Sparse Delta Memory: Scaling the State of Linear RNNs through Sparsity

Loïc Cabannes[1,2], **Pierre-Emmanuel Mazaré**[1], **Gergely Szilvasy**[1], **Matthijs Douze**[1], **Maria Lomeli**[1], **Ilze Amanda Auzina**[1,3], **Justin Carpentier**[2], **Gabriel Synnaeve**[1], **Hervé Jégou**[1]

[1]Meta FAIR, [2]Inria Paris & ENS-PSL University, [3]University of Tübingen

Linear attention models allow a fixed state size and a fixed amount of compute per token. However, due to their limited state size, linear attention models fall behind in long-context recall compared to softmax-attention-based transformer architectures. Increasing the state size of linear attention improves recall performance but at the cost of higher FLOPs. In this work, we introduce **Sparse Delta Memory (SDM)**, an architecture that scales the hidden state of gated linear RNNs to orders of magnitude higher capacity using a sparse addressing scheme. SDM extends the Gated DeltaNet architecture by replacing the dense key-value outer product with sparse reads and writes to a large explicit memory. We show that, under an isoFLOP constraint and with an identical number of parameters, a higher state memory capacity significantly improves performance on in-context learning and long-context retrieval tasks. Moreover, by learning the initial state of the SDM memory and therefore using it as a parametric memory, we show that the model further improves on a wide range of common-knowledge and reasoning tasks.

Date: July 9, 2026
Code: https://github.com/facebookresearch/sparse-delta-memory

Meta

1 Introduction

As frontier models continue to progress, they are leveraged in increasingly more complex tasks. In particular, the emergence of agentic settings involve sustained reasoning over long contexts, including software engineering, research assistance, and personal assistants.
These applications demand memory mechanisms that preserve long-range dependencies across extended interactions. In the standard transformer architectures equipped with a vanilla softmax attention, the interaction memory is stored in a Key-Value (KV) cache. It effective on long-context tasks, yet the KV cache and therefore the compute and memory per token all grow linearly with the sequence length, (Fig. 1 green line). This unbounded growth limits in-context learning over very long sequences, such as entire codebases or extended reasoning traces, which are increasingly central to autonomous agents, but also videos which play a central role notably in world models and robotics.

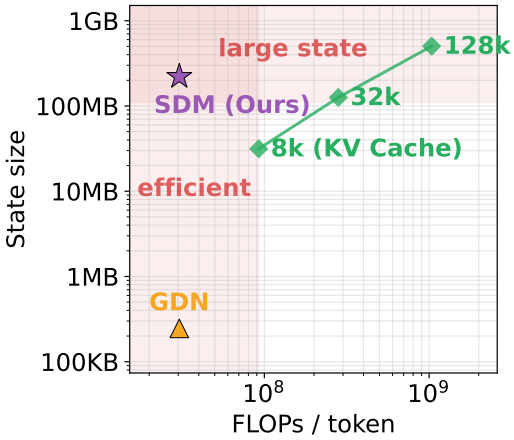


Figure 1 State size vs FLOPs per token (global layer, 1.4B). KV cache scales linearly with the sequence length. Our SDM approach offers a large state with constant FLOPs.

A possible way to avoid this growing KV cache is to replace the explicit storage of all past tokens with a compressed recurrent state. Recurrent Neural Networks (RNNs), including State Space Models (Gu & Dao, 2024) and Linear Attention variants (Katharopoulos et al., 2020; Beck et al., 2024), rely on this strategy: they compress information into a fixed-size hidden state, maintaining constant memory and compute per token

regardless of sequence length. This enables the processing of arbitrarily long contexts without an explicit token limit. However, their extremely small state sizes limit recall capability compared to transformers (Fu et al., 2023). Indeed, Arora et al. (2025) show that long-context performance is fundamentally bounded by the hidden state size. Simply increasing the RNN memory size would improve recall, but modern linear RNNs like Mamba2 (Dao & Gu, 2024) and Gated DeltaNet (GDN) (Yang et al., 2025) are bottlenecked by dense state updates that become prohibitively expensive as the state size grows.

To address the limitations of dense state updates, we introduce Sparse Delta Memory (SDM), a novel architecture based on the observation that the GDN update rule can be sparsified. This enables a three-order-of-magnitude increase in memory state size while maintaining the same compute budget, as shown in Fig. 1. Thanks to its larger state size, SDM significantly outperforms GDN on long-context recall tasks from the RULER (Hsieh et al., 2024) benchmark and also shows better in-context learning capabilities on sequences of up to 1 million tokens. Moreover, we show that contrary to GDN, learning the initial state of SDM allows the model to store meaningful pretraining knowledge that significantly improves performance on a wide range of metrics. Indeed, in isoFLOP comparisons, an SDM with a learned initial state consistently achieves a lower training loss than GDN across all parameter scales. We validate our findings by training 8B activated parameter models on more than 1 trillion tokens. At this scale, SDM reaches an even lower loss and a slightly better short-context accuracy than a model trained with full attention. Overall, we show that SDM, thanks to its large state, can keep the constant-space and memory advantages of Linear RNNs while significantly improving on long-context tasks which have been the main limitation of Linear RNNs like GDN and Mamba so far. Given the constant compute and memory footprint of SDM and its strong long-context performance, we believe that SDM opens up new possibilities in developing agents with improved long-term memory and in-context understanding over extended sequences and also has the potential to address long-term memory issues found in other modalities such as long-video processing.

2 Background

Linear Attention as an Associative Memory. A fundamental perspective introduced by Katharopoulos et al. (2020) is that attention writes outer products of keys $\mathbf{k}_t \in \mathbb{R}^{d_{\text{qk}}}$ and values $\mathbf{v_t} \in \mathbb{R}^{d_{\text{v}}}$ into a memory. Then, to read from the memory, a query $\mathbf{q_t} \in \mathbb{R}^{d_{\text{qk}}}$ is compared to the previous keys using a similarity metric, usually the inner product $\langle \mathbf{q}, \mathbf{k} \rangle$. Moreover, to improve the accuracy of information retrieval, a pre-processing feature mapping ϕ can be applied to the keys and queries. We can then define the memory tensor $\mathbf{M}_t$ and normalization factor $\mathbf{z}_t$:

$$\mathbf{M}_t = \sum_{i=1}^{t} \phi(\mathbf{k}_i)\, \mathbf{v}_i^\top \in \mathbb{R}^{d_{\text{qk}} \times d_{\text{v}}}, \qquad \mathbf{z}_t = \sum_{i=1}^{t} \phi(\mathbf{k}_i) \in \mathbb{R}^{d_{\text{qk}}}.$$

A normalized read is then:

$$\mathbf{y}_t = \frac{\mathbf{M}_t^\top \phi(\mathbf{q}_t)}{\mathbf{z}_t^\top \phi(\mathbf{q}_t)} = \frac{\sum_{i \le t} \langle \phi(\mathbf{q}_t), \phi(\mathbf{k}_i) \rangle\, \mathbf{v}_i}{\sum_{i \le t} \langle \phi(\mathbf{q}_t), \phi(\mathbf{k}_i) \rangle}. \tag{1}$$

Note that there exists an infinite-dimensional feature mapping ϕ such that $\langle \phi(\mathbf{q}_t), \phi(\mathbf{k}_i) \rangle = e^{\langle \mathbf{q}_t, \mathbf{k}_i \rangle}$, in which case Equation (1) computes exactly softmax attention. On the contrary, if ϕ is a finite-dimensional mapping, then the feature mapped keys and queries as well as the memory tensor $\mathbf{M}_t$ are finite-dimensional and can thus be materialized and cached for future retrievals. That is, $(\mathbf{M}_t, \mathbf{z}_t)$ is stored in *constant* memory, no matter the sequence length.

Gated DeltaNet (GDN). DeltaNet (Schlag et al., 2021) improves upon vanilla linear attention by introducing the *delta rule*: before writing a new association into the memory, the model first retrieves and subtracts the existing value associated with the key, thereby preventing interference and keeping the memory norm bounded. Gated DeltaNet (GDN) further adds a decay (forget) gate $\alpha_t \in (0, 1)$ to control how quickly old associations are forgotten. This gives the following state update formula:

$$\mathbf{M}_t \leftarrow \alpha_t\, \mathbf{M}_{t-1} + \beta_t\, \mathbf{k}_t \left(\mathbf{v}_t - \alpha_t \mathbf{M}_{t-1}^\top \mathbf{k}_t \right)^\top, \tag{2}$$

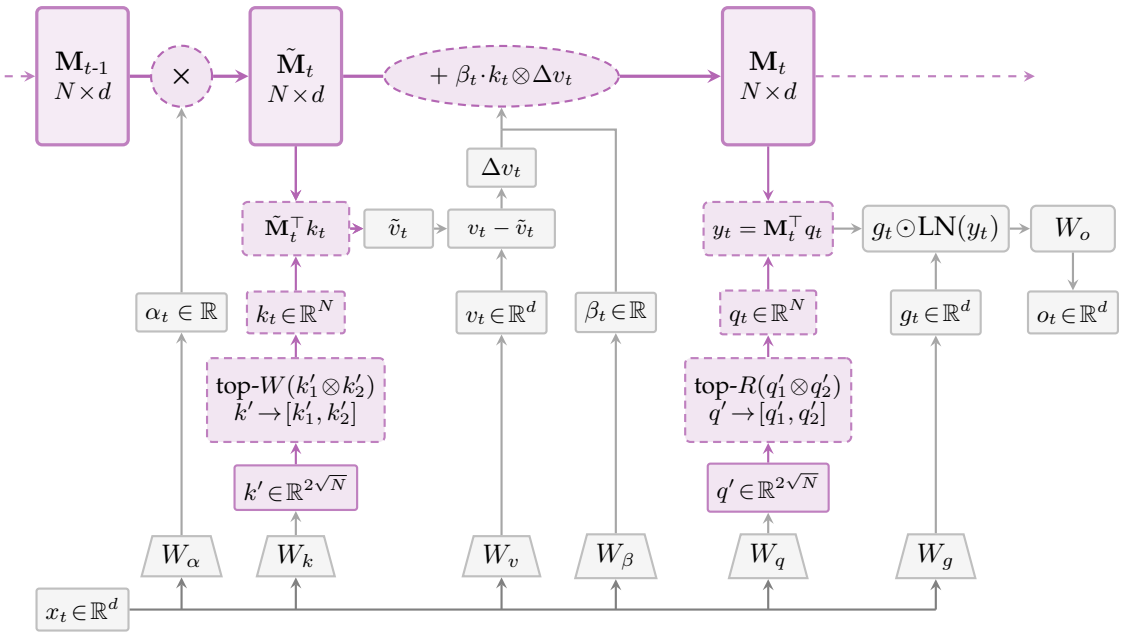


Figure 2 SDM layer. Gray: operations present in GDN. Purple: operations modified in SDM. Dashed borders indicate sparse operations (W or R out of N slots).

