DEPTH MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Residual connections propagate information over depth by additively accumulating layer outputs. Recently proposed alternatives, such as Manifold-Constrained HyperConnections (mHC) and Attention Residuals (AttnRes), instead use more expressive depth aggregation mechanisms that improve language modeling performance at scale. In this work, we consider the mixing of representations over depth a problem of storing and retrieving information across layers. Building on this perspective, we propose *Depth Memory (DM)*, a framework that interprets the standard transformer, as well as mHC and AttnRes, as propagating information forward using different streams. One such stream can be a memory stream, which is queried to enrich the input to every layer, and whose output is subsequently written back into the memory for use by later layers. For AttnRes, this memory is a growing cache of explicit layer representations. For mHC, it is an implicit representation stored in an expanded stream. We then instantiate DM as a *Linear Depth Memory (LDM)*, which propagates forward three streams: a standard residual stream, a memory stream, and an inverse covariance stream. We formulate the dynamics of the memory stream as a depth regression problem that is solved online via recursive least squares. We furthermore propose a hardware-aware implementation of LDM that reduces peak activation memory and memory traffic, enabling high-throughput training and inference. LDM consistently achieves the lowest perplexities among the evaluated PreLN, mHC and AttnRes baselines at the 400M and 1.5B parameter scales, alongside competitive zero-shot accuracy, showcasing the practical utility of our framework.

1 INTRODUCTION

The transformer architecture underpins modern language modeling (Vaswani et al., 2023). Since its introduction, architecture research has mainly seen innovation along two axes: sequence mixing, which includes softmax attention (Vaswani et al., 2023), linear attention (Katharopoulos et al., 2020), and state-space models (Gu et al., 2022), and channel mixing, through works like gated MLPs (Shazeer, 2020) and MoEs (Fedus et al., 2022). The standard mechanism for propagating information across depth, however, has remained largely unchanged since the introduction of ResNet (He et al., 2016): layer outputs are accumulated additively through residual connections. Recent works have established alternatives to this standard mechanism at scale (Luo et al., 2026a; DeepSeek-AI et al., 2026). For example, HyperConnections (HC, Zhu et al. (2025)) and Manifold-Constrained HyperConnections (mHC, Xie et al. (2026)) maintain an expanded residual stream, while Attention Residuals (AttnRes, Team et al. (2026)) explicitly attend to previous layer outputs (Figure 2b). Analogously to sequence and feature mixing operators, we refer to these methods as *depth mixing* mechanisms.

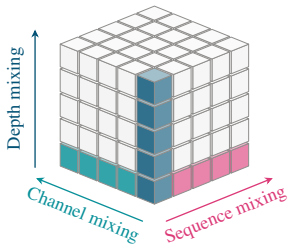
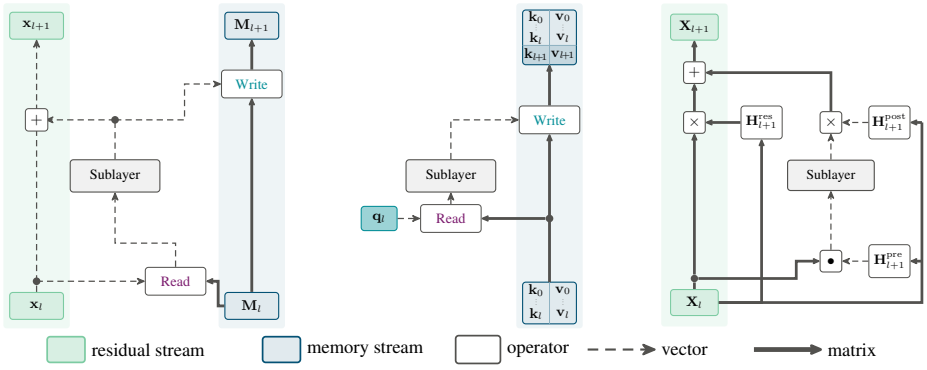


Figure 1: The axes of representation mixing: sequence (tokens), channel (features), and depth (layers).

In this work, we approach depth mixing from a different angle. Viewing it as the problem of storing and retrieving information across layers, we introduce *Depth Memory*, a framework that interprets the propagation of information over depth as a problem of managing layer representations using a set of *streams*. At each layer, and independently for each token, a Read operation forms the layer input from these streams, and a Write operation updates the streams using the layer

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) LDM (ours) (b) AttnRes (c) mHC

Figure 2: (a) LDM carries a separate per-token depth memory alongside the residual stream. Read includes the residual addition; Write shows only the memory update. The inverse covariance stream and the write-address generation are omitted. (b) AttnRes reads from a growing key–value memory $\mathbf{M}_l$ and writes the sublayer output back into it. (c) mHC generates its three mappings from the current expanded state and applies them around one transformer branch.

output. This framework encompasses PreLN's residual stream, AttnRes's growing cache of layer representations, and mHC's expanded residual stream, covering both explicit and implicit forms of depth mixing (Figure 2).

Besides taxonomizing existing residual methods using our depth memory framework, we propose instantiating it as a *Linear Depth Memory (LDM)*. LDM maintains three streams: the standard residual stream, a fixed-size matrix-valued memory stream, and an inverse covariance stream (Figure 2a). Analogous to test-time regression for sequence mixing (Wang et al., 2025), we formulate linear associative memory updates as the solution to a *Depth Regression* problem, which fits a linear map to reconstruct previous layer outputs from their corresponding keys. LDM solves this optimization problem optimally through recursive least squares (RLS, Section 3.2).

Our hardware-aware implementation of LDM addresses efficiency challenges posed by a naive instantiation of the framework. First, we implement an unwinding mechanism that does not require storing each layer's memory stream. Second, we employ a windowed Write algorithm that defers parts of associative memory updates. Implemented as efficient CuTeDSL kernels, LDM achieves higher throughput than full AttnRes and mHC and is competitive with block AttnRes (Section 3.3).

We evaluate LDM on language modeling benchmarks across 400M Llama-like (Touvron et al., 2023) and 1.5B Qwen-like (Yang et al., 2025a) transformer backbones. LDM achieves lower perplexity than standard residual connections, mHC and AttnRes at both scales, alongside competitive zero-shot accuracy. We perform extensive ablations to isolate the contribution of different architectural components of LDM.

In summary, our main contributions are:

1. We introduce depth memory, a framework that describes the propagation of information over depth through a collection of streams and their Read and Write operations.
2. We instantiate this framework in the form of LDM, a linear depth memory that solves a depth regression problem using recursive least squares to store layer representations in a fixed-size state.
3. We provide an efficient implementation of LDM that combines unwinding and deferred writing techniques to reduce peak memory overhead and memory traffic of LDM.
4. We show that LDM consistently achieves the lowest perplexities among tuned PreLN, mHC, and AttnRes baselines at the 400M and 1.5B scales, alongside competitive zero-shot accuracy.

2 Background

2.1 Residual Connections and Notation

Consider a standard pre-normalized transformer (PreLN, Xiong et al. (2020)) with B blocks and $L = 2B$ layers. We write equations for a single token and suppress its index. Let $\mathbf{x}_l \in \mathbb{R}^D$ denote

its representation after layer l, with $\mathbf{x}_0 = \mathbf{e}$ as the token embedding. In PreLN, representations across depth follow the recurrence

$$\mathbf{x}_l = \mathbf{x}_{l-1} + \boldsymbol{\delta}_l, \qquad \boldsymbol{\delta}_l = F_l\left(\mathrm{Norm}_l(\mathbf{x}_{l-1})\right), \qquad l = 1, \ldots, L, \tag{1}$$

where Norm_l is typically RMSNorm (Zhang & Sennrich, 2019), and F_l is either an attention layer or an MLP. Unrolling this recurrence gives

$$\mathbf{x}_l = \mathbf{e} + \sum_{i=1}^{l} \boldsymbol{\delta}_i.$$

In particular, residual connections establish a *residual stream* that aggregates layer outputs additively (Figure 2a). Recent work extends this design with richer representations and learned mechanisms for mixing information across depth. We review two prominent approaches below, then unify them within our depth memory framework in Section 3.1.

2.2 Attention Residuals

Attention Residuals (AttnRes, Team et al. (2026)) replace simple additive accumulation of layer outputs via the residual stream with softmax attention over earlier layer outputs. Set $\mathbf{v}_0 = \mathbf{k}_0 = \mathbf{e}$ and $\mathbf{v}_i = \mathbf{k}_i = \boldsymbol{\delta}_i$, for $i \geq 1$. Full AttnRes forms the input to layer l as

$$\mathbf{a}_l = \sum_{i=0}^{l-1} \alpha_{l,i} \mathbf{v}_i, \qquad \alpha_{l,i} = \frac{\exp\left(\mathbf{q}_{l-1}^\top \mathrm{RMSNorm}(\mathbf{k}_i)\right)}{\sum_{j=0}^{l-1} \exp\left(\mathbf{q}_{l-1}^\top \mathrm{RMSNorm}(\mathbf{k}_j)\right)}. \tag{2}$$

The layer then produces $\boldsymbol{\delta}_l = F_l(\mathrm{Norm}_l(\mathbf{a}_l))$, which is stored as $\mathbf{v}_l = \mathbf{k}_l = \boldsymbol{\delta}_l$ for subsequent layers to attend to. The query $\mathbf{q}_{l-1} \in \mathbb{R}^D$ is a learned per-layer parameter. Through the attention mechanism, each layer can selectively weight previous representations, while PreLN gives every previous layer the same weight (Figure 2b).

The above formulation of AttnRes, which we refer to as full AttnRes, incurs memory overhead during inference because it needs to retain previous representations. Block AttnRes groups layers into blocks and summarizes each block by the sum of its layer outputs. At every layer, attention forms the input from the embedding, the completed block sums, and the current block's partial sum, if nonempty.

2.3 HyperConnections

HyperConnections (HC, Zhu et al. (2025)) expand the residual representation into N streams, collected as columns of $\mathbf{X}_l \in \mathbb{R}^{D \times N}$. Three learned mappings take the incoming state $\mathbf{X}_{l-1}$ and produce vectors $\mathbf{H}_l^{\mathrm{pre}}(\mathbf{X}_{l-1}), \mathbf{H}_l^{\mathrm{post}}(\mathbf{X}_{l-1}) \in \mathbb{R}^N$ as well as the matrix $\mathbf{H}_l^{\mathrm{res}}(\mathbf{X}_{l-1}) \in \mathbb{R}^{N \times N}$. A layer acts on this expanded residual stream via

$$\boldsymbol{\delta}_l = F_l\left(\mathrm{Norm}_l(\mathbf{X}_{l-1}\mathbf{H}_l^{\mathrm{pre}}(\mathbf{X}_{l-1}))\right), \quad \mathbf{X}_l = \mathbf{X}_{l-1}\mathbf{H}_l^{\mathrm{res}}(\mathbf{X}_{l-1}) + \boldsymbol{\delta}_l\left(\mathbf{H}_l^{\mathrm{post}}(\mathbf{X}_{l-1})\right)^\top.$$

The pre-mapping $\mathbf{H}_l^{\mathrm{pre}}$ combines streams into a D-dimensional layer input, the post-mapping $\mathbf{H}_l^{\mathrm{post}}$ distributes the layer output across streams, and the residual mapping $\mathbf{H}_l^{\mathrm{res}}$ mixes the streams (Figure 2c).

Manifold-Constrained HyperConnections (mHC, Xie et al. (2026)) constrain this architecture to stabilize signal propagation over depth. In particular, $\mathbf{H}_l^{\mathrm{res}}$ is constrained to be doubly stochastic through applications of the Sinkhorn-Knopp algorithm (Sinkhorn & Knopp, 1967).

Hyper connections as a memory. (m)HC updates the residual stream with a rank-1 matrix, and contracts it to generate layer inputs. Informally, this mechanism can be interpreted as a linear associative memory over depth (Kohonen, 1972). The pre-mapping $\mathbf{H}_l^{\mathrm{pre}}$ acts as a query that reads from the state, and the post-mapping $\mathbf{H}_l^{\mathrm{post}}$ acts as a key under which the layer output is written to the expanded residual stream through an outer product update. The residual mapping $\mathbf{H}_l^{\mathrm{res}}$ acts as a transition matrix on the state. Since this state combines earlier layer outputs, subsequent reads mix information across depth without explicitly retaining each output. HC and mHC can therefore be understood as performing a form of implicit depth mixing, with both queries and keys generated from the incoming matrix-valued state itself. Building on this perspective, we next establish depth memory as a framework that encompasses both (m)HC and AttnRes, before instantiating the framework as a linear depth memory.

Table 1: Overview of depth mixing methods. Each method maintains streams that are updated via Write and read via Read. LDM uses the RLS update derived in Section 3.2, with prediction error $\mathbf{u}_l$ and gain $\boldsymbol{\kappa}_l$. Gates and the inverse covariance update for $\mathbf{P}_l$ are omitted for brevity.

Method	Streams	State update (Write)	Readout (Read)
PreLN	$(\mathbf{x}_l)$	$\mathbf{x}_l = \mathbf{x}_{l-1} + \mathbf{v}_l$	$\mathbf{x}_l$
AttnRes	$((\mathbf{K}_{\leq l}, \mathbf{V}_{\leq l}))$	$\mathrm{Append}(\mathbf{k}_l, \mathbf{v}_l)$	$\mathrm{Attend}(\mathbf{M}_l, \mathbf{q}_l)$
mHC	$(\mathbf{X}_l)$	$\mathbf{X}_l = \mathbf{X}_{l-1}\mathbf{H}_l^{\mathrm{res}}(\mathbf{X}_{l-1}) + \mathbf{v}_l\mathbf{H}_l^{\mathrm{post}}(\mathbf{X}_{l-1})^\top$	$\mathbf{X}_l\mathbf{H}_{l+1}^{\mathrm{pre}}(\mathbf{X}_l)$
LDM	$(\mathbf{x}_l, \mathbf{M}_l, \mathbf{P}_l)$	$(\mathbf{x}_l, \mathbf{M}_l) = (\mathbf{x}_{l-1} + \mathbf{v}_l,\ \mathbf{M}_{l-1} + \mathbf{u}_l\boldsymbol{\kappa}_l^\top)$	$\mathbf{x}_l + \mathbf{M}_l\mathbf{q}_l$

3 Depth Memory

In this section, we first introduce depth memory (DM), a framework that takes a memory-centric view to describe how information in a transformer flows over depth and encompasses PreLN as well as AttnRes and mHC. We then instantiate DM as a linear depth memory, which we formulate as solving a regression problem over depth. Finally, we describe how to use unrolling and windowed updates to implement linear depth memory efficiently.

3.1 The Framework

The depth memory framework assumes a tuple $\mathbf{S}_l = (\mathbf{S}_l^{(1)}, \ldots, \mathbf{S}_l^{(P)})$ of P *streams*, which are read from and written to using Read and Write operators. For a single stream, we identify $\mathbf{S}_l = \mathbf{S}_l^{(1)}$. The input to sublayer F_l is enhanced by reading from the streams $\mathbf{S}_{l-1}$ using a query $\mathbf{q}_{l-1}$, and it is updated using keys $\mathbf{k}_l$ and values $\mathbf{v}_l$,

$$\boldsymbol{\delta}_l = F_l\left(\mathrm{Norm}_l\left(\mathrm{Read}(\mathbf{S}_{l-1}, \mathbf{q}_{l-1})\right)\right), \qquad \mathbf{S}_l = \mathrm{Write}\left(\mathbf{S}_{l-1}, \mathbf{k}_l, \mathbf{v}_l\right).$$

Any instantiation of DM requires specifying the tuple $\mathbf{S}_l$ of streams as well as the Read and Write operators. The following paragraphs incorporate PreLN, AttnRes and mHC.

