EFFICIENT TRAINING OF LINEAR TRANSFORMERS WITH BLOCK-DIAGONAL GATES

Anonymous authors
Paper under double-blind review

ABSTRACT

Linear Transformers have established themselves as efficient alternatives to softmax attention for sequence modeling, to the point that modern LLMs increasingly rely on hybrid designs for large-scale deployment. Their primary advantage stems from compressing context into a fixed-size hidden state that is updated recurrently, thus disposing of an ever-growing KV cache. The widespread adoption of Linear Transformers drives the push for their improvement, which hinges on enriching the structure of the state-transition matrix, or *gate*, governing their recurrent update. This structure has progressed from scalar, to diagonal, to diagonal plus a low-rank perturbation: each step broadened the transformations the state could undergo, and was unlocked by dedicated *chunk-wise parallel* algorithms that enabled efficient training at scale. Block-diagonal transitions have recently emerged as the relevant next step, to boost expressivity via richer state mixing. However, the lack of an adequate training algorithm has so far restricted their application to either small scales or to specific block families. In this work, we develop a *generalized chunk-wise parallel* algorithm that lifts these restrictions, together with efficient hardware-aware kernels for its implementation. We use it to introduce and train a Block-diagonal Gated Linear Transformer (BGLT) at 7B scale. To keep parameter count comparable to existing methods, BGLT employs 16×16 transition blocks, further expressed as a Kronecker product of two 4×4 factors. Despite this compact representation, BGLT attains superior performance on a range of synthetic tasks, solving S_5 in a single layer. On language modeling tasks, it shows results competitive with baselines at 7B. Ultimately, BGLT provides a blueprint for deploying more general recurrences that remain efficiently trainable at scale.

1 INTRODUCTION

Linear Transformers, and more broadly linear RNNs, represent efficient alternatives to softmax attention for sequence modeling, combining the linear-time, constant-memory cost of classical RNNs (Hochreiter & Schmidhuber, 1997) with the parallelism of Transformers (Vaswani et al., 2017). With their recent adoption in large-scale hybrid foundation models (Kimi Team, 2026; Qwen Team, 2026), improving their underlying recurrence has become a practical priority. These improvements primarily target the recurrence state-transition matrix, or *gate*. By governing how different state components interact across tokens, the gate determines the richness of the hidden state dynamics that the model can track (see Fig. 1), and ultimately its expressivity (Cirone et al., 2024). To date, these interactions have been kept minimal: scalar (Sun et al., 2023; Dao & Gu, 2024) and diagonal (Yang et al., 2024a; Qin et al., 2024) transitions leave coordinates entirely uncoupled, provably capping state-tracking capabilities (Merrill et al., 2024). Even a rank-1 perturbation of a diagonal gate, which permits mixing along a single direction, yields sizable gains (Yang et al., 2024b; 2025a; Peng et al., 2025; Kimi Team et al., 2025), but capturing more complex trajectories requires richer state interactions. While this can be achieved by composing several rank-1 perturbations (Siems et al., 2025), doing so incurs a computational cost growing with rank. Block-diagonal transitions (Walker et al., 2025; Dubinin et al., 2026) offer a more effective alternative, since they freely mix coordinates within blocks at fixed cost. Yet, despite their theoretical appeal, their deployment at scale remains confined to restricted block families (Movahedi et al., 2025; Lahoti et al., 2026).

The main reason behind these structural limitations is computational. To train efficiently, the RNN application must be parallelized across the sequence length. This is feasible in principle via *parallel*

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.

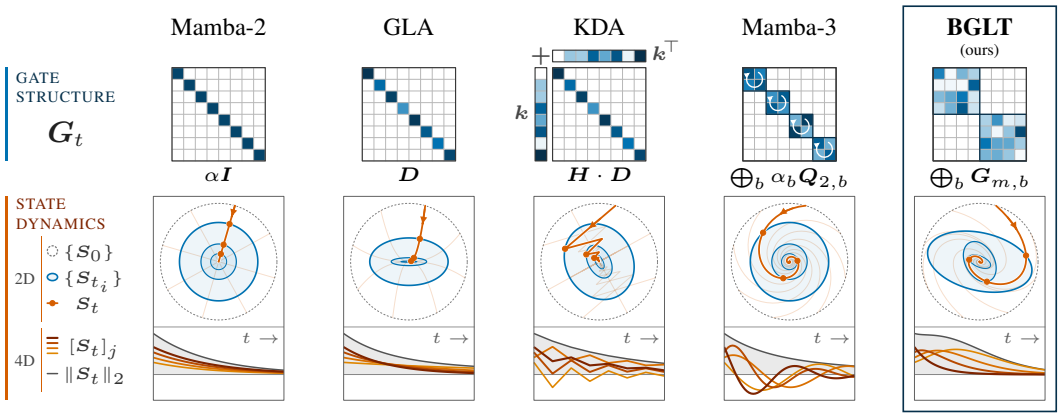


Figure 1: **General gates unlock expressivity** A Linear Transformer can be interpreted as an RNN, whose hidden-state $\boldsymbol{S}_t$ dynamics are tied to the transition matrix $\boldsymbol{G}_t$. Imposing structure on $\boldsymbol{G}_t$ (e.g., diagonal $\boldsymbol{D}$, Householder $\boldsymbol{H}$, or orthogonal $\boldsymbol{Q}$) restricts the range of dynamics the model can track, affecting its expressive power. Only unstructured gates provide the whole range. We report examples trajectories for $\boldsymbol{S}_t$ (orbits in 2D, individual components in 4D) for structures currently in use.

scan (Blelloch, 1989), provided the gates are cheap to store and multiply. But even when these conditions are met, parallel scan remains suboptimal for large-scale training: it suffers from low arithmetic intensity, and requires fitting both states and transitions within the memory budget of modern accelerators (NVIDIA Corporation, 2022), which caps hidden-state size, hurting the model recall capacity (Arora et al., 2024a). For training at scale, a better alternative to parallel scan is *chunk-wise parallel application* (Hua et al., 2022; Dao & Gu, 2024), which consists in splitting the sequence into chunks, and evaluating intra-chunk updates in parallel via matrix products, thus leveraging the efficiency of Tensor Cores. In particular, this requires computing chunk-local analogues of Attention matrices, whose entries are given by inner products of queries and keys at different positions, weighted by the products of the gates in between. However, their efficient assembly introduces an additional requirement on the gates, as it currently relies on *factorizing* their products onto queries and keys. With scalar or diagonal gates, factorization is free. For products of diagonal-plus-low-rank (DPLR) this is also doable, via their UT representation (Bischof & Van Loan, 1987). General block-diagonal matrices, however, do not lend themselves to factorization: naïve attempts relying on matrix inversion are both computationally expensive and prone to numerical instability. This is what restricted prior work to use orthogonal blocks (Lahoti et al., 2026), reducing state dynamics to mere rotations.

We resolve this bottleneck by developing a *generalized chunk-wise parallel form* which bypasses the need for gate inversion to efficiently assemble chunk-local attention matrices, thus unlocking large-scale training of general block-diagonal recurrences. Enabled by this algorithm, we introduce *BGLT*, a highly-expressive Linear Transformer employing input-dependent Block-diagonal Gates. To match the parameter count of current baselines, BGLT parameterizes its 16×16 gate blocks as Kronecker products of two 4×4 factors. Despite parameter parity, BGLT exhibits significantly broader expressivity than DPLR baselines, solving a range of synthetic tasks, including S_5 state-tracking, in a single layer. We provide the parameterizations and training recipes to scale BGLT stably to 7B parameters, where it shows performance competitive with KDA (Kimi Team et al., 2025) on language modeling. In summary, our contributions are as follows:

- **Generalized chunk-wise parallel algorithm (Sec. 3.1):** An efficient, Tensor-Core friendly algorithm for assembling chunk-local attention matrices with general block-diagonal transitions, with optimized CuTe (Cecka, 2026) implementations, to be released on GitHub. Additionally, we improve the lower-triangular solver for Delta Rule via bidiagonally-preconditioned Newton-Schulz.
- **BGLT (Sec. 3.2):** A Linear Transformer with input-dependent Kronecker-factorized block-diagonal gates parameterized via SVD. Our parameterization trains stably at scale, and offers significantly more expressivity over competitors, at matched parameter count and comparable throughput.
- **Empirical evaluation (Sec. 4):** BGLT attains superior performance on a range of synthetic tasks, solving S_5 in a single layer and beating baselines on NCA (Lee et al., 2026). At 7B scale, it shows results competitive with KDA (Kimi Team et al., 2025) on language modeling tasks.

2 BACKGROUND

In this section we introduce the linear recurrence forming the backbone of modern Linear Transformers, and highlight the two competing forces shaping its design: on the one hand, the drive to boost expressivity; on the other, the necessity for efficient parallelism. Both are governed by the transition matrix parameterization. We refer to App. A for an extended discussion of related work.

Linear RNNs Inside a modern Linear Transformer architecture, the core operation responsible for mixing information across the sequence comes in the form of a linear recurrence with a matrix-valued hidden state. At position t, the input $\boldsymbol{x}_t \in \mathbb{R}^d$ is mapped by learned projections to a query and a key $\boldsymbol{q}_t, \boldsymbol{k}_t \in \mathbb{R}^{d_k}$, a value $\boldsymbol{v}_t \in \mathbb{R}^{d_v}$, and a state-transition matrix $\boldsymbol{G}_t \in \mathbb{R}^{d_k \times d_k}$, also known as *gate*. This matrix dictates how the hidden state evolves from the previous position $\boldsymbol{S}_{t-1} \in \mathbb{R}^{d_k \times d_v}$ to the next. Concurrently, the key and value are applied as an outer-product additive update, while the query reads from the current state to produce the output $\boldsymbol{o}_t \in \mathbb{R}^{d_v}$:

$$\text{UPDATE:} \quad \boldsymbol{S}_t = \boldsymbol{G}_t \boldsymbol{S}_{t-1} + \boldsymbol{k}_t \boldsymbol{v}_t^\top, \qquad \text{OUTPUT:} \quad \boldsymbol{o}_t = \boldsymbol{S}_t^\top \boldsymbol{q}_t. \tag{1}$$

Crucially, this outer-product structure yields a matrix-valued hidden state $\boldsymbol{S}_t$: this is particularly appealing as it expands the memory capacity of the recurrence without having to materialize the full state (Gu & Dao, 2023). This structure also directly connects the recurrence to attention, allowing its application to be recast into tensor operations, thus providing the foundation for efficient large-scale training. The primary design choice, and the main differentiator among modern Linear RNNs, lies in the parameterization of the transition matrix $\boldsymbol{G}_t$. As summarized in Tab. 1 and (Yang et al., 2024b, Tab.2), choices range from scalar in Mamba-2 (Dao & Gu, 2024), to Householder×Diagonal in KDA (Kimi Team et al., 2025). This choice ultimately balances expressivity against parallelizability. While enriching $\boldsymbol{G}_t$ captures more complex state dynamics (cf. Fig. 1), doing so naïvely breaks the applicability of the parallel algorithms needed to train at scale. We examine this trade-off next.

Expressivity We broadly interpret the *expressivity* of an RNN as its capacity for general function approximation and the class of computations it can represent, often measured on synthetic tasks. Particularly relevant are *state-tracking* tasks (Merrill et al., 2024), testing the ability to maintain a running summary of sequential updates. The simplest such task is *parity* (tracking the even/odd count of a binary sequence), which generalizes to tracking permutations of n elements: this is known as the *word problem* of the symmetric group S_n. As n increases, the complexity class of this task shifts from TC^0 to NC^1-complete at $n = 5$ (Barrington, 1989), making S_5 particularly relevant. Crucially, an RNN's expressivity hinges on the structure of its transition matrix $\boldsymbol{G}$: as shown in Fig. 1, this determines the richness of its hidden-state dynamics, and ultimately the breadth of functions the RNN can model (Huang et al., 2025). As illustrated in Fig. 1 and Tab. 1, current architectures are confined to restrictive structures. Scalar and diagonal gates (Dao & Gu, 2024; Yang et al., 2024a) limit dynamics to exponential decays and oscillations, which suffice for parity (Grazzi et al., 2025) but fail on S_5. Orthogonal (Lahoti et al., 2026) and DPLR (Kimi Team et al., 2025) gates introduce rotations and reflections. Combining m DPLR matrices progressively enriches the transition dynamics, eventually solving S_5 (Siems et al., 2025) for $m \geq 4$, but at a parameter and application cost growing in m. Ultimately, unlocking the full spectrum of trajectories requires unstructured blocks (Dubinin et al.,

Table 1: **Design space of state transitions** BGLT balances expressivity, efficient parallel training, and rich state dynamics while controlling parameter count. Here, $\bigoplus$ denotes the direct (block-diagonal) sum, d denotes model width, m block size. Gate structures notation follows Fig. 1. Under **Complexity Class**, in brackets are the conditions to solve parity (for TC^0) and S_5 (for NC^1).

Method	Gate G Structure	# Parameters	Parallelism	Complexity Class	Supported State Dynamics
Mamba-2 (Dao & Gu, 2024)	$\alpha \boldsymbol{I}$	1	Chunk-wise	$\mathsf{TC}^0_{(\alpha \in [-1,1])}$	Uniform decay
GLA (Yang et al., 2024a)	$\boldsymbol{D}$	d	Chunk-wise	$\mathsf{TC}^0_{(\boldsymbol{D} \in [-1,1])}$	Per-channel decay
KDA (Kimi Team et al., 2025)	$\boldsymbol{H} \cdot \boldsymbol{D}$	d	Chunk-wise	TC^0	Per-channel decay + (oblique) reflection
Mamba-3 (Lahoti et al., 2026)	$\bigoplus_b \alpha_b \boldsymbol{Q}^{(b)}_{(2)}$	d	Chunk-wise	TC^0	Per-block decay + 2-D rotations
(Hypothetical)	$\bigoplus_b \alpha_b Q^{(b)}_{(m)}$	$\frac{(m-1)d}{2} + \frac{d}{m}$	Chunk-wise	$\mathsf{NC}^1_{(m \geq 4)}$	Per-block decay + m-D rotations
DeltaProd (Siems et al., 2025)	$\boldsymbol{D} \cdot \prod_{j=1}^{m} \boldsymbol{H}_j$	md	$m\times$ Chunk-wise	$\mathsf{NC}^1_{(m \geq 4)}$	Per-channel decay + m (oblique) reflections
BD-RNN (Dubinin et al., 2026)	$\bigoplus_b \boldsymbol{G}^{(b)}_{(m)}$	md	Parallel Scan	$\mathsf{NC}^1_{(m \geq 4)}$	Rich block dynamics
BGLT (ours)	$\bigoplus_b \left(\boldsymbol{G}^{(b,A)}_{(\sqrt{m})} \otimes \boldsymbol{G}^{(b,B)}_{(\sqrt{m})} \right)$	$2d$	**Chunk-wise**	$\mathsf{NC}^1_{(\sqrt{m} \geq 4)}$	**Rich block dynamics**

2026), which can freely combine mixing and decay with more complex dynamics driven by non-normality, like shearing and transients (Kerg et al., 2019). Yet, as shown next, general block-diagonal matrices cannot be efficiently trained with current algorithms.

