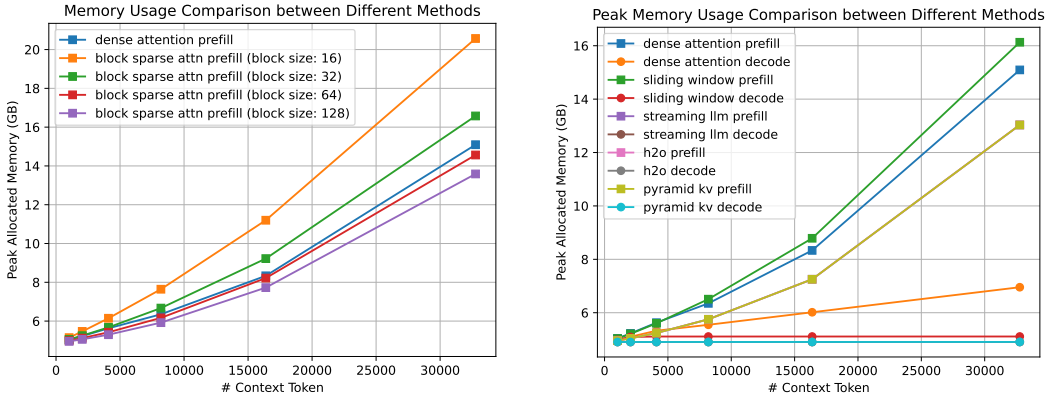
Figure 6: Needle-in-a-haystack results on Gemma2-2b-it (maximum context length = 8k). The budget is specified per query per layer. Block size and query chunk size are both set to 64 for block sparse and DHSA, respectively.



(a) Prefill-stage sparse attention

(b) Decode-stage sparse attention

Figure 7: Latency comparison on Gemma2-2b-it with varying context lengths. Sliding-window attention uses a budget of 2048, block sparse attention 512 and KV compression (streaming LLM, h2o, pyramidKV) 128. The number of generated tokens is 100, and all methods are implemented with `torch.sdpa`.

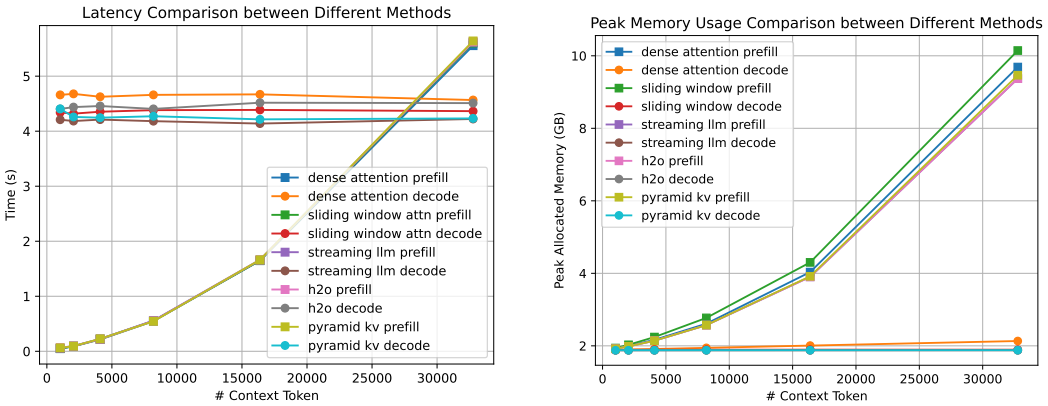


(a) Prefill-stage sparse attention

(b) Decode-stage sparse attention

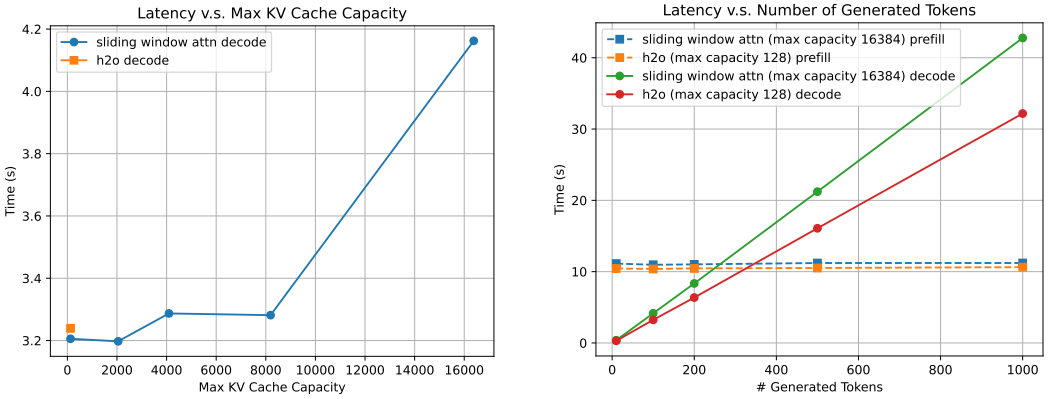
Figure 8: Peak memory usage comparison on Gemma2-2b-it with varying context lengths. Sliding-window attention uses a budget of 2048, block sparse attention 512 and KV compression (streaming LLM, h2o, pyramidKV) 128. The number of generated tokens is 100, and all methods are implemented with `torch.sdpa`.

Finally, we note several potential influencing factors in KV compression methods (Fig. 10). Large changes in wall-clock inference time occur only with substantial reductions in KV cache size. While KV compression may not drastically shorten inference time, it clearly reduces memory usage. In addition, decoding time is largely proportional to the number of newly generated tokens.



(a) Latency (b) Memory Usage

Figure 9: Latency and peak memory usage comparison across different baselines with varying context length on Gemma3-1b-it. Sliding-window attention uses a budget of 2048, block sparse attention 512 and KV compression (streaming LLM, h2o, pyramidKV) 128. The number of generated tokens is 100, and all methods are implemented with `torch.sdpa`.



(a) Latency vs. Maximum KV Cache Capacity (context length = 32k, generated tokens = 100)

(b) Latency vs. Number of Generated Tokens (context length = 32k)

Figure 10: Potential factors affecting the latency of KV compression methods on Gemma2-2b-it. The attention is implemented with `torch.sdpa`.