PreLN. For PreLN there exists exactly one stream, which is the classic residual stream: $\mathbf{S}_l = \mathbf{x}_l$. The Read and Write operators retrieve the stream and write layer outputs into it,

$$\mathrm{Read}\left(\mathbf{x}_{l-1}, \cdot\right) = \mathbf{x}_{l-1}, \qquad \mathbf{x}_l = \mathrm{Write}\left(\mathbf{x}_{l-1}, \cdot, \boldsymbol{\delta}_l\right) = \mathbf{x}_{l-1} + \boldsymbol{\delta}_l.$$

AttnRes. Like PreLN, AttnRes maintains exactly one stream. This stream consists of a growing set of key-value pairs: $\mathbf{S}_l = \mathbf{M}_l = (\mathbf{K}_{\leq l}, \mathbf{V}_{\leq l})$. Here, $\mathbf{M}_l$ acts as a memory for the model that explicitly maintains every previous layer's representation. Reading and writing correspond to attending and appending to the cache:

$$\mathrm{Read}(\mathbf{M}_{l-1}, \mathbf{q}_{l-1}) = \mathrm{Attend}(\mathbf{M}_{l-1}, \mathbf{q}_{l-1})$$
$$\mathbf{M}_l = \mathrm{Write}(\mathbf{M}_{l-1}, \mathbf{k}_l, \mathbf{v}_l) = \mathbf{M}_{l-1}.\mathrm{Append}(\mathbf{k}_l, \mathbf{v}_l).$$

Since this mechanism keeps a copy of every representation, we refer to it as an *explicit* depth mixer. Its memory stream grows over depth, as incoming representations are appended.

mHC. For mHC, there again is exactly one stream. This time it is the expanded residual stream, $\mathbf{S}_l = (\mathbf{X}_l)$. The queries and keys are generated via $\mathbf{q}_l = \mathbf{H}_{l+1}^{\mathrm{pre}}(\mathbf{X}_l)$ and $\mathbf{k}_l = \mathbf{H}_l^{\mathrm{post}}(\mathbf{X}_{l-1})$. Reading and writing correspond to

$$\mathrm{Read}\left(\mathbf{X}_{l-1}, \mathbf{q}_{l-1}\right) = \mathbf{X}_{l-1}\mathbf{q}_{l-1}, \qquad \mathbf{X}_l = \mathrm{Write}\left(\mathbf{X}_{l-1}, \mathbf{k}_l, \mathbf{v}_l\right) = \mathbf{X}_{l-1}\mathbf{H}_l^{\mathrm{res}}(\mathbf{X}_{l-1}) + \mathbf{v}_l\mathbf{k}_l^\top.$$

The memory stream of mHC has a fixed size (see also Section 2). Compared to AttnRes, incoming representations are additively written into the stream, not appended. We thus refer to it as an *implicit* depth mixer. Table 1 provides an overview of depth mixing architectures in our framework. In the next section, we introduce linear depth memory, an implicit depth mixer derived from first principles.

3.2 Linear Depth Memory

In this section, we develop *linear depth memory (LDM)*, a novel depth mixing module. At its core, LDM implements a linear associative memory (Kohonen, 1972). Such memories store key-value associations and lay the foundation for linear attention (Schlag et al., 2021).

The most simple version of LDM maintains two streams, $\mathbf{S}_l = (\mathbf{x}_l, \mathbf{M}_l)$, where $\mathbf{x}_l$ is the standard PreLN residual stream and $\mathbf{M}_l$ is a matrix-valued hidden state. The read and write operations are

$$\begin{aligned} \text{Read}((\mathbf{x}_{l-1}, \mathbf{M}_{l-1}), \mathbf{q}_{l-1}) &= \mathbf{x}_{l-1} + \mathbf{M}_{l-1}\mathbf{q}_{l-1}, \\ (\mathbf{x}_l, \mathbf{M}_l) = \text{Write}((\mathbf{x}_{l-1}, \mathbf{M}_{l-1}), \mathbf{k}_l, \mathbf{v}_l) &= (\mathbf{x}_{l-1} + \mathbf{v}_l, \mathbf{M}_{l-1} + \mathbf{v}_l\mathbf{k}_l^\top). \end{aligned} \tag{3}$$

Note how, since simple LDM carries two streams, Read and Write need to incorporate both streams. One of the streams is the classic PreLN residual stream, while the other one maintains a matrix-valued memory stream via outer-product writes. Our ablations in Section 4.2 reveal the importance of maintaining these two streams separately.

Note that the memory stream is closely related to softmax depth attention (Equation (2)). Indeed, the latter takes the form

$$\mathbf{o}_{l-1} = \mathbf{V}_{<l}^\top \operatorname{softmax}\left(\mathbf{K}_{<l}\mathbf{q}_{l-1}\right). \tag{4}$$

Following Katharopoulos et al. (2020), replacing the exponential kernel in Equation (4) with the linear dot-product kernel without normalization yields

$$\mathbf{o}_{l-1} = \mathbf{V}_{<l}^\top \mathbf{K}_{<l}\mathbf{q}_{l-1} = \left(\sum_{i=0}^{l-1} \mathbf{v}_i\mathbf{k}_i^\top\right)\mathbf{q}_{l-1} = \mathbf{M}_{l-1}\mathbf{q}_{l-1},$$

where $\mathbf{M}_{l-1} = \sum_{i=0}^{l-1} \mathbf{v}_i\mathbf{k}_i^\top$ is exactly maintained by the LDM outer product recurrence in Equation (3). Thus, simple LDM implements linear attention over depth, replacing a growing collection of layer representations by a fixed-size matrix summary. The additive update rule has been shown to approximate the solution of a regression problem (Wang et al., 2025). We adapt this perspective for LDM.

An exact solution to depth regression. Recent work formulates linear attention as a test-time regression problem. Following Wang et al. (2025), we interpret the simple outer product update in Equation (3) as an approximate solution to

$$\mathbf{M}_l^* = \arg\min_{\mathbf{M}_l} \sum_{i=0}^{l} \|\mathbf{v}_i - \mathbf{M}_l\mathbf{k}_i\|_2^2. \tag{5}$$

We call this the *depth regression* objective, by analogy with test-time regression along the sequence axis. If $\mathbf{K}_{\le l}^\top\mathbf{K}_{\le l}$ is invertible, the unique solution to Equation (5) is $\mathbf{M}_l^* = \mathbf{V}_{\le l}^\top\mathbf{K}_{\le l}\left(\mathbf{K}_{\le l}^\top\mathbf{K}_{\le l}\right)^{-1}$. Approximating the key Gram matrix by the identity recovers the update in Equation (3).

For sequence mixing, efficient training and prefill motivate parallelizable (approximate) solvers of Equation (5). Delta-rule-based models take an online gradient step on each incoming key–value pair (Schlag et al., 2021; Yang et al., 2024; 2025b; Team et al., 2025). DeltaProduct takes multiple steps per token (Siems et al., 2025). MesaNet uses an iterative conjugate-gradient solve (von Oswald et al., 2026).

For depth mixing, however, transformer layers are already evaluated sequentially, allowing the use of non-parallelizable solvers. LDM maintains the exact regression solution through recursive least squares (RLS), as in MesaLayer (von Oswald et al., 2024). We apply RLS to a generalized version of Equation (5), introducing learnable per-layer forgetting factors $\lambda_j > 0$, dynamic importance weights $\omega_i > 0$, and a weighted ridge penalty:

$$\mathbf{M}_l^* = \arg\min_{\mathbf{M}} \left\{ \sum_{i=0}^{l} \left(\prod_{j=i+1}^{l} \lambda_j\right) \omega_i \|\mathbf{v}_i - \mathbf{M}\mathbf{k}_i\|_2^2 + \left(\prod_{j=0}^{l} \lambda_j\right) \alpha^{-1}\|\mathbf{M}\|_F^2 \right\}. \tag{6}$$

Equation (6) admits an exact recursive solution via Sherman-Morrison. Initialize $\mathbf{M}_{-1} = 0$ and the inverse key covariance as $\mathbf{P}_{-1} = \alpha\mathbf{I} \in \mathbb{R}^{D_k \times D_k}$, and define the weighted observations $\overline{\mathbf{k}}_l = \sqrt{\omega_l}\mathbf{k}_l$ and $\overline{\mathbf{v}}_l = \sqrt{\omega_l}\mathbf{v}_l$. The RLS gain and prediction error are

$$\boldsymbol{\kappa}_l = \mathbf{P}_{l-1}\overline{\mathbf{k}}_l / (\lambda_l + \overline{\mathbf{k}}_l^\top \mathbf{P}_{l-1}\overline{\mathbf{k}}_l), \qquad \mathbf{u}_l = \overline{\mathbf{v}}_l - \mathbf{M}_{l-1}\overline{\mathbf{k}}_l. \tag{7}$$

These can be used to solve Equation (6) online,

$$\mathbf{M}_l = \mathbf{M}_{l-1} + \mathbf{u}_l\boldsymbol{\kappa}_l^\top, \qquad \mathbf{P}_l = (\mathbf{P}_{l-1} - \boldsymbol{\kappa}_l\overline{\mathbf{k}}_l^\top\mathbf{P}_{l-1})/\lambda_l. \tag{8}$$

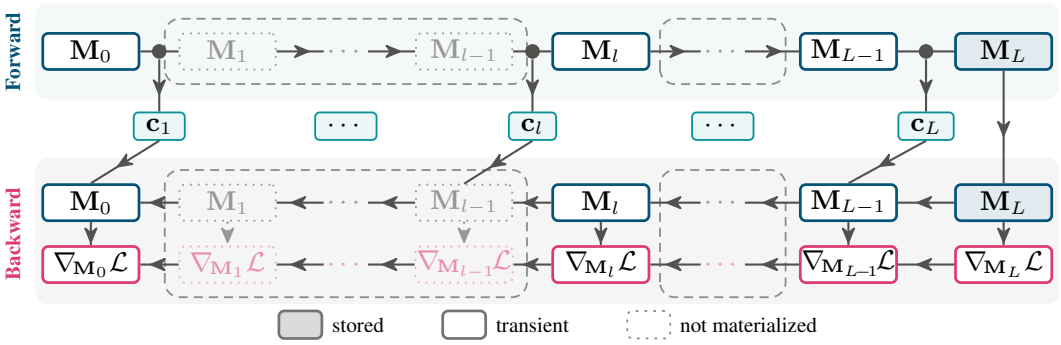


Figure 3: Hardware-aware implementation of LDM. The Forward pass updates the state in-place using windowed writes and keeps a small $\mathbf{c}_l$ per layer. The Backward pass then reconstructs preceding states from $\mathbf{c}_l$.

The complete derivation is given in Appendix E.1. The RLS update thus introduces a third stream, an inverse covariance stream $\mathbf{P}_l$, to LDM.

Multiple groups. We find that partitioning the memory state $\mathbf{M}_l$ into G groups, each with value dimension $D_v = D/G$, improves expressivity and empirically improves performance (Section 4.2). Each group maintains its own inverse covariance matrix and is queried independently. The resulting D_v-dimensional readouts are concatenated to form the full D-dimensional memory readout. The state then becomes $\mathbf{S}_l = (\mathbf{x}_l, \mathbf{M}_l, \mathbf{P}_l)$, where $\mathbf{P}_l \in \mathbb{R}^{G \times D_k \times D_k}$.

Efficient parameterization through shared latent representations. Having established LDM as an abstract architectural component above, we now present how to parametrize it efficiently. Let Aff denote a learned affine map. For the Read operator, we generate a shared latent representation

$$\mathbf{h}_{l-1}^{(R)} = \mathrm{SiLU}\left(\mathrm{Aff}(\,\mathrm{Norm}_{\mathrm{RMS}}(\mathbf{x}_{l-1}))\right) \in \mathbb{R}^H.$$

From this latent representation, we obtain the query as well as a per-group read gate

$$\mathbf{q}_{l-1} = \mathrm{Norm}_{L^2}(\mathrm{Aff}(\mathbf{h}_{l-1}^{(R)})) \in \mathbb{R}^{G \times D_k}, \quad \mathbf{g}_{l-1} = 2\sigma(\mathrm{Aff}(\mathbf{h}_{l-1}^{(R)})) \in \mathbb{R}^G.$$

The gated query retrieves data from the streams as

$$\mathrm{Read}((\mathbf{x}_{l-1}, \mathbf{M}_{l-1}, \mathbf{P}_{l-1}), \mathbf{q}_{l-1}) = \mathbf{x}_{l-1} + \gamma_{l-1}\mathbf{M}_{l-1}\left(\,\mathbf{g}_{l-1} \odot \mathbf{q}_{l-1}\right),$$

where $\boldsymbol{\gamma}_{l-1} \in \mathbb{R}^G$ is a learnable per-group gain that controls the magnitude of the read. For the Write operator, we equivalently first generate a latent representation

$$\mathbf{h}_l^{(W)} = \mathrm{SiLU}\left(\mathrm{Aff}\left(\,\mathrm{Norm}(\mathbf{x}_l)\right)\right) \in \mathbb{R}^H,$$

from which we obtain the keys $\mathbf{k}_l$ and groupwise weight $\boldsymbol{\omega}_l$,

$$\mathbf{k}_l = \mathrm{Norm}(\mathrm{Aff}(\mathbf{h}_l^{(W)})) \in \mathbb{R}^{G \times D_k}, \qquad \boldsymbol{\omega}_l = 2\sigma(\mathrm{Aff}(\mathbf{h}_l^{(W)})) \in \mathbb{R}^G.$$

The RLS write then computes

$$(\mathbf{x}_l, \mathbf{M}_l, \mathbf{P}_l) = \mathrm{Write}((\mathbf{x}_{l-1}, \mathbf{M}_{l-1}, \mathbf{P}_{l-1}), \mathbf{k}_l, \mathbf{v}_l, \boldsymbol{\omega}_l, \lambda_l)$$

in the sense of Equation (8). We additionally write the embedding into the first memory state and perform a read before the LM head (see Appendix E.2).

3.3 Hardware-Aware Implementation

A naive implementation of LDM (1) stores a state $\mathbf{M}_l$ per layer for later reuse during backpropagation and (2) reads and writes the current state at every Read and Write, which incurs memory traffic. We propose an efficient implementation that reduces both activation memory overhead and traffic:

(1) State unwinding. The RLS recurrence is reversible given compact per-layer auxiliary information. Indeed, Equation (8) shows that $\mathbf{M}_l$ can be reconstructed from $\mathbf{M}_{l+1}$ via

$$\mathbf{M}_l = \mathbf{M}_{l+1} - \mathbf{u}_{l+1}\boldsymbol{\kappa}_{l+1}^\top. \tag{9}$$

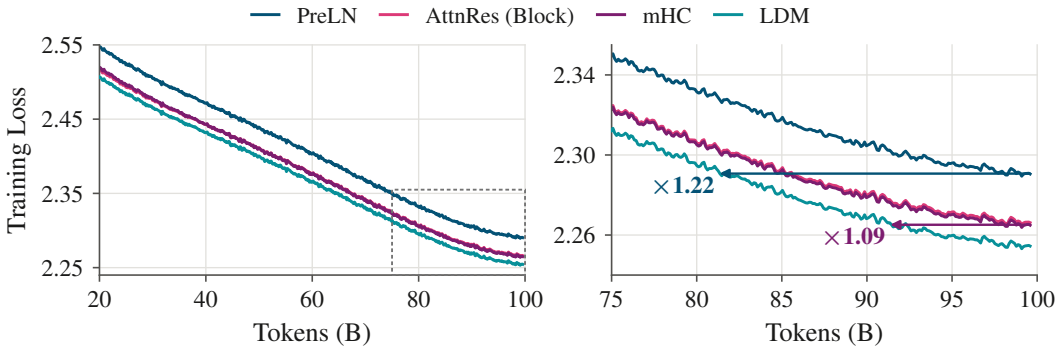


Figure 4: Left: Training loss at 1.5B model size (100B tokens, learning rate 10^{-3}). Right: the last 25B tokens. Arrows: tokens LDM needs to reach each baseline's final loss (mHC and AttnRes have nearly identical losses).