Parallel Training To train efficiently at large scale, an architecture must account for the properties of modern accelerators, which reward parallelism and tensor operations. While Linear RNNs admit sequence-parallelism via parallel scan (Blelloch, 1989; Gu & Dao, 2023; Smith et al., 2023), this operation maps poorly to current hardware, due to its low arithmetic intensity and large memory requirements, especially for matrix-valued hidden states. A better balance is struck by applying the RNN in a *chunk-wise parallel* fashion (Hua et al., 2022; Sun et al., 2023), whereby the hidden state is updated sequentially *across* chunks, while quantities *within* a chunk are computed in parallel, leveraging matrix products on Tensor Cores. To illustrate this procedure, we split the sequence into non-overlapping chunks of size C, starting at $n_S = (n-1)C+1$ and ending at $n_E = nC$. Unrolling the recurrence in (1) across a given chunk n yields (see App. F.1 for the full derivation)

$$\begin{aligned} &\text{UPDATE: } \boldsymbol{S}_{[n]} = \boldsymbol{G}_{n_S \to n_E} \boldsymbol{S}_{[n-1]} + \overrightarrow{\boldsymbol{K}}_{[n]}^\top \boldsymbol{V}_{[n]} \\ &\text{OUTPUT: } \boldsymbol{O}_{[n]} = \overleftarrow{\boldsymbol{Q}}_{[n]} \boldsymbol{S}_{[n-1]} + \left(\boldsymbol{A}_{[n]}^{qk} \odot \boldsymbol{M}\right) \boldsymbol{V}_{[n]} \end{aligned} \quad \text{with} \quad \begin{aligned} \boldsymbol{G}_{i \to j} &:= \textstyle\prod_{k=i}^{j} \boldsymbol{G}_k = \boldsymbol{G}_j \cdots \boldsymbol{G}_i \\ \left[\boldsymbol{A}_{[n]}^{qk}\right]_{t,s} &:= \boldsymbol{q}_t^\top \boldsymbol{G}_{s+1 \to t} \boldsymbol{k}_s, \end{aligned} \tag{2}$$

where $[\boldsymbol{M}]_{t,s} := \delta_{t \geq s}$ is the causal mask, and $\boldsymbol{G}_{i \to j} = \boldsymbol{I}$ whenever $i > j$. Note that (2) cleanly reduces (1) to a number of operations between tensors: $\boldsymbol{S}_{[n]} := \boldsymbol{S}_{n_E} \in \mathbb{R}^{d_k \times d_v}$ is the hidden state at the *end* of chunk n; $\boldsymbol{V}_{[n]}, \boldsymbol{O}_{[n]} \in \mathbb{R}^{C \times d_v}$ stack row-wise the values and outputs of this chunk; finally, $\overrightarrow{\boldsymbol{K}}_{[n]}, \overleftarrow{\boldsymbol{Q}}_{[n]} \in \mathbb{R}^{C \times d_k}$ stack keys and queries, *pushed* respectively to the *end* and the *beginning* of the chunk by applying the appropriate sequence of gates, i.e.,

$$[\overrightarrow{\boldsymbol{K}}_{[n]}]_s^\top := \overrightarrow{\boldsymbol{k}}_s := \boldsymbol{G}_{s+1 \to n_E} \boldsymbol{k}_s, \qquad \text{and} \qquad [\overleftarrow{\boldsymbol{Q}}_{[n]}]_t^\top := \overleftarrow{\boldsymbol{q}}_t := \boldsymbol{G}_{n_S \to t}^\top \boldsymbol{q}_t. \tag{3}$$

The performance of this formulation rests entirely on whether we can efficiently assemble the tensors in (2), which in turn restricts the gate structure. Assembling the pushed queries and keys $\overleftarrow{\boldsymbol{Q}}_{[n]}, \overrightarrow{\boldsymbol{K}}_{[n]}$, requires that gate cumulative products can be computed fast. For diagonal gates, this reduces to a trivial chunk-wise associative scan; for DPLR, this is done implicitly via their UT representation (Bischof & Van Loan, 1987); for block-diagonal, associative scan is viable, but the block size must be constrained for it to be efficient. Still, the true computational bottleneck lies in assembling the chunk-local attention matrix $\boldsymbol{A}_{[n]}^{qk} \in \mathbb{R}^{C \times C}$. Unlike for vanilla attention, this cannot immediately be computed as a simple matrix product of queries and keys, as the presence of the intermediate gates $\boldsymbol{G}_{s+1 \to t}$ couples query and key indices. To decouple them, standard implementations rely on a *factorization* of the gates. DPLR structures achieve this implicitly, again via the UT representation. For (non-zero) scalar, diagonal, and orthogonal gates, factorization is achievable by forcibly extracting an inverse:

$$[\boldsymbol{A}_{[n]}^{qk}]_{t,s} = \boldsymbol{q}_t^\top \boldsymbol{G}_{s+1 \to t} \boldsymbol{k}_s = (\boldsymbol{G}_{n_S \to t}^\top \boldsymbol{q}_t)^\top (\boldsymbol{G}_{n_S \to s}^{-1} \boldsymbol{k}_s) \quad \Longrightarrow \quad \boldsymbol{A}_{[n]}^{qk} = \overleftarrow{\boldsymbol{Q}}_{[n]} (\boldsymbol{K}_{[n]}^{\text{inv}})^\top, \tag{4}$$

with $[\boldsymbol{K}_{[n]}^{\text{inv}}]_s^\top = \boldsymbol{G}_{n_S \to s}^{-1} \boldsymbol{k}_s$. This approach, however, does not generalize to unstructured blocks: in this case, inverses are not only computationally expensive and numerically unstable (cf. Lem. 3), but are not even guaranteed to exist. To employ more expressive block-diagonal transitions at scale, then, we must overcome this barrier to parallelizability. We do so in the following, by developing a generalized chunk-wise algorithm that bypasses the need for inversion entirely.

3 Efficient Training with Block-diagonal Gates

As established in Sec. 2, extending the chunk-wise parallel form to support general block-diagonal transitions hinges on our ability to maximize the use of tensor operations when assembling the chunk-local attention matrix $\boldsymbol{A}_{[n]}^{qk}$, without resorting to gate inversion. In Sec. 3.1, we present the algorithmic foundation for a *generalized chunk-wise parallel formulation* which unlocks this capability. The procedure, illustrated in Fig. 2, works by hierarchically tiling $\boldsymbol{A}_{[n]}^{qk}$: we assemble off-diagonal tiles using a locally-factorized form that fully exploits Tensor Cores, and only fall back to a non-factorized, sequential assembly for tiles along the main diagonal. This algorithm unlocks large-scale trainability for BGLT, our Linear Transformer with Block-diagonal gates, which we describe in Sec. 3.2.

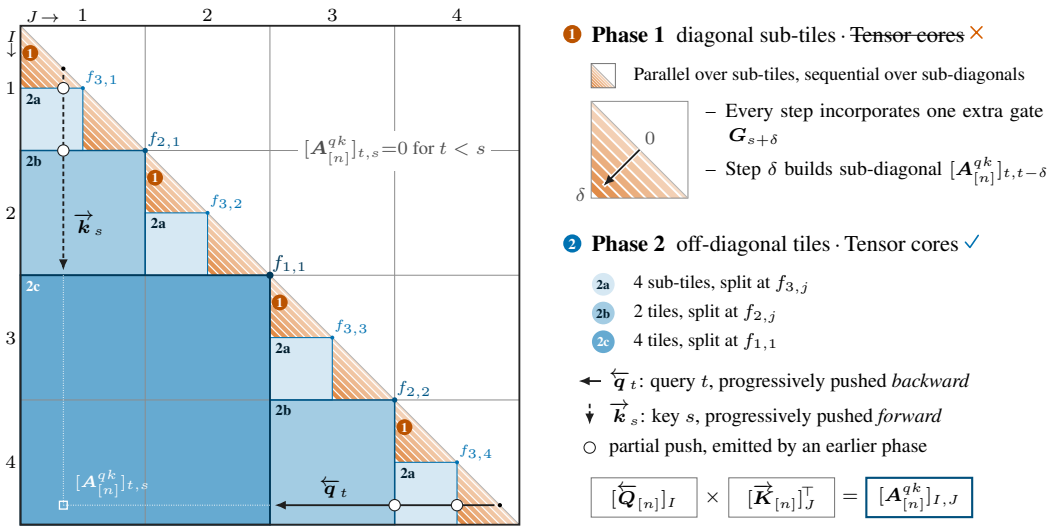


Figure 2: **Attention matrix assembly** Inside one chunk of size $C = 64$, $\boldsymbol{A}^{qk}_{[n]}$ gets partitioned into 16×16 tiles. Keys get progressively pushed *down* along the columns multiplying them by subsequent gates; queries get pushed *left* along the rows multiplying them by previous gates. Whenever they meet at a common frame $f_{i,j}$, we can assemble whole tiles of $\boldsymbol{A}^{qk}_{[n]}$ via matrix multiplications.

3.1 Generalized Chunk-wise Parallel Application

To illustrate the mechanics of our algorithm, consider the task of assembling the entries $[\boldsymbol{A}^{qk}_{[n]}]_{t,s} = \boldsymbol{q}_t^\top \boldsymbol{G}_{s+1\to t} \boldsymbol{k}_s$ for a given chunk of size C. A naïve implementation would compute all intermediate quantities $\boldsymbol{G}_{s+1\to t}$ by fixing the starting gate $\boldsymbol{G}_{s+1}$ and progressively incorporating subsequent gates $\boldsymbol{G}_{s+2}, \ldots, \boldsymbol{G}_t$ until all $t > s$ are covered. Doing so in parallel across s effectively assembles the matrix one sub-diagonal at a time. While this procedure does avoid matrix inversion, it still requires C sequential steps, and entirely forgoes the computational efficiency of Tensor Cores. Fortunately, the global *semi-separable* structure of the attention matrix (Dao & Gu, 2024; Chandrasekaran et al., 2005) still allows us to recover tensor parallelism. By definition, every off-diagonal sub-block of a semi-separable matrix admits a low-rank factorization. Hence, if we can identify and build said factorization, we can still assemble $\boldsymbol{A}^{qk}_{[n]}$ efficiently via matrix multiplication.

Local Attention Factorization To construct this factorization, notice we can separate t and s indices by splitting the gate product about a common *reference frame* f sitting strictly between them:

$$[\boldsymbol{A}^{qk}_{[n]}]_{t,s} = \boldsymbol{q}_t^\top \boldsymbol{G}_{s+1\to t} \boldsymbol{k}_s = \left(\boldsymbol{G}^\top_{f\to t} \boldsymbol{q}_t\right)^\top \left(\boldsymbol{G}_{s+1\to f-1} \boldsymbol{k}_s\right), \qquad \forall s < f \le t. \tag{5}$$

This identity yields precisely the low-rank decomposition we need: the left factor consists of queries pushed *backward* to f, and depends only on $t \ge f$; the right one contains keys pushed *forward* to f, and depends only on $s < f$. In summary, even though general block-diagonal transitions do not allow a *global* factorization of $\boldsymbol{A}^{qk}_{[n]}$, we can still factorize *locally*, about f, and assemble an entire $(t, s) \in [f, C] \times [1, f)$ sub-block via a single matrix multiplication.

Hierarchical Tiling To optimally cover the full lower-triangular attention matrix, we apply this frame-splitting strategy in a hierarchical fashion, effectively mirroring the nodes $f_{i,j} = \frac{2j-1}{2^i} C$ of a parallel associative scan tree. Setting $f_{1,1} = \frac{C}{2}$, we concurrently assemble the largest possible tile (the entire bottom-left quadrant); subsequently setting $f_{2,1} = \frac{C}{4}$ and $f_{2,2} = \frac{3C}{4}$ covers the next largest sub-tiles, and so on. The size of the remaining unfilled tiles shrinks as this hierarchical partitioning approaches the main diagonal, to the point that dispatching them to Tensor Cores introduces more overhead than benefit. At this point, we gracefully fall back to naïve sequential assembly. The resulting algorithm, outlined in Fig. 2, executes this logic bottom-up: it begins by assembling the smallest diagonal tiles sequentially, one sub-diagonal at a time (Phase 1). It then progressively pushes queries and keys to common reference frames $f_{i,j}$, traversing the tree depth i backwards, and

assembling ever-larger off-diagonal tiles via dense matrix multiplications (Phase 2). We point out that a similar sub-tiling strategy is also employed by GLA and KDA (Yang et al., 2024a; Kimi Team et al., 2025) with the goal of mitigating loss of numerical precision; in our formulation, instead, this is a structural necessity dictated by the need to avoid inverses altogether. Comprehensive details on this algorithm implementation are provided in App. G.

Limitations Even though our algorithm unlocks Tensor Cores for assembling $\boldsymbol{A}^{qk}_{[n]}$ with generic block-diagonal gates, for it to be practical the blocks must remain cheap to store and multiply. Specifically, their products must fit within GPU thread registers (NVIDIA Corporation, 2022), as failing to do so would result in excessive SRAM memory traffic, severely impacting runtime. This constraint is not unique to our formulation: it mirrors similar limitations required by standard parallel scans to run efficiently. Ultimately, though, this forces us to introduce additional structure into the block definition, motivating the Kronecker parameterization employed in BGLT, as discussed next.

3.2 BGLT: Block-diagonal Gated Linear Transformer

BGLT extends current Linear Transformers to general block-diagonal gates, thus unlocking richer state dynamics and expressivity while still retaining large-scale trainability. Its recurrent form reads

$$\boldsymbol{S}_t = \alpha_t\big(\bigoplus_{b=1}^{n_B} \boldsymbol{G}_t^{(b)}\big)\boldsymbol{S}_{t-1} + \beta_t \boldsymbol{k}_t \boldsymbol{v}_t^\top, \qquad \text{with} \quad \boldsymbol{G}_t^{(b)} \coloneqq \boldsymbol{G}_t^{(b,A)} \otimes \boldsymbol{G}_t^{(b,B)} \in \mathbb{R}^{m\times m} \quad \forall b, t, \tag{6}$$

where α_t, β_t are data-dependent scalar decay and input gate, respectively, and $\bigoplus$ denotes the *direct sum*, so that $\boldsymbol{G}_t$ results in a block-diagonal matrix with $\boldsymbol{G}_t^{(b)}$ as its blocks. As mentioned previously, further structuring the block definition is a practical necessity: deploying unstructured $m \times m$ blocks would incur a parameter cost that grows *quadratically* in m, forcing smaller blocks that limit mixing. The choice of employing Kronecker factors not only recovers linear parameter scaling while retaining broad state interactions, but also exhibits remarkable synergies across multiple aspects of the layer performance, spanning kernel efficiency, stability, and expressivity, as outlined next.