where $\beta_t \in [0, 1]$ is a learned input gate modulating the memory update strength. In GDN, $\mathbf{k}_t$ and $\mathbf{q}_t$ are dense vectors in $\mathbb{R}^{d_{\text{qk}}}$, and the state $\mathbf{M}_t \in \mathbb{R}^{d_{\text{qk}} \times d_{\text{v}}}$ is a dense matrix. The per-token computational cost is $O(d_{\text{qk}} \times d_{\text{v}})$. Thus, increasing the size of the state represents a linear increase in FLOPs.

Product Key Memory. Product-Key Memories (PKM) (Lample et al., 2019) propose an indexing scheme that allows indexing to k arbitrary indices among N possible memory slots, while scaling the number of computations sublinearly with respect to N. Indeed, PKM can scale to memories of size $N \times d$ with $N \approx 10^6$ (Berges et al., 2024), while previous dense approaches are limited to $N \approx 10^3$ or 10^4. Given two sets of scores $\mathbf{s}_1 \in \mathbb{R}^{\sqrt{N}}$ and $\mathbf{s}_2 \in \mathbb{R}^{\sqrt{N}}$, one can get N scores, one for each possible memory index, by doing the outer sum between the two score vectors: $\mathbf{s} \in \mathbb{R}^{\sqrt{N} \times \sqrt{N}} = \mathbf{s}_1 \oplus \mathbf{s}_2$. We simply get the top_k of the flattened scores $\mathbf{s}$. Since $\text{top}_k(\mathbf{s}_1 \oplus \mathbf{s}_2) = \text{top}_k(\text{top}_k(\mathbf{s}_1) \oplus \text{top}_k(\mathbf{s}_2))$, we do not need to materialize the entire N scores $\mathbf{s}$ but only k^2 scores, making the indexing operation very efficient. Since obtaining the scores $\mathbf{s_1}$ and $\mathbf{s_2}$ requires $O(\sqrt{N} \times d)$ operations and obtaining the final top_k scores requires $O(k^2)$ operations, the indexing scheme of PKM has a $O(\sqrt{N} \times d + k^2)$ time complexity. It is important to note that in PKM, the memory state is not updated by the context and is learned only through training, analogous to the FFN weights in the transformer architecture.

Other Related Works. The work most closely related to ours is the recent Fast-Weight Product Key Memory (Zhao & Jones, 2026), which aims at sparsifying the existing Test-Time-Training methods (Sun et al., 2025; Zhang et al., 2025) using a sparse PKM memory. Their design diverges from ours in their update rule, their hybridization design and ablation choices and other experimental settings. Moreover, their work does not provide results in an iso-FLOPs setting. Finally, their analysis remains limited to small-scale experiments involving models with at most 100M non-embedding parameters. Nonetheless, their design of a sparse online Memory shows promising results, and we concur with their motivation that sparsity enables significant improvements in RNN long-context capabilities.

3 Method

3.1 Sparse Delta Memory

Our key insight is that the GDN update rule in Eq. (2) can be *sparsified*: rather than applying the decay and delta update to the entire dense state $\mathbf{M}_t \in \mathbb{R}^{d_{\text{qk}} \times d_{\text{v}}}$, we maintain an explicit memory table $\mathbf{M}_t \in \mathbb{R}^{N \times d_{\text{v}}}$ with N slots, and apply gated delta updates *only to the W slots selected by sparse keys*. Concretely, at each timestep t and for each SDM head, we propose:

1. **Sparse Key Selection.** Pre-PKM write keys $\mathbf{k'}_t$ and read queries $\mathbf{q'}_t$ are projected from the input $\mathbf{x}_t$ via learned linear projections $W_k, W_q \in \mathbb{R}^{d \times 2\sqrt{N}}$. Each projected vector is split into two halves ($\mathbf{k}'_{1,t}, \mathbf{k}'_{2,t} \in \mathbb{R}^{\sqrt{N}}$ for keys; $\mathbf{q}'_{1,t}, \mathbf{q}'_{2,t} \in \mathbb{R}^{\sqrt{N}}$ for queries), and their outer sum $\mathbf{k}'_{1,t} \oplus \mathbf{k}'_{2,t} \in \mathbb{R}^{\sqrt{N} \times \sqrt{N}}$ yields N scores: one per memory slot. Applying top-W to the write scores and top-R to the read scores selects the W write indices $\mathcal{I}_t^w$ and R read indices $\mathcal{I}_t^r$ from N possible slots with $O(\sqrt{N} \times d + W^2 + R^2)$ compute, as this can be done without materializing the full score matrix (since $\text{top}_k(\mathbf{s}_1 \oplus \mathbf{s}_2) = \text{top}_k(\text{top}_k(\mathbf{s}_1) \oplus \text{top}_k(\mathbf{s}_2))$).

2. **Gated Delta Write.** For each selected write slot $i \in \mathcal{I}_t^w$:

$$\tilde{\mathbf{M}}_t[i] \leftarrow \underbrace{\alpha_t}_{\text{forget gate}} \cdot \mathbf{M}_{t-1}[i] \tag{3}$$

$$\mathbf{M}_t[i] \leftarrow \tilde{\mathbf{M}}_t[i] + \underbrace{\beta_t}_{\text{input gate}} \cdot k_t^{(i)} \cdot \left(\mathbf{v}_t - \tilde{\mathbf{M}}_t[i]\right) \tag{4}$$

   where $\alpha_t = \exp(-A \cdot \text{softplus}(W_a \mathbf{x}_t + b_{\text{dt}}))$ is a per-head forget gate, $\beta_t = \sigma(W_b \mathbf{x}_t)$ is the input gate, $k_t^{(i)}$ is the sparse key value (writing weight) for slot i, $\mathbf{v}_t = W_v \mathbf{x}_t$ is the value vector, and A is a learnable decay parameter. Unselected slots ($i \notin \mathcal{I}_t^w$) remain unchanged: $\mathbf{M}_t[i] = \mathbf{M}_{t-1}[i]$.

3. **Sparse Read.** The memory is then read by a weighted sum of the R selected read slots:

$$\mathbf{y}_t = \mathbf{M}_t^\top q_t = \sum_{i \in \mathcal{I}_t^r} q_t^{(i)} \cdot \mathbf{M}_t[i] \tag{5}$$

4. **Norm, Gating, and Head Mixing.** The retrieved memory $\mathbf{y}_t$ is normalized through RMS-Norm, element-wise gated with $\mathbf{g} \in \mathbb{R}^{d_v}$, and finally a projection W_o mixes the outputs from all SDM heads to produce the final layer output $\mathbf{o}_t \in \mathbb{R}^d$.

Connection to GDN. When $N = d_{\text{qk}}$, $W = R = d_{\text{qk}}$ (all slots selected), and the sparse key values $k_t^{(i)}$ form a dense vector, Eq. (4) recovers exactly the GDN update. In this case, the only difference is the lack of 1D convolutions on the qkv vectors, present in GDN but not in SDM.

Learned Initial State $\mathbf{M_0}$. Compared to GDN, which has a very small state size, SDM has a much larger state. This property might not serve only as a storage mechanism for in-context knowledge, but also, if one considers $\mathbf{M_0}$ as a learnable parameter of the model, the SDM memory can learn knowledge during pretraining and reuse it at test time. Since having a learned $\mathbf{M_0}$ does not add any FLOPs at inference time compared to a null-initialized $\mathbf{M_0}$, we chose the learned $\mathbf{M_0}$ variant as the default setting for SDM. We ablate the impact of learning $\mathbf{M_0}$ in Section 6.

Efficient Training. We detail how SDM is trained efficiently using chunk-wise parallelism (via the WY representation from GDN/FLA) and a memory-efficient backward pass in Appendix A.

3.2 IsoFLOP Design: Matching GDN Parameters and FLOPs

We ensure that SDM uses the same number of parameters and FLOPs as the dense GDN baseline. Hence, any improvement stems from the larger memory capacity alone.

Parameters. Both GDN and SDM share identically-sized linear projections: $W_q, W_k \in \mathbb{R}^{d \times d_{\text{qk}}^{\text{total}}}$ and $W_v \in \mathbb{R}^{d \times d_{\text{v}}^{\text{total}}}$, where $d_{\text{qk}}^{\text{total}} = H \times d_{\text{qk}}^{\text{GDN}} = d/2$ and $d_{\text{v}}^{\text{total}} = H \times d_{\text{v}}^{\text{GDN}} = d$.