In particular, the backward pass can undo the state update given the context $\mathbf{c}_{l+1} = (\overline{\mathbf{k}}_{l+1}, \lambda_{l+1}, \mathbf{P}_l, \mathbf{u}_{l+1})$, from which $\boldsymbol{\kappa}_{l+1}$ can be computed using Equation (7). Since $\mathbf{P}_l \in \mathbb{R}^{G \times D_k \times D_k}$ and typically D_k is small (see Section 4), storing $\mathbf{c}_l$ is a lot cheaper than storing $\mathbf{M}_l \in \mathbb{R}^{D_v \times D_k}$. Our implementation of LDM updates the state $\mathbf{M}_l$ in place during the forward pass and unrolls the depth recurrence from the final state $\mathbf{M}_L$ during the backward pass (Figure 3).

(2) Windowed writes. To reduce memory traffic, we defer Write operations within a window $\mathcal{W} = \{l, \ldots, l + E - 1\}$ of E layers, folding the deferred updates into Read via the identity

$$\mathbf{M}_j \mathbf{q}_j = \mathbf{M}_{l-1} \mathbf{q}_j + \sum_{i=l}^{j} \mathbf{u}_i (\boldsymbol{\kappa}_i^\top \mathbf{q}_j), \tag{10}$$

for any $j \in \mathcal{W}$, with the pseudo-values $\mathbf{u}_i$ and RLS gains $\boldsymbol{\kappa}_i$ of Equation (7). No intermediate state is materialized inside a window, reducing Write traffic. The window size E trades state traffic against accesses to the pseudo-values. Our main experiments use $E = 4$.

State unwinding and windowed updates also apply to the memory gradients $\nabla_{\mathbf{M}_l} \mathcal{L}$ (Appendix F). We implement LDM in CuTe DSL for Hopper and Blackwell GPUs.

4 Experiments

In this section, we evaluate LDM by pretraining 400M Llama-like and 1.5B Qwen-like transformers on 50B and 100B FineWeb-Edu tokens, respectively (Penedo et al., 2024). Unless stated otherwise, the queries and keys are generated from hidden representations of dimension $H = 32$ at 400M and $H = 64$ at 1.5B. We set the per-group value dimension to the attention head dimension, yielding $G = D/d_{\text{head}} = 16$ groups for both models. We furthermore fix the ridge weight to $\alpha = 1$, which we found to work better than making it learnable.

4.1 Main Results

Setup. For each setting, we compare LDM to PreLN, mHC and AttnRes using matched backbone configurations. Relative to PreLN, LDM has a $\sim$1% parameter overhead at 400M and 1.5B, approximately matching the parameter overhead of mHC (Table 4). Our main configuration uses $D_k = 8$ for LDM. We also include a configuration with $D_k = 4$ at 400M. At that scale, we furthermore train both Full and Block AttnRes at 400M. Since Full AttnRes is expensive to train at larger scales, we train only Block AttnRes at 1.5B. Following Team et al. (2026), we keep the number of depth blocks close to eight: exactly eight at 400M and seven at 1.5B, which prevents transformer blocks from being split between AttnRes blocks. All models use AdamW with a weight decay of 0.1 and linear warmup followed by cosine decay to 10% of the peak learning rate. We sweep the peak learning rate for each method and scale using a grid spaced by factors of two, selecting the value with the lowest held-out validation loss. Full implementation details are in Appendix B.

Table 2: Pretraining evaluations for both 400M and 1.5B models, generated using the lm-eval-harness (Gao et al., 2024). Val is the pretraining validation perplexity.

	Method	ARC-E acc ↑	ARC-C acc_n ↑	Hella acc_n ↑	PIQA acc ↑	Wino acc ↑	OBQA acc_n ↑	Lmb acc ↑	Avg acc ↑	Lmb ppl ↓	Wiki ppl ↓	Val ppl ↓
400M @ 50B	PreLN	65.66	31.23	46.80	68.93	55.25	37.00	38.46	49.05	25.32	22.59	12.73
	AttnRes (Full)	64.94	32.08	49.34	70.08	55.64	36.40	41.35	49.98	19.62	21.46	12.35
	AttnRes (Block)	**67.63**	**32.85**	**49.55**	70.08	**56.04**	**37.20**	40.91	**50.61**	20.71	20.86	12.12
	mHC	65.11	31.66	49.26	70.02	55.09	35.00	40.69	49.55	20.23	21.20	12.23
	LDM, D_k=4	66.79	32.34	48.71	69.42	54.70	**37.20**	42.27	50.20	19.59	20.84	12.14
	LDM (Ours)	67.13	32.34	49.41	**70.40**	54.93	36.40	**42.58**	50.46	**19.10**	**20.70**	**12.08**
1.5B @ 100B	PreLN	74.07	41.47	62.04	75.14	62.35	42.60	51.19	58.41	10.49	15.06	9.48
	AttnRes (Block)	75.88	**42.92**	63.51	75.03	60.77	43.60	54.22	59.42	8.84	14.54	9.26
	mHC	75.67	41.72	63.32	**75.57**	**62.98**	**44.40**	54.08	59.68	9.05	14.44	9.25
	LDM (Ours)	**76.35**	42.66	**63.98**	75.14	62.83	42.00	**55.00**	**59.71**	**8.42**	**14.21**	**9.15**

Lower perplexity and competitive accuracy. Table 2 reports our main pretraining results. It reveals that our flagship LDM configuration ($D_k = 8$) achieves the lowest validation, WikiText-2, and LAMBADA perplexity at both model sizes. It also achieves the highest average zero-shot accuracy at 1.5B and ranks second at 400M.

Matched memory-matrix size. Both mHC and LDM maintain states of fixed size across depth, whereas Full AttnRes retains a growing cache of layer outputs. With $D_k = 8$, LDM's memory matrix contains $8D$ entries per token, twice the $4D$ entries in mHC's expanded residual state. We therefore also evaluate LDM with $D_k = 4$ at 400M, matching the number of entries in its memory matrix to mHC's expanded residual state, although LDM additionally maintains a classic residual stream and an inverse covariance stream. This smaller configuration still achieves lower validation, WikiText-2, and LAMBADA perplexity than mHC, alongside higher average zero-shot accuracy. Thus, LDM remains competitive with other baselines even with a smaller memory matrix.

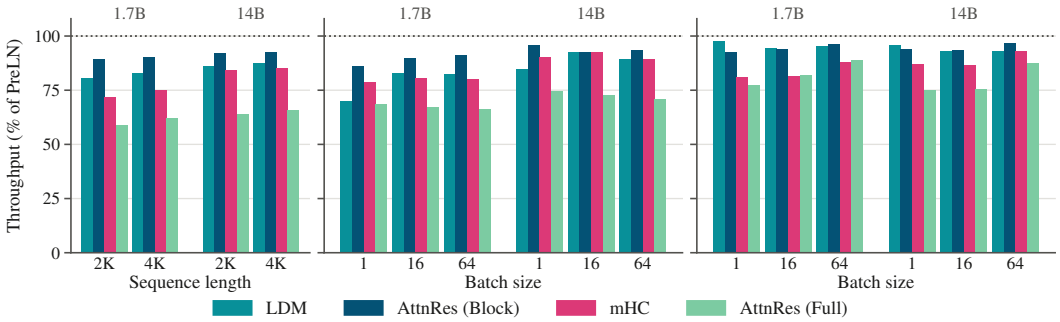
Throughput. Figure 5 compares LDM's training, prefill and decode throughput with AttnRes, implemented using Flash Linear Attention kernels (Yang & Zhang, 2024), and mHC using an improved version of the official mHC kernels. Baseline modifications, benchmarking details, and comparisons with the released mHC kernels can be found in Appendix C. Across 1.7B–14B models, sequence lengths, context lengths, and batch sizes, LDM exceeds Full AttnRes throughput in every setting and mHC in all training and most inference settings, while remaining competitive with Block AttnRes, especially during decode.

4.2 Ablations and Architectural Choices

We ablate LDM's main design choices at 400M, training each variant on 15B FineWeb-Edu tokens with a peak learning rate of 10^{-3} and otherwise following our main training recipe. Table 3 reports changes in held-out perplexity relative to the default LDM configuration in the same setting. Appendix D.2 provides full definitions of each variant. Our main findings are as follows:

(a) Keeping the residual stream is crucial. We highlight the importance of LDM maintaining a separate residual stream by interrupting it at either of two points: before the layer, by feeding the layer only the memory readout, or after the layer, by replacing the stream with the layer output. Both variants perform worse than PreLN.

(b) Depth regression approximations. Replacing RLS with a plain outer-product or delta-rule update (Schlag et al., 2021; Yang et al., 2024) still clearly beats PreLN, but RLS remains best, consistent with sequence-modeling findings that more accurate test-time regression solvers perform better (von Oswald et al., 2026; Tumma et al., 2026).



(a) Training step (b) Prefill, 2K (c) Decode, 2K

Figure 5: Throughput relative to PreLN on B200 for Qwen-like 1.7B and 14B models, comparing LDM, mHC and AttnRes. The mHC results use an improved version of the official mHC kernels. (a) Training step with 2K and 4K sequence length. (b) Prefill of 2K prompts at different batch sizes. (c) KV-cached decode after 2k prefill at different batch sizes. More settings in Appendix C.

Table 3: LDM ablations, 400M @ 15B. Δ denotes change in held-out perplexity relative to flagship LDM ($\downarrow$). (a) $F(\mathbf{r})$: memory-only layer input (normalization omitted). $\mathbf{x} = \boldsymbol{\delta}$: no post-layer residual addition. (b) Delta/Outer: replace RLS with delta-rule/outer-product update. (c) $\mathbf{v} = \mathbf{x}$: store the residual stream. $\mathbf{QK}(\boldsymbol{\delta})$: queries and keys from layer outputs. Static $\mathbf{q}$: learned query per read site with read gate fixed to one. (d) $-\gamma$, $\mathbf{g} = \mathbf{1}$, $\boldsymbol{\omega} = \mathbf{1}$: fix read gain, read gate, or write weight to one, respectively. $\lambda = 1$: disable forgetting. Selected training curves in Figure 17, full definitions and results in Appendix D.2 and Table 8.

	Baselines		(a) Residual		(b) Update		(c) Addressing			(d) Gain, gates, forgetting			
	PreLN	**LDM**	$F(\mathbf{r})$	$\mathbf{x} = \boldsymbol{\delta}$	Delta	Outer	$\mathbf{v} = \mathbf{x}$	$\mathbf{QK}(\boldsymbol{\delta})$	Static $\mathbf{q}$	$-\gamma$	$\mathbf{g} = \mathbf{1}$	$\boldsymbol{\omega} = \mathbf{1}$	$\lambda = 1$
Δ	0.78	0.00	1.88	0.87	0.06	0.07	0.33	0.07	0.19	0.04	0.07	0.01	−0.01

(c) Addressing and stored values. Storing $\mathbf{x}_l$ instead of $\boldsymbol{\delta}_l$ hurts most, likely due to redundancy in the stored information. Deriving queries and keys from $\boldsymbol{\delta}_l$ instead of $\mathbf{x}_l$ also hurts, and static per-layer queries (read gate fixed to one) hurt more. The group count matters little (Figure 6): $G = 1$ is slightly worse and $G = 8$ slightly better than our $G = 16$, which matches the value width to the attention head dimension.

(d) Gain, gates and forgetting. Disabling either the read gain γ or the read gate $\mathbf{g}$ hurts slightly. Fixing the write weight $\boldsymbol{\omega}_l$ to one or disabling forgetting ($\lambda_l = 1$) has a negligible effect in this setting.

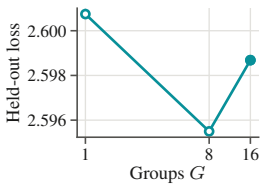


Figure 6: Held-out loss versus group count (400M, 15B tokens). LDM uses $G = 16$.

5 Discussion and Conclusion

In this work, we introduce Depth Memory, a framework that interprets the propagation of information over depth as a problem of managing layer representations using a set of streams. This perspective unifies standard residual connections, AttnRes, and mHC, while distinguishing methods that retain explicit layer representations from those that mix them into a fixed-size state. It also provides a starting point for designing new depth mixers. We instantiate this framework as Linear Depth Memory (LDM), which maintains a standard residual stream, a memory stream, and an inverse covariance stream. LDM formulates memory updates as a depth regression problem, which it solves using recursive least squares. Each token thus carries a compressed, queryable representation of earlier layer outputs, which enriches the inputs to subsequent layers. Our implementation uses state unwinding and windowed updates to reduce activation storage and memory traffic, making LDM feasible in practice. At both 400M and 1.5B, trained on 50B and 100B tokens, respectively, our main LDM configuration achieves the lowest reported perplexities among the tuned PreLN, AttnRes, and mHC baselines, alongside competitive zero-shot accuracy.

AI USE STATEMENT

In this work, we use generative AI tools to design or give feedback on our methodology and experiments, to create or edit code, to implement methods, to propose or refine hypotheses, to support qualitative and thematic data analysis, to interpret results, and to do a literature review, all apart from the manual efforts of the authors. Additionally, we used generative AI tools for LaTeX formatting, to draft parts of the text that were edited afterwards, and for discussion and refinement of ideas. We have not used generative AI tools to generate synthetic datasets, to help develop theoretical models or conceptual frameworks, to assist with translation, to clean or reformat datasets, to formulate mathematical claims, or to provide critical ingredients for proving them. An important instance of AI use is kernel development. We used AI to search the space of possible algorithmic implementations with substantial human involvement. Given our algorithmic framework, we used AI to further search through the space of kernels and to write them. Aside from human involvement, we substantially tested these with a wide array of synthetic tests and real training scenarios to ensure their validity. We have reviewed all AI-assisted work and checked the code as well as the reformatted LaTeX tables. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have taken several steps to ensure the reproducibility of our work. First, the datasets used are readily available online. We experiment with standard settings, which are thoroughly described in the appendix. We have also attached our codebase, used to obtain the results presented here, to the supplementary material. Upon acceptance, we will also release different checkpoints (ours and those of the baselines) throughout the training. In particular, we will release checkpoints every 2.5K training steps for the 400M models and every 5K for the 1.5B ones, with the objective of aiding future architecture research.

REFERENCES

Badr AlKhamissi, Mohamed Gabr, Muhammad ElNokrashy, and Khaled Essam. Adapting MARBERT for improved Arabic dialect identification: Submission to the NADI 2021 shared task. In Nizar Habash, Houda Bouamor, Hazem Hajj, Walid Magdy, Wajdi Zaghouani, Fethi Bougares, Nadi Tomeh, Ibrahim Abu Farha, and Samia Touileb (eds.), *Proceedings of the Sixth Arabic Natural Language Processing Workshop*, pp. 260–264, Kyiv, Ukraine (Virtual), April 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.wanlp-1.29/.

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.

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.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang

Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Muhammad ElNokrashy, Badr AlKhamissi, and Mona Diab. Depth-wise attention (dwatt): A layer fusion method for data-efficient classification, 2024. URL https://arxiv.org/abs/2209.15168.

Yanwen Fang, Yuxi Cai, Jintai Chen, Jingyu Zhao, Guangjian Tian, and Guodong Li. Cross-layer retrospective retrieving via layer attention, 2023. URL https://arxiv.org/abs/2302.03985.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL http://jmlr.org/papers/v23/21-0998.html.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Gleb Gerasimov, Yaroslav Aksenov, Nikita Balagansky, Viacheslav Sinii, and Daniil Gavrilov. You do not fully utilize transformer's representation capacity, 2025. URL https://arxiv.org/abs/2502.09245.

Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. The reversible residual network: Backpropagation without storing activations, 2017. URL https://arxiv.org/abs/1707.04585.

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel,

Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, William Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah Smith, and Hannaneh Hajishirzi. OLMo: Accelerating the science of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15789–15809, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.841. URL https://aclanthology.org/2024.acl-long.841/.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022. URL https://arxiv.org/abs/2111.00396.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition . In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Los Alamitos, CA, USA, June 2016. IEEE Computer Society. doi: 10.1109/CVPR.2016.90. URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.90.