Computational efficiency The properties of Kronecker products (cf. App. B) are ideal for our chunk-wise algorithm. Transposition and matrix multiplication distribute across the individual Kronecker factors, allowing us to manipulate a full $m \times m$ block without ever materializing it, by simply working on two smaller $\sqrt{m} \times \sqrt{m}$ matrices. Building on this, our choice of 4×4 factors, spanning an $m = 16$ block, is directly dictated by hardware. As described in Sec. 3.1, assembling the off-diagonal Attention tiles requires multiplying pushed queries and keys. By setting $m = 16$, these operations translate into 16×16 dense matrix multiplications that perfectly fit the shapes expected by Tensor Cores. At the same time, register overhead remains limited to two small 4×4 Kronecker factors. We refer to App. G for additional kernel-related considerations.

Stability and gate parameterization To ensure stable training, we must control the norm growth of both hidden states (in the forward pass) and gradients (in the backward pass). This translates into constraining the gate norm $\|\boldsymbol{G}_t\|$. The Kronecker structure simplifies this, as bounding each gate reduces to bounding its individual factors. To enforce this without sacrificing expressivity, we parameterize each 4×4 factor directly via its Signed Singular Value Decomposition $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are strictly orthogonal matrices and $\boldsymbol{\Sigma}$ is diagonal, containing the signed singular values. This decomposition requires the same 16 parameters as a dense 4×4 matrix, so it incurs no parameter overhead. Most importantly, SVD grants direct access to the singular values: by constraining $\|\boldsymbol{\Sigma}\| \leq 1$, we inherently bound $\|\boldsymbol{G}_t\|_2 \leq 1$, guaranteeing stable products across arbitrary sequence lengths (see App. C.2).

Expressivity and state-tracking Despite the compact Kronecker structure, our gate parameterization can still solve the S_5 word problem (Merrill et al., 2024). To show this, let us consider a single 4×4 Kronecker factor, fixing the other to identity. Solving S_5 requires tracking permutations of 5 elements, representable as rigid rotations and reflections within a 4-dimensional space (Grazzi et al., 2025). Setting $\boldsymbol{V} = \boldsymbol{I}$ in our signed SVD parameterization for $\boldsymbol{G}^{(b,A)} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$, we have that $\boldsymbol{U}$ by itself tracks rigid rotations, while negating singular values in $\boldsymbol{\Sigma}$ allow us to express reflections. Consequently, a single BGLT layer can track the general orthogonal group $\mathrm{O}(4)$ and solve S_5. We verify this empirically in Fig. 6.

3.2.1 ΔBGLT: INTEGRATING BGLT WITH THE DELTA RULE CORRECTION

While block-diagonal transitions unlock a richer range of hidden-state dynamics, the state components can still only mix *locally*, within their m-dimensional blocks. To introduce a degree of *global* mixing, we integrate our recurrence with the Delta Rule (Yang et al., 2024b; Widrow & Hoff, 1988; Schlag et al., 2021). This applies an additional data-dependent, pseudo-Householder reflection $\boldsymbol{H}_t$ to the state, creating a synergy between gate factors akin to the mechanics of Talking Heads Attention (Shazeer et al., 2020). We denote the resulting model ΔBGLT: its combined state update reads

$$\boldsymbol{S}_t = \alpha_t \left(\boldsymbol{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top\right) \boldsymbol{G}_t \boldsymbol{S}_{t-1} + \beta_t \boldsymbol{k}_t \boldsymbol{v}_t^\top = \alpha_t \boldsymbol{H}_t \boldsymbol{G}_t \boldsymbol{S}_{t-1} + \beta_t \boldsymbol{k}_t \boldsymbol{v}_t^\top, \tag{7}$$

whose overall transition takes a *block*-diagonal-plus-low-rank form. Stability is still preserved, provided $\beta_t \in [0, 2]$, as this bounds $\|\boldsymbol{H}_t\|_2 \leq 1$. While this theoretically enables a single ΔBGLT layer to track even S_6 (Grazzi et al., 2025), in practice we set $\beta \in [0, 1]$ for our experiments.

Gram matrix assembly and unit lower-triangular solve The application of the Delta Rule additionally requires assembling the chunk-local key-Gram matrix (cf. App. F.1 for derivations)

$$[\boldsymbol{A}^{kk}_{[n]}]_{t,s} := \boldsymbol{k}_t^\top \boldsymbol{G}_{s+1\to t} \boldsymbol{k}_s, \qquad \boldsymbol{A}^{kk}_{[n]} \in \mathbb{R}^{C\times C}. \tag{8}$$

Comparing this with (2), note that the structure is equivalent to that of the Attention matrix, except $\boldsymbol{k}$ simultaneously serves as both key and query: its assembly follows then the same procedure illustrated in Sec. 3.1. As an extra step, however, implementing the Delta Rule requires *inverting* a unit lower-triangular matrix built from $\boldsymbol{A}^{kk}_{[n]}$. This inverse always exists, and can be efficiently recovered by tile-wise forward substitution (Yang et al., 2024b). To maximize Tensor Core utilization, Chen et al. (2026) invert the diagonal tiles via a Neumann series with repeated squaring. However, this gives rise to a numerically unstable algorithm: the intermediate matrix powers in the Neumann series can grow large in norm, hitting round-off errors that corrupt the inverse. To address this and control transient growth, we propose a bi-diagonally preconditioned Newton-Schulz scheme, recovering numerical stability at no additional cost. We detail the complete algorithm in App. G.2.

4 EXPERIMENTS

We validate BGLT across three axes: language modeling ability, benchmarking at 7B against KDA; efficiency, profiling throughput; expressivity, measuring performance on synthetic tasks. To isolate the performance of BGLT, we standardize the surrounding architecture across evaluated models. We implement it as stacks of pre-norm (Xiong et al., 2020) Transformer blocks with skip connections and SwiGLU MLPs (Shazeer, 2020), omitting biases on linear projections (Chowdhery et al., 2022). We ℓ_2-normalize $\boldsymbol{k}_t$ and $\boldsymbol{q}_t$, and apply RMSNorm followed by an input-dependent sigmoid gate (parameterized via a bottleneck MLP) to the attention output (Qiu et al., 2025; Kimi Team et al., 2025), prior the down-projection. See App. C for the detailed layer definition. As optimizer, we use Kimi-Muon (Jordan et al., 2024; Liu et al., 2025) for attention and MLP projections, and AdamW (Loshchilov & Hutter, 2019) for the rest, including low-rank gate projections (Tab. E.2).

4.1 LANGUAGE MODELING

We train BGLT, ΔBGLT, and the KDA (Kimi Team et al., 2025) and Transformer++ baselines at scales from 400M to 7B total parameters on $2-5\times$ chinchilla (Hoffmann et al., 2022) tokens from a filtered version of the high-quality split of Nemotron-CC (Su et al., 2025). We use the smallest scale at $5\times$ chinchilla to conduct hyperparameter sweeps for all methods, which we then transfer to larger scales; we detail the scaling rules and hyperparameters in App. E.2 and App. E.1. During our hyperparameter exploration at 400M we found that moving part of the gate low-rank projections from Muon to AdamW lowers the loss and widens the range of stable learning rates, in line with Qiu et al. (2026); at scale, we therefore optimize all low-rank projections with AdamW.

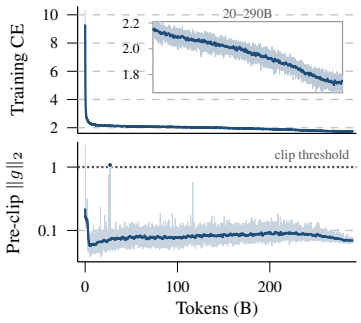


Figure 3: **Training stability of ΔBGLT at 3B**

Training throughput BGLT is able to achieve competitive training and prefill throughput compared to linear Transformer baselines such as GLA and KDA. In Fig. 4 we show that the prefill throughput at 7B at $128k$ tokens per step of BGLT matches KDA and is about $5-9\%$ slower than GLA. The Delta Rule correction introduced in ΔBGLT reduces throughput by $9-11\%$: the gap is comparable to that between KDA and GLA, and corresponds to the cost of including a Householder factor in the recursion. Training throughput of BGLT is lower relative to the baselines since the backward takes relatively longer: we report full timing results in Fig. G.3. Since our block-diagonal gates hold more parameters than diagonal baselines, the activation memory required is $\sim 9-13\%$ higher than KDA and GLA, and BGLT relies on recomputation to reduce memory consumption. In App. G.4 we provide additional results, and a full per-kernel break-down of runtimes for our models.

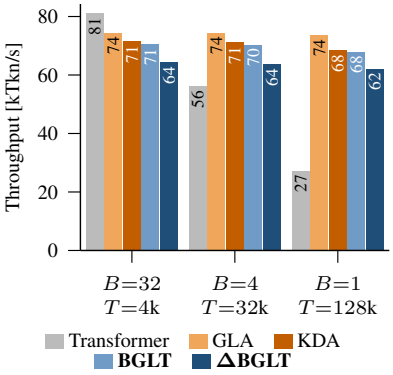


Figure 4: **7B scale prefill throughput** at 128k tokens per step.

Results We report general language modeling results for our 1.5B to 7B models in Tab. 2. ΔBGLT trains stably at scale, without loss spikes over the full 3B run (Fig. 3). We find that the performance of ΔBGLT improves as we increase model size, suggesting that the benefit from more expressive state transitions manifests more prominently with longer training horizons and more parameters. On general language modeling, ΔBGLT performs on average slightly better than KDA, the gap widening with model scale. On recall datasets we find that the Transformer++ reference outperforms both linear models, in line with expectations. We also find that the ranking between KDA and ΔBGLT flips as we scale the models from 1.5B to 3B. We remark, however, that the gap remains narrow and does not allow us to draw conclusions with regard to their relative performance on recall. We also test models on multi-hop in-context retrieval (Fig. 5) as introduced in Allen-Zhu (2025) and find that in the single-hop case ΔBGLT outperforms KDA as context grows.

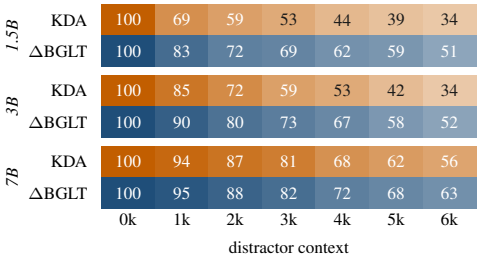


Figure 5: **Single-hop entity retrieval against distractor length** (Allen-Zhu, 2025).

4.2 Synthetic Tasks

We test BGLT on a range of synthetic tasks including recall, state tracking, and tasks from the Chomsky hierarchy (Beck et al., 2024), further described in App. D. As baselines, we consider both linear and non-linear RNNs, as well as Transformer.

Table 2: **General language modeling and recall results** All models trained at sequence length 4096 with per-architecture optimal hyperparameters. Recall-intensive tasks are evaluated in the cloze/extraction setting. All metrics are accuracies in %. Averages are unweighted over the tasks in each group. Bold marks the best overall entry per scale, underline the best non-Transformer entry. Transformer++ (gray) is a softmax-attention reference trained with its own recipe (App. E.3). The factor next to each scale is its training token budget as a multiple of Chinchilla-optimal of 20 tokens per parameter (Hoffmann et al., 2022).

	Model	Perplexity		General language modeling							Recall				
		LMB. ppl ↓	Wiki. ppl ↓	LMB. acc ↑	PIQA acc ↑	Hella. acc_n ↑	Wino. acc ↑	ARC-e acc ↑	ARC-c acc_n ↑	**Avg.** ↑	SWDE ↑	FDA ↑	SQuAD ↑	TQA ↑	**Avg.** ↑
1.5B 5× chin.	Transformer++	7.21	15.23	58.2	76.0	64.7	61.5	72.2	41.6	62.4	**56.1**	**73.8**	**49.1**	65.8	**61.2**
	KDA	**6.33**	14.89	**60.2**	76.2	66.4	62.1	**75.7**	42.5	63.9	48.6	58.4	47.5	65.3	55.0
	ΔBGLT	6.38	**14.72**	59.9	**76.3**	**66.6**	**64.8**	74.6	**42.7**	**64.2**	46.6	53.0	45.8	**66.1**	52.9
3B 5× chin.	Transformer++	5.13	12.95	64.4	77.6	71.3	64.6	76.3	45.6	66.6	**64.4**	**72.6**	**52.8**	**70.3**	**65.0**
	KDA	4.80	**12.56**	65.9	**78.5**	72.8	67.7	**77.4**	45.1	67.9	50.4	57.9	48.9	68.4	56.4
	ΔBGLT	**4.76**	12.66	**66.5**	78.2	**72.9**	**69.3**	77.1	**48.8**	**68.8**	54.5	54.6	50.3	69.1	57.1
7B 2× chin.	Transformer++	4.35	11.71	68.1	78.5	74.1	69.0	78.5	49.0	69.5	**63.2**	**74.5**	**54.4**	**71.0**	**65.8**
	KDA	4.06	11.27	69.4	**79.5**	**76.3**	**72.0**	81.1	52.9	71.9	55.5	63.3	50.4	70.2	59.9
	ΔBGLT	**4.00**	**11.03**	**69.9**	78.6	**76.3**	71.3	**81.9**	**53.8**	**72.0**	58.2	66.1	52.2	71.0	61.9

State tracking In Sec. 3.2 we prove that BGLT can learn S_5 provided $\mathbf{\Sigma}$ is extended to the $[-1, 1]$ range (Grazzi et al., 2025), and is otherwise limited to A_5. We empirically show this in Fig. 6, where we compare it with KDA and DeltaProduct$_4$ (Siems et al., 2025). We find that, as expected, KDA cannot fit the training sequence length, while DeltaProduct$_4$ is able to extrapolate beyond it. We also find that BGLT is able to extrapolate perfectly to length 512, while DeltaProduct performance decays soon beyond the training T. We hypothesize this is due to DeltaProduct$_4$ having to maintain an exact $\beta = 2$ for $4\times$ as many steps compared to BGLT, which instead only needs to set one $\mathbf{\Sigma} \in \{\pm 1\}^4$ per token, thus decaying slower.

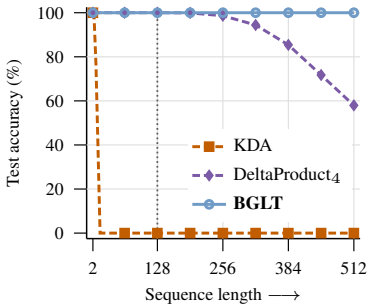


Figure 6: **Length generalization on** S_5 Dotted vertical line shows training T.

Synthetic task suite Beyond S_5, we train single-layer non-linear RNN, attention, and linear-attention models on nine synthetic tasks covering recall (MQAR, k-hops, copying), counting and arithmetic, group state tracking and Neural Cellular Automata (NCA) (Lee et al., 2026), with three seeds per model and a learning rate sweep per task (setup in App. D). Table 3 reports the four tasks that best separate the models; Tab. D.1 reports all nine. These results indicate that the additional expressivity granted by BGLT effectively translates into accuracy gains for various tasks. In particular for NCA, BGLT and ΔBGLT consistently outperform other linear transformers (details in App. D.5) confirming our method can better track complex state transitions.