FLOPs. GDN's per-token cost is $O(H \times d_{\text{qk}}^{\text{GDN}} \times d_{\text{v}}^{\text{GDN}}) = O(d_{\text{qk}}^{\text{GDN}} \times d_{\text{v}}^{\text{total}})$ because every head accesses its full $d_{\text{qk}} \times d_{\text{v}}$ state. SDM's cost is instead $O(H^{\text{SDM}} \times (W+R) \times d_{\text{v}}^{\text{SDM}}) = O((W+R) \times d_{\text{v}}^{\text{total}})$, independent of the memory size N. Setting $W = R = d_{\text{qk}}^{\text{GDN}}$ thus yields matching FLOPs. To be precise, PKM's top-k on the outersum of scores is an additional computation, but it amounts for less than 1% of the layer's FLOPs.

We set $d_{\text{qk}}^{\text{GDN}} = 64$, $d_{\text{v}}^{\text{GDN}} = 128$, giving $W_q, W_k \in \mathbb{R}^{d \times d/2}$ and $W_v \in \mathbb{R}^{d \times d}$ for both GDN and SDM.

3.3 Limiting the State Size Expansion

With SDM (and unlike GDN), fewer heads do not incur increased FLOPs but still increasing memory size, which improves performance (Arora et al., 2025). Having a single head under a parameter constraint maximizes the size of the memory. Indeed, the Memory size of SDM scales as such:

$$M_{\text{size}} = H \times \left[\frac{d_{\text{qk}}^{\text{total}}}{2H} \times \frac{d_{\text{qk}}^{\text{total}}}{2H} \times \frac{d_{\text{v}}^{\text{total}}}{H} \right] = \frac{(d_{\text{qk}}^{\text{total}})^2 \cdot d_{\text{v}}^{\text{total}}}{4H^2}. \tag{6}$$

However, the total memory size per layer of SDM would scale as $O(d^3)$, faster than both GDN's state growth ($O(d)$ with GDN heads and O(d^2 without heads) and even faster than the model's parameter count ($O(d^2)$). This would make the sparse memory impractically large at scale. Therefore, H serves as a hyper-parameter with no impact on FLOPs but allowing one to control the state size in SDM.

4 Experimental Setup

Architecture. All models use a hybrid architecture with Multi-Head Attention (MHA) layers using Sliding Window Attention (SWA) and interleaved global receptive field layers in a 3:1 short:long ratio. To avoid harmful competition between the short and long layers (Cabannes et al., 2025), we choose a small window size of 128 tokens which has been adopted for similar reasons in Team et al. (2026) and in OpenAI et al. (2025). All full-attention layers use grouped-query attention (GQA) with group size 2 ($n_{kv} = n_h/2$) and gated attention output (Qiu et al., 2025). Each block uses a gated MLP (Liu et al., 2021) with SiLU activation (Elfwing et al., 2017) and a hidden dimension made to match the parameters of an equivalent non-gated MLP with a hidden dimension of $4d$. Position encoding uses RoPE (Su et al., 2023) with $\theta = 500{,}000$.

SDM Configuration. The SDM layers use $W = R = 64$ reads and writes, and a number of slots $N = (d/4H)^2$ memory slots (Section 3.2). Read and write activations use softmax-normalization. The forget gate parameter A is initialized uniformly in $[0, 16]$ and the time-step bias b_{dt} is initialized from inv_softplus($\mathcal{U}(0.001, 0.1)$), both matching GDN/FLA conventions. The number of SDM heads H is set per scale to keep the state-to-parameter ratio around 1:1 (see Section 3.3 and Table 1).

GDN and Mamba2 Baselines. The GDN and Mamba2 baseline uses $d_{\text{qk}} = 64$, a value dimension $v_{\text{dim}} = 128$, and n_h heads, matching the number of attention heads.

Training. All models were pretrained on 8192-token sequences of diverse text data. Training follows a Warmup–Stable–Decay (WSD) LR schedule with gradient clipping at 1.0 using the AdamW optimizer (Loshchilov & Hutter, 2019) with $\beta_1 = 0.9$ and $\beta_2 = 0.95$. Learning rates were tuned for the transformer baseline and confirmed optimal for GDN via small grid searches. We thus use a uniform LR across all architectures, varying only by model scale (Table 5.1). To maximize recall performance, 1.4B and 8B models undergo a long-context fine-tuning stage on 128k-token sequences using 4B and 16B tokens, respectively.

Scaling Ladder. To evaluate the capability of our approach to scale at larger scale, we train a ladder of model sizes, where we adopt a 160 tokens-per-parameter (160TPP) compute budget. The architectures considered for the ladder are FullAttn, GDN and SDM. We do not rely on optimal token budget following the Chinchilla scaling laws (Hoffmann et al., 2022) because most architectures and models currently deployed at scale are trained way beyond their training-optimal compute budget. We thus choose this 160 tokens per non-embedding parameter budget to compare the architectures in a rather inference-optimal setting. Except for the head number H, GDN, SDM and FullAttn share identical hyperparameters at each level.

Evaluation. We evaluate models using validation NLL on held-out natural text and coding data. We also evaluate them on a diverse set of reasoning and commonsense tasks as listed in Table 2.

Table 1 Scaling ladder configurations. Params include embedding weights but excludes SDM memory state. The State columns correspond to the total state size across all global layers, with St:Param reporting this size as a fraction of the non-embedding parameters.

						GDN		SDM		
Level	d	Layers	Params	Tokens	LR ($\times 10^{-3}$)	State	St:Param	H	State	St:Param
1	768	9	257M	10.7B	1.78	98k	0.16%	1	57M	94%
2	768	11	280M	14.9B	1.50	98k	0.12%	1	57M	68%
3	1024	11	407M	24.3B	1.35	131k	0.09%	1	134M	93%
4	1024	14	465M	34.0B	1.24	197k	0.10%	1	201M	99%
5	1280	14	658M	52.8B	1.20	246k	0.07%	1	393M	119%
6	1536	15	847M	81.0B	0.99	295k	0.07%	1	679M	150%
8	1920	21	1.48B	168.4B	0.87	614k	0.06%	2	553M	56%
13	3840	38	8.14B	1.141T	0.50	2.2M	0.03%	2	7.963B	111%

5 Results

In the following sections, we report SDM performance compared to GDN, Mamba2 and FullAttn as global layers. First, we examine scaling laws across the scaling ladder after pre-training (Section 5.1). Second, we report the performance on short and long-context tasks of the long-context finetuned models (Sections 5.2, 5.3,5.4). Last, we perform extensive ablations verifying SDM architecture's utility and memory use (Section 6).

5.1 Power-Law Scaling: SDM Outperforms GDN at All Compute Levels

We start by evaluating SDM's compute efficiency by measuring how training loss decreases as total training compute (FLOPs) increases. Specifically, we compare SDM against GDN at each level of the scaling ladder (Table 1), keeping both architectures matched in FLOPs and parameters (excluding SDM's sparse embedding memory). We used the final losses of the levels from 1 to 8 to compute scaling laws and predict the loss at 8B (level13) scale. As shown in Figure 3, SDM outperforms GDN at every level of the scaling ladder. Moreover, SDM provides predictable scaling with a correlation coefficient $R^2 = 0.999$. As predicted by our scaling law, when trained at level-13 (8B parameters) scale, SDM reaches a significantly lower loss than GDN (which slightly underperforms its predicted loss but stays within the 95% confidence interval computed using bootstrapping) and even outperforms the 8B model with FullAttn. These results demonstrate that SDM consistently outperforms an iso-FLOP GDN across all compute levels while remaining competitive or even surpassing FullAttn at larger scales.

5.2 Sparse Memory Improves Short-Context Performance

We evaluate how SDM's large sparse memory impacts performance on reasoning and knowledge tasks that do not specifically involve large context capabilities, referred to as "short context". At both 1.4B and 8B scales, SDM achieves lower DCLM NLL and higher average accuracy than GDN, see Table 2. Notably, SDM obtains the lowest DCLM NLL among all models at both scales, outperforming even FullAttn. at 1.4B scale SDM improves over GDN on 13/15 tasks and achieves an average accuracy much higher than GDN and closer to FullAttn. At 8B scale, SDM again improves on most benchmarks and reaches an average accuracy not only higher than GDN but even higher than the model with FullAttn. This underscores the advantage of having a large state containing not only in-context information but also pretraining knowledge in the form of a learned initial state compared to the states of GDN or FullAttn which only store in-context knowledge and not pre-training knowledge.

5.3 Scaling Memory Improves Long-Context Retrieval

RULER results demonstrate that scaling the hidden memory state significantly improves long-context recall. SDM achieves the highest overall scores among fixed-state models at both 1.4B and 8B scales (31.2 and 50.2 respectively), outperforming GDN (20.0 and 34.2) by a wide margin (Table 2). Indeed, at both scale SDM

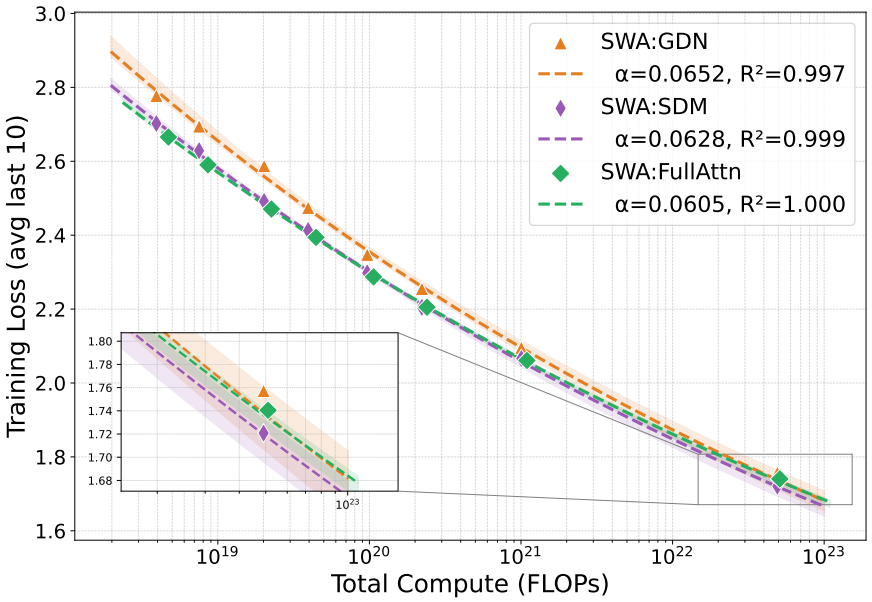


Figure 3 Scaling laws: loss vs FLOPs. Average of last 10 log entries vs total compute. SDM consistently outperforms GDN across all scales. As predicted by the scaling law, SDM outperforms full attention at 8B scale.

improves (or matches in the case of the single1 where GDN is already at 100% accuracy) on 6 out of the 6 RULER tasks we evaluated the models on. At 8B, FullAttn achieves 76.2 accuracy overall thanks to its unbounded KV cache. However, SDM actually matches or exceeds FullAttn on 4 of 6 tasks at 1.4B and 3 of 6 at 8B, despite using a fixed memory representation. On multikey 2 however, FullAttn maintains a large advantage over both SDM and GDN. Detailed results on the obtained performance gap at every sequence length scale are reported in Figures 9 and 10. Overall, the RULER results provide strong evidence that the larger state size enabled by the SDM architecture significantly improves performance on long-context recall.