Mike Heddes, Adel Javanmard, Kyriakos Axiotis, Gang Fu, Mohammadhossein Bateni, and Vahab Mirrokni. Deepcrossattention: Supercharging transformer residual connections. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=j3JBfFnGYh.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/katharopoulos20a.html.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong, Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun, Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian, Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang,

Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang, Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Teuvo Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, C-21(4):353–359, 1972. doi: 10.1109/TC.1972.5008975.

Tianyu Li, Dongchen Han, Zixuan Cao, Haofeng Huang, Mengyu Zhou, Ming Chen, Erchao Zhao, Xiaoxi Jiang, Guanjun Jiang, and Gao Huang. Siamesenorm: Breaking the barrier to reconciling pre/post-norm, 2026. URL https://arxiv.org/abs/2602.08064.

Ze-Wei Liou and Tri Dao. ReplaySSM: Cache SSM inputs, not state. https://tridao.me/blog/2026/replayssm/, June 2026. Blog post.

Cheng Luo, Zefan Cai, and Junjie Hu. Delta attention residuals, 2026a. URL https://arxiv.org/abs/2605.18855.

Cheng Luo, Zefan Cai, and Junjie Hu. Multi-head attention residuals, 2026b. URL https://arxiv.org/abs/2607.27230.

Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger B Grosse. Reversible recurrent neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/4ff6fa96179cdc2838e8d8ce64cd10a7-Paper.pdf.

Brian Mak and Jeffrey Flanigan. Residual matrix transformers: Scaling the size of the residual stream, 2025. URL https://arxiv.org/abs/2506.22696.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. URL https://arxiv.org/abs/1609.07843.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering, 2018. URL https://arxiv.org/abs/1809.02789.

Matteo Pagliardini, Amirkeivan Mohtashami, Francois Fleuret, and Martin Jaggi. Denseformer: enhancing information flow in transformers via depth weighted averaging. In *Proceedings of the*

38th International Conference on Neural Information Processing Systems, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context, 2016. URL https://arxiv.org/abs/1606.06031.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024. URL https://arxiv.org/abs/2406.17557.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, Fan Zhou, Hao Luo, Haofeng Huang, Haoran Lian, Haoyan Huang, Hongqing Chen, Jianwei Zhang, Jing Xu, Junjie Wang, Langshi Chen, Liangyu Wang, Linlang Jiang, Man Yuan, Minmin Sun, Peng Jin, Siqi Zhang, Siyu Wang, Xingzhang Ren, Yakai Wang, Yi Zhang, Yiming Dong, Yizhong Cao, Yubo Ma, Yunfei Mao, Bo Zheng, and Dayiheng Liu. On the design of qwen3.8-next architecture: Evaluation, efficiency, and training stability, 2026. URL https://arxiv.org/abs/2608.30320.

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. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9355–9366. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/schlag21a.html.

Noam Shazeer. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. Deltaproduct: Improving state-tracking in linear rnns via householder products, 2025. URL https://arxiv.org/abs/2502.10297.

Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

Jonathan Su. Low-rank attention residuals, 2026. URL https://arxiv.org/abs/2607.09694.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, et al. Kimi linear: An expressive, efficient attention architecture. *arXiv preprint arXiv:2510.26692*, 2025.

Kimi Team, Guangyu Chen, Yu Zhang, Jianlin Su, Weixin Xu, Siyuan Pan, Yaoyu Wang, Yucheng Wang, Guanduo Chen, Bohong Yin, Yutian Chen, Junjie Yan, Ming Wei, Y. Zhang, Fanqing Meng, Chao Hong, Xiaotong Xie, Shaowei Liu, Enzhe Lu, Yunpeng Tai, Yanru Chen, Xin Men, Haiqing Guo, Y. Charles, Haoyu Lu, Lin Sui, Jinguo Zhu, Zaida Zhou, Weiran He, Weixiao Huang, Xinran Xu, Yuzhi Wang, Guokun Lai, Yulun Du, Yuxin Wu, Zhilin Yang, and Xinyu Zhou. Attention residuals, 2026. URL https://arxiv.org/abs/2603.15031.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William

Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 Furious, 2024. URL https://arxiv.org/abs/2501.00656.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Neehal Tumma, Noel Loo, and Daniela Rus. Preconditioned deltanet: Curvature-aware sequence modeling for linear recurrences, 2026. URL https://arxiv.org/abs/2604.21100.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Johannes von Oswald, Maximilian Schlegel, Alexander Meulemans, Seijin Kobayashi, Eyvind Niklasson, Nicolas Zucchet, Nino Scherrer, Nolan Miller, Mark Sandler, Blaise Agüera y Arcas, Max Vladymyrov, Razvan Pascanu, and João Sacramento. Uncovering mesa-optimization algorithms in transformers, 2024. URL https://arxiv.org/abs/2309.05858.

Johannes von Oswald, Nino Scherrer, Seijin Kobayashi, Luca Versari, Songlin Yang, Sarthak Mittal, Maximilian Schlegel, Kaitlin Maile, Yanick Schimpf, Oliver Sieberling, Alexander Meulemans, Rif A. Saurous, Guillaume Lajoie, Charlotte Frenkel, Razvan Pascanu, Blaise Agüera y Arcas, and João Sacramento. Mesanet: Sequence modeling by locally optimal test-time training, 2026. URL https://arxiv.org/abs/2506.05233.

Ke Alexander Wang, Jiaxin Shi, and Emily B Fox. Test-time regression: a unifying framework for designing sequence models with associative memory. *arXiv preprint arXiv:2501.12352*, 2025.

Da Xiao, Qingye Meng, Shengping Li, and Xingyuan Yuan. MUDDFormer: Breaking residual bottlenecks in transformers via multiway dynamic dense connections. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=qkhgzNiEdj.

Zhenda Xie, Yixuan Wei, Huanqi Cao, Chenggang Zhao, Chengqi Deng, Jiashi Li, Damai Dai, Huazuo Gao, Mingyu Xu, Kuai Yu, Liang Zhao, Shangyan Zhou, Zhean Xu, Zhengyan Zhang, Wangding Zeng, Shengding Hu, Yuqing Wang, Jingyang Yuan, Lean Wang, and Wenfeng Liang. mHC: Manifold-constrained hyper-connections. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=mDhyxu8WRb.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 10524–10533. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/xiong20b.html.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

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. *Advances in neural information processing systems*, 37:115491–115522, 2024.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations*, volume 2025, pp. 29687–29707, 2025b.

Yongyi Yang and Jianyang Gao. mhc-lite: You don't need 20 sinkhorn-knopp iterations, 2026. URL https://arxiv.org/abs/2601.05732.

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.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *CoRR*, abs/1910.07467, 2019. URL http://arxiv.org/abs/1910.07467.

Yifan Zhang, Yifeng Liu, Mengdi Wang, and Quanquan Gu. Deep delta learning, 2026. URL https://arxiv.org/abs/2601.00417.

Defa Zhu, Hongzhi Huang, Zihao Huang, Yutao Zeng, Yunyao Mao, Banggu Wu, Qiyang Min, and Xun Zhou. Hyper-connections. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9FqARW7dwB.

A Extended Related Work

Our work sits at the intersection of two lines of research: mechanisms that enrich information flow across transformer depth, particularly through depth mixing, and associative memory mechanisms developed primarily for sequence modeling in the past. We discuss both connections below.

A.1 Information Flow Across Transformer Depth

Enriching the residual representation. The standard transformer propagates information across depth through a single additive residual stream. A number of works have explored richer alternatives to this formulation. SiameseNorm (Li et al., 2026) maintains two streams that complement each other with different normalization approaches. Moreover, residual matrix transformers (Mak & Flanigan, 2025) propagate a matrix-valued residual representation. HyperConnections (Zhu et al., 2025) introduce multiple residual channels along with learned mappings between them, and Manifold-Constrained HyperConnections (mHC) (Xie et al., 2026) (and its follow-up mHC-lite (Yang & Gao, 2026)) enhance their stability. Related approaches include Deep Delta Learning (Zhang et al., 2026), which uses a delta-like rule to update a (possibly matrix-sized) residual stream and other architectures that increase the capacity or flexibility of the representations propagated between layers.

These methods modify the residual representation itself. LDM instead preserves the standard additive residual stream and introduces a separate recurrent associative state, providing an additional pathway for information flow across depth.

Explicit depth mixing. A second family of methods directly combines representations produced at different depths. We refer to this general class of mechanisms as *depth mixing*. Unlike ordinary residual accumulation, depth mixing methods allow the computation at one layer to selectively incorporate representations produced at earlier layers.

Several works have explored representation aggregation across layers. DenseFormer (Pagliardini et al., 2024) combines preceding hidden representations using learned depth-dependent weights. Cross-Layer Retrospective Retrieving via Layer Attention (Fang et al., 2023) uses attention to retrieve representations from earlier layers. They share closer connections to us as they also consider a "linearized" version as motivation, but their final method is different from ours. DeepCrossAttention (Heddes et al., 2025) considers richer cross-layer mixing, including input-dependent mechanisms. MUDDFormer (Xiao et al., 2025) similarly constructs layer inputs by dynamically combining representations across depth, and LIMe (Gerasimov et al., 2025) also revisits the use of intermediate

transformer representations. Earlier work also introduced Vertical Attention (AlKhamissi et al., 2021) and Depth-Wise Attention (ElNokrashy et al., 2024), likewise exploring attention or fusion across network layers.

Most closely related to our framing, Attention Residuals (AttnRes) (Team et al., 2026) replaces fixed residual accumulation with softmax attention over representations from preceding depth sites. Figure 2b illustrates Full AttnRes. Its blockwise variant applies the same operation to coarser groups of layers. Subsequent work has considered alternative parameterizations of attention-based residual mixing (Su, 2026; Luo et al., 2026b).

LDM differs from these approaches in how previous depth representations are made available. Explicit depth mixing mechanisms retain or otherwise expose a collection of previous representations and directly form a mixture over them. LDM instead recurrently compresses key–value associations from previous depth sites into a fixed-size state. A later layer retrieves from this state using a content-dependent query. As shown in the main text, the simplest LDM recurrence can therefore be interpreted as implementing linearized attention over depth *implicitly*, without explicitly materializing the collection of previous depth representations.

Use in large-scale runs. Richer mechanisms for information propagation across depth have also begun to appear in large-scale language models. Recent models, including DeepSeek-V4 (DeepSeek-AI et al., 2026), Kimi K3 (Kimi Team et al., 2026), and Qwen3.8 (Qiu et al., 2026), incorporate variants of enriched residual or cross-layer routing mechanisms. This provides additional evidence that modifying information flow across transformer depth works at large scale and appears to be a promising scaling axis.

A.2 Associative Memories, Linear Attention and Regression

Our second connection is to the literature on linear attention and associative memory. A linear associative memory stores key–value associations through a matrix-valued state and retrieves values using a query. Under the simplest outer product update, this recurrence is equivalent to unnormalized linear attention. In particular, the recurrent state suffices to compute the readout of any query over all previous key–value pairs.

Recent work has further interpreted such memories as online regression problems. In particular, Test-Time Regression (Wang et al., 2025) provides a regression perspective on sequence models with associative memory. Closely related ideas appear in work on mesa-optimization. The original MesaLayer (von Oswald et al., 2024) uses a recursive least squares update implemented through the Sherman–Morrison identity, maintaining the exact regression solution recurrently along the sequence axis. This recurrence is inherently sequential, which makes efficient sequence training challenging. To address this, subsequent MesaNet work (von Oswald et al., 2026) instead develops iterative solvers that permit efficient parallel training over the sequence dimension.

LDM transfers this perspective from the sequence axis to the depth axis. Each layer supplies a key–value observation, and the memory stream propagated through the network maintains a regression solution over representations produced by preceding layers. This distinction removes an important systems constraint. In particular, transformer layers are naturally evaluated sequentially in depth, so the memory update does not need to be parallelizable along that axis. We can therefore use a Sherman–Morrison rank-one recursive least-squares update directly, obtaining the exact solution to the weighted depth-regression objective described in the main text.

This gives another view of LDM's relationship to prior work. From the depth mixing perspective, it replaces explicit attention over stored layer representations with a recurrent associative state. From the associative-memory perspective, it applies regression-based memory updates (previously developed for associations across sequence positions) to associations across transformer depth.

A.3 Relation of Our Hardware-Aware Implementation to Past Work

During training, our hardware-aware implementation avoids storing intermediate states during the forward pass by "unwinding" the memory, as explained in Appendix F. Although different, this has some connections to past work. Reversible residual networks (Gomez et al., 2017) split the residual stream into two halves that update each other additively, so that the input of a layer can

be recomputed from its output and no activations have to be stored for the backward pass of "reversible" layers. Reversible recurrent networks (MacKay et al., 2018) use a similar technique along the sequence axis to reconstruct earlier hidden states from later ones. Since forgetting makes their update not exactly invertible, they store the information it discards, similar to how LDM stores its pseudo-values.

On the other hand, the windowed writes are also a natural approach for the sequence axis (though only for inference, given that training is generally done using chunkwise-parallel approaches). This was very recently explored for linear attention (in particular, Mamba-2 and Gated DeltaNet) acceleration by ReplaySSM (Liou & Dao, 2026), which, during sequential decoding, caches recent inputs of layers (for Gated DeltaNet, the delta rule correction, the counterpart to our pseudo-values), and then uses each read from the last stored state added to these cached rank-one terms, only writing back the updated state when the cached state is "flushed". Our windowed writes apply the same idea across depth, but across both training and inference (as well as for the backward pass).

B Experimental Details

We train on the FineWeb-Edu dataset (Penedo et al., 2024). In particular, we train on (subsets of) the publicly available 100B-token subset [1]. For all experiments, data ordering is fixed for every baseline to ensure fairness. Validation loss is taken from a held-out shard of the 100B subset.

Data is tokenized with OLMo-core's GPT-NeoX Dolma v1.5 tokenizer (Groeneveld et al., 2024) (with a vocabulary size of 50,280). Documents are packed into sequences of 2,048 tokens.

All models are trained with AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay 0.1) and global gradient norm clipping at 1.0. We use a global batch of 512 sequences, i.e., 1,048,576 tokens per optimizer step. The learning rate follows a cosine schedule, decaying to 10% of its peak, with a linear warmup from 0 over the first 1,000 steps at 400M and 2,000 steps at 1.5B. The 400M models are trained for 47,684 steps (50.0B tokens) and the 1.5B models for 94,972 steps (99.6B tokens).

B.1 Architecture

For a matched comparison, all baselines use the same backbone configurations, and only the method-specific details vary. Details about our Llama-like and Qwen-like architectures can be found in Table 4.

[1] https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu/viewer/sample-100BT

Table 4: Backbone configurations used in our experiments. The bottom rows list the parameters that each depth mixing method adds to the backbone, and their share of it.

Configuration	400M Llama-like	1.5B Qwen-like
Backbone parameters	405.4M	1.512B
Transformer blocks	24	28
Model width D	1,024	2,048
Attention type	MHA	GQA
Query heads	16	16
Key/value heads	16	8
Queries per KV head	1	2
Head dimension	64	128
Query projection width	1,024	2,048
Key/value projection width	1,024	1,024
FFN type	SwiGLU	SwiGLU
FFN intermediate width	2,736	6,144
Normalization	Pre-RMSNorm	Pre-RMSNorm
RMSNorm ϵ	10^{-6}	10^{-6}
Query/key normalization	None	RMSNorm
RoPE base θ	10^4	10^6
Maximum sequence length	2,048	2,048
Vocabulary size	50,280	50,280
Input/output embeddings	Untied	Tied
Attention bias	No	No
FFN bias	No	No
Dropout	0	0
Master/compute precision	FP32/BF16	FP32/BF16
Initialization std.	0.02	0.02
Parameters added by the depth mixing method		
AttnRes (Block or Full)	0.10M (0.02%)	0.23M (0.02%)
mHC	4.94M (1.22%)	11.51M (0.76%)
LDM	3.78M (0.93%)	16.25M (1.07%)
LDM, $D_k = 4$	3.58M (0.88%)	–

B.2 Implementation and baselines

Our codebase is adapted from OLMo-core (Team OLMo et al., 2024).[2] During training, all methods use FP32 master parameters and optimizer states, BF16 backbone computation and residual streams, and FP32 gradient reductions unless noted otherwise. Each baseline is implemented in our codebase following its description. For some unspecified details, we used public conventions or tried both in our initial explorations.