Table 3: **Synthetic tasks** Test accuracy (%) of single-layer models. Bold: best performance (note $\geq 99\%$, early-stopping threshold). Underlined: best linear transformer.

Task	LSTM	Transf.	KDA	**BGLT**	**ΔBGLT**
A_5 state tracking	**100.0**	6.3	6.2	**100.0**	**100.0**
Mod. arithmetic w/o()	97.7	43.0	68.3	**99.1**	96.6
Copy memory	11.5	**100.0**	62.0	**99.1**	**99.3**
k-hops, $k = 4$	54.7	48.6	**99.5**	49.1	**99.2**
NCA	**87.1**	78.6	71.3	77.0	78.9

5 Conclusion

In this work, we introduce BGLT, the first Linear Transformer architecture equipped with input-dependent, block-diagonal gates that remains efficiently trainable at scale. Our primary motivation is to overcome the expressivity limitations of scalar and diagonal transitions, unlocking the richer hidden-state dynamics necessary for complex sequence modeling. The key enabler for BGLT is our *generalized chunk-wise parallel* algorithm. By allowing the efficient assembly of chunk-local attention matrices via Tensor Cores irrespectively of the gate structure, the algorithm expands the space of feasible transitions, easing the tension between expressivity and large-scale trainability. We demonstrate that BGLT, alongside its Delta Rule-augmented variant ΔBGLT, translates these theoretical advantages into concrete empirical gains. On synthetic benchmarks, BGLT performs remarkably well, and a single layer perfectly solves the S_5 state-tracking task. Scaled to 7B parameters, ΔBGLT trains stably and achieves language modeling performance competitive with state-of-the-art baselines like KDA, all the while maintaining parameter parity and tractable training throughput.

Limitations and Future Work While our $(4 \times 4) \otimes (4 \times 4)$ block-Kronecker formulation strikes an ideal balance between expressivity, parameter count, and efficiency, the absolute size and structure of deployable gate blocks remains fundamentally bottlenecked by the strict register and SRAM budgets of modern GPUs. In spite of this, our Kronecker parameterization is only one of many admissible choices: cheaper sub-structures for the gate factors could further reduce the memory footprint, and a more detailed analysis on their spectrum's impact on performance might inform better initialization schemes than the current SVD-based one. Most importantly, much remains to be learned about the way these architectures scale, and in experimenting with ΔBGLT, we found that not all lessons from Transformers transfer directly: we see identifying tailored scaling rules as the key next step.

More broadly, by loosening the algorithmic restrictions on gate structures, BGLT provides a blueprint for discovering, training, and deploying even more general and expressive recurrences at scale, further boosting the capabilities of Linear Transformers.

AI USE STATEMENT

In this work, we used generative AI tools, primarily Claude Sonnet 5 and Opus 5 through Claude Code, for aid in implementation and experiments setup, and assistance in writing the manuscript. In particular, AI assistants helped implementing the training and evaluation tooling, and accelerated the iteration through design choices for the Triton and CuTe kernels. They also helped writing the code to compute summary statistics from logged training curves and hyperparameter sweeps. Finally, they reviewed claims against the underlying data and flagged eventual inconsistencies. AI was also used for drafting and editing portions of the paper, adjusting figures, and checking references and cross-references. AI was not used to produce synthetic data, develop the theoretical framework, state or prove mathematical results, or propose the research hypotheses.

We have reviewed all AI-assisted work. All kernels are validated against naïve reference implementations in both passes. AI-drafted text was reviewed and edited by the authors. We take responsibility for the final content of this work.

ETHICS STATEMENT

This work is foundational, aimed at improving the capabilities of linear RNNs. All experiments use publicly available datasets and involve no human subjects or sensitive data. We identify no ethical concerns specific to our contributions beyond those inherent to general-purpose sequence modeling. The authors adhere to the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our results, the models presented in this work have been trained using publicly available datasets and standard open-source libraries. Our kernels are implemented on top of FLA (Yang & Zhang, 2024). The complete source code to reproduce the BGLT architecture and the custom hardware-efficient kernels described in this work is currently undergoing final preparation and will soon be made publicly available.

This manuscript has been structured to provide all information necessary to independently implement our algorithms and reproduce our experiments. To this end, App. F contains comprehensive mathematical derivations detailing BGLT operations, and App. G provides pseudo-code for the novel kernels required for its efficient application. In App. D we illustrate in exhaustive detail the setup used to train our models on various synthetic tasks. Finally, in App. E.1 we provide the hyperparameter and optimizers configurations used to train our models at scale on language modeling tasks.

REFERENCES

Kwangjun Ahn, Byron Xu, Natalie Abreu, Ying Fan, Gagik Magakyan, Pratyusha Sharma, Zheng Zhan, and John Langford. Dion: Distributed orthonormalized updates, 2025. URL https://arxiv.org/abs/2504.05295.

Zeyuan Allen-Zhu. Physics of Language Models: Part 4.1, Architecture Design and the Magic of Canon Layers. In Proceedings of the 39th Conference on Neural Information Processing Systems, NeurIPS '25, 2025. Full version available at https://ssrn.com/abstract=5240330.

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks, 2016. URL https://arxiv.org/abs/1511.06464.

S. Arora, S. Eyuboglu, A. Timalsina, I. Johnson, M. Poli, J. Zou, A. Rudra, and C. Ré. Zoology: Measuring and Improving Recall in Efficient Language Models. In The Twelfth International Conference on Learning Representations (ICLR'24). ICLR, 2024a.

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes, 2023.

Simran Arora, Aman Timalsina, Aaryan Singhal, Benjamin Spector, Sabri Eyuboglu, Xinyi Zhao, Ashish Rao, Atri Rudra, and Christopher Ré. Just read twice: closing the recall gap for recurrent language models, 2024b. URL https://arxiv.org/abs/2407.05483.

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

David A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. Journal of Computer and System Sciences, 38(1):150–164, 1989. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(89)90037-8. URL https://www.sciencedirect.com/science/article/pii/0022000089900378.

M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. xLSTM: Extended Long Short-Term Memory. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), Proceedings of the 38th International Conference on Advances in Neural Information Processing Systems (NeurIPS'24). Curran Associates, 2024.

Christian Bischof and Charles Van Loan. The wy representation for products of householder matrices. SIAM Journal on Scientific and Statistical Computing, 8(1):s2–s13, 1987. doi: 10.1137/0908009. URL https://doi.org/10.1137/0908009.

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.

G.E. Blelloch. Scans as primitive parallel operations. IEEE Transactions on Computers, 38(11): 1526–1538, 1989. doi: 10.1109/12.42122.

Cris Cecka. Cute layout representation and algebra, 2026. URL https://arxiv.org/abs/2603.02298.

S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A. J. van der Veen, and D. White. Some fast algorithms for sequentially semiseparable representations. SIAM Journal on Matrix Analysis and Applications, 27(2):341–364, 2005. doi: 10.1137/S0895479802405884. URL https://doi.org/10.1137/S0895479802405884.

Y. Chen, Z. Li, Y. Wang, and M. Wei. Flashkda: Flash kimi delta attention. https://github.com/MoonshotAI/FlashKDA, 2026.

K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller. Rethinking attention with performers. In The Ninth International Conference on Learning Representations (ICLR'21). ICLR, 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. URL https://arxiv.org/abs/2204.02311.

N. M. Cirone, A. Orvieto, B. Walker, C. Salvi, and T. Lyons. Theoretical Foundations of Deep Selective State-Space Models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), Proceedings of the 38th International Conference on Advances in Neural Information Processing Systems (NeurIPS'24). Curran Associates, 2024.

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.

Federico Danieli, Pau Rodriguez, Miguel Sarabia, Xavier Suau, and Luca Zappella. Pararnn: Unlocking parallel training of nonlinear rnns for large language models, 2025. URL https://arxiv.org/abs/2510.21450.

T. Dao and A. Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp (eds.), Proceedings of the 41st International Conference on Machine Learning (ICML'24), volume 235 of Proceedings of Machine Learning Research. PMLR, 2024.

T. Dao, N. Sohoni, A. Gu, M. Eichhorn, A. Blonder, M. Leszczynski, A. Rudra, and C. Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In The Eighth International Conference on Learning Representations (ICLR'20). ICLR, 2020a.

Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations, 2020b. URL https://arxiv.org/abs/1903.05895.

Tri Dao, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training, 2022. URL https://arxiv.org/abs/2204.00595.

S. De, S. L. Smith, A. Fernando, A. Botev, G. Cristian-Muraru, A. Gu, R. Haroun, L. Berrada, Y. Chen, S. Srinivasan, G. Desjardins, A. Doucet, D. Budden, Y. W. Teh, R. Pascanu, N. De Freitas, and C. Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models. arXiv:2402.19427 [cs.LG], 2024.

Nolan Dey, Bin Claire Zhang, Lorenzo Noci, Mufan Li, Blake Bordelon, Shane Bergsma, Cengiz Pehlevan, Boris Hanin, and Joel Hestness. Don't be lazy: Completep enables compute-efficient deep transformers, 2026. URL https://arxiv.org/abs/2505.01618.

Igor Dubinin, Antonio Orvieto, and Felix Effenberger. Improved state mixing in higher-order and block diagonal linear recurrent networks, 2026. URL https://arxiv.org/abs/2602.12021.

Ting-Han Fan, Ta-Chung Chi, and Alexander I. Rudnicky. Advancing regular language reasoning in linear recurrent neural networks, 2024. URL https://arxiv.org/abs/2309.07412.

D. Fu, T. Dao, K. Saab, A. Thomas, A. Rudra, and C. Re. Hungry Hungry Hippos: Towards Language Modeling with State Space Models. In The Eleventh International Conference on Learning Representations (ICLR'23). ICLR, 2023.

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.

X. Gonzalez, A. Warrington, J. T.H. Smith, and S. Linderman. Towards Scalable and Stable Parallelization of Nonlinear RNNs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), Proceedings of the 38th International Conference on Advances in Neural Information Processing Systems (NeurIPS'24). Curran Associates, 2024.

R. Grazzi, J. Siems, A. Zela, J. Franke, F. Hutter, and M. Pontil. Unlocking State-Tracking in Linear RNNs Through Negative Eigenvalues. In The Thirteenth International Conference on Learning Representations (ICLR'25). ICLR, 2025.

A. Gu and T. Dao. Mamba: Linear time sequence modeling with selective state spaces. arXiv:2312.00752 [cs.LG], 2023.

A. Gu, K. Goel, and C. Re. Efficiently Modeling Long Sequences with Structured State Spaces. In The Tenth International Conference on Learning Representations (ICLR'22). ICLR, 2022a.

A. Gu, A. Gupta, K. Goel, and C. Ré. On the Parameterization and Initialization of Diagonal State Space Models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), Proceedings of the 36th International Conference on Advances in Neural Information Processing Systems (NeurIPS'22). Curran Associates, 2022b.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), Advances in Neural Information Processing Systems, volume 35, pp. 22982–22994. Curran Associates, Inc., 2022. doi: 10.52202/068431-1670. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9156b0f6dfa9bbd18c79cc459ef5d61c-Paper-Conference.pdf.

R. Hasani, M. Lechner, T. Wang, M. Chahine, A. Amini, and D. Rus. Liquid Structural State-Space Models. In The Eleventh International Conference on Learning Representations (ICLR'23). ICLR, 2023.

Ali Hatamizadeh, Yejin Choi, and Jan Kautz. Gated deltanet-2: Decoupling erase and write in linear attention, 2026. URL https://arxiv.org/abs/2605.22791.

Kyle Helfrich, Devin Willmott, and Qiang Ye. Orthogonal recurrent neural networks with scaled cayley transform, 2018. URL https://arxiv.org/abs/1707.09520.

Alex Henry, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. Query-key normalization for transformers. In Trevor Cohn, Yulan He, and Yang Liu (eds.), Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 4246–4253, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.379. URL https://aclanthology.org/2020.findings-emnlp.379/.

Nicholas J. Higham. Accuracy and Stability of Numerical Algorithms. Society for Industrial and Applied Mathematics, second edition, 2002. doi: 10.1137/1.9780898718027. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation, 9(8):1735–1780, 1997. Based on TR FKI-207-95, TUM (1995).

J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre. Training compute-optimal large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), Proceedings of the 36th International Conference on Advances in Neural Information Processing Systems (NeurIPS'22). Curran Associates, 2022.

J. Hu, Y. Pan, J. Du, D. Lan, X. Tang, Q. Wen, Y. Liang, and W. Sun. Comba: Improving Bilinear RNNs with Closed-loop Control. arXiv:2506.02475 [cs.LG], 2025.

W. Hua, Z. Dai, H. Liu, and Q. Le. Transformer quality in linear time. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato (eds.), Proceedings of the 39th International Conference on Machine Learning (ICML'22), volume 162 of Proceedings of Machine Learning Research. PMLR, 2022.

Ningyuan Huang, Miguel Sarabia, Abhinav Moudgil, Pau Rodriguez, Luca Zappella, and Federico Danieli. Understanding input selectivity in mamba: Impact on approximation power, memorization, and associative recall capacity, 2025. URL https://arxiv.org/abs/2506.11891.

Yulong Huang, Xiang Liu, Hongxiang Huang, Xiaopeng Lin, Zunchang Liu, Xiaowen Chu, Zeke Xie, and Bojun Cheng. Mdn: Parallelizing stepwise momentum for delta linear attention, 2026. URL https://arxiv.org/abs/2605.05838.

L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Soljacic. Tunable Efficient Unitary Neural Networks (EUNN) and their application to RNNs. In D. Precup and Y. Teh (eds.), Proceedings of the 34th International Conference on Machine Learning (ICML'17), volume 70. Proceedings of Machine Learning Research, 2017.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

C. Jose, M. Cisse, and F. Fleuret. Kronecker recurrent units. In J. Dy and A. Krause (eds.), Proceedings of the 35th International Conference on Machine Learning (ICML'18), volume 80. Proceedings of Machine Learning Research, 2018.

Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017. URL https://arxiv.org/abs/1705.03551.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In H. Daume III and A. Singh (eds.), Proceedings of the 37th International Conference on Machine Learning (ICML'20), volume 119. Proceedings of Machine Learning Research, 2020.

Tobias Katsch. Gateloop: Fully data-controlled linear recurrence for sequence modeling, 2024. URL https://openreview.net/forum?id=02Ug9N8DCI.

Matthijs Van Keirsbilck, Alexander Keller, and Xiaodong Yang. Rethinking full connectivity in recurrent neural networks, 2019. URL https://arxiv.org/abs/1905.12340.