5.4 SDM Reaches Lower Perplexity with More Context

SDM achieves consistently lower perplexity on code data compared to Mamba2 and GDN (Fig. 4). Even at short sequences (512 tokens), SDM outperforms both baselines, a benefit we attribute to the learned initial state $\mathbf{M_0}$. The advantage grows substantially at long contexts (32k–512k tokens), where SDM's perplexity decreases to near 2.0 while Mamba2 and GDN remain around 2.2–2.3. On this evaluation data, the validation perplexity of all models seems to increase for token positions beyond 256k. However, this is an artifact of the local, token-level perplexity being higher at those positions. When we measure only the perplexity gain contributed by the long-context layers, SDM continues to improve beyond 256k tokens, up to 1 million tokens. This highlights SDM's key strength: its large sparse memory retains information across extremely long sequences, whereas fixed-size state methods suffer from capacity constraints.

6 Ablations: What Makes SDM Work?

Disentangling Memory Capacity from Learned Initialization. To verify that SDM's gains stem from increased memory capacity rather than the learned initial state $\mathbf{M_0}$, we ablate both components. Figure 5 shows that SDM without a learned $\mathbf{M_0}$ substantially outperforms GDN, confirming that state size is the primary driver of performance. Adding a learned $\mathbf{M_0}$ to a vanilla GDN does not measurably improve performance, which is not surprising considering the limited state size. For detailed results, see Table 3. Overall, the ablation confirms that the in-context learning gains come from the increased state size more than from the learned initial state.

Impact of Memory State Size. To confirm that memory size drives SDM's long-context advantage, we ablate the number of memory slots N by varying the PKM key dimension d_{qk} (which controls $N = (d_{\text{qk}}/2)^2$)

Table 2 Performance of different global layer choices at 1.4B (L08) and 8B (L13) scale. All models use SWA as the local layer. Delta column shows delta in performance between SDM and the isoFLOP GDN. Top: Validation NLL, reasoning and common-knowledge tasks. Bottom: exact-match accuracy averaged across lengths 4k–131k on all 13 RULER long-context tasks.

	1.4B (L08)				8B (L13)		
Local layer →		SWA				SWA	
Global layer →	FullAttn	Mamba2	GDN	SDM	FullAttn	GDN	SDM
Validation text NLL ↓	2.660	2.686	2.683	**2.654** -0.029	2.285	2.298	**2.253** -0.040
HellaSWAG (2019) ↑	61.47	61.31	61.27	**62.04** +0.77	79.33	79.10	**80.02** +0.92
WinoGrande (2019) ↑	60.54	**62.67**	61.56	62.12 +0.55	73.64	73.24	**75.30** +2.06
ARC easy (2018) ↑	61.56	62.88	62.49	**63.00** +0.51	78.10	**79.15**	78.52 -0.63
ARC challenge (2018) ↑	33.56	34.08	34.59	**34.76** +0.17	50.82	52.27	**53.05** +0.78
PIQA (2019) ↑	73.83	73.67	73.83	**74.27** +0.44	79.98	**81.01**	80.47 -0.54
OpenBookQA (2018) ↑	35.80	36.40	**36.80**	36.00 -0.80	43.80	43.00	**44.60** +1.60
RACE.mid (2017) ↑	**53.55**	50.28	50.42	49.65 -0.77	**64.83**	60.17	62.05 +1.88
RACE.high (2017) ↑	**39.62**	36.25	36.68	37.76 +1.09	**47.94**	43.40	44.97 +1.57
CommonsenseQA (2019) ↑	19.57	20.31	19.33	**20.72** +1.39	68.88	66.83	**70.60** +3.77
BoolQ (2019) ↑	62.42	62.75	**63.03**	62.78 -0.24	71.83	**74.98**	67.13 -7.85
TQA (2017) ↑	24.24	23.48	23.68	**26.61** +2.93	55.23	55.36	**59.11** +3.75
HumanEval+/pass@1 (2023) ↑	8.54	**9.76**	8.54	**9.76** +1.22	**24.39**	18.29	**24.39** +6.10
NaturalQuestions (2019) ↑	7.70	8.09	7.76	**8.23** +0.47	22.94	22.60	**25.93** +3.33
MMLU (2021) ↑	24.54	25.03	26.02	**27.24** +1.22	**58.73**	57.24	57.81 +0.57
GSM8K (2021) ↑	2.81	**3.03**	2.96	2.73 -0.23	**29.34**	28.81	28.66 -0.15
Average accuracy ↑	37.98	38.00	37.93	**38.51** +0.58	56.65	55.70	**56.84** +1.14
single_1 avg ↑	64.2	53.8	99.9	**100.0** +0.1	99.3	**100.0**	**100.0** +0.0
single_2 avg ↑	53.1	19.2	20.7	**70.8** +50.1	**89.4**	45.1	71.5 +26.4
single_3 avg ↑	41.1	8.0	12.1	**46.3** +34.2	70.1	32.6	**74.9** +42.3
multikey_1 avg ↑	**44.0**	14.7	13.6	35.0 +21.4	**75.3**	32.9	59.3 +26.4
multikey_2 avg ↑	**25.3**	1.0	0.7	0.8 +0.1	**58.0**	1.0	4.7 +3.7
multikey_3 avg ↑	**8.6**	0.2	0.1	0.2 +0.1	**29.8**	0.1	1.2 +1.1
multivalue avg ↑	**39.8**	17.5	11.7	37.4 +25.7	**74.3**	31.1	66.1 +35.0
multiquery avg ↑	39.0	10.5	11.8	**41.1** +29.3	**73.0**	31.2	68.6 +37.4
vt avg ↑	**32.1**	8.6	16.6	23.7 +7.1	67.2	46.2	**72.3** +26.1
cwe avg ↑	5.8	3.6	6.6	**8.7** +2.1	5.5	11.4	**12.8** +1.5
fwe avg ↑	30.9	**37.5**	35.6	12.3 -23.2	**77.8**	62.7	65.0 +2.4
qa_1 avg ↑	**19.3**	11.5	13.3	13.3 +0.0	**39.0**	23.6	27.2 +3.6
qa_2 avg ↑	**20.9**	**18.5**	18.3	18.0 -0.3	**38.4**	28.2	30.4 +2.3
Average accuracy on RULER ↑	32.5	17.9	20.0	**31.2** +11.2	**61.2**	34.2	50.2 +16.0

while keeping W, R, d_v, and all non-SDM layers fixed. Table 3 (bottom) shows monotonic NLL degradation as memory shrinks from 432 MB to 27 MB (0.914 → 0.947), confirming that larger memory state improves modeling. All SDM variants outperform GDN on long-context recall and show monotonic improvement with larger state sizes.

Training Efficiency. A common limitation of sparse approaches is that despite matching dense models in FLOPs, their larger memory footprints incur slower memory accesses in practice. GDN's compact state fits in fast GPU SRAM, while SDM's state must remain in HBM, which has a 10× lower bandwidth than SRAM. This is why the current SDM kernel MFU (Model FLOPs Utilization) is around an order of magnitude lower than the highly optimized GDN kernel from the FLA library (Yang & Zhang, 2024). We expect that many gains remain to be made in developing more efficient SDM kernels that reach an MFU closer to that of GDN. Despite this, thanks to the hybrid attention architecture design, SDM's end-to-end training overhead is modest: at 8B scale, the training throughput of SDM was 1.49x slower than the GDN model. This is remarkable given SDM's state is ∼4,000× larger at that scale. For the 1.4B SDM, inference decode is around

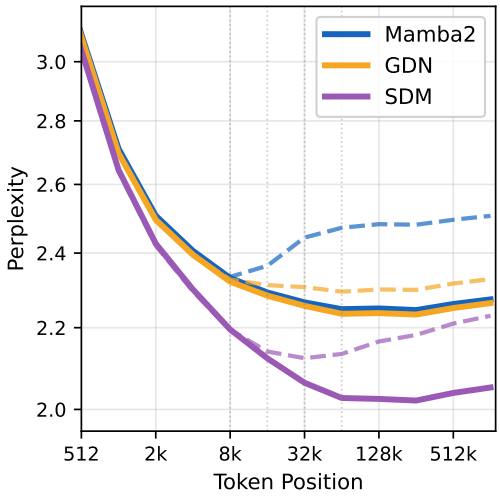


Figure 4 Perplexity by token position on code data (1M token documents, 1.4B model size). Solid: 128k long-context finetuned, dashed: pre-trained. Local layers are SWA for all models.