Pre-LN. We retain the standard OLMo-core pre-normalized transformer implementation, including its final RMSNorm before the vocabulary projection. For the Qwen-like backbone, attention Q/K normalization is applied independently within each head. The same backbone normalization settings are used across methods.

AttnRes. We adapt FLA's fused AttnRes implementation[3]. We use zero-initialized depth queries as described in the original paper. Routing parameters follow the backbone's mixed precision policy. Following the recommendations of Team et al. (2026), for the block variant, we use eight depth blocks at 400M and seven at 1.5B. For the latter, we use seven so that every block contains whole transformer layers (in this case, this gives four per block).

The RMSNorm used for the keys has a learnable gain but no bias, following popular implementations like FLA or Megatron (Shoeybi et al., 2020). We did not test removing it.

The original paper does not state whether the depth queries receive weight decay. Decay pushes the queries towards zero and thus biases the mixing towards uniform weights. In preliminary experi-

[2] https://github.com/allenai/OLMo-core

[3] https://github.com/fla-org/flash-linear-attention/blob/02e270330b4fc5bb2fd738281b5d594bc89e6d5f/fla/ops/attnres/fused.py

ments, both choices performed similarly, with a slight but consistent advantage for using decay for the queries. We hence use weight decay for the queries in every run.

mHC. We use DeepSeek's official mHC implementation in TileKernels,[4] with four residual streams and 20 Sinkhorn iterations. We use their fused TileLang kernels for the three coefficient mappings and the two application kernels, along with their precision policy (TF32 controller projection, FP32 controller parameters and scalars, BF16 residual streams, and transformer branches), as well as their block-wise activation recomputation, which stores one residual entry per L_r mHC layers and rebuilds the remaining widened streams during the backward pass. We use the block size described in the mHC paper, $L_r \approx \sqrt{nL/(n+2)} = 6$.

Following Xie et al. (2026), the controller projection is initialized to zero, and its biases are initialized as follows. First, the pre-layer mixing bias is -3 except for the layer's selected stream, which is $+3$, and the residual mixing bias is $+3$ on the diagonal and -3 elsewhere. The post-mapping bias is initialized to zero, and the selected stream of each connector cycles with the connector's index modulo n. The learned logit scales are initialized to 0.01. In some preliminary experiments where we used other initializations, this did not appear to impact performance. This detail appears to only be available in their ICML version.

We exclude biases and logit scales from weight decay. Controller RMS normalization, pre-mixing sigmoid stabilization, and Sinkhorn stabilization use $\epsilon = 10^{-6}$. We did not try ablating this.

There are other details undisclosed in the mHC paper. First, in our implementation, the first layer expands the D-dimensional residual stream into an $n \times D$-dimensional one by replicating it n times. This appears to be in line with other public implementations like Megatron and the official DeepSeek-V4 checkpoints, so we use it. Second, for the final layer, Megatron offers the option to use an average of the n channels to reduce it to a single vector; however, it also supports a final "head" that increases expressivity. The released DeepSeek-V4 (DeepSeek-AI et al., 2026) checkpoint also uses such a learnable head. In early experiments, we found that it had little effect on the training curves but seemed consistently equal to or better than the alternatives. For that, and to stay close to DeepSeek-V4, we use the learnable head. Its projection is initialized from $\mathcal{N}(0, 0.02^2)$, its bias to zero, and its logit scale to 0.01.

LDM. Our ablations run on more flexible code. On the other hand, the main training results follow the implementation in Appendix F; in particular, we use state unwinding (Equation (9)) with a window size $E = 4$. The settings are those of Section 3.2: $G = 16$ groups, $D_k = 8$ (except for the $D_k = 4$ result at 400M scale), and H set to 32 at 400M and 64 at 1.5B scales, respectively. We do not use weight decay on the read gains, the gate biases and the forgetting logits.

B.3 Training

We use FSDP with FP32 master parameters and BF16 computation, with gradient reductions in FP32. For the 400M models, we use the equivalent of DDP (no sharding) but utilize the FSDP API to ensure an identical precision policy.

Hyperparameter tuning. To enable a fair comparison, we tune the learning rates of each baseline at both scales. We use a grid anchored at 10^{-3}, with candidates multiplied or divided by powers of two (e.g. 2.5×10^{-4}, 5×10^{-4}, 10^{-3}, 2×10^{-3}, 4×10^{-3}). For each baseline and scale, we run candidate learning rates until we find either three consecutive values, where the middle one has the lowest held-out loss, or two consecutive values whose held-out losses differ by less than 0.001 nats (in these cases, training and held-out validation losses often flip), and we select the value with the lowest held-out loss. The final checkpoint of the selected run is used for all evaluations. Other hyperparameters are kept fixed.

The resulting sweeps are reported in Appendix D.1.

B.4 Evaluation

We evaluate the final checkpoint of each selected pretraining run with the `lm-eval-harness` (Gao et al., 2024) in zero-shot mode, with a maximum length of 2,048 tokens. The suite includes

[4] https://github.com/deepseek-ai/TileKernels

ARC-Easy and ARC-Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2019), WinoGrande (Sakaguchi et al., 2019), OpenBookQA (Mihaylov et al., 2018), WikiText (Merity et al., 2016) (word perplexity), and LAMBADA (Paperno et al., 2016) (accuracy and perplexity).

C Efficiency Measurements

We measure different notions of efficiency for every method on Qwen3 models (1.7B to 14B parameters; we retain the native vocabulary of 151936 for all of them) on NVIDIA B200 GPUs, in BF16 with compiled models, and report measurements as a percentage of the corresponding PreLN baseline. The models follow the 1.5B Qwen-like configuration of Table 4, with the changes listed in Table 5.

Table 5: Qwen3 shapes of the efficiency measurements. All other settings are those of the 1.5B Qwen-like model in Table 4.

	1.7B	4B	8B	14B
Transformer blocks	28	36	36	40
Model width D	2,048	2,560	4,096	5,120
Query heads	16	32	32	40
Key/value heads	8	8	8	8
Head dimension	128	128	128	128
FFN intermediate width	6,144	9,728	12,288	17,408
Vocabulary size	151,936	151,936	151,936	151,936
Input/output embeddings	Tied	Tied	Untied	Untied
LDM groups G	16	20	32	40

We execute all methods of one setting in three interleaved rounds, with each method in a new process and different orderings per round to ensure fairness. Every point represents the median over the rounds.

We made several adjustments to ensure the fairness of measurements. Every mHC controller folds its normalization weight into the projection once at load time, following their description in (Xie et al., 2026). "mHC (released kernels)" denotes DeepSeek's TileKernels as released (though with several adaptations around the exact kernels to make them compatible with our codebase). The mHC row replaces their projection backward with a faster matmul operation and splits the decode projection, both of which we found to work better in terms of efficiency (with exact improvements varying per setting). "mHC, no recompute" keeps every stream instead of recomputing them (recomputation follows the approach in (Xie et al., 2026)). Our main figures, for fairness, always report the "mHC" setting. Here we report the entire set for completeness.

Training (Figure 7). We measure the optimizer step of about 1M tokens on 8 GPUs with FSDP and PyTorch's efficient implementation of scaled dot product attention (SDPA). We use the largest microbatch at which every method fits.

Prefill (Figure 8). We measure forward over the whole prompt on one GPU, keeping only the last position's logits, from a single prompt up to the largest batch size that fits for all methods.

Decode (Figures 9 and 10). KV-cached greedy decoding on one GPU with FlashAttention 4. The prompt is prefilled in 512-token chunks, as in serving, which gives the time to first token; each generated token then replays a CUDA graph of one decode step.

Peak memory (Table 6 and Figures 11 to 13). We report the peak memory allocated per GPU in GiB in the settings of the main text figure (Table 6): the training step with sequences of 2K tokens, and prefill and decode at batch 64 with a 2K token prompt or context. Figures 11 to 13 give it as a percentage of PreLN in every setting above.

Table 6: Peak memory allocated per GPU (GiB). Training runs on 8 GPUs with sequences of 2K tokens; prefill and decode run on one GPU at batch 64 with a 2K token prompt or context. Recomputation does not apply at inference, so the mHC row without recompute is training only.

	1.7B	4B	8B	14B
Training				
PreLN	80.2	82.6	107.6	97.3
LDM	94.0	92.8	123.2	107.4
AttnRes (Block)	80.3	82.8	107.7	97.4
AttnRes (Full)	90.5	94.0	126.7	111.0
mHC	80.3	81.1	106.2	95.3
mHC (released kernels)	80.3	81.1	104.8	94.9
mHC, no recompute	115.3	110.3	151.5	126.7
Prefill				
PreLN	7.8	14.3	24.4	39.9
LDM	14.7	22.9	38.1	57.1
AttnRes (Block)	11.3	19.9	33.4	52.4
AttnRes (Full)	34.8	58.0	94.4	137.4
mHC	10.8	18.1	30.5	47.5
mHC (released kernels)	10.8	18.1	30.5	47.5
Decode				
PreLN	19.4	28.6	36.9	52.1
LDM	21.2	30.8	40.4	56.5
AttnRes (Block)	20.4	30.2	39.4	55.6
AttnRes (Full)	26.3	39.7	54.7	76.8
mHC	20.2	29.6	38.6	54.2
mHC (released kernels)	20.2	29.6	38.6	54.2

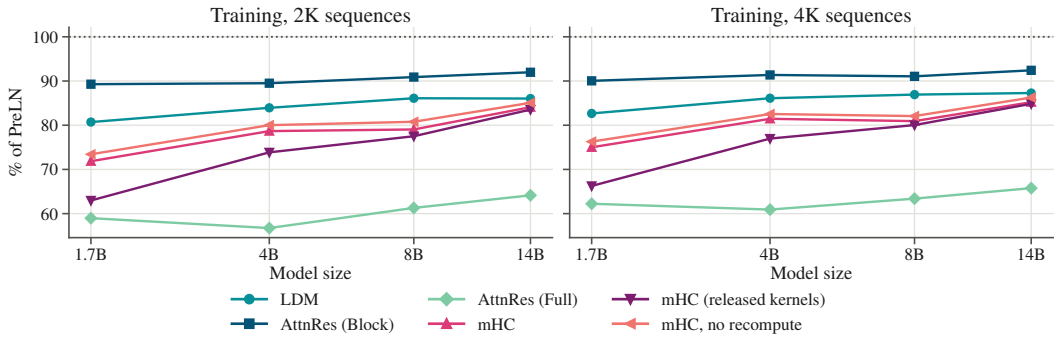


Figure 7: Training throughput as a percentage of PreLN.

D Extended Results

D.1 Learning Rate Sweeps

Table 7 lists the final held-out validation loss of every run in the learning rate sweeps (we describe our protocol in Appendix B). We bold the selected learning rate of each method and scale. Figure 14 plots the losses of each model at each learning rate.

Figures 15 and 16 show the training loss of every run in Table 7. Each panel compares all methods trained at shared peak learning rates. At 400M scale, the last panel shows each method at its selected rate. For better visibility, the top rows show the whole run, and the bottom rows show the run after its first quarter of tokens, where the methods separate more.

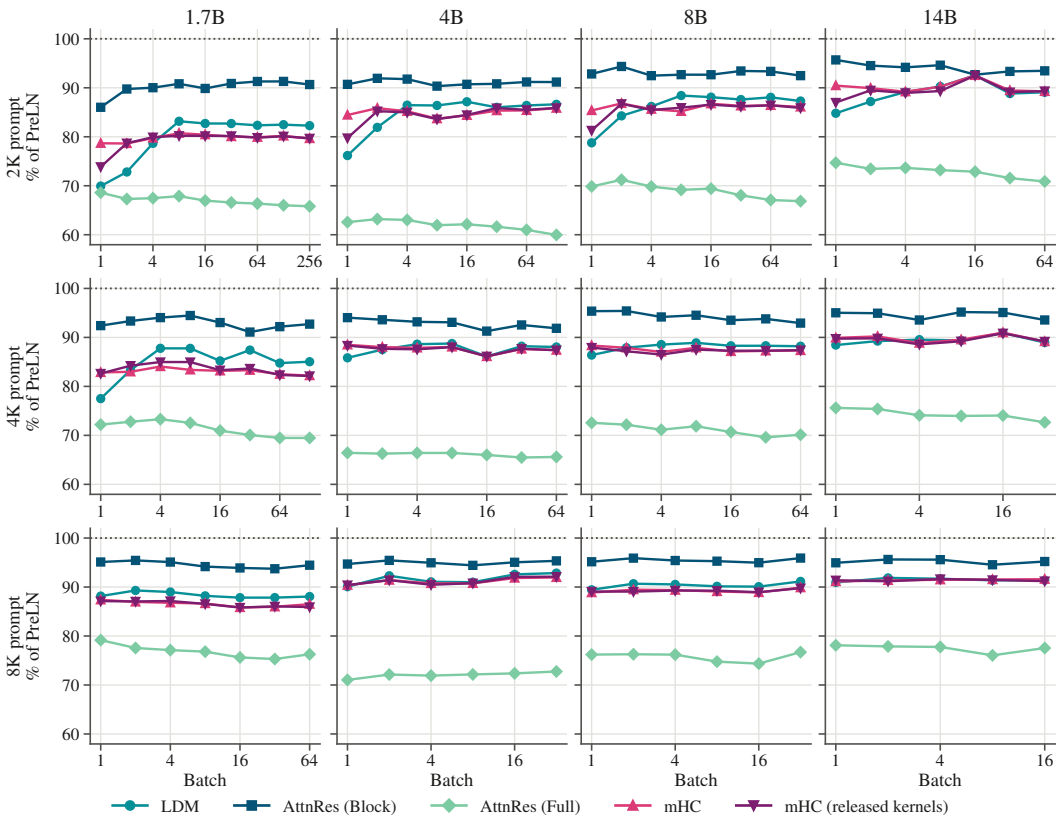


Figure 8: Prefill throughput as a percentage of PreLN, by prompt length (rows) and model size (columns).

Table 7: Learning rate sweeps: final held-out FineWeb-Edu cross-entropy of each method, model scale and peak learning rate. The selected learning rate of each method is in bold. – means not run.