Giancarlo Kerg, Kyle Goyette, Maximilian Puelma Touzel, Gauthier Gidel, Eugene Vorontsov, Yoshua Bengio, and Guillaume Lajoie. Non-normal recurrent neural network (nnrnn): learning long time dependencies while improving expressivity with transient dynamics. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/9d7099d87947faa8d07a272dd6954b80-Paper.pdf.

Kimi Team. Kimi k3: Open frontier intelligence. Technical report, July 2026. URL https://github.com/MoonshotAI/Kimi-K3/blob/main/k3_tech_report.pdf.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Aakash Lahoti, Kevin Li, Berlin Chen, Caitlin Wang, Aviv Bick, J Zico Kolter, Tri Dao, and Albert Gu. Mamba-3: Improved sequence modeling using state space principles. In The Fourteenth International Conference on Learning Representations, 2026. URL https://openreview.net/forum?id=HwCvaJOiCj.

Dan Lee, Seungwook Han, Akarsh Kumar, and Pulkit Agrawal. Training language models via neural cellular automata, 2026. URL https://arxiv.org/abs/2603.10055.

Mario Lezcano-Casado and David Martínez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group, 2019. URL https://arxiv.org/abs/1901.08428.

Y. Lim, Q. Zhu, J. Selfridge, and M. Kasim. Parallelizing non-linear sequential models over the sequence length. In The Twelfth International Conference on Learning Representations (ICLR'24). ICLR, 2024.

B. Liu, J. Ash, S. Goel, A. Krishnamurthy, and C. Zhang. Transformers Learn Shortcuts to Automata. In The Eleventh International Conference on Learning Representations (ICLR'23). ICLR, 2023.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for llm training, 2025. URL https://arxiv.org/abs/2502.16982.

Colin Lockard, Prashant Shiralkar, and Xin Luna Dong. OpenCeres: When open information extraction meets the semi-structured web. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 3047–3056, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1309. URL https://aclanthology.org/N19-1309.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In The Seventh International Conference on Learning Representations (ICLR'19). ICLR, 2019.

E. Martin and C. Cundy. Parallelizing Linear Recurrent Neural Nets Over Sequence Length. In The Sixth International Conference on Learning Representations (ICLR'18). ICLR, 2018.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

W. Merrill, J. Petty, and A. Sabharwal. The Illusion of State in State-Space Models. In R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp (eds.), Proceedings of the 41st International Conference on Machine Learning (ICML'24), volume 235 of Proceedings of Machine Learning Research. PMLR, 2024.

Z. Mhammedi, A. Hellicar, A. Rahman, and J. Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In D. Precup and Y. Teh (eds.), Proceedings of the 34th International Conference on Machine Learning (ICML'17), volume 70. Proceedings of Machine Learning Research, 2017.

Mayank Mishra, Shawn Tan, Ion Stoica, Joseph Gonzalez, and Tri Dao. M^2rnn: Non-linear rnns with matrix-valued states for scalable language modeling, 2026. URL https://arxiv.org/abs/2603.14360.

Bruno Kacper Mlodozeniec, Pierre Ablin, Louis Béthune, Dan Busbridge, Michal Klein, Jason Ramapuram, and Marco Cuturi. Completed hyperparameter transfer across modules, width, depth, batch and duration. In The Fourteenth International Conference on Learning Representations, 2026. URL https://openreview.net/forum?id=elB9k4nTL1.

Sajad Movahedi, Timur Carstensen, Arshia Afzal, Frank Hutter, Antonio Orvieto, and Volkan Cevher. Selective rotary position embedding, 2025. URL https://arxiv.org/abs/2511.17388.

NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. Whitepaper, NVIDIA Corporation, 2022. URL https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c. Accessed: 2026-07-27.

A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, and S. De. Resurrecting recurrent neural networks for long sequences. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett (eds.), Proceedings of the 40th International Conference on Machine Learning (ICML'23), volume 202 of Proceedings of Machine Learning Research. PMLR, 2023.

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.

Rom Parnichkun, Neehal Tumma, Armin W Thomas, Alessandro Moro, Qi An, Taiji Suzuki, Atsushi Yamashita, Michael Poli, and Stefano Massaroli. Quantifying memory utilization with effective state-size. In Forty-second International Conference on Machine Learning, 2025. URL https://openreview.net/forum?id=YKAnIvY5hf.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In S. Dasgupta and D. McAllester (eds.), Proceedings of the 30th International Conference on Machine Learning (ICML'13), pp. 1310–1318. Omnipress, 2013.

B. Peng, R. Zhang, D. Goldstein, E. Alcaide, X. Du, H. Hou, J. Lin, J. Liu, J. Lu, W. Merrill, G. Song, K. Tan, S. Utpala, N. Wilce, J. Wind, T. Wu, D. Wuttke, and C. Zhou-Zheng. RWKV-7 "Goose" with Expressive Dynamic State Evolution. arXiv:2503.14456 [cs.CL], 2025.

Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, Kranthi Kiran GV, Jan Kocoń, Bartłomiej Koptyra, Satyapriya Krishna, Ronald McClelland Jr., Jiaju Lin, Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Cahya Wirawan, Stanisław Woźniak, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Peng Zhou, Jian Zhu, and Rui-Jie Zhu. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence, 2024. URL https://arxiv.org/abs/2404.05892.

H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. Smith, and L. Kong. Random Feature Attention. In The Ninth International Conference on Learning Representations (ICLR'21). ICLR, 2021.

Tiziano Politi. A formula for the exponential of a real skew-symmetric matrix of order 4. BIT Numerical Mathematics, 41:842–845, 2001. URL https://api.semanticscholar.org/CorpusID:116954675.

Tomer Porian, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon. Resolving discrepancies in compute-optimal scaling of language models, 2025. URL https://arxiv.org/abs/2406.19146.

Korbinian Pöppel, Maximilian Beck, and Sepp Hochreiter. Flashrnn: I/o-aware optimization of traditional rnns on modern hardware, 2025. URL https://arxiv.org/abs/2412.07752.

Z. Qin, S. Yang, and Y. Zhong. Hierarchically Gated Recurrent Neural Network for Sequence Modeling. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), Proceedings of the 37th International Conference on Advances in Neural Information Processing Systems (NeurIPS'23). Curran Associates, 2023.

Z. Qin, S. Yang, W. Sun, X. Shen, D. Li, W. Sun, and Y. Zhong. HGRN2: Gated Linear RNNs with State Expansion. arXiv:2404.07904 [cs.CL], 2024.

S. Qiu, A. Potapczynski, M. A. Finzi, M. Goldblum, and A. G. Wilson. Compute better spent: Replacing dense layers with structured matrices. In R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp (eds.), Proceedings of the 41st International Conference on Machine Learning (ICML'24), volume 235 of Proceedings of Machine Learning Research. PMLR, 2024.

Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, Dayiheng Liu, Jingren Zhou, and Junyang Lin. Gated attention for large language models: Non-linearity, sparsity, and attention-sink-free. In The Thirty-ninth Annual Conference on Neural Information Processing Systems, 2025. URL https://openreview.net/forum?id=1b7whO4SfY.

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.

Qwen Team. Qwen3-coder-next technical report. Technical report, 2026. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018.

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.

I. Schlag, K. Irie, and J. Schmidhuber. Linear transformers are secretly fast weight programmers. In M. Meila and T. Zhang (eds.), Proceedings of the 38th International Conference on Machine Learning (ICML'21), volume 139 of Proceedings of Machine Learning Research. PMLR, 2021.

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

Noam Shazeer, Zhenzhong Lan, Youlong Cheng, Nan Ding, and Le Hou. Talking-heads attention, 2020. URL https://arxiv.org/abs/2003.02436.

J. Siems, T. Carstensen, A. Zela, F. Hutter, M. Pontil, and R. Grazzi. DeltaProduct: Increasing the expressivity of deltanet through products of householders. arXiv:2502.10297 [cs.LG], 2025.

J. Smith, A. Warrington, and S. Linderman. Simplified State Space Layers for Sequence Modeling. In The Eleventh International Conference on Learning Representations (ICLR'23). ICLR, 2023.

Aleksandros Sobczyk, Gioele Gottardo, Christos K. Matzoros, Mirko De Vita, Filip Skogh, Anastasios Zouzias, and Jiawei Zhuang. Fast and stable triangular inversion for delta-rule linear transformers, 2026. URL https://arxiv.org/abs/2605.21325.

Dan Su, Kezhi Kong, Ying Lin, Joseph Jennings, Brandon Norick, Markus Kliegl, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. Nemotron-cc: Transforming common crawl into a refined long-horizon pretraining dataset, 2025. URL https://arxiv.org/abs/2412.02595.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. Neurocomputing, 568:127063, 2024.

Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei. Retentive Network: A Successor to Transformer for Large Language Models. arXiv:2307.08621 [cs.CL], 2023.

Aleksandar Terzić, Michael Hersche, Giacomo Camposampiero, Thomas Hofmann, Abu Sebastian, and Abbas Rahimi. On the expressiveness and length generalization of selective state-space models on regular languages. In Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. ISBN 978-1-57735-897-8. doi: 10.1609/aaai.v39i19.34301. URL https://doi.org/10.1609/aaai.v39i19.34301.

Aleksandar Terzić, Nicolas Menet, Michael Hersche, Thomas Hofmann, and Abbas Rahimi. Structured sparse transition matrices to enable state tracking in state-space models, 2025. URL https://arxiv.org/abs/2509.22284.

Aleksandar Terzić, Francesco Carzaniga, Nicolas Menet, Yannick Biehl, Michael Hersche, Thomas Hofmann, and Abbas Rahimi. Flash pd-ssm: Memory-optimized structured sparse state-space models, 2026. URL https://arxiv.org/abs/2605.19150.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems (NeurIPS'17). Curran Associates, Inc., 2017.

Benjamin Walker, Lingyi Yang, Nicola Muca Cirone, Cristopher Salvi, and Terry Lyons. Structured linear cdes: Maximally expressive and parallel-in-time sequence models, 2025. URL https://arxiv.org/abs/2505.17761.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits, pp. 123–134. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

Scott Wisdom, Thomas Powers, John R. Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks, 2016. URL https://arxiv.org/abs/1611.00035.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020. URL https://arxiv.org/abs/2002.04745.

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer, 2022. URL https://arxiv.org/abs/2203.03466.

S. Yang, B. Wang, Y. Shen, R. Panda, and Y. Kim. Gated Linear Attention Transformers with Hardware-Efficient Training. In R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp (eds.), Proceedings of the 41st International Conference on Machine Learning (ICML'24), volume 235 of Proceedings of Machine Learning Research. PMLR, 2024a.

S. Yang, B. Wang, Y. Zhang, Y. Shen, and Y. Kim. Parallelizing Linear Transformers with the Delta Rule over Sequence Length. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), Proceedings of the 38th International Conference on Advances in Neural Information Processing Systems (NeurIPS'24). Curran Associates, 2024b.

S. Yang, J. Kautz, and A. Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In The Thirteenth International Conference on Learning Representations (ICLR'25). ICLR, 2025a.

S. Yang, Y. Shen, K. Wen, S. Tan, M. Mishra, L. Ren, R. Panda, and Y. Kim. PaTH Attention: Position encoding via accumulating householder transformations. arXiv:2505.16381 [cs.CL], 2025b.

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.

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.

Chenyu Zheng, Rongzhen Wang, Xinyu Zhang, and Chongxuan Li. Spectral condition for μP under width-depth scaling. arXiv preprint arXiv:2603.00541, 2026.

APPENDIX CONTENTS

A Extended Literature Review **20**

B Notation, Useful Properties, and Proofs **21**

C Layer Design **25**

C.1 Model architecture 25

C.2 State transition parameterization 26

D Measuring Model Expressivity through Synthetic Tasks **28**

D.1 Model Setup 28

D.2 Training Details 30

D.3 Synthetic Tasks Definitions 30

D.4 Results & Analysis 31

D.5 NCA: performance under different transition functions 31

E Language Model Training and Scaling **35**

E.1 Hyperparameter Selection 35

E.2 Scaling 37

E.3 Transformer++ Baseline 41

E.4 Evaluation 41

F Chunk-wise Parallel Formulation of BGLT **43**

F.1 Forward pass 43

F.2 Backward pass 46

G Kernels **49**

G.1 Forward pass: Attention and Gram assembly 49

G.2 Forward pass: improving the Unit Triangular Solver for Delta Rule 51

G.3 Backward pass: Attention and Gram assembly 54

G.4 Kernel profiling 62

A Extended Literature Review

Linear RNNs The development of linear recurrences can be characterized as navigating a parallelism–expressivity trade-off. Linear Attention (LA) (Katharopoulos et al., 2020; Peng et al., 2021; Choromanski et al., 2021) rephrases softmax attention as a matrix-valued recurrence with constant-time *inference* and quadratic training complexity. Non input-dependent state space models (SSMs) were simultaneously developed (Gu et al., 2022a; Gupta et al., 2022; Gu et al., 2022b; Fu et al., 2023) and could be parallelized either as a convolution via FFT (Gu et al., 2022a; Fu et al., 2023) or as a linear recurrence via parallel scan (Smith et al., 2023; Orvieto et al., 2023). Both LA and non-selective SSMs were limited in their recall ability either by the lack of a forgetting mechanism and state saturation (Parnichkun et al., 2025) or due to their fixed decay and vector valued hidden states (Arora et al., 2024a). Input-dependent scalar or diagonal gating (Hasani et al., 2023; Qin et al., 2023; Gu & Dao, 2023; De et al., 2024; Qin et al., 2024; Beck et al., 2024) improved recall in both, at first parallelized with scans (Martin & Cundy, 2018; Blelloch, 1989) and then with the chunk-wise parallel form (Hua et al., 2022; Sun et al., 2023), which recovers linear-time *training* (cf. Sec. 2). Mamba-2 (Dao & Gu, 2024) unified both lines by showing that gated linear attention (Katsch, 2024; Yang et al., 2024a; Peng et al., 2024) and selective SSMs are two views of the same sequentially semiseparable token-mixing matrix. Expressivity in the form of state tracking was enabled through negative eigenvalues (Grazzi et al., 2025), and DeltaNet (Yang et al., 2024b; Widrow & Hoff, 1988; Schlag et al., 2021) extended diagonal transitions to DPLR (Yang et al., 2025a; Peng et al., 2025; Hu et al., 2025; Kimi Team et al., 2025; Hatamizadeh et al., 2026; Huang et al., 2026) while remaining chunk-wise parallelizable; further expressivity has only been achieved through products of DPLR (Siems et al., 2025; Yang et al., 2025b), at a cost linear in the number of factors involved.

Block-diagonal recurrences Block-diagonal non input-dependent transitions were first introduced in nonlinear RNNs (Keirsbilck et al., 2019) which are not easily parallelized over the sequence length (Danieli et al., 2025; Gonzalez et al., 2024; Lim et al., 2024) and instead require efficient I/O-aware kernels to saturate modern accelerators (Pöppel et al., 2025; Mishra et al., 2026). Linear RNNs with block diagonal transitions (Fan et al., 2024) are parallelizable across sequence length but limited in model (Walker et al., 2025; Dubinin et al., 2026) and hidden state size (Terzić et al., 2025; Terzić et al., 2025; 2026) since they are currently being trained with parallel scans (cf. Sec. 3). Orthogonal parameterizations are easily integrated with current chunk-wise parallel forms using the RoPE-trick (Su et al., 2024) but are constrained to TC^0 (Liu et al., 2023) for blocks $m \leq 2$ (Lahoti et al., 2026; Movahedi et al., 2025).