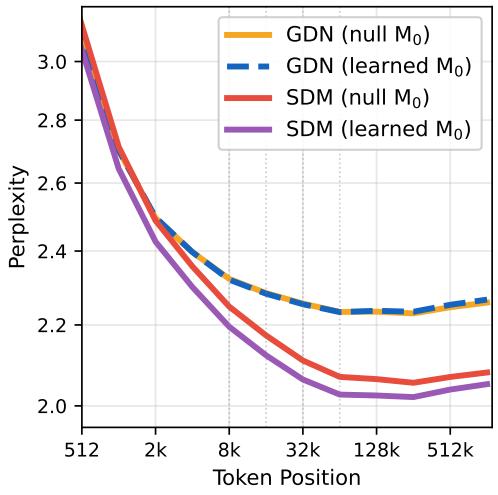


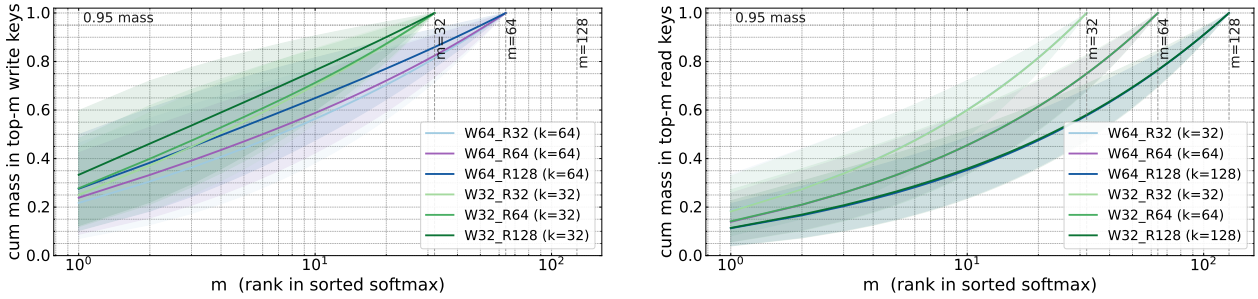
Figure 5 Learned initial state ablation on code data (1M token documents, 1.4B model size, post-trained). PPL by token position. Learning $\mathbf{M_0}$ benefits SDM but not GDN.

	Model	$\mathbf{M_0}$	State/layer	Code NLL ↓	Acc↑	RULER↑
ablation: learned initial state (1.4B)	GDN	null	0.2 MB	0.849	37.9	20.0
	GDN	learned	0.5 MB	0.850	38.0	20.6
	SDM	null	211 MB	0.845	37.3	28.0
	SDM	learned	211 MB	**0.822**	**38.5**	**31.2**
ablation: memory size (0.8B)	GDN	null	0.2 MB	0.963	33.8	16.0
	SDM	learned	27 MB	0.947	33.7	20.2
	SDM	learned	108 MB	0.937	33.6	20.7
	SDM	learned	432 MB	**0.914**	**34.6**	**21.5**

Table 3 SDM ablations. *Top*: learned initial state ablation (L08, 1.4B). Making the memory $\mathbf{M}_0$ a learnable parameter improves code-data NLL, accuracy (%), and RULER recall (%). *Bottom*: memory size ablation (L06). SDM degrades gracefully when memory is reduced; even at 27 MB it outperforms GDN on RULER.

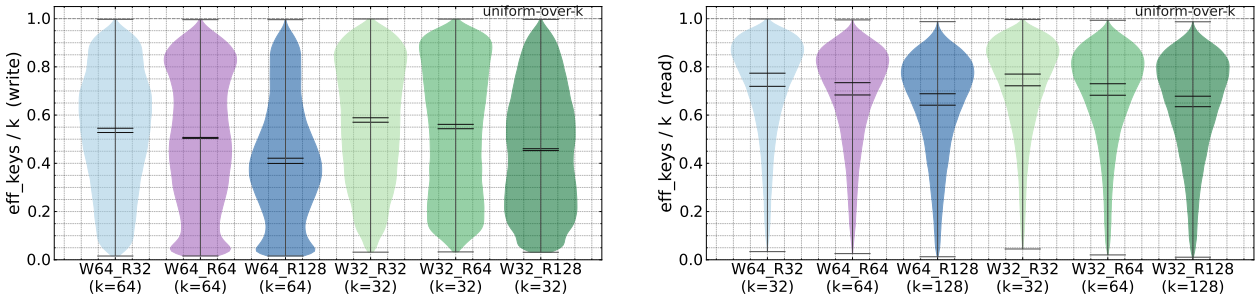
10% slower than GDN but 6 times faster than FullAttn.

Adaptive Memory Access Patterns. The memory slot utilization is controlled by the softmax over read and write keys. We analyze the peakiness of these distributions to assess whether all keys are actively used. We pre-train SWA:SDM 1.4B models with varying read/write configurations ($W \in \{32, 64\}$, $R \in \{32, 64, 128\}$) and extract softmax values per token across all layers and heads (for details see Appendix D.1). First, we compute cumulative mass in the top-m keys: $C(m) = \sum_{i=1}^{m} p_{(i)}$ where $p_{(1)} \geq p_{(2)} \geq \cdots \geq p_{(k)}$. Figure 6 shows that write distributions are moderately peaked: with $k = 64$ writes, the top 32 keys capture ∼85% of probability mass. Read distributions are more uniform: with $k = 128$ reads, the top 64 keys hold only ∼77% of mass, indicating broader access patterns. Second, we compute the effective number of keys per token as $u(p) = \exp(H(p))/k \in (0, 1]$, which measures utilization independent of k. Figure 7 reveals distinct read/write behaviors: reads are consistently more uniform than writes across all configurations. Increasing reads to 128 makes the distribution *less* uniform (more selective), while decreasing writes to 32 makes writes *more* uniform. Notably, read and write distributions adapt to each other: with limited reads ($R = 32$), writes become more uniform to compensate; with abundant reads ($R = 128$), writes become more peaked. This demonstrates SDM's adaptive memory access, the model dynamically balances read selectivity against write dispersion based on available capacity. For a more detailed performance comparison across runs, we refer to reader to Table 4 in Appendix D .



(a) Write keys **(b)** Read keys

Figure 6 Coverage of the top-$\mathbf{m}$ selected memory keys. In each panel, the solid line is the layer-averaged mean cumulative mass curve across all tokens; the shaded band shows the corresponding $[p10, p90]$ range. Panel (a) shows write keys and panel (b) shows read keys. Write keys exhibit less uniform distribution than read.



(a) Write keys **(b)** Read keys

Figure 7 Per-token effective-key utilization: analysis of $\mathbf{eff\_keys}/k$. For each ablation, each violin is built from all pooled token-level values for each run; the overlaid markers indicate the mean and median. Panel (a) shows write keys and panel (b) shows read keys. Read keys have a more uniform distribution than write keys, while the write key distribution varies more with the assigned read and write key budget, indicating that SDM memory access adapts to the given key constraints.

7 Conclusion

In this work, we introduced Sparse Delta Memory, a sparse extension of the Gated DeltaNet architecture based on a Product-Key Memory sparse design. This sparsity offers state sizes thousands of times larger than GDN while keeping the FLOPs identical. Thanks to this much larger state size, SDM reaches much better training loss and NLL than GDN and even than Full Attention. It also demonstrates much better long-context performance as measured across a wide range of long-context tasks from the RULER benchmark. Finally, we provide ablations showing which gains are attributed to the larger state size and which are obtained from learning the initial state of the large memory.

Limitations. Although our implementation of SDM gives a training speed allowing us to scale to 8B models, more research is needed to design more efficient kernels to further scale the SDM models. The main limitation of SDM is also its strength: SDM memory requirements are not negligible, as the memory footprint may be as large as the model parameters, which is not adapted to certain resource-constrained contexts. However, the KV cache memory usage of FullAttn models is significant when processing sequence lengths of hundreds of thousands of tokens. More precisely, the SDM state for the 8B model occupies as much memory as 203400 tokens in the KV cache of the 8B FullAttn model.

References

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff, 2025. URL https://arxiv.org/abs/2402.18668.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory, 2024. URL https://arxiv.org/abs/2405.04517.

Vincent-Pierre Berges, Barlas Oğuz, Daniel Haziza, Wen tau Yih, Luke Zettlemoyer, and Gargi Ghosh. Memory layers at scale, 2024. URL https://arxiv.org/abs/2412.09764.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language, 2019. URL https://arxiv.org/abs/1911.11641.

Loïc Cabannes, Maximilian Beck, Gergely Szilvasy, Matthijs Douze, Maria Lomeli, Jade Copet, Pierre-Emmanuel Mazaré, Gabriel Synnaeve, and Hervé Jégou. Short window attention enables long-term memorization, 2025. URL https://arxiv.org/abs/2509.24552.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions, 2019. URL https://arxiv.org/abs/1905.10044.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018. URL https://arxiv.org/abs/1803.05457.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning (ICML)*, 2024.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, 2017. URL https://arxiv.org/abs/1702.03118.

Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models, 2023. URL https://arxiv.org/abs/2212.14052.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL https://aclanthology.org/P17-1147/.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention, 2020. URL https://arxiv.org/abs/2006.16236.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering

research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations, 2017. URL https://arxiv.org/abs/1704.04683.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys, 2019. URL https://arxiv.org/abs/1907.05242.

Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay attention to MLPs, 2021. URL https://arxiv.org/abs/2105.08050.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

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

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Gated attention for large language models: Non-linearity, sparsity, and attention-sink-free, 2025. URL https://arxiv.org/abs/2505.06708.

Jack W Rae, Jonathan J Hunt, Tim Harley, Ivo Danihelka, Andrew Senior, Greg Wayne, Alex Graves, and Timothy P Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes, 2016. URL https://arxiv.org/abs/1610.09027.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019. URL https://arxiv.org/abs/1907.10641.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers, 2021. URL https://arxiv.org/abs/2102.11174.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (learn at test time): RNNs with expressive hidden states, 2025. URL https://arxiv.org/abs/2407.04620.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge, 2019. URL https://arxiv.org/abs/1811.00937.