Scale	Method	5×10^{-4}	1×10^{-3}	2×10^{-3}	4×10^{-3}
400M	PreLN	2.5627	**2.5441**	2.5465	–
	AttnRes (Full)	–	2.5172	2.5135	**2.5133**
	AttnRes (Block)	–	2.5064	**2.4950**	2.5026
	mHC	2.5291	2.5043	**2.5041**	–
	LDM (Ours)	2.5138	**2.4913**	2.4957	–
	LDM, $D_k = 4$	2.5178	**2.4965**	2.4991	–
1.5B	PreLN	2.2554	**2.2496**	2.2633	–
	AttnRes (Block)	2.2333	**2.2257**	2.2437	–
	mHC	2.2336	**2.2244**	2.2366	–
	LDM (Ours)	2.2207	**2.2141**	2.2260	–

D.2 Ablation Definitions

We use the notation of Section 3.1: at layer l the incoming stream is $\mathbf{x}_{l-1}$, the layer output is $\boldsymbol{\delta}_l$, the state is $\mathbf{S}_l = (\mathbf{x}_l, \mathbf{M}_l, \mathbf{P}_l)$, and the LDM equations are

$$\mathbf{q}_{l-1} = \mathrm{Query}(\mathbf{x}_{l-1}), \qquad \mathbf{r}_{l-1} = \gamma_{l-1} \odot \mathbf{M}_{l-1}(\mathbf{g}_{l-1} \odot \mathbf{q}_{l-1}), \tag{11}$$

$$\boldsymbol{\delta}_l = F_l\big(\mathrm{Norm}_l(\mathbf{x}_{l-1} + \mathbf{r}_{l-1})\big), \qquad \mathbf{x}_l = \mathbf{x}_{l-1} + \boldsymbol{\delta}_l,$$

$$\mathbf{k}_l = \mathrm{Key}(\mathbf{x}_l), \quad \mathbf{v}_l = \boldsymbol{\delta}_l, \qquad \mathbf{S}_l = \mathrm{Write}\big(\mathbf{S}_{l-1}, \mathbf{k}_l, \mathbf{v}_l\big), \tag{12}$$

with the RLS Write of Equation (8), the boundary layers of the main text, G groups of width D/G, the per-group read gate $\mathbf{g}_{l-1} \in \mathbb{R}^G$, the per-group write weight $\boldsymbol{\omega}_l \in \mathbb{R}^G$ and the learned per-group

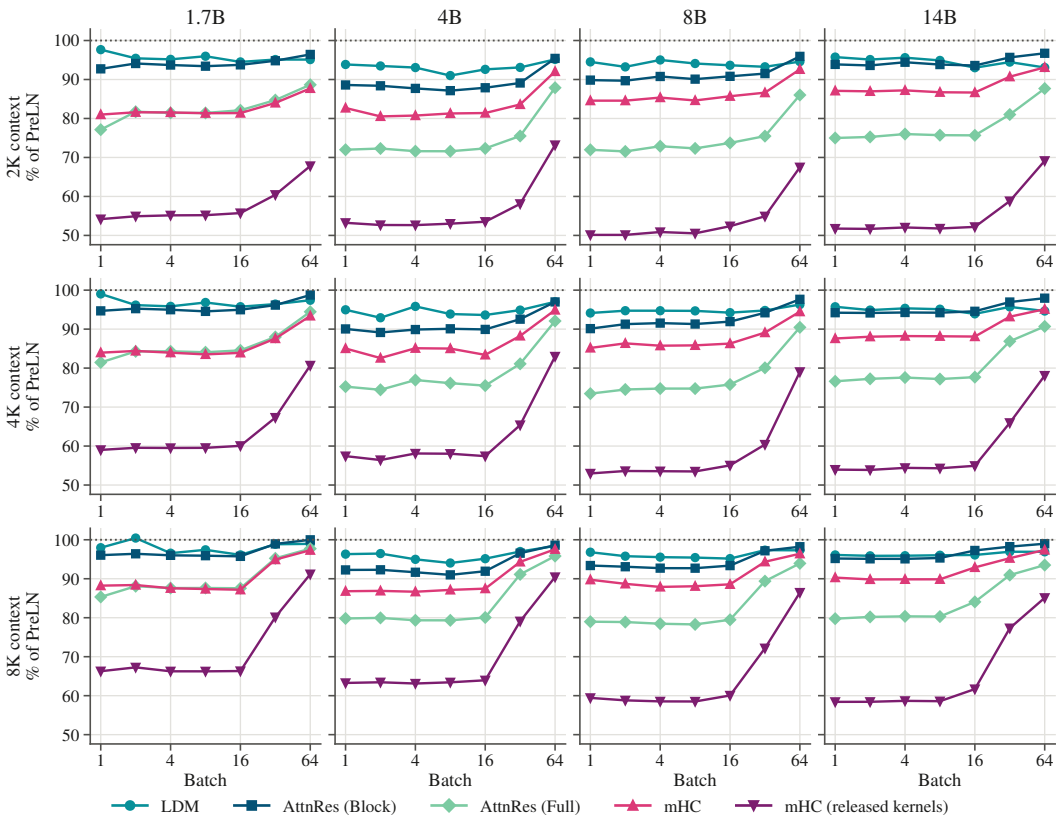


Figure 9: Decode throughput per generated token as a percentage of PreLN, by context length (rows) and model size (columns).

Table 8: Final held-out FineWeb-Edu cross-entropy after 15B training tokens (400M; step 14,305; seed 42). Groups match Table 3; the group counts $G = 1$ and $G = 8$ are those plotted in Figure 6. Δ is relative to LDM. Lower is better.

Group	Variant	Loss ↓	Δ
Baselines	PreLN	2.6554	+0.0567
	LDM	**2.5987**	0.0000
(a) Residual stream	$F_l(\mathbf{r}_{l-1})$	2.7296	+0.1310
	$\mathbf{x}_l = \boldsymbol{\delta}_l$	2.6611	+0.0625
(b) Recurrence	Delta	2.6028	+0.0041
	Outer	2.6037	+0.0050
(c) Addressing	$\mathbf{v}_l = \mathbf{x}_l$	2.6227	+0.0241
	$\mathbf{QK}(\boldsymbol{\delta}_l)$	2.6040	+0.0053
	Static $\mathbf{q}$	2.6128	+0.0141
	$G = 1$	2.6007	+0.0021
	$G = 8$	2.5955	−0.0032
(d) Gain, gates and forgetting	$-\gamma$	2.6014	+0.0027
	$\mathbf{g}_{l-1} = \mathbf{1}$	2.6035	+0.0048
	$\boldsymbol{\omega}_l = \mathbf{1}$	2.5992	+0.0006
	$\lambda_l = 1$	2.5981	−0.0006

read gain $\gamma_{l-1} \in \mathbb{R}^G$. Each ablation changes one of these equations. Everything else, including the training recipe, is fixed. The ablations are grouped as in Table 3.

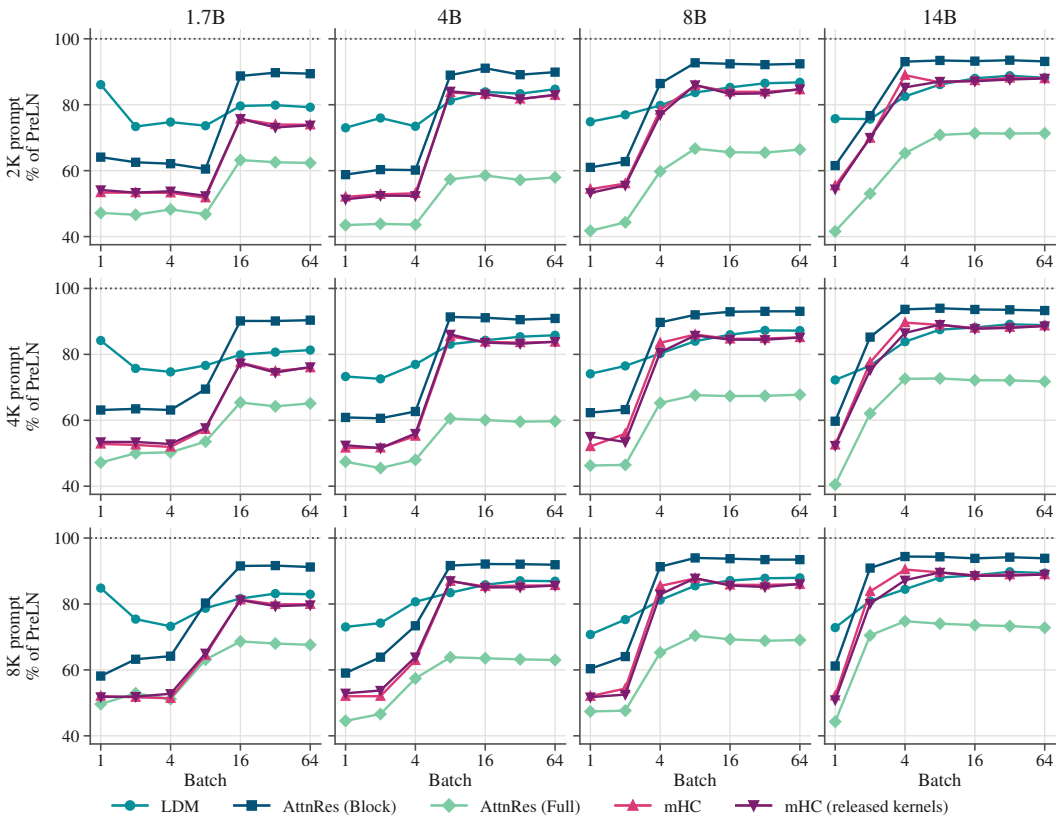


Figure 10: Time to first token relative to PreLN (PreLN's time divided by the method's), by prompt length (rows) and model size (columns).

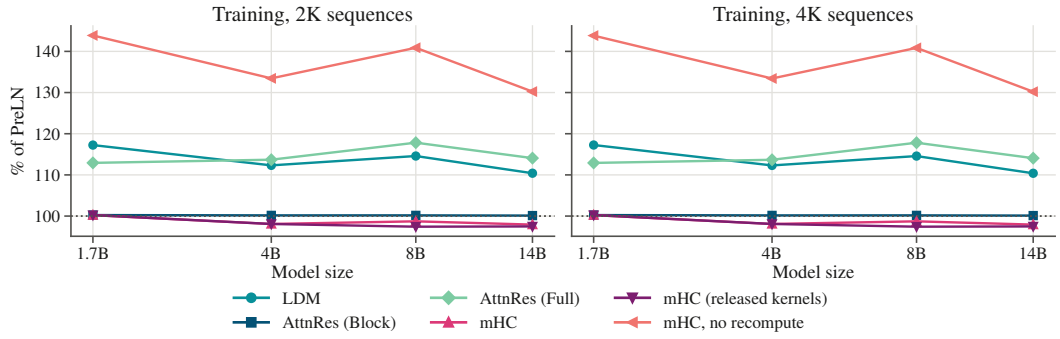


Figure 11: Peak memory per GPU of the training step as a percentage of PreLN.

(a) Residual stream: remove read addition/remove post-function residual addition. Both remove one of the two direct residual paths. *Remove read addition* replaces the stream by the read at every layer, including the final boundary read,

$$\boldsymbol{\delta}_l = F_l\big(\mathrm{Norm}_l(\mathbf{r}_{l-1})\big), \qquad \mathbf{x}_{\mathrm{out}} = \mathbf{r}_{\mathrm{out}}.$$

Remove post-function residual addition drops the skip connection after each layer, $\mathbf{x}_l = \boldsymbol{\delta}_l$ (the written value stays $\mathbf{v}_l = \boldsymbol{\delta}_l$). Each variant removes one of the two identity paths of the residual stream, the first at the layer input and the second at the layer output. For a fixed memory, the readout of each group lies in the column space of its memory, of dimension at most D_k; the memory and the addresses themselves still depend on the input.

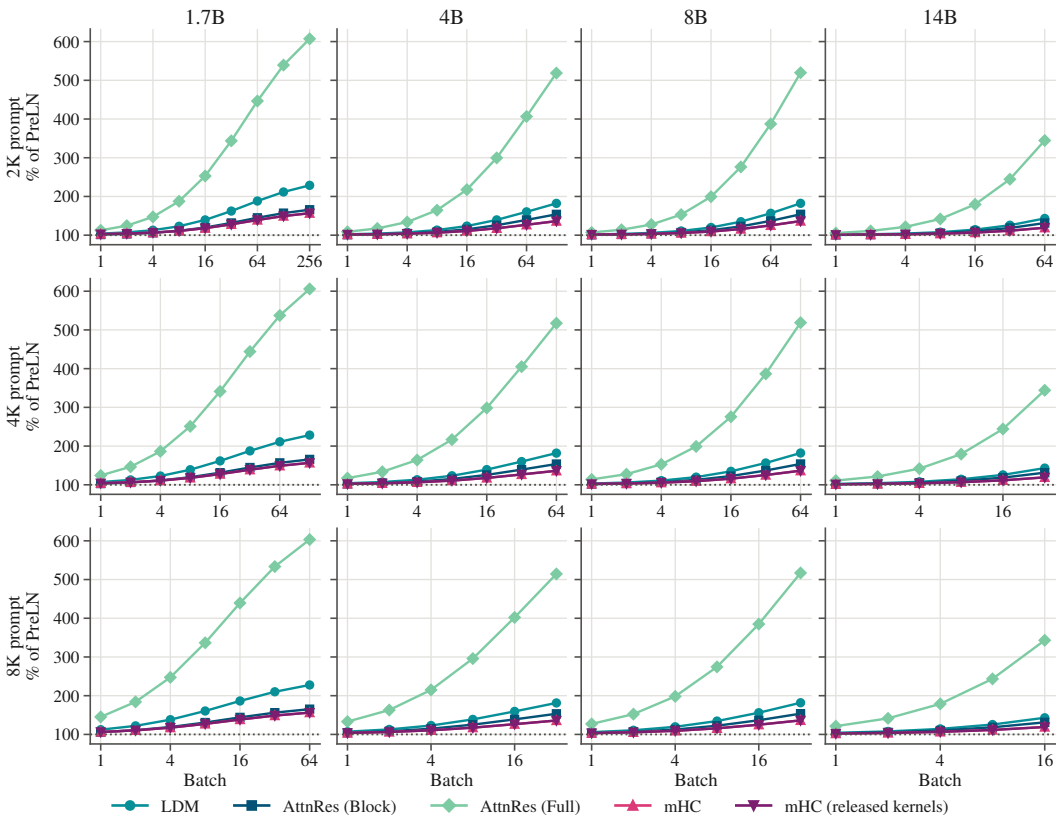


Figure 12: Peak memory of prefill as a percentage of PreLN, by prompt length (rows) and model size (columns).

(b) Recurrence: delta rule / outer product. Only Write changes. Addresses, gates, boundaries and the learned forgetting factor λ_l are those of LDM. With the weighted observations $\overline{\mathbf{k}}_l = \sqrt{\omega_l}\,\mathbf{k}_l$ and $\overline{\mathbf{v}}_l = \sqrt{\omega_l}\,\mathbf{v}_l$,

$$\begin{aligned}
\text{RLS (LDM):}\quad & \mathbf{M}_l = \mathbf{M}_{l-1} + \mathbf{u}_l\,\boldsymbol{\kappa}_l^\top, \qquad \boldsymbol{\kappa}_l = \frac{\mathbf{P}_{l-1}\overline{\mathbf{k}}_l}{\lambda_l + \overline{\mathbf{k}}_l^\top \mathbf{P}_{l-1}\overline{\mathbf{k}}_l}, \quad \mathbf{u}_l = \overline{\mathbf{v}}_l - \mathbf{M}_{l-1}\overline{\mathbf{k}}_l, \\
\text{Delta rule:}\quad & \mathbf{M}_l = \lambda_l \mathbf{M}_{l-1} + \mathbf{u}_l\,\overline{\mathbf{k}}_l^\top, \qquad \mathbf{u}_l = \overline{\mathbf{v}}_l - \lambda_l \mathbf{M}_{l-1}\overline{\mathbf{k}}_l, \\
\text{Outer product:}\quad & \mathbf{M}_l = \lambda_l \mathbf{M}_{l-1} + \overline{\mathbf{v}}_l\,\overline{\mathbf{k}}_l^\top.
\end{aligned}$$

The delta rule first decays the state by λ_l and then takes one gradient step of unit size on the instantaneous objective $\frac{1}{2}\|\overline{\mathbf{v}}_l - \mathbf{M}\overline{\mathbf{k}}_l\|_2^2$. The outer product decays the state and adds the raw association $\overline{\mathbf{v}}_l\overline{\mathbf{k}}_l^\top$, i.e., linear attention over depth with forgetting (Equation (3)). Neither maintains $\mathbf{P}_l$. RLS can be understood as applying forgetting through $\mathbf{P}_l$ instead of decaying the state.