Dense parameterizations Factorizations for linear maps allow for efficient computation of the application of dense weight matrices (Dao et al., 2020b;a; Qiu et al., 2024) and e.g., Monarch matrices (Dao et al., 2022) are factorized using block-diagonal matrices. However, our method requires closure and fixed-cost under repeated multiplication, which these methods generally do not provide. Kronecker Recurrent Unit (Jose et al., 2018) is the closest predecessor to our work but is a non-linear RNN with a non input-dependent transition and vector valued states. Classical non-linear RNNs faced numerous training issues due to vanishing and exploding gradients (Pascanu et al., 2013) which were attempted to resolve with orthogonal parameterizations (Arjovsky et al., 2016; Wisdom et al., 2016; Jing et al., 2017; Mhammedi et al., 2017; Helfrich et al., 2018; Lezcano-Casado & Martínez-Rubio, 2019). nnRNN (Kerg et al., 2019) proposed extending parameterizations to non-normal matrices with explicit eigenvalue control to improve expressivity; BGLT leaves normality unconstrained. While previous chunk-wise parallel forms forced orthogonality for unstructured, dense transitions in LA, our method allows for blocks with arbitrary, even singular, structures so long as the norm of the cumulative product of blocks is bounded independently of sequence length.

B NOTATION, USEFUL PROPERTIES, AND PROOFS

***Pushed* quantities**

- Gates product:

$$\boldsymbol{G}_{i\to j} \coloneqq \prod_{k=i}^{j} \boldsymbol{G}_k = \boldsymbol{G}_j \cdots \boldsymbol{G}_i, \qquad \text{with} \qquad \boldsymbol{G}_{i\to j} \coloneqq \boldsymbol{I} \quad \text{if} \quad i > j. \tag{9}$$

- Indexed quantity $\boldsymbol{x}_i$, pushed forward to end of chunk n_E:

$$\overrightarrow{\boldsymbol{x}}_i \coloneqq \boldsymbol{G}_{i+1\to n_E} \boldsymbol{x}_i. \tag{10}$$

- Indexed quantity $\boldsymbol{x}_i$, pushed backward to beginning of chunk n_S:

$$\overleftarrow{\boldsymbol{x}}_i \coloneqq (\boldsymbol{G}_{n_S\to i})^\top \boldsymbol{x}_i. \tag{11}$$

- Indexed quantity $\boldsymbol{x}_i$, pushed forward to index $j > i$:

$$\overrightarrow{\boldsymbol{x}}_i^j \coloneqq \boldsymbol{G}_{i+1\to j} \boldsymbol{x}_i. \tag{12}$$

- Indexed quantity $\boldsymbol{x}_i$, pushed backward to index $j < i$:

$${}^j\overleftarrow{\boldsymbol{x}}_i \coloneqq (\boldsymbol{G}_{j\to i})^\top \boldsymbol{x}_i. \tag{13}$$

Tensor slicing

- Rows / columns of 2D tensor $\boldsymbol{X} \in \mathbb{R}^{m\times n}$:

$$\begin{aligned} [\boldsymbol{X}]_{i,-} &\coloneqq [\boldsymbol{X}_{i,1}, \ldots, \boldsymbol{X}_{i,n}] \in \mathbb{R}^{1\times n}, \\ [\boldsymbol{X}]_{-,j} &\coloneqq [\boldsymbol{X}_{1,j}, \ldots, \boldsymbol{X}_{m,j}]^\top \in \mathbb{R}^{m\times 1}. \end{aligned} \tag{14}$$

- Slice of 3D tensor $\boldsymbol{X} \in \mathbb{R}^{m\times \hat{m}\times \hat{m}}$:

$$[\boldsymbol{X}]_{i,-,-} \coloneqq [\boldsymbol{X}_{i,p,q}]_{p,q=1}^{\hat{m}} \in \mathbb{R}^{\hat{m}\times \hat{m}}. \tag{15}$$

- Tile of 2D tensor $\boldsymbol{X} \in \mathbb{R}^{m\times n}$ for given sets of indices $I = \{i_1, \ldots, i_{|I|}\}$, $J = \{j_1, \ldots, j_{|J|}\}$:

$$\begin{aligned} [\boldsymbol{X}]_I &\coloneqq [\boldsymbol{X}_{p,q}]_{(p,q)\in I\times\{1,\ldots,n\}} \in \mathbb{R}^{|I|\times n}, \\ [\boldsymbol{X}]_{I,J} &\coloneqq [\boldsymbol{X}_{p,q}]_{(p,q)\in I\times J} \in \mathbb{R}^{|I|\times|J|}. \end{aligned} \tag{16}$$

- Vector $\boldsymbol{x}$, restricted to block b of dimension $\hat{m}^2$:

$$\boldsymbol{x}\big|_{(b)} \coloneqq [\boldsymbol{x}_i]_{i=\hat{m}^2(b-1)+1}^{\hat{m}^2 b}. \tag{17}$$

- Block-diagonal matrix $\boldsymbol{X}$ with $\hat{m}^2 \times \hat{m}^2$ blocks, restricted to block b:

$$\boldsymbol{X}\big|_{(b)} \coloneqq [\boldsymbol{X}_{i,j}]_{i,j=\hat{m}^2 b+1}^{(\hat{m}^2+1)b}. \tag{18}$$

Reshaping operations

- Matricization (row-major):

$$\operatorname{mat}(\cdot) : \mathbb{R}^{\hat{m}^2} \to \mathbb{R}^{\hat{m}\times\hat{m}}, \quad \big[\operatorname{mat}(\boldsymbol{x})\big]_{ij} \coloneqq \boldsymbol{x}_{\hat{m}(i-1)+j}, \tag{19}$$

- Vectorization (row-major):

$$\operatorname{vec}(\cdot) : \mathbb{R}^{\hat{m}\times\hat{m}} \to \mathbb{R}^{\hat{m}^2}, \quad \big[\operatorname{vec}(\boldsymbol{X})\big]_{\hat{m}(i-1)+j} \coloneqq \boldsymbol{X}_{ij}, \tag{20}$$

Gradients

- Gradient of loss function $\mathcal{L}$ with respect to parameters $\boldsymbol{X}$ (with some abuse of notation, regardless of the dimensions of $\boldsymbol{X}$):

$$\delta \boldsymbol{X} \coloneqq \partial \mathcal{L} / \partial \boldsymbol{X}. \tag{21}$$

- Definition of differential:

$$d\mathcal{L}(\boldsymbol{Y}) = \operatorname{Tr}\left(\delta \boldsymbol{Y}^\top d\boldsymbol{Y}\right) = \operatorname{vec}\left(\delta \boldsymbol{Y}\right)^\top \operatorname{vec}\left(d\boldsymbol{Y}\right). \tag{22}$$

Selected properties of Kronecker products

- Mixed product:

$$(\boldsymbol{A}_1 \otimes \boldsymbol{B}_1)(\boldsymbol{A}_2 \otimes \boldsymbol{B}_2) = (\boldsymbol{A}_1 \boldsymbol{A}_2) \otimes (\boldsymbol{B}_1 \boldsymbol{B}_2). \tag{23}$$

- Transposition:

$$(\boldsymbol{A} \otimes \boldsymbol{B})^\top = \boldsymbol{A}^\top \otimes \boldsymbol{B}^\top. \tag{24}$$

- Vectorization identity:

$$(\boldsymbol{A} \otimes \boldsymbol{B})\boldsymbol{x} = \operatorname{vec}\left(\boldsymbol{A} \operatorname{mat}\left(\boldsymbol{x}\right) \boldsymbol{B}^\top\right). \tag{25}$$

Selected properties of trace operator

- Commutativity:

$$\operatorname{Tr}(\boldsymbol{A}\boldsymbol{B}) = \operatorname{Tr}(\boldsymbol{B}\boldsymbol{A}).$$

- Invariance under cyclic permutation:

$$\operatorname{Tr}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \operatorname{Tr}(\boldsymbol{C}\boldsymbol{A}\boldsymbol{B}).$$

- Mixed standard-Hadamard matrix product:

$$\operatorname{Tr}(\boldsymbol{A}^\top (\boldsymbol{B} \odot \boldsymbol{C})) = \operatorname{Tr}((\boldsymbol{A} \odot \boldsymbol{C})^\top \boldsymbol{B}).$$

Gradients through Kronecker factors. For the derivations in App. F.2, we rely on the following useful lemmas to compute gradients through Kronecker factors:

Lemma 1. *Consider the relationship*

$$\boldsymbol{y} = (\boldsymbol{A}_{1\to t} \otimes \boldsymbol{B}_{1\to t})^\top \boldsymbol{x} = \operatorname{vec}\left(\boldsymbol{A}_{1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t}\right), \tag{26}$$

for given input and output vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{\hat{m}^2}$, the row-wise matrix reshape $\boldsymbol{X} = \operatorname{mat}\left(\boldsymbol{x}\right) \in \mathbb{R}^{\hat{m}\times\hat{m}}$ (and similarly for $\boldsymbol{Y}$), and the Kronecker factors $\boldsymbol{A}_j, \boldsymbol{B}_j \in \mathbb{R}^{\hat{m}\times\hat{m}}$, $j = 1, \dots, t$. Given the gradient of a given loss function $\mathcal{L}(\boldsymbol{y})$ with respect to $\boldsymbol{y}$, $\delta\boldsymbol{y} = d\mathcal{L}(\boldsymbol{y})/d\boldsymbol{y}$, the gradient with respect to any factor $\boldsymbol{A}_i, \boldsymbol{B}_i$ is given by:

$$\begin{aligned} \delta \boldsymbol{A}_i &= \boldsymbol{A}_{i+1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t} \delta \boldsymbol{Y}^\top \boldsymbol{A}_{1\to i-1}^\top, \\ \delta \boldsymbol{B}_i &= \boldsymbol{B}_{i+1\to t}^\top \boldsymbol{X}^\top \boldsymbol{A}_{1\to t} \delta \boldsymbol{Y} \boldsymbol{B}_{1\to i-1}^\top. \end{aligned} \tag{27}$$

Proof. The proof follows from a series of algebraic manipulations:

$$\begin{aligned} d\mathcal{L} &= \delta\boldsymbol{y}^\top d \operatorname{vec}\left(\boldsymbol{A}_{1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t}\right) = \delta\boldsymbol{y}^\top \operatorname{vec}\left(\boldsymbol{A}_{1\to i-1}^\top d\boldsymbol{A}_i^\top \boldsymbol{A}_{i+1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t}\right) \\ &= \delta\boldsymbol{y}^\top \left(\boldsymbol{A}_{1\to i-1}^\top \otimes \left(\boldsymbol{A}_{i+1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t}\right)^\top\right) \operatorname{vec}\left(d\boldsymbol{A}_i^\top\right) \\ &= \operatorname{vec}\left(\underbrace{\boldsymbol{A}_{1\to i-1} \delta \boldsymbol{Y} \left(\boldsymbol{A}_{i+1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t}\right)^\top}_{=\delta \boldsymbol{A}_i^\top}\right)^\top \operatorname{vec}\left(d\boldsymbol{A}_i^\top\right) \end{aligned} \tag{28}$$

$$\implies \quad \delta \boldsymbol{A}_i = \boldsymbol{A}_{i+1\to t}^\top \boldsymbol{X} \boldsymbol{B}_{1\to t} \delta \boldsymbol{Y}^\top \boldsymbol{A}_{1\to i-1}^\top,$$

and similarly for $\delta \boldsymbol{B}_i = \boldsymbol{B}_{i+1\to t}^\top \boldsymbol{X}^\top \boldsymbol{A}_{1\to t} \delta \boldsymbol{Y} \boldsymbol{B}_{1\to i-1}^\top$. □

Lemma 2. *Consider the relationship*

$$\boldsymbol{Y} = \boldsymbol{A}_{1\to C} \otimes \boldsymbol{B}_{1\to C}, \tag{29}$$

for a given output matrix $\boldsymbol{Y} \in \mathbb{R}^{\hat{m}^2\times\hat{m}^2}$, and the Kronecker factors $\boldsymbol{A}_j, \boldsymbol{B}_j \in \mathbb{R}^{\hat{m}\times\hat{m}}$, $j = 1,\dots,C$. If the gradient of a given loss function $\mathcal{L}(\boldsymbol{Y})$ with respect to $\boldsymbol{Y}$, $\delta\boldsymbol{Y} = d\mathcal{L}(\boldsymbol{Y})/d\boldsymbol{Y}$ comes in an outer-product form, i.e.,

$$\delta\boldsymbol{Y} = \boldsymbol{x}_1\boldsymbol{x}_2^\top \qquad \text{for given } \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^{\hat{m}^2}, \tag{30}$$

then the gradient with respect to any factor $\boldsymbol{A}_i, \boldsymbol{B}_i$ is given by:

$$\begin{aligned} \delta\boldsymbol{A}_i &= \boldsymbol{A}_{i+1\to C}^\top \boldsymbol{X}_1 \boldsymbol{B}_{1\to C} \boldsymbol{X}_2^\top \boldsymbol{A}_{1\to i-1}^\top, \\ \delta\boldsymbol{B}_i &= \boldsymbol{B}_{i+1\to C}^\top \boldsymbol{X}_1^\top \boldsymbol{A}_{1\to C} \boldsymbol{X}_2 \boldsymbol{B}_{1\to i-1}^\top, \end{aligned} \tag{31}$$

with $\boldsymbol{X}_i = \mathrm{mat}\,(\boldsymbol{x}_i)$, $i = 1, 2$.