Core Team, Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, Gang Xie, Hailin Zhang, Hanglong Lv, Hanyu Li, Heyu Chen, Hongshen Xu, Houbin Zhang, Huaqiu Liu, Jiangshan Duo, Jianyu Wei, Jiebao Xiao, Jinhao Dong, Jun Shi, Junhao Hu, Kainan Bao, Kang Zhou, Lei Li,

Liang Zhao, Linghao Zhang, Peidian Li, Qianli Chen, Shaohui Liu, Shihua Yu, Shijie Cao, Shimao Chen, Shouqiu Yu, Shuo Liu, Tianling Zhou, Weijiang Su, Weikun Wang, Wenhan Ma, Xiangwei Deng, Bohan Mao, Bowen Ye, Can Cai, Chenghua Wang, Chengxuan Zhu, Chong Ma, Chun Chen, Chunan Li, Dawei Zhu, Deshan Xiao, Dong Zhang, Duo Zhang, Fangyue Liu, Feiyu Yang, Fengyuan Shi, Guoan Wang, Hao Tian, Hao Wu, Heng Qu, Hongfei Yi, Hongxu An, Hongyi Guan, Xing Zhang, Yifan Song, Yihan Yan, Yihao Zhao, Yingchun Lai, Yizhao Gao, Yu Cheng, Yuanyuan Tian, Yudong Wang, Zhen Tang, Zhengju Tang, Zhengtao Wen, Zhichao Song, Zhixian Zheng, Zihan Jiang, Jian Wen, Jiarui Sun, Jiawei Li, Jinlong Xue, Jun Xia, Kai Fang, Menghang Zhu, Nuo Chen, Qian Tu, Qihao Zhang, Qiying Wang, Rang Li, Rui Ma, Shaolei Zhang, Shengfan Wang, Shicheng Li, Shuhao Gu, Shuhuai Ren, Sirui Deng, Tao Guo, Tianyang Lu, Weiji Zhuang, Weikang Zhang, Weimin Xiong, Wenshan Huang, Wenyu Yang, Xin Zhang, Xing Yong, Xu Wang, Xueyang Xie, Yilin Jiang, Yixin Yang, Yongzhe He, Yu Tu, Yuanliang Dong, Yuchen Liu, Yue Ma, Yue Yu, Yuxing Xiang, Zhaojun Huang, Zhenru Lin, Zhipeng Xu, Zhiyang Chen, Zhonghua Deng, Zihan Zhang, and Zihao Yue. Mimo-v2-flash technical report, 2026. URL https://arxiv.org/abs/2601.02780.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/fla-org/flash-linear-attention.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length, 2024. URL https://arxiv.org/abs/2406.06484.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule, 2025. URL https://arxiv.org/abs/2412.06464.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019. URL https://arxiv.org/abs/1905.07830.

Tianyuan Zhang, Sai Bi, Yicong Hong, Kai Zhang, Fujun Luan, Songlin Yang, Kalyan Sunkavalli, William T. Freeman, and Hao Tan. Test-time training done right, 2025. URL https://arxiv.org/abs/2505.23884.

Tianyu Zhao and Llion Jones. Fast-weight product key memory, 2026. URL https://arxiv.org/abs/2601.00671.

Appendices

A Efficient Training of SDM

Training SDM requires computing exact outputs and gradients for the gated delta rule (Eq. 4–5) across long sequences. We decompose this into *intra-chunk parallel* computation (batched across all chunks) and *chunkwise recurrent* computation (sequential across chunks), following the WY representation approach from GDN (Yang et al., 2024).

A.1 Intra-Chunk Parallel vs. Chunkwise Recurrent

We split the sequence into chunks of size C. Within each chunk, the gated delta rule creates causal dependencies between tokens—token t's memory read depends on all prior writes within the same chunk. The WY representation (Yang et al., 2024) resolves these dependencies analytically via a triangular solve, enabling parallel computation within each chunk.

Phase 1: Intra-chunk parallel (batched). All chunks are processed simultaneously. Let $\mathbf{V} \in \mathbb{R}^{C\times d}$ denote the stacked value vectors for a chunk, and $\boldsymbol{\beta} \in \mathbb{R}^{C}$ the per-token input gates. For each chunk, we compute:

- **Segmented cumulative decay.** For each slot i accessed in the chunk, accumulate the log-decay $\log \alpha_t$ across tokens that write to slot i. This is a segmented prefix sum over slot indices, implemented via scatter-add in log space: for each entry (t, w) with slot index i, we atomically add $\log \alpha_t$ to an accumulator indexed by i, yielding the cumulative log-decay $\lambda_{t,w} = \sum_{t'\le t,\, \mathcal{I}^w_{t'}\ni i} \log \alpha_{t'}$.
- **Sparse interaction matrices $\mathbf{A}$ and $\mathbf{QK}$.** We define two $C \times C$ lower-triangular matrices via the gated sparse inner product (Eq. 7): (i) $\mathbf{A}[i, j]$, the *write–write* self-interaction using write keys on both sides, capturing how token j's write to shared slots affects token i's delta-rule subtraction; and (ii) $\mathbf{QK}[i, j]$, the *read–write* cross-interaction using read queries for token i and write keys for token j, capturing how token j's write affects token i's read output. Both are extremely sparse since two tokens interact only if their top-W slot indices overlap ($W \ll N$). We compute them via a *sparse inner product* kernel (Section A.2).
- **Triangular solve.** Construct the lower-triangular system $\mathbf{M}_{\text{sys}}[i, j] = \delta_{ij} + \beta_i \cdot \mathbf{A}[i, j]$ for $j < i$, and solve $\mathbf{M}_{\text{sys}}\, \Delta\mathbf{V}_{\text{const}} = \text{diag}(\boldsymbol{\beta}) \cdot \mathbf{V}$ via `solve_triangular`. This yields the intra-chunk delta-v values assuming reads from the chunk-initial memory state. We also derive the correction matrix $\mathbf{B} = \mathbf{M}^{-1}_{\text{sys}} \cdot \text{diag}(-\boldsymbol{\beta})$, which accounts for intra-chunk memory modifications.

All operations above are batched over chunks (one `bmm`/`solve` call for all chunks simultaneously), achieving high GPU utilization.

Phase 2: Chunkwise recurrent (sequential). Chunks are processed one at a time, since each chunk's memory writes depend on the previous chunk's memory state. Per chunk:

1. **Read:** Gather memory (from HBM) at write indices to obtain $\texttt{retrieved}[t] = \sum_n k^{(n)}_{\text{eff},t} \cdot \mathbf{M}[\mathcal{I}^w_t[n]]$ (where $k_{\text{eff}} = k_{\text{val}} \cdot e^{\lambda}$ incorporates cumulative decay), and at read indices to obtain the inter-chunk output $\texttt{inter}[t] = \sum_n q^{(n)}_{\text{eff},t} \cdot \mathbf{M}[\mathcal{I}^r_t[n]]$.
2. **Correct:** Compute the full delta-v incorporating the current memory state: $\delta\mathbf{v} = \Delta\mathbf{V}_{\text{const}} + \mathbf{B} \cdot \texttt{retrieved}$. The correction term $\mathbf{B} \cdot \texttt{retrieved}$ accounts for the fact that earlier tokens in the chunk have already modified the memory that later tokens read from.
3. **Write:** Apply per-slot decay and scatter the delta-v contributions back to memory.
4. **Output:** Compute the final chunk output combining inter-chunk reads with intra-chunk corrections: $\mathbf{y} = \texttt{inter} + \text{tril}(\mathbf{QK}) \cdot \delta\mathbf{v}$.

The sequential Phase 2 is memory-bound: it is dominated by random gathers and scatters to the $N \times d$ memory table. Phase 1 is compute-bound and runs once for all chunks. This decomposition keeps the total compute at $O(T \cdot (W^2 + W \cdot d))$ per layer, while the sequential bottleneck scales as $O((T/C) \cdot W \cdot d)$ random memory accesses.

A.2 Sparse Inner Product via Two-Pointer Merge

The interaction matrices $\mathbf{A}[i, j]$ and $\mathbf{QK}[i, j]$ measure how much tokens i and j interact through shared memory slots. For GDN with dense keys, these are standard matrix products ($\mathbf{A} = \mathbf{K}\mathbf{K}^\top$). For SDM with sparse keys, most entries are zero—two tokens interact only if their top-W slot indices overlap.

We compute the *gated sparse inner product*:

$$\mathbf{A}[i, j] = \sum_{n,m \,:\, \mathcal{I}_i^w[n] = \mathcal{I}_j^w[m]} k_i^{(n)} \cdot k_j^{(m)} \cdot \exp(\lambda_{i,n} - \lambda_{j,m}), \quad j < i \tag{7}$$

where the sum ranges only over matching slot indices between tokens i and j, and λ are the cumulative log-decays.

Two-pointer algorithm. The PKM top-W selection includes an additional sort-by-index step ($O(W \log W)$, which does not change the overall PKM complexity of $O(\sqrt{N} \cdot d + W^2)$). The resulting indices are thus already sorted by slot index, enabling the classical two-pointer merge to find matching slots between two tokens in $O(W)$ time rather than $O(W^2)$:

Input: Sorted indices $\mathcal{I}_i[0..W-1]$, $\mathcal{I}_j[0..W-1]$; values k_i, k_j; log-decays λ_i, λ_j
$a, b \leftarrow 0, 0$; result $\leftarrow 0$
while $a < W$ **and** $b < W$ **do**
  if $\mathcal{I}_i[a] = \mathcal{I}_j[b]$ **then**
    result $+= k_i^{(a)} \cdot k_j^{(b)} \cdot \exp(\lambda_i^{(a)} - \lambda_j^{(b)})$
    $a \mathrel{+}= 1$; $b \mathrel{+}= 1$
  else if $\mathcal{I}_i[a] < \mathcal{I}_j[b]$ **then**
    $a \mathrel{+}= 1$
  else
    $b \mathrel{+}= 1$
return result

This is implemented as a CUDA kernel with one thread per (i, j) pair in the $C \times C$ interaction matrix, yielding $O(C^2 \cdot W)$ total work. Only the *causal* entries ($j < i$ for $\mathbf{A}$, $j \leq i$ for $\mathbf{QK}$) are computed; non-causal entries are zero.