(c) Addressing: store stream / function-output query and key / static query / group count. *Store stream* writes the updated residual stream instead of the layer output, $\mathbf{v}_l = \mathbf{x}_l$ in Equation (12). Since $\mathbf{x}_l = \mathbf{e} + \sum_{i=1}^{l} \boldsymbol{\delta}_i$, the memory then stores cumulative sums of earlier outputs rather than the outputs themselves. *Function-output query and key* changes the address sources while the stored value does not. In particular $\mathbf{q}_{l-1} = \mathrm{Query}(\boldsymbol{\delta}_{l-1})$, the most recent layer output, instead of $\mathrm{Query}(\mathbf{x}_{l-1})$, and $\mathbf{k}_l = \mathrm{Key}(\boldsymbol{\delta}_l)$ instead of $\mathrm{Key}(\mathbf{x}_l)$. For the embedding write $\boldsymbol{\delta}_0 = \mathbf{x}_0$, so the two coincide there. *Static query* replaces the query network by a vector learned for each layer, $\mathbf{q}_{l-1} = \mathbf{q}_{l-1}^{\mathrm{static}}$, the same for every token. The read gate is computed from the same input as the query, so it is fixed as well, $\mathbf{g}_{l-1} = \mathbf{1}$. *Group count* changes G, the number of memories per token: G memories of size $(D/G) \times D_k$, each with its own key, query and gate. LDM uses $G = 16$, which makes the value

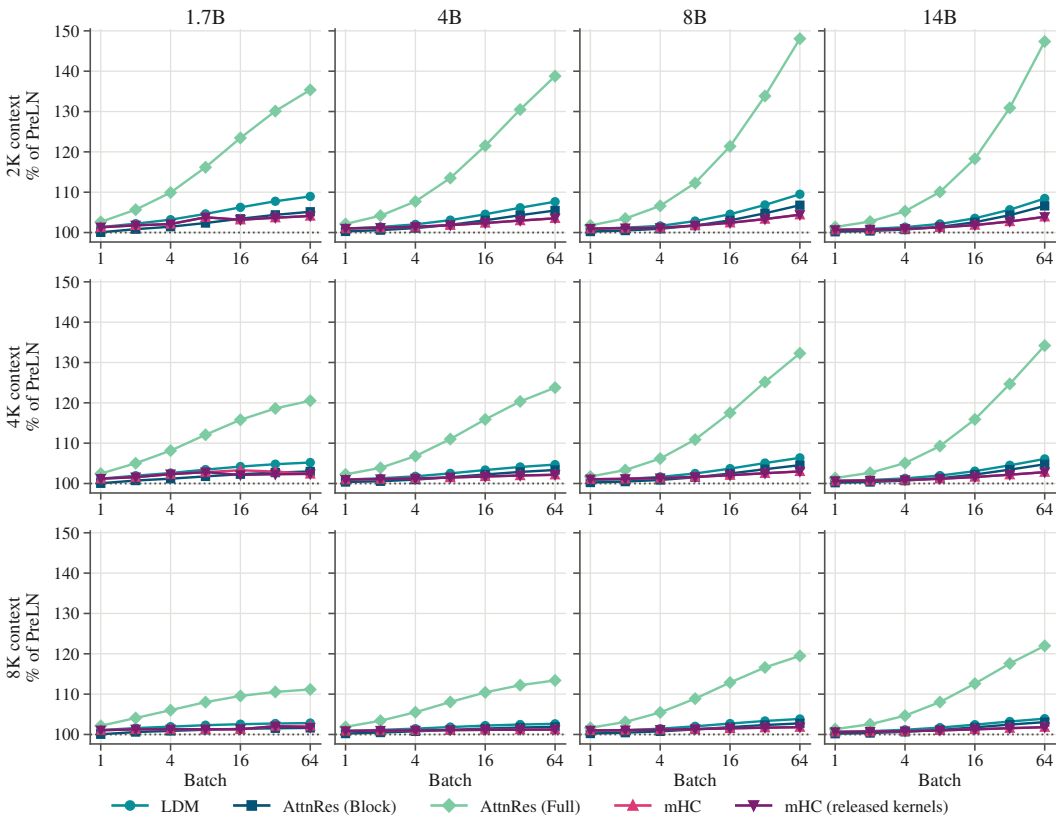


Figure 13: Peak memory of decode as a percentage of PreLN, by context length (rows) and model size (columns).

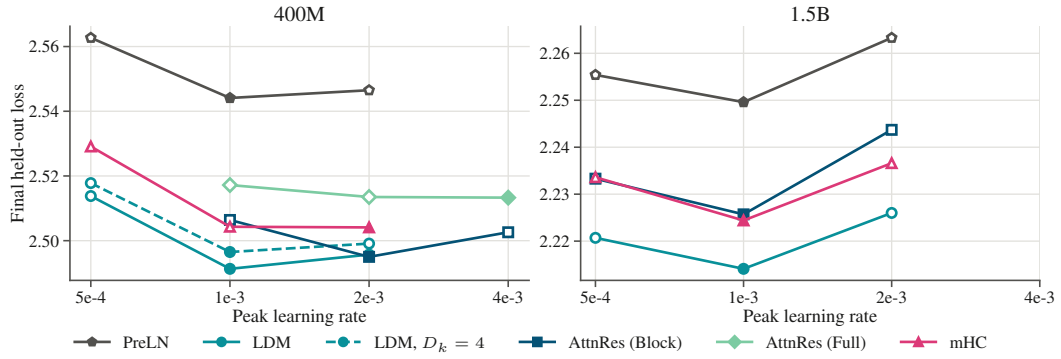


Figure 14: Final held-out loss against peak learning rate (the numbers of Table 7); the filled marker is each method's lowest loss.

width of a group the attention head dimension ($D/G = 64$ at 400M). *Single group* sets $G = 1$, one $D \times D_k$ memory per token; *eight groups* sets $G = 8$. The address networks keep their hidden width.

(d) Gain, gates and forgetting: no read gain / no R gate / no W gate / no forgetting. *No read gain* fixes $\gamma_{l-1} = \mathbf{1}$ in Equation (11). *No R gate* fixes $\mathbf{g}_{l-1} = \mathbf{1}$, so the read is $\mathbf{r}_{l-1} = \gamma_{l-1} \odot \mathbf{M}_{l-1}\mathbf{q}_{l-1}$. *No W gate* fixes $\boldsymbol{\omega}_l = \mathbf{1}$, so every write enters the least-squares objective of Equation (6) with unit weight. The corresponding gate heads are removed from the address networks. *No forgetting* fixes the forgetting factor at $\lambda_l = 1$ instead of learning it, so every earlier write keeps its full weight in the least-squares objective.

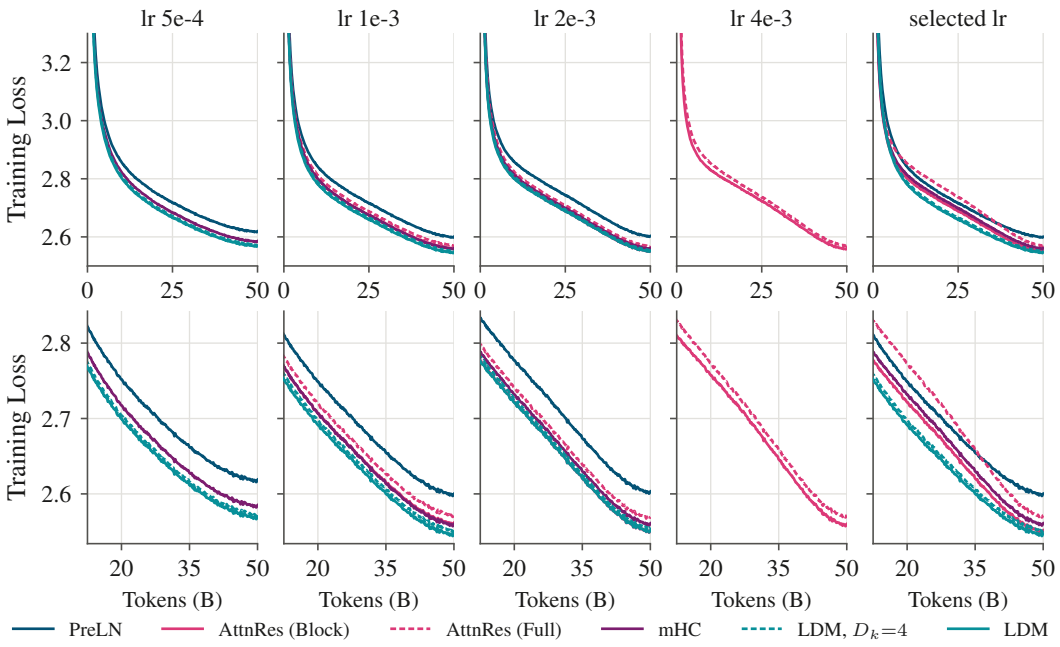


Figure 15: Training loss of the 400M learning rate sweeps over 50B FineWeb-Edu tokens. Each panel shows every method trained at one peak learning rate. The last panel shows each method at its selected LR. Top: all tokens. Bottom: from 12.5B tokens (25% of training) onwards.

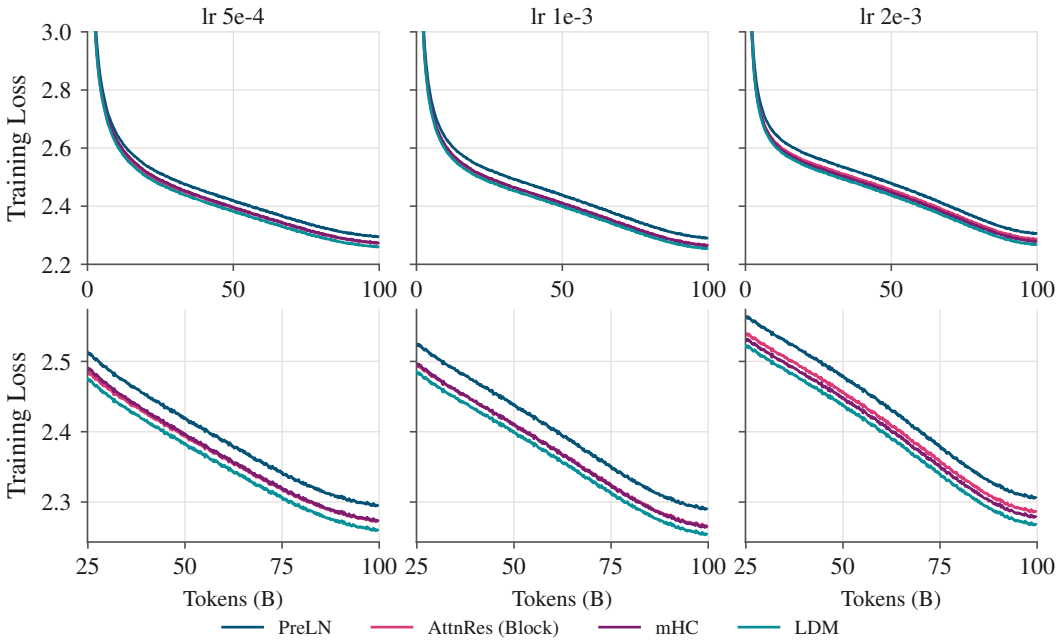


Figure 16: Training loss of the 1.5B learning rate sweeps over 100B FineWeb-Edu tokens. Each panel shows every method trained at one peak learning rate. Every method's selected rate is 10^{-3}. Top: the whole run. Bottom: from 24.9B tokens (25% of training) onwards.

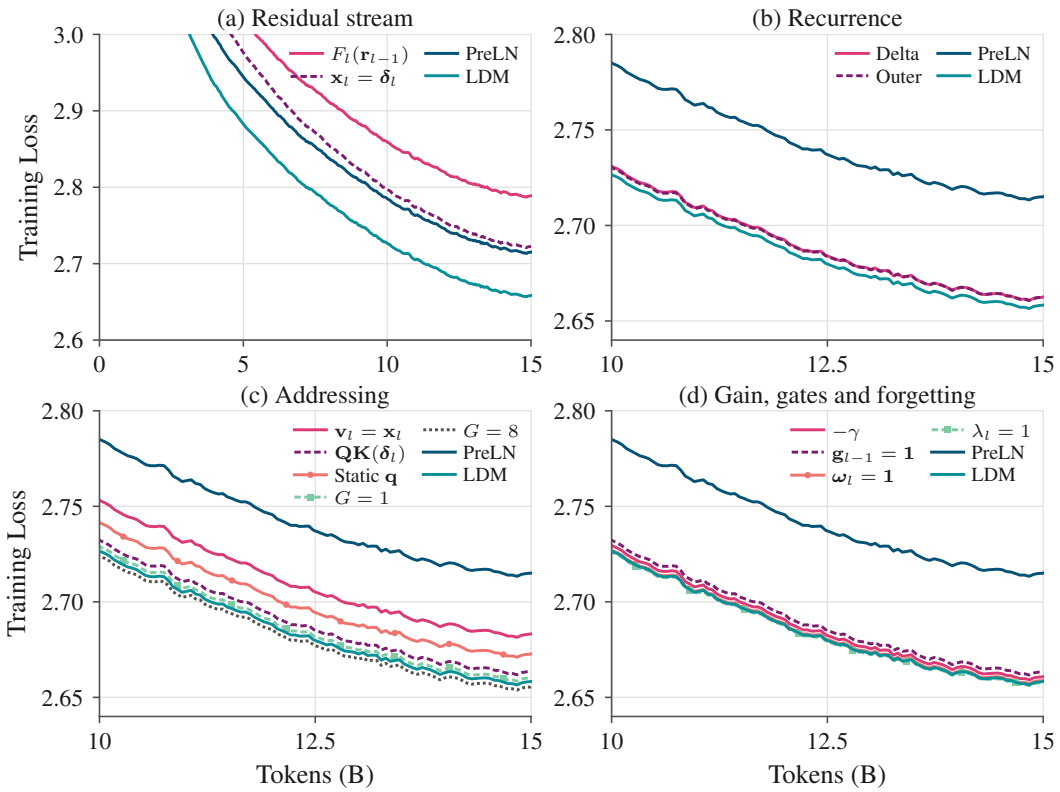


Figure 17: Training loss of the 400M ablation runs trained on 15B FineWeb-Edu tokens, grouped as in Table 8. (a) Residual stream: giving only the memory read to the layer, $F_l(\text{Read})$ (normalization and read arguments are omitted), and removing the residual addition after the layer, $\mathbf{x}_l = \boldsymbol{\delta}_l$. (b) Recurrence: LDM uses RLS, Delta uses the delta rule, and Outer uses a plain outer-product update. (c) Addressing: storing the residual stream, $\mathbf{v}_l = \mathbf{x}_l$, generating queries and keys from layer outputs, $\mathbf{QK}(\boldsymbol{\delta}_l)$, a static query per layer (Static $\mathbf{q}$), and one or eight groups ($G = 1$, $G = 8$). (d) Gain, gates and forgetting: fixing the read gain ($-\gamma$), the read gate ($\mathbf{g}_{l-1} = \mathbf{1}$), the write weight ($\boldsymbol{\omega}_l = \mathbf{1}$) or the forgetting factor ($\lambda_l = 1$) to one. PreLN and LDM are shown as references in every panel. (b) to (d) show the last third of training. Final held-out losses for all variants appear in Table 8.

E LDM DETAILS

E.1 FULL DERIVATION OF OUR LEAST SQUARES OBJECTIVE

For conciseness, fix one group, with $\mathbf{M}_l \in \mathbb{R}^{D_v \times D_k}$ and $\mathbf{P}_l \in \mathbb{R}^{D_k \times D_k}$. We write l maps the state after write $l-1$ to the state after write l. Write at index 0 stores the embedding. Let $\alpha > 0$, $\lambda_l > 0$, and $\omega_l \geq 0$. The weighted objective is Equation (6), with empty products equal to one. Define

$$\mathbf{A}_l = \sum_{i=0}^{l} \left(\prod_{j=i+1}^{l} \lambda_j \right) \omega_i \mathbf{k}_i \mathbf{k}_i^\top + \left(\prod_{j=0}^{l} \lambda_j \right) \alpha^{-1} \mathbf{I},$$
$$\mathbf{B}_l = \sum_{i=0}^{l} \left(\prod_{j=i+1}^{l} \lambda_j \right) \omega_i \mathbf{v}_i \mathbf{k}_i^\top.$$

The objective gradient is $2(\mathbf{M}\mathbf{A}_l - \mathbf{B}_l)$. Since $\mathbf{A}_l$ is positive definite, the unique minimizer is $\mathbf{M}_l = \mathbf{B}_l \mathbf{A}_l^{-1}$. With $\mathbf{A}_{-1} = \alpha^{-1}\mathbf{I}$ and $\mathbf{B}_{-1} = 0$,

$$\mathbf{A}_l = \lambda_l \mathbf{A}_{l-1} + \overline{\mathbf{k}}_l \overline{\mathbf{k}}_l^\top, \qquad \mathbf{B}_l = \lambda_l \mathbf{B}_{l-1} + \overline{\mathbf{v}}_l \overline{\mathbf{k}}_l^\top,$$

where $\overline{\mathbf{k}}_l = \sqrt{\omega_l}\mathbf{k}_l$ and $\overline{\mathbf{v}}_l = \sqrt{\omega_l}\mathbf{v}_l$.