Proof. Even in this case, a series of algebraic manipulations recovers the result:

$$\begin{aligned} d\mathcal{L} &= \mathrm{Tr}\left(\delta\boldsymbol{Y}^\top d\left(\boldsymbol{A}_{1\to C}\otimes\boldsymbol{B}_{1\to C}\right)\right) \\ &= \mathrm{Tr}\left(\boldsymbol{x}_2\boldsymbol{x}_1^\top\left((\boldsymbol{A}_{i+1\to C}\otimes\boldsymbol{B}_{i+1\to C}) d(\boldsymbol{A}_i\otimes\boldsymbol{B}_i)(\boldsymbol{A}_{1\to i-1}\otimes\boldsymbol{B}_{1\to i-1})\right)\right) \\ &= \mathrm{Tr}(\underbrace{(\boldsymbol{A}_{1\to i-1}\otimes\boldsymbol{B}_{1\to i-1})\boldsymbol{x}_2}_{=:\hat{\boldsymbol{x}}_2}(\underbrace{(\boldsymbol{A}_{i+1\to C}\otimes\boldsymbol{B}_{i+1\to C})^\top\boldsymbol{x}_1}_{=:\hat{\boldsymbol{x}}_1})^\top d(\boldsymbol{A}_i\otimes\boldsymbol{B}_i)) \\ &= \mathrm{Tr}\left(\hat{\boldsymbol{x}}_2\hat{\boldsymbol{x}}_1^\top d(\boldsymbol{A}_i\otimes\boldsymbol{B}_i)\right) = \hat{\boldsymbol{x}}_2^\top d(\boldsymbol{A}_i\otimes\boldsymbol{B}_i)^\top\hat{\boldsymbol{x}}_1 \qquad \text{fix } \boldsymbol{B}_i\text{, focus on } d\boldsymbol{A}_i\text{:} \\ &= \mathrm{vec}\left(\hat{\boldsymbol{X}}_2\right)^\top \mathrm{vec}\left(d\boldsymbol{A}_i^\top\hat{\boldsymbol{X}}_1\boldsymbol{B}_i\right) = \mathrm{Tr}\left(\hat{\boldsymbol{X}}_2^\top d\boldsymbol{A}_i^\top\hat{\boldsymbol{X}}_1\boldsymbol{B}_i\right) \\ \implies \quad \delta\boldsymbol{A}_i &= \hat{\boldsymbol{X}}_1\boldsymbol{B}_i\hat{\boldsymbol{X}}_2^\top = \boldsymbol{A}_{i+1\to C}^\top\boldsymbol{X}_1\boldsymbol{B}_{i+1\to C}\boldsymbol{B}_i\boldsymbol{B}_{1\to i-1}\boldsymbol{X}_2^\top\boldsymbol{A}_{1\to i-1}^\top \\ &= \boldsymbol{A}_{i+1\to C}^\top\boldsymbol{X}_1\boldsymbol{B}_{1\to C}\boldsymbol{X}_2^\top\boldsymbol{A}_{1\to i-1}^\top, \end{aligned} \tag{32}$$

and similarly for $\delta\boldsymbol{B}_i = \boldsymbol{B}_{i+1\to C}^\top\boldsymbol{X}_1^\top\boldsymbol{A}_{1\to C}\boldsymbol{X}_2\boldsymbol{B}_{1\to i-1}^\top$. □

Numerical instability of inverse-based factorization In Sec. 2 we mention how relying on computing inverses of products of general gate blocks for the factorization of the attention matrix $\boldsymbol{A}_{[n]}^{qk}$ quickly results in loss of numerical precision. Here we quantify this claim, outlining the unreasonable constraints we need to impose on the gates maximal decay rate (i.e., their *minimal* singular value $\sigma_{\min}(\boldsymbol{G}_t)$) to ensure numerical stability.

Lemma 3 (Precision lost to the chunk-wise inverse)**.** *Let every $\boldsymbol{G}_t$ have singular values $\sigma \in [\sigma_{\min}, 1]$. Consider their cumulative products stored at precision u. Then the computed factorized keys $\boldsymbol{k}_s^{inv} = \boldsymbol{G}_{n_S\to s}^{-1}\boldsymbol{k}_s$ carry a relative error of*

$$\varepsilon \le \kappa(\boldsymbol{G}_{n_S\to n_E})u \le u\sigma_{\min}^{-C}, \tag{33}$$

where $\kappa(*)$ is the condition number of matrix $(*)$. In other words, inversion loses $C\log_{10}(1/\sigma_{\min})$ decimal digits of the $\log_{10}(1/u)$ available at the stored precision. Targeting a numerical precision tolerance of $\varepsilon \le \mathrm{tol}$ therefore imposes

$$C\log_{10}\frac{1}{\sigma_{\min}} \;\le\; \log_{10}\frac{1}{u} - \log_{10}\frac{1}{\mathrm{tol}}, \qquad \implies \qquad \sigma_{\min} \ge \left(\frac{u}{\mathrm{tol}}\right)^{\frac{1}{C}}. \tag{34}$$

Proof. We have that

$$\sigma_{\min}(\boldsymbol{G}_{n_S\to n_E}) \ge \prod_{t=n_S}^{n_E}\sigma_{\min}(\boldsymbol{G}_t) \ge \sigma_{\min}^C, \tag{35}$$

with equality holding in the worst-case scenario where all gates share the singular vectors associated with $\sigma_{\min}$. Similarly, $\sigma_{\max}(\boldsymbol{G}_{n_S\to n_E}) \le 1$, implying $\kappa(\boldsymbol{G}_{n_S\to n_E}) \le \sigma_{\min}^{-C}$. Since we obtain the

factorized keys $\boldsymbol{k}_t^{\text{inv}}$ by numerically inverting $\boldsymbol{G}_{n_S \to t}$, then, following the standard perturbation bound for a linear solve (Higham, 2002) we get that their relative error ε is bounded by

$$\varepsilon \leq \kappa(\boldsymbol{G}_{n_S \to t}) u \leq \sigma_{\min}^{-C} u, \tag{36}$$

which recovers (33). Taking $\log_{10}$ turns the amplification $\sigma_{\min}^{-C}$ into $C \log_{10}(1/\sigma_{\min})$ lost digits, and imposing $u\sigma_{\min}^{-C} \leq \text{tol}$ yields (34), completing the proof. □

We point out that the bound in (33) is *optimistic*, in that it start from a numerical round-off of the cumulative products. In reality, each gate is *by itself* stored at precision u and the product round-off errors accumulate in finite precision. Nonetheless, this error is incurred by virtually any chunk-wise formulation, while the error we track in (33) is only associated to factorization via forced inversion. We also point out that diagonal gates do not pay the same approximation price, as gates in that case are accumulated and inverted in log space.

To quantify the constraint imposed by the bounds in Lem. 3, consider that a given datatype provides a $\log_{10}(1/u)$ precision budget: 2.41 digits for `bf16`, 7.22 for `fp32`, 15.95 for `fp64`; of these, we would like at least $\log_{10}(1/\text{tol})$ to survive, which at $\text{tol} = 1\%$ translates into 0.41, 5.22 and 13.95 digits, respectively. Assuming a typical chunk length of $C = 64$, imposing $\sigma_{\min} = 0.9$ onto our gates already destroys 2.93 digits: $7\times$ over the `bf16` budget. Inverting the requirement, `bf16` at $C = 64$ allows at best $\sigma_{\min} \geq 0.985$. Even pushing to `fp32` only relaxes this constraint to $\sigma_{\min} \geq 0.83$. In Tab. B.1 we report additional evaluations of the constraints imposed by (34) on $\sigma_{\min}$.

Table B.1: Precision cost of the chunk-wise inverse, from Lem. 3, at a tolerance $\text{tol} = 1\%$, with available precision budget of $\log_{10}(1/u) - \log_{10}(1/\text{tol})$, i.e. 0.41 digits for `bf16`, 5.22 for `fp32` and 13.95 for `fp64`.

(a) Digits corrupted by inversion, $C \log_{10}\left(\frac{1}{\sigma_{\min}}\right)$

	C			
$\sigma_{\min}$	16	32	64	128
0.95	0.36	0.71	1.43	2.85
0.90	0.73	1.46	2.93	5.86
0.85	1.13	2.26	4.52	9.03
0.80	1.55	3.10	6.20	12.40
0.70	2.48	4.96	9.91	19.83

(b) Resulting constraint on $\boldsymbol{G}_t$, $\sigma_{\min} \geq \left(\frac{u}{\text{tol}}\right)^{\frac{1}{C}}$

	datatype		
C	`bf16`	`fp32`	`fp64`
16	0.943	0.471	0.134
32	0.971	0.687	0.366
64	0.985	0.829	0.605
128	0.993	0.910	0.778
256	0.996	0.954	0.882

C LAYER DESIGN

Summary

This appendix describes the structure of a BGLT layer in full. As a drop-in replacement for self-attention in a standard Transformer block, its surrounding architecture (App. C.1, such as projections, short convolutions, normalization, and multi-head layout), is largely borrowed from existing linear RNNs like KDA. The main novelty lies in the assembly of the state transitions (App. C.2). Each 16×16 block is a Kronecker product of two 4×4 factors, and each factor is built via its Singular Value Decomposition (SVD) $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$: this gives direct access to its singular values $\boldsymbol{\Sigma}$, which we explicitly bound by 1 in modulo to ensure stability both in the forward and the backward pass. Moreover, SVD comes at the same 16-parameter cost as a 4×4 dense matrix. At initialization, the rotations $\boldsymbol{U}, \boldsymbol{V}$ are set to identity, meaning the model starts as a KDA-like diagonal gate and learns to rotate and mix during training.

C.1 MODEL ARCHITECTURE

The BGLT block acts as a substitute for self-attention inside a standard pre-norm Transformer block: the layer receives as input the normalized residual $\boldsymbol{x}_t \in \mathbb{R}^d$ and returns a vector of the same dimension, leaving MLPs and residual structure untouched. Figure C.1 reports schematic for the block, whose components we describe next. For the actual functioning of the core sequence mixer, we refer instead to App. F.1.

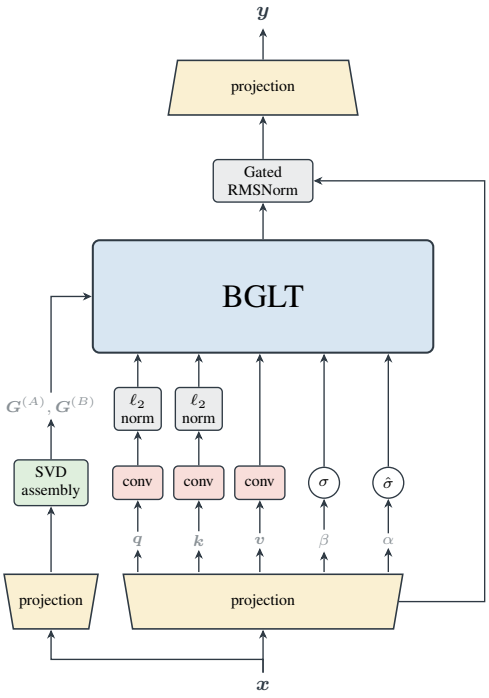


Figure C.1: Schematics of BGLT block.

Projections, convolutions, and normalization All quantities fed to the BGLT cell are produced from $\boldsymbol{x}_t$ via a linear projection. These include queries $\boldsymbol{q}$, keys $\boldsymbol{k}$, values $\boldsymbol{v}$, the read-out gate, and, when the Delta Rule correction is enabled, the corresponding coefficients β. Analogously to the procedure introduced in Mamba-2 (Dao & Gu, 2024), queries, keys, and values are then processed by a short convolution of width 4, with $\boldsymbol{k}$ and $\boldsymbol{q}$ further ℓ_2-normalized, and $\boldsymbol{q}$ scaled by $d_k^{-1/2}$, before being passed as inputs to the core RNN cell. While $\beta \in [0, 2]$ suffices to guarantee stability, in practice we restrict $\beta \in [0, 1]$ via a sigmoid nonlinearity, following similar standards in KDA (Kimi Team et al., 2025). The read-out gate is employed for the final Gated RMSNorm, right before the output projection. A second, narrower input projection produces the parameters for the assembly of the state-transition matrix $\boldsymbol{G}$: we describe it separately in App. C.2.

Multi-head implementation The layer application is split into heads following grouped-value attention (Dao & Gu, 2024). As such, H query- and key-heads of size d_k are produced, with d_k a multiple of 16 to ensure consistency with the dimension of the blocks in the transition matrices. Values, however, live on a separate set of H_v heads, with H_v a multiple of H, so that multiple value-heads are associated with a same query/key-head.

C.2 State transition parameterization

Recall that BGLT uses block-diagonal state transition matrices composed of 16×16 blocks, each a Kronecker product of two 4×4 factors,

$$\boldsymbol{G}_t := \bigoplus_{b=1}^{n_B} \boldsymbol{G}_t^{(b)}, \qquad \text{with} \qquad \boldsymbol{G}_t^{(b)} := \boldsymbol{G}_t^{(b,A)} \otimes \boldsymbol{G}_t^{(b,B)}. \tag{37}$$

The parameterization of each factor is, in principle, a free design choice, but for the training procedure to remain stable we must ensure that both forward and backward passes are well-behaved. As the recurrence multiplies the hidden state by $\boldsymbol{G}_t$ at each step, this translates into a bound on the norm of both the transition matrix and its transpose. This necessity was highlighted already in the BD-RNN work (Dubinin et al., 2026): there, the authors apply a row-wise softmax operation to the transitions, in order to make them row-stochastic, thereby bounding their L_∞-norm $\|\boldsymbol{G}_t\|_\infty$. While row-softmax is cheap to apply, this normalization procedure does not tightly bound the norm of the transpose, which is responsible for propagating gradients: in fact, we only have $\|\boldsymbol{G}_t^\top\|_\infty = \|\boldsymbol{G}_t\|_1 \leq m\|\boldsymbol{G}_t\|_\infty$. In this work we opt to control the gates L_2-norm instead, given its invariance under transposition, and impose $\|\boldsymbol{G}_t\|_2 \leq 1$. As illustrated in Fig. C.2, this resulted in better training behavior during our initial ablations, justifying the larger cost associated with the procedure for L_2-normalization. We report corresponding timing results in App. G.4, and outline said procedure in the following.

For block-diagonal matrices, their norm is given by the maximum norm of its individual blocks, and for a Kronecker product we have $\|\boldsymbol{G}_t^{(b,A)} \otimes \boldsymbol{G}_t^{(b,B)}\|_2 = \|\boldsymbol{G}_t^{(b,A)}\|_2 \|\boldsymbol{G}_t^{(b,B)}\|_2$, so it suffices to individually constrain each of its factors: this translates into bounding their singular values. One could achieve this by spectrally-normalizing any given matrix, but relying from the get-go on an SVD parameterization gives us more fine-grained control on the resulting gate: this is the strategy we follow for BGLT. In practice, $\forall b, t$ we assemble the SVD components

$$\boldsymbol{G}_t^{(b,A)} = \boldsymbol{U}_t^{(b,A)} \boldsymbol{\Sigma}_t^{(b,A)} (\boldsymbol{V}_t^{(b,A)})^\top, \qquad \text{and} \qquad \boldsymbol{\Sigma}_t^{(b,A)} = \operatorname{diag}[\sigma_{t,1}^{(b,A)}, \ldots, \sigma_{t,4}^{(b,A)}], \tag{38}$$

with both $\boldsymbol{U}_t^{(b,A)}, \boldsymbol{V}_t^{(b,A)}$ orthogonal, and similarly for $\boldsymbol{G}_t^{(b,B)}$. This parameterization gives us the freedom to independently tweak all spectral properties of the gates: their singular values distribution, as well as their degree of non-normality. Moreover, we can assemble the full SVD parameterization starting from just $\boldsymbol{z} \in \mathbb{R}^{16}$ features produced from an input projection, despite involving three 4×4 matrices. The details are provided next.