Compared to a dense inner product ($O(C^2 \cdot d_{\text{qk}})$ for GDN), the sparse version scales as $O(C^2 \cdot W)$ where $W = 64 \ll N$, making it significantly cheaper than addressing the full memory and enabling SDM to scale to large memory tables without increasing the interaction cost.

A.3 Memory-Efficient Backward

Following Sparse Access Memory (Rae et al., 2016), we apply sparse updates to the memory *in-place* during the forward pass and *undo* these sparse updates during the backward pass to recover the correct memory state for each timestep. This avoids storing a full copy of the $N \times d$ memory at each chunk boundary, reducing peak memory from $O((T/C) \times N \times d)$ to $O(N \times d + T \times W \times d)$. for example at 16384 token training sequence length, and for an 8B training, chunk size $C = 128$ $W = 64$, this reduces memory consumption for the memory state checkpoints of an SDM layer from 226 GB to 8GB. Nonetheless, for longer sequence length such as 128k long-context finetuning, this memory snapshot can become prohibitively large. We therefore investigated quantization of the snapshot to fp8 and int4 and found that both quantization levels seem to have no detrimental on training loss or end performance and therefore recommend it for long-context finetuning.

B SDM Memory Utilization During Training

Figure 8 shows the evolution of key memory access statistics during training of the SDM model at 1.4B scale (L08). All metrics are averaged across the 5 SDM layers.

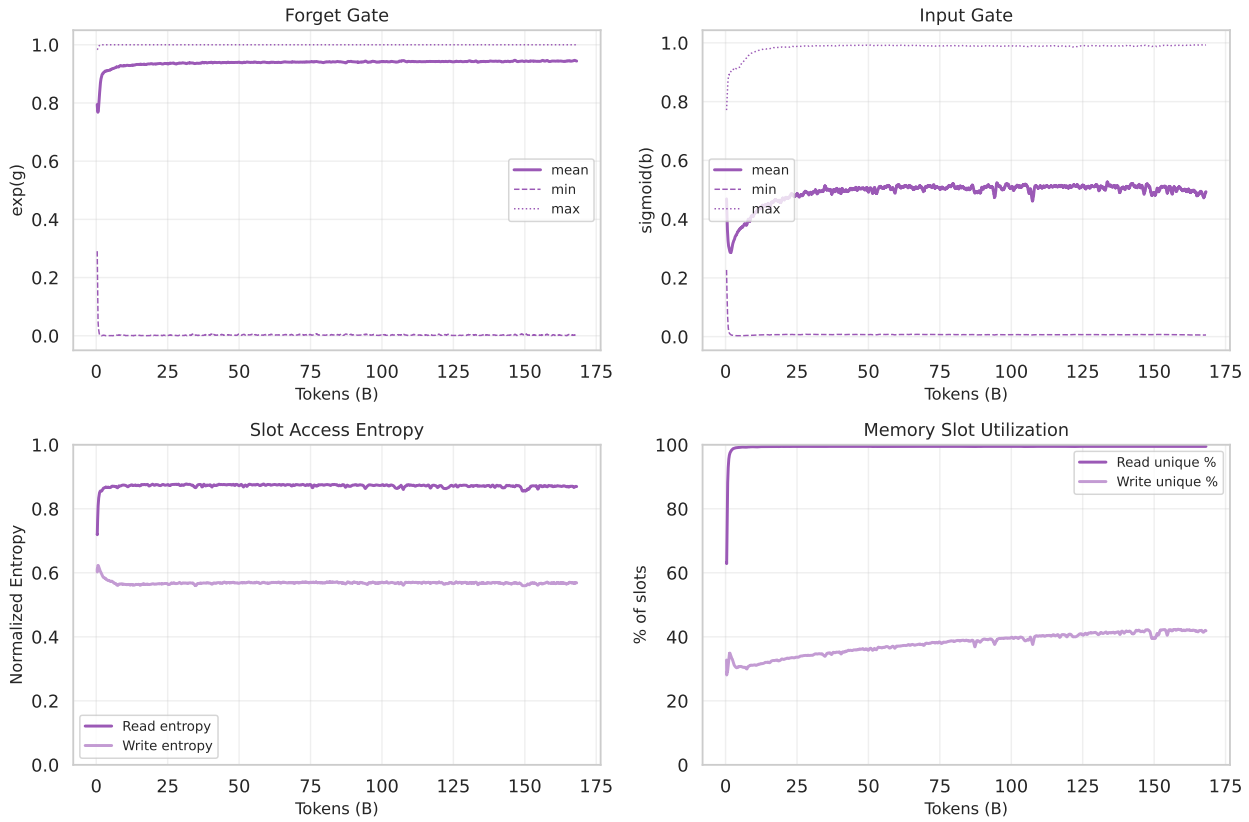


Figure 8 SDM memory utilization during training (1.4B, L08). Statistics are accumulated over 10 training steps (∼21M tokens) per data point and averaged across 5 SDM layers. *Top*: forget and input gate statistics (min/mean/max). *Bottom left*: normalized entropy of slot access distributions. *Bottom right*: percentage of unique memory slots accessed.

The forget gate (top left) converges early to $\exp(g) \approx 0.95$. The minimum forget gate stays near zero, indicating that some slots are fully decayed when overwritten. The input gate (top right) stabilizes at $\sigma(b) \approx 0.50$, with the maximum and minimum reaching ∼1 and ∼0 respectively. Overall, the model learns a wide dynamic range for both forget and input strength, showing the importance of both mechanisms in the model.

The read utilization (bottom right) quickly reaches 100% of the memory, showing that no memory slot is left unutilized. The write utilization (bottom right) grows steadily from 28% to 42% during training, suggesting that the model learns to diversify its write patterns over time.

C RULER Per-Task Accuracy by Sequence Length

Figure 9 and Figure 10 show the per-task RULER accuracy at each sequence length for different architectures at 1.4B scale and 8B scale respectively. All 13 RULER tasks are shown. Solid lines denote post-trained models (128k finetuned); dashed lines denote pre-trained models.

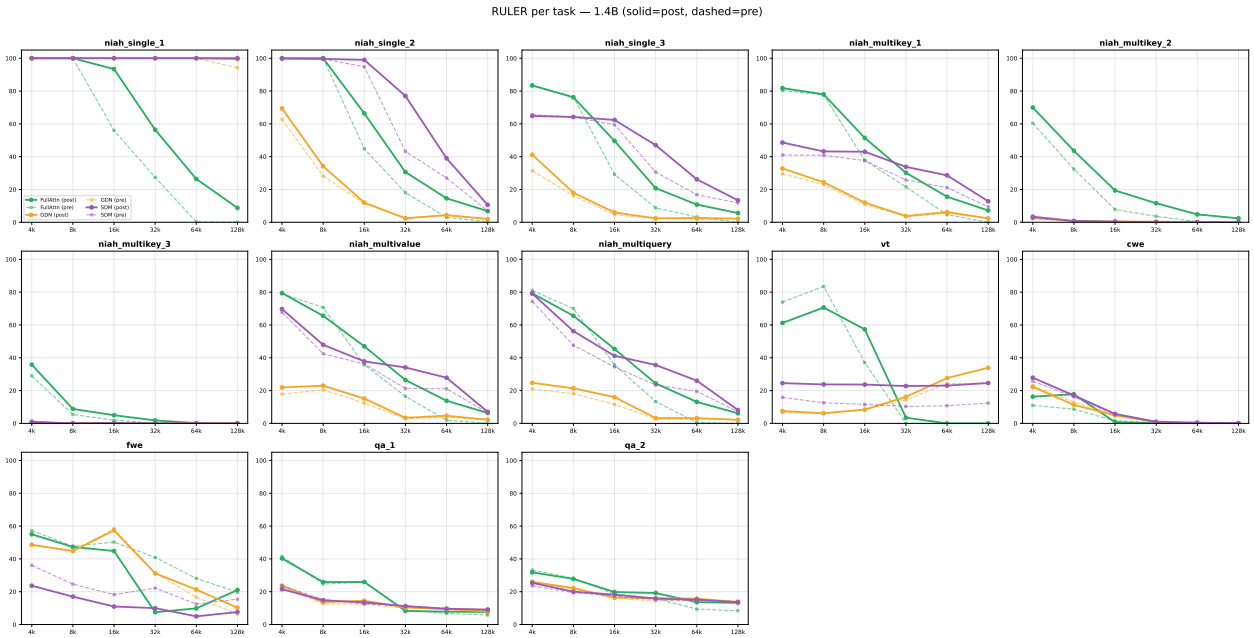


Figure 9 RULER per-task accuracy (%) by sequence length at 1.4B scale. Solid lines: after 128k post-training. Dashed lines: pre-trained only. SDM maintains strong recall on NIAH single-needle tasks across all lengths, while Full Attention degrades rapidly beyond its training length. GDN struggles on multi-step retrieval (single_2, single_3, multiquery). For some reason on the variable tracking task, GDN performs better and better on longer sequence lengths. We believe this to be due to the design on the vt task which clusters distractor variables early in the sequence, unintentionally making their influence weaker and weaker the longer the sequence is.