Forward inverse covariance update. Set $\mathbf{P}_l = \mathbf{A}_l^{-1}$. Sherman–Morrison gives

$$\mathbf{P}_l = \frac{1}{\lambda_l} \left(\mathbf{P}_{l-1} - \frac{\mathbf{P}_{l-1} \overline{\mathbf{k}}_l \overline{\mathbf{k}}_l^\top \mathbf{P}_{l-1}}{\lambda_l + \overline{\mathbf{k}}_l^\top \mathbf{P}_{l-1} \overline{\mathbf{k}}_l} \right).$$

The denominator is strictly positive. With the gain from Equation (7), it is easy to see that this is the covariance update in Equation (8). In particular, $\overline{\mathbf{k}}_l^\top \mathbf{P}_l = \boldsymbol{\kappa}_l^\top$.

Forward memory update. Using $\mathbf{B}_{l-1} = \mathbf{M}_{l-1}\mathbf{A}_{l-1}$, we have that

$$\begin{aligned} \mathbf{M}_l &= (\lambda_l \mathbf{B}_{l-1} + \overline{\mathbf{v}}_l \overline{\mathbf{k}}_l^\top)\mathbf{P}_l \\ &= \mathbf{M}_{l-1}(\mathbf{I} - \overline{\mathbf{k}}_l \overline{\mathbf{k}}_l^\top \mathbf{P}_l) + \overline{\mathbf{v}}_l \overline{\mathbf{k}}_l^\top \mathbf{P}_l \\ &= \mathbf{M}_{l-1} + (\overline{\mathbf{v}}_l - \mathbf{M}_{l-1}\overline{\mathbf{k}}_l)\boldsymbol{\kappa}_l^\top = \mathbf{M}_{l-1} + \mathbf{u}_l \boldsymbol{\kappa}_l^\top. \end{aligned}$$

Gating and regularization. The dynamic weight ω_l sets the "importance" of observation l and λ_l discounts all earlier observations (and the initial ridge penalty). A fixed ridge penalty would instead introduce an additional full-rank term $(1-\lambda_l)\alpha^{-1}\mathbf{I}$ into the covariance recurrence. That alternative remains solvable, but no longer has the same simple Sherman–Morrison rank-one update we use.

E.2 BOUNDARY CASES

Before writing the embedding $\mathbf{e}$, we initialize the memory as

$$\mathbf{M}_{-1} = \mathbf{0}, \qquad \mathbf{P}_{-1} = \alpha \mathbf{I}.$$

The initial value written into the memory is $\mathbf{v}_0 = \mathbf{e}$, and its key and weight are generated from $\mathbf{e}$ using the same write-side parametrization, yielding

$$(\mathbf{e}, \mathbf{M}_0, \mathbf{P}_0) = \text{Write}\left((\mathbf{0}, \mathbf{M}_{-1}, \mathbf{P}_{-1}), \mathbf{k}_0, \mathbf{v}_0\right).$$

After the final transformer layer, one final read enters the residual stream directly. We generate $\mathbf{q}_{\text{out}}$ from $\mathbf{x}_L$ using the same parametrization and define

$$\mathbf{x}_{\text{out}} = \text{Read}\left((\mathbf{x}_L, \mathbf{M}_L, \mathbf{P}_L), \mathbf{q}_{\text{out}}\right).$$

The final representation $\mathbf{x}_{\text{out}}$ is then passed to the LM head.

F HARDWARE-AWARE IMPLEMENTATION

This appendix shows the implementation details in Section 3.3. All quantities below are per token and per group; groups are processed in parallel.

We write $\overline{\mathbf{k}}_l = \sqrt{\omega_l}\mathbf{k}_l$ and $\overline{\mathbf{v}}_l = \sqrt{\omega_l}\mathbf{v}_l$ for the weighted key and value of a Write, and $\tilde{\mathbf{q}}_l = \gamma_l\, \mathbf{g}_l \odot \mathbf{q}_l$ for the gated query so that the read is $\mathbf{r}_l = \mathbf{M}_l\tilde{\mathbf{q}}_l$. We also define $\rho_l = \lambda_l + \overline{\mathbf{k}}_l^\top \mathbf{P}_{l-1}\overline{\mathbf{k}}_l$, the denominator in the gain $\boldsymbol{\kappa}_l$ of Equation (7). Step l is the Write at layer l followed by the Read that supplies layer $l+1$. Step 0 writes the embedding, while step L ends with the final read. Thus an L-layer model can be understood as having $L+1$ steps.

F.1 FORWARD

We partition the steps into windows $\{b+1, \dots, e\}$ of at most E steps, with $e = \min(b+E, L)$. Within a window, the kernel keeps the boundary state $\mathbf{M}_b$ together with the pseudo-values and gains $(\mathbf{u}_i, \boldsymbol{\kappa}_i)$ already computed in that window. Using Equation (10), step l is evaluated as

$$\mathbf{u}_l = \overline{\mathbf{v}}_l - \mathbf{M}_b\overline{\mathbf{k}}_l - \sum_{b<i<l} \mathbf{u}_i\,(\boldsymbol{\kappa}_i^\top \overline{\mathbf{k}}_l), \qquad \mathbf{r}_l = \mathbf{M}_b\tilde{\mathbf{q}}_l + \sum_{b<i\le l} \mathbf{u}_i\,(\boldsymbol{\kappa}_i^\top \tilde{\mathbf{q}}_l).$$

The memory itself is written only at the end of the window,

$$\mathbf{M}_e = \mathbf{M}_b + \sum_{b<i\le e} \mathbf{u}_i\boldsymbol{\kappa}_i^\top.$$

The inverse covariance does not depend on $\mathbf{M}$ and is still updated in place at every step.

For the backward pass, each step stores $\overline{\mathbf{k}}_l$, $\tilde{\mathbf{q}}_l$, $\boldsymbol{\kappa}_l$, ρ_l, ω_l, and λ_l in FP32. It also stores the pre-update covariance $\mathbf{P}_{l-1}$ in packed form and the pseudo-value $\mathbf{u}_l$ in BF16. These quantities make up the saved context referred to in Section 3.3. Saving $\mathbf{M}_l$ at every step instead would require D_k rows of width D per step.

F.2 BACKWARD

Let $\mathrm{d}a = \partial\mathcal{L}/\partial a$, and let $\mathrm{d}\mathbf{r}_l$ denote the gradient arriving at the read of step l from the residual stream. All steps in the next window read from and write through $\mathbf{M}_e$, so the gradient associated with the end-of-window state is

$$\mathrm{d}\mathbf{M}_e = \mathrm{d}\mathbf{M}_{e'} + \sum_{j=e+1}^{e'} \left(\mathrm{d}\mathbf{r}_j\tilde{\mathbf{q}}_j^\top - \mathrm{d}\overline{\mathbf{v}}_j\overline{\mathbf{k}}_j^\top\right), \qquad e' = \min(e+E, L). \tag{13}$$

This gradient is carried from one window to the preceding one. Here $\mathrm{d}\mathbf{M}_e$ contains the contribution from steps after e; reads inside the current window are handled by Equation (14). For the last window, $\mathrm{d}\mathbf{M}_L = \mathbf{0}$.

The dependence on later windows enters step i through

$$\mathbf{f}_i = \mathrm{d}\mathbf{M}_e\boldsymbol{\kappa}_i, \qquad \mathbf{z}_i = \mathrm{d}\mathbf{M}_e^\top\mathbf{u}_i.$$

The intermediate states

$$\mathbf{M}_i = \mathbf{M}_b + \sum_{b<m\le i} \mathbf{u}_m\boldsymbol{\kappa}_m^\top$$

are reconstructed from $\mathbf{M}_b$ and the saved rank-one updates, just as in the forward pass. The resulting vector–Jacobian products within the window are

$$\begin{aligned}
&\mathrm{d}\tilde{\mathbf{q}}_i = \mathbf{M}_i^\top\mathrm{d}\mathbf{r}_i, \quad \mathrm{d}\overline{\mathbf{v}}_i = \mathrm{d}\mathbf{u}_i = \mathbf{f}_i + \sum_{j\ge i}(\boldsymbol{\kappa}_i^\top\tilde{\mathbf{q}}_j)\,\mathrm{d}\mathbf{r}_j - \sum_{j>i}(\boldsymbol{\kappa}_i^\top\overline{\mathbf{k}}_j)\,\mathrm{d}\overline{\mathbf{v}}_j,\\
&\mathrm{d}\boldsymbol{\kappa}_i = \mathbf{z}_i + \sum_{j\ge i}(\mathbf{u}_i^\top\mathrm{d}\mathbf{r}_j)\,\tilde{\mathbf{q}}_j - \sum_{j>i}(\mathbf{u}_i^\top\mathrm{d}\overline{\mathbf{v}}_j)\,\overline{\mathbf{k}}_j, \quad \mathrm{d}\overline{\mathbf{k}}_i = -\mathbf{M}_{i-1}^\top\mathrm{d}\overline{\mathbf{v}}_i + \mathrm{d}\overline{\mathbf{k}}_i^{\mathbf{P}}.
\end{aligned} \tag{14}$$

Here $\mathrm{d}\overline{\mathbf{k}}_i^{\mathbf{P}}$ is the contribution through the gain $\boldsymbol{\kappa}_i$ and the covariance recursion. This part is differentiated one step at a time, with the gradient of the symmetric covariance stored in upper-triangular form (which slightly reduces memory). The same calculation yields $\mathrm{d}\lambda_i$. The write weight has gradient

$$\mathrm{d}\omega_i = \frac{\overline{\mathbf{k}}_i^\top \mathrm{d}\overline{\mathbf{k}}_i + \overline{\mathbf{v}}_i^\top \mathrm{d}\overline{\mathbf{v}}_i}{2\omega_i},$$

after which $\mathrm{d}\mathbf{q}_i$, $\mathrm{d}\mathbf{g}_i$, and $\mathrm{d}\gamma_i$ follow directly from $\mathrm{d}\tilde{\mathbf{q}}_i$.

Unwinding. At the end of each window, backward reconstructs the state at the start of the window using Equation (9),

$$\mathbf{M}_b = \mathbf{M}_e - \sum_{b<i\le e} \mathbf{u}_i \boldsymbol{\kappa}_i^\top.$$

The reconstructed $\mathbf{M}_b$ overwrites $\mathbf{M}_e$. Training therefore needs only one state matrix per token at any point in the backward pass. States inside the window are generated from $\mathbf{M}_b$ and the saved pseudo-values and gains in the same way as in the forward pass. Algorithm 1 summarizes the computation for one window.

Algorithm 1 Backward pass over the window $\{b+1,\dots,e\}$, for one group and one token.

Require: $\mathbf{M}_e$; $\mathrm{d}\mathbf{M}_{e'}$ and the gradients of the next window's steps ($\mathbf{0}$ for the last window); for $b < i \le e$ the saved $\overline{\mathbf{k}}_i, \tilde{\mathbf{q}}_i, \boldsymbol{\kappa}_i, \rho_i, \omega_i, \lambda_i, \mathbf{P}_{i-1}$ and $\mathbf{u}_i$, the values $\overline{\mathbf{v}}_i$, and the read gradients $\mathrm{d}\mathbf{r}_i$
Ensure: $\mathbf{M}_b$; $\mathrm{d}\mathbf{M}_e$; the gradients of the inputs of every step in the window
1: $\mathrm{d}\mathbf{M}_e \leftarrow$ Equation (13)
2: $\mathbf{M}_b \leftarrow \mathbf{M}_e - \sum_{b<i\le e} \mathbf{u}_i \boldsymbol{\kappa}_i^\top$ (Equation (9)), overwriting $\mathbf{M}_e$
3: $\mathbf{f}_i \leftarrow \mathrm{d}\mathbf{M}_e \boldsymbol{\kappa}_i$, $\mathbf{z}_i \leftarrow \mathrm{d}\mathbf{M}_e^\top \mathbf{u}_i$ for $b < i \le e$
4: **for** $i = e, \dots, b+1$ **do**
5: $\quad \mathrm{d}\overline{\mathbf{v}}_i,\ \mathrm{d}\boldsymbol{\kappa}_i,\ \mathrm{d}\tilde{\mathbf{q}}_i \leftarrow$ Equation (14)
6: $\quad \mathrm{d}\overline{\mathbf{k}}_i,\ \mathrm{d}\lambda_i,\ \mathrm{d}\mathbf{P}_{i-1} \leftarrow$ backward of the gain and covariance update; $\quad \mathrm{d}\omega_i$

In the implementation, lines 1–3 together with step e are handled by a single "window end" kernel. Every other step uses an "interior" kernel. This leaves the backward pass through each step's address networks in the usual form. The residual addition after each read and the following branch normalization are recomputed during backward rather than saved.

F.3 Memory of the State

Table 9 counts memory accesses due to the LDM state traffic, per token and per step, measured in rows of D BF16 values. The state consists of the memory $\mathbf{M}$ (D_k rows across all groups), the packed covariance $\mathbf{P}$ ($p = GD_k(D_k+1)/(2D)$ rows), and the pseudo-value $\mathbf{u}$ (one row).

The "storing every state" baseline saves both $\mathbf{M}_l$ and $\mathbf{P}_l$ for backward. Unwinding removes the need to save $\mathbf{M}_l$, retaining only $\mathbf{u}_l$ and the covariance before its update. Windowing further reduces traffic by replacing the write to $\mathbf{M}$ at every step with one write per window, in exchange for reading the pseudo-values from earlier steps in that window. All three variants also save a small FP32 "context" containing keys, queries, gains, and scalar factors. At $D = 2048$, this amounts to 0.42 rows per step and is omitted from the table.

Table 9: Memory of LDM's state per token and step, in rows of D BF16 values, for $G = 16$, $D_k = 8$, $E = 4$, and $D = 2048$ ($p = 0.28$).

	Forward accesses	Saved for the backward pass
Storing every state	$2D_k + 2p = 16.6$	$D_k + p = 8.3$
Unwinding	$2D_k + 3p + 1 = 17.8$	$1 + p = 1.3$
Unwinding + Windowing	$D_k + D_k/E + (E-1)/2 + 3p + 1 = 13.3$	$1 + p = 1.3$

F.4 KERNELS

Our kernels are implemented in CuTe DSL for Hopper and Blackwell GPUs and are released with our code. Launch configurations can be tuned offline for each model shape and GPU.

The two address networks at a step consume the same normalized residual stream and their first layers are evaluated together as a single GEMM to save memory traffic. We store $\mathbf{M}$, $\mathbf{u}$, and $\mathbf{P}$ in BF16. Keys, queries, gains, denominators, and arithmetic are generally FP32.

Inference uses separate forward kernels and does not save backward context. During prefill, a given layer's Write and the following layers's Read are split across three kernels: a fused kernel for the preceding residual addition, normalization, and first layers of both address networks. Also a GEMM for their heads and gates and a final kernel for the gains, covariance update, memory step, and branch normalization. During decoding, where the forward pass contains only a small number of tokens, all three stages are fused into a single kernel to avoid latency.