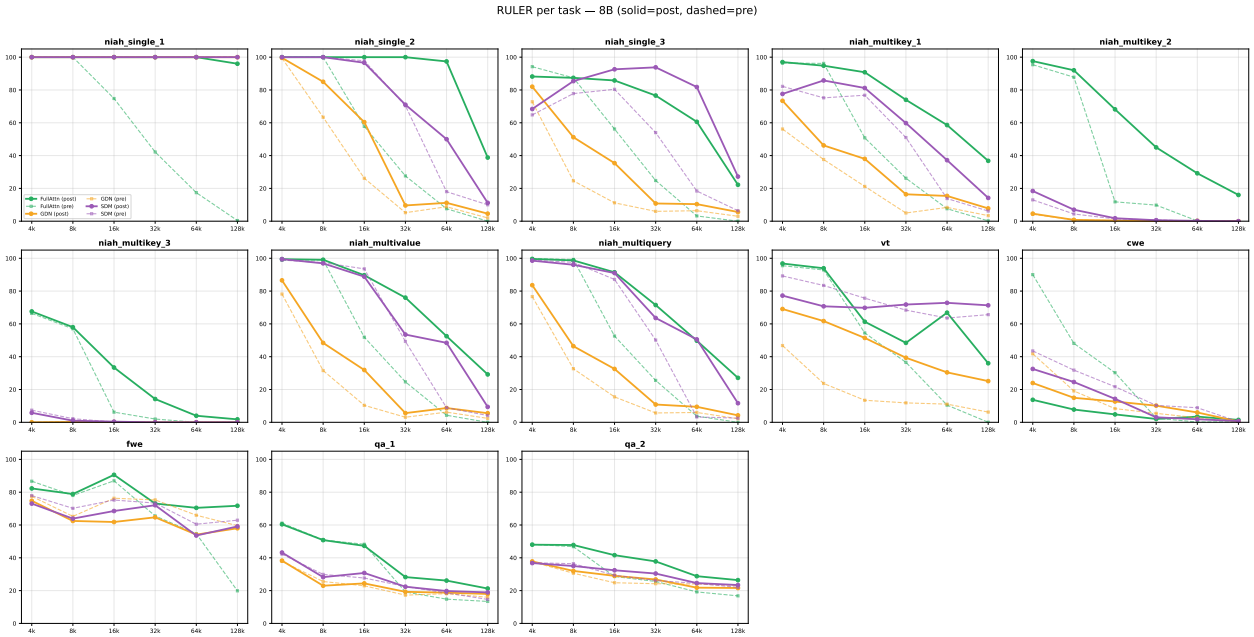


Figure 10 RULER per-task accuracy (%) by sequence length at 8B scale. Solid lines: after 128k post-training. Dashed lines: pre-trained only. At 8B, post-training dramatically improves FullAttn (green) which achieves near-perfect recall on single-needle tasks. SDM maintains strong recall and outperforms FullAttn on single_3 and vt tasks. The multikey_2 task remains challenging for all fixed-state models.

D Ablations

D.1 Memory Utilization and Access

The values are collected by an offline probe run on each ablation model's final checkpoint using a fixed held-out batch from coding data. Concretely, for every run, the script reads the first $B = 8$ documents whose tokenized length is at least $T = 8192$, truncates each document to 2048 tokens, and performs a single deterministic forward pass on this 8×8192 batch. During this pass, forward hooks are attached to every memory-controller layer to capture the post-softmax weights over the selected top-k memory slots for each token and attention head, with $k = R$ on the read side and $k = W$ on the write side. All reported statistics are then computed from these per-token top-k distributions and aggregated across tokens, heads, and layers.

Cumulative Mass in top-k Keys For the cumulative mass in the top-m keys, let $p \in \mathbb{R}^k$ denote the post-softmax distribution over the selected keys for a single token, and let $p_{(1)} \geq p_{(2)} \geq \cdots \geq p_{(k)}$ be the same values sorted in descending order. We then define the cumulative mass curve as

$$C(m) = \sum_{i=1}^{m} p_{(i)}, \qquad m = 1, \ldots, k.$$

This quantity measures how quickly the probability mass concentrates in the highest-weighted keys: if $C(m)$ rises rapidly toward 1, then only a few keys account for most of the mass and the distribution is highly peaked; if it rises more slowly, the controller is making broader use of its available top-k budget. In the notebook, we summarize this curve by reporting its mean and token-level quantiles across all captured $(\text{batch} \times \text{head} \times \text{time})$ positions.

Effective Number of Keys per Token For the effective-keys calculation, we compute the entropy of the per-token top-k distribution,

$$H(p) = -\sum_{i=1}^{k} p_i \log p_i,$$

and convert it into an entropy-based effective number of keys,

$$\text{eff\_keys}(p) = \exp(H(p)).$$

This is equal to 1 for a delta-like distribution concentrated on a single key and approaches k when the distribution is close to uniform over the selected budget. To make the quantity directly comparable across runs with different values of k, we normalize it as

$$u(p) = \frac{\text{eff\_keys}(p)}{k} \in (0, 1].$$

Thus, $u(p) \approx 1$ indicates near-uniform use of the available top-k slots, whereas $u(p) \ll 1$ indicates that the controller effectively relies on only a small fraction of them.

Performance across different read and write key pairs for SDM All values are reported on 1.4B model scale after pre-training on 8k sequence length.

Run	DCLM NLL ↓	Δ	Avg. RULER ↑	Δ	Avg. Reasoning ↑	Δ
W64_R128	**2.6703**	-0.0016	0.4011	-0.0164	0.3842	-0.0026
W64_R64	2.6719	0.0000	**0.4176**	0.0000	0.3868	0.0000
W32_R64	2.6726	+0.0007	0.3501	-0.0675	**0.3890**	+0.0022
SWA:FullAttn	2.6729	+0.0010	0.3474	-0.0702	0.3805	-0.0063
W32_R128	2.6737	+0.0018	0.4039	-0.0136	0.3821	-0.0047
W64_R32	2.6746	+0.0027	0.3764	-0.0412	0.3887	+0.0019
W32_R32	2.6766	+0.0047	0.3612	-0.0564	0.3871	+0.0003

Table 4 Comparison across varying reads (R) and writes (W). Deltas are computed w.r.t. `W64_R64`. Lower is better for DCLM NLL; higher is better for Avg. RULER and Avg. Reasoning. Bold indicates the best value in each non-delta metric column. RULER is averaged across 6 sub-tasks for computational reasons (niah_single 1/2/3, multikey 2, multiquery, vt).

E Mixed FullAttn + SDM Hybrid

We explore a three-way hybrid architecture combining SWA (local), FullAttn (global), and SDM (global) layers within the same model. In this configuration, global layer positions alternate between FullAttn and SDM: for 5 global layers, the pattern is FA, SDM, FA, SDM, FA. All other layers remain SWA with a window of 128 tokens. We denote this architecture SWA:(FA/SDM).

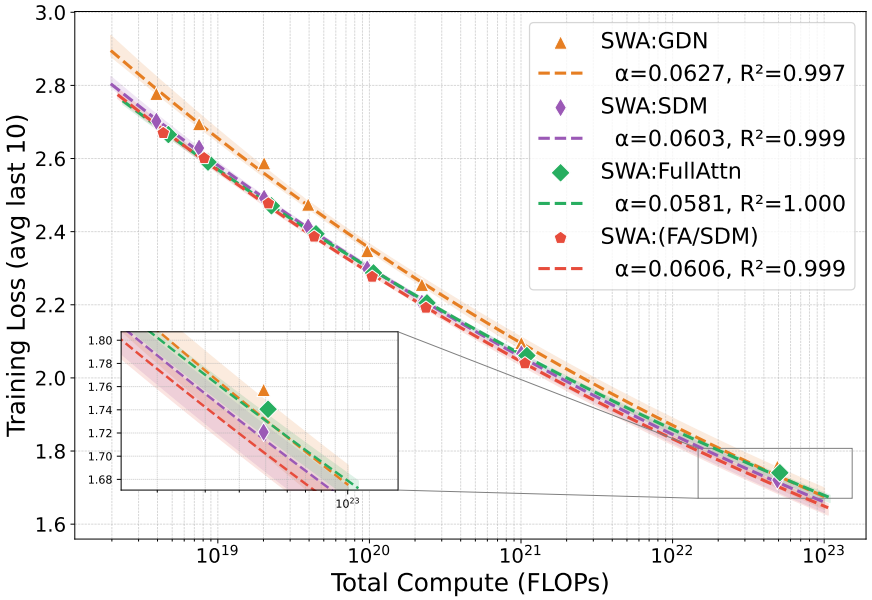


Figure 11 Training loss scaling law for four architectures at levels 1–8.

As shown in Figure 11, the 3-way hybrid architecture outperforms both 2-way hybrids at larger scale thanks to it better scaling coefficient. This hints at the fact that SDM and FullAttn might provide complementary capabilities and hints at the possibility of augmenting existing local-global architectures with SDM layers.

F Compute Resources

All experiments in this work were conducted using NVIDIA H100 Hopper 80GB GPUs using the latest stable PyTorch release available at run time. We estimate the GPU-hour usage of this work for pre-training, post-training, ablations and evaluation to be around 200k gpu hours in total.

	SWA:FA	SWA:(FA/SDM)	SWA:SDM	SWA:GDN
Validation text data NLL ↓	2.658	**2.645**	2.654	2.684
Validation code data NLL ↓	0.798	**0.797**	0.821	0.849
Avg. Accuracy (%) ↑	38.1	**38.6**	**38.6**	37.9
RULER (6-task, 4k–8k) ↑	**80.1**	64.3	56.9	32.9

Table 5 Performance of the mixed SWA:(FA/SDM) hybrid at 1.4B scale (L08, pre-trained). RULER is averaged across 6 tasks (niah_single 1/2/3, multikey_2, multiquery, vt) at sequence lengths 4k–8k (within training length).