Singular values $\boldsymbol{\Sigma}$ The SVD parameterization in (38) allows us to explicitly bound the singular values of $\boldsymbol{G}_t^{(b,A)}$: to ensure stability, it suffices to constrain $\max_i |\sigma_i| \leq 1$ (where we dropped

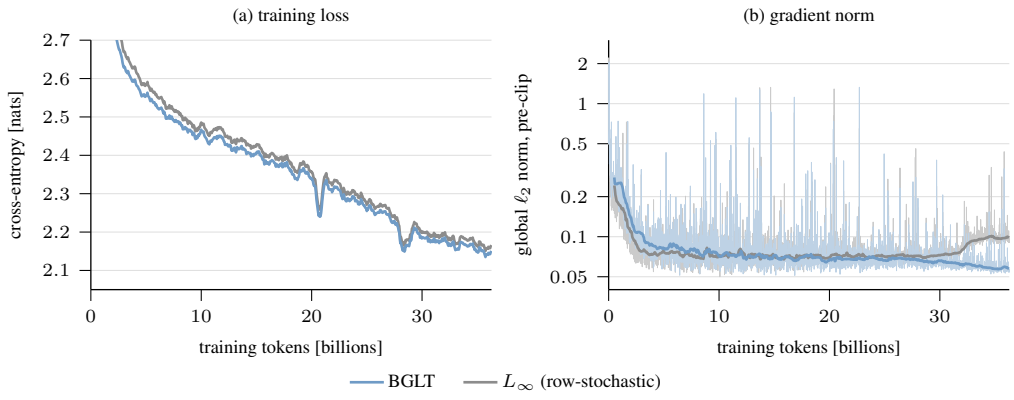


Figure C.2: Training cross-entropy (a) and pre-clip gradient norm (b) of BGLT with Kronecker factors L_2-normalized via SVD, as in (38), or L_∞-normalized via row-wise softmax, as used by Dubinin et al. (2026). 400M parameters over 36B tokens (5× Chinchilla), both without the Delta Rule. Matched parameter counts and hyperparameters. L_∞-normalization, despite being cheaper to apply, results in worse CE loss and larger gradient norms towards later stages in training.

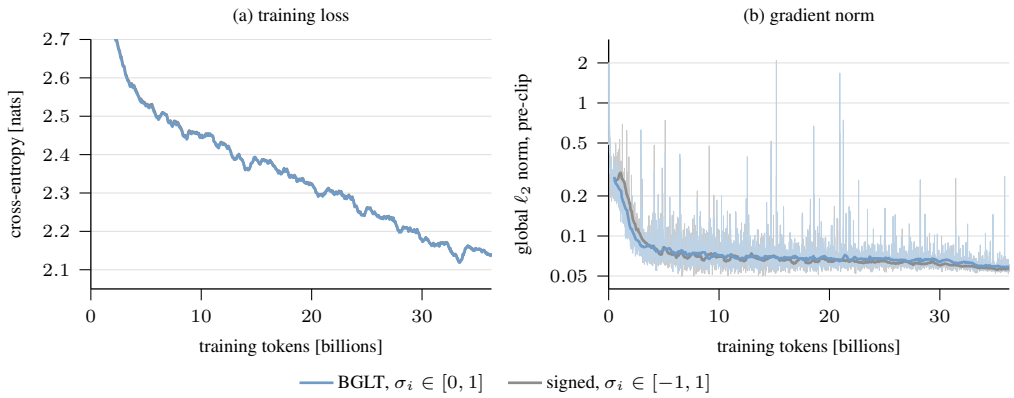


Figure C.3: Training cross-entropy (a) and pre-clip gradient norm (b) of BGLT with singular values $\sigma_i \in [0, 1]$, as in (39), or signed, remapped to $2\sigma_i - 1 \in [-1, 1]$. 400M parameters over 36B tokens (5× Chinchilla), both without the Delta Rule.

references to block b, factor A/B, and position t to slim notation). While employing signed singular values is useful for solving the S_5 state-tracking task, as described in Sec. 3.2 and shown in Fig. 6, in practice early ablations showed marginally better training behavior by restricting $\sigma_i \in [0, 1]$, which is what we use for our large-scale training runs: see also Fig. C.3. Note that the singular values cover a similar role to the diagonal gates in KDA: fixing $\boldsymbol{U} \equiv \boldsymbol{V} \equiv \boldsymbol{I}$, in fact, reduces each gate block to a purely diagonal one, whose elements are however constrained to the form $\sigma_i^{(A)}\sigma_j^{(B)}$. In light of this, we parameterize the singular values similarly to KDA, using

$$\sigma_i = \exp\Big(-\exp(A_{\log}) \cdot \mathrm{softplus}(z_i + \delta_\sigma)\Big) \in [0, 1]. \tag{39}$$

for a given scale parameter $A_{\log}$ and bias δ_σ, and four features z_i produced by the input projection.

Also for their initialization, we follow the blueprint of KDA: the scale is drawn as $\exp(A_{\log}) \sim \mathcal{U}(1, 16)$ and the bias δ_σ is set by sampling a target step size $\Delta \sim \mathcal{U}(\Delta_{\min}, \Delta_{\max})$ and inverting the softplus, $\delta_\sigma = \Delta + \log(1 - e^{-\Delta})$: this way, (39) reproduces the intended distribution. Since in our case the singular the singular values of the two Kronecker factors appear multiplicatively, we keep KDA's own range $\Delta_{\min} = 10^{-3}$, $\Delta_{\max} = 10^{-1}$, but split each realization randomly between the two Kronecker factors, so that $\sigma_i^{(A)}\sigma_i^{(B)}$ exactly reproduces KDA's initial diagonal-gate distribution.

Orthogonal factors $\boldsymbol{U}, \boldsymbol{V}$ To assemble each of the $\boldsymbol{U}, \boldsymbol{V}$ matrices for the SVD parameterization (38), we use the fact that matrix exponentials of skew-symmetric matrices are orthogonal. This follows directly from their definition $\boldsymbol{Z}^\top = -\boldsymbol{Z}$, which allows us to write:

$$\boldsymbol{U}^\top = \exp(\boldsymbol{Z})^\top = \exp(-\boldsymbol{Z}) = \exp(\boldsymbol{Z})^{-1} = \boldsymbol{U}^{-1}. \tag{40}$$

The exponential of a 4×4 skew-symmetric matrix admits a closed-form expression of Rodrigues type (Politi, 2001), but this presents a removable singularity at 0, so we revert to a Taylor series approximation when $\boldsymbol{Z}$ has small norm. For 4×4 matrices, we just need 6 elements to assemble a skew-symmetric matrix (its strictly upper-triangular part), which we fetch from the features $\boldsymbol{z}$ obtained from the input projection (see Fig. C.1). This brings the total of features needed to perform a full SVD parameterization to $4 + 6 + 6 = 16$, exactly matching the 4×4 elements composing the factor itself.

Note that, in loose terms, the norm of $\boldsymbol{Z}$ directly ties to the amount of *rotation* prescribed by $\boldsymbol{U} = \exp(\boldsymbol{Z})$: $\boldsymbol{Z} \equiv \boldsymbol{0}$ maps to $\boldsymbol{U} \equiv \boldsymbol{I}$, with the change of basis encoded by $\boldsymbol{U}$ progressively rotating as we increase $\|\boldsymbol{Z}\|$. This allows to tweak the amount of rotation prescribed by the gates by directly controlling $\|\boldsymbol{Z}\|$. At initialization, we fix $\|\boldsymbol{Z}\| = 0$: this collapses our gate to diagonal, matching KDA (except with fewer degrees of freedom, due to the Kronecker factorization). The cell can then generalize KDA by increasing $\|\boldsymbol{Z}\|$, which causes gates to rotate, but also by increasing the difference between $\boldsymbol{U}$ and $\boldsymbol{V}$, which increases the degree of non-normality of the gate factor.

D MEASURING MODEL EXPRESSIVITY THROUGH SYNTHETIC TASKS

Summary

This appendix evaluates the expressivity of single-layer models (LSTM, GRU, Transformer, GLA, KDA, BGLT, and ΔBGLT) across nine synthetic tasks, with full results presented in Table D.1. These tasks test a wide range of algorithmic capabilities: recall (MQAR, k-hops, copying), counting and arithmetic, group state tracking, and Neural Cellular Automata (NCA). We carry out additional analyses on NCA as it allows us to systematically verify the role of block-diagonal gates in tracking complex state dynamics.
Take-home: Among the ones considered, BGLT and ΔBGLT are the *only* linear models capable of solving state tracking. They also outperform KDA on modular arithmetic without brackets ($\sim 98\%$ vs. 76%) and copy memory ($\sim 99\%$ vs. 59%). Overall, ΔBGLT is the strongest (or tied for strongest) linear model across most tasks, including every NCA setting. Furthermore, the inclusion of the Delta Rule proves critical for complex recall, as only KDA and ΔBGLT solve the 4-hop task. While nonlinear RNNs (LSTM/GRU) remain the upper bound for pure state-tracking, they fail on recall tasks. Finally, results on NCA show that ΔBGLT outperforms other linear transformers, especially for sequences with higher complexity.

As discussed in the main text, in order to verify the expressivity of different linear transformers, we trained multiple *single-layer* models against a variety of synthetic tasks, which are presented in Tab. D.1 below. We are especially interested in testing the ability of our models to track complex state transitions, which is why in App. D.5 we analyze NCA performance in terms of task difficulty, as well as rollout complexity.

We provide further details on model setup, training regime, synthetic task definitions as well as analyze our results in what follows.

D.1 MODEL SETUP

For all synthetic tasks covered in this section, we employ a single-layer model composed of: *i.* embedding layer, *ii.* LayerNorm, *iii.* linear projection(s), *iv.* short-convolution (with kernel-size set to four), *v.* multi-head attention mechanism (with four heads, each with a fourth of the overall dimension d), *vi.* Gated RMSNorm, *vii.* another linear projection, *viii.* a residual connection, *ix.* a second LayerNorm, *x.* a two-layer MLP with a SwiGLU non-linearity which first up-projects by a variable ratio (which we use to parameter match all our models), *xi.* another residual conection, *xii.* a final LayerNorm, and *xiii.* an unembedding layer, tied to the embedding layer *i*. Layers *iii.*–*vii.* mirror the schematics in Fig. C.1, adapted to multiple attention mechanisms.

The attention mechanism can be one of LSTM, GRU, Self-Attention with RoPE (which we denote as Transformer), GLA, KDA, BGLT, and ΔBGLT. For LSTM and GRU, the recurrent cells are fed by a single projection and short-convolution; for the other attention mechanisms, instead, we have independent projections and short-convolutions for keys, queries, and values, with a LayerNorm further applied to the keys and queries. Note that when equipped with self-attention the resulting architecture differs from the standard transformer; we opted to follow the conventional architecture of linear transformers, keeping all models with as few differences as possible.

We use a model dimension $d = 128$ by default. In order to keep all models at the same parameter count, we modify the up-projection ratio in layer *x.* above. This way, models whose attention mechanism has larger number of parameters, will have fewer parameters on the last projection[1]. This likewise means that all models have the same effective hidden dimension for each of their four heads, i.e. 32 when $d = 128$.

[1]For instance, for a vocabulary size $|V| = 16$ and model dimension d=128, a Transformer with an up-projection of 4.0, will be parameter-matched by a ΔBGLT with an up-projection ratio of ≈ 3.43.

Table D.1: **Accuracy of *single-layer* models across nine synthetic tasks** Model architecture is detailed in App. D.1. All models were trained following the recipe in App. D.2. Tasks and their hyper-parameters are described in App. D.3. Due to early-stopping, tasks with 99.0% accuracy or higher are equivalent. We report the accuracy of the best and worst seeds for each task/model combination.

	LSTM	GRU	TRANSF.	GLA	KDA	BGLT	ΔBGLT
MQAR $\kappa = 16$ $\lvert V\rvert = 256$ $T = 128$ 305k	67.4% 60.9% η: 10^{-3} 27min	75.9% 60.5% η: 10^{-2} 32min	100.0% 100.0% η: 5×10^{-3} 5min	99.9% 99.8% η: 5×10^{-2} 5min	99.7% 99.4% η: 10^{-2} 5min	99.8% 99.5% η: 5×10^{-3} 5min	99.5% 99.2% η: 5×10^{-2} 10min
k-HOPS $k = 1$ $\lvert V\rvert = 16$ $T = 128$ 275k	100.0% 99.5% η: 10^{-2} 4min	100.0% 99.9% η: 5×10^{-2} 5min	99.7% 99.6% η: 5×10^{-2} 5min	99.8% 99.7% η: 5×10^{-2} 7min	100.0% 99.9% η: 5×10^{-2} 7min	100.0% 99.8% η: 5×10^{-2} 5min	100.0% 100.0% η: 5×10^{-2} 7min
k-HOPS $k = 4$ $\lvert V\rvert = 16$ $T = 128$ 275k	54.7% 50.7% η: 5×10^{-3} 26min	50.1% 48.9% η: 5×10^{-3} 31min	48.6% 48.5% η: 5×10^{-3} 31min	49.2% 48.3% η: 5×10^{-4} 34min	99.5% 99.0% η: 5×10^{-4} 7min	49.1% 48.8% η: 5×10^{-3} 36min	99.2% 99.0% η: 5×10^{-4} 14min
CYCLE NAV $c = 10$ $s = 2$ $\lvert V\rvert = 16$ $T \in [4, 128]$ 275k	99.0% 98.8% η: 5×10^{-4} 17min	99.1% 96.5% η: 10^{-4} 28min	99.2% 98.7% η: 5×10^{-4} 24min	99.2% 98.8% η: 5×10^{-4} 18min	99.1% 98.0% η: 5×10^{-4} 29min	99.9% 9.7% η: 5×10^{-3} 11min	99.2% 99.0% η: 10^{-3} 11min
MOD. ARITHM. (w/o brackets) $\lvert V\rvert = 12$ $T \in [4, 128]$ 274k	97.7% 78.3% η: 10^{-3} 29min	98.6% 80.9% η: 10^{-3} 34min	43.0% 38.5% η: 5×10^{-3} 26min	42.8% 37.9% η: 10^{-3} 31min	68.3% 48.9% η: 5×10^{-3} 35min	99.1% 77.0% η: 10^{-3} 25min	96.6% 89.8% η: 5×10^{-4} 40min
MOD. ARITHM. (with brackets) $\lvert V\rvert = 16$ $T \in [4, 32]$ 275k	88.7% 79.5% η: 10^{-3} 26min	86.6% 83.7% η: 5×10^{-3} 25min	62.0% 54.7% η: 10^{-2} 26min	63.2% 61.5% η: 5×10^{-3} 26min	65.3% 62.1% η: 5×10^{-3} 37min	71.9% 70.3% η: 10^{-2} 40min	75.5% 69.9% η: 5×10^{-3} 43min
COPY MEMORY $M = 63$ $\lvert V\rvert = 16$ $T = 128$ 275k	11.5% 7.1% η: 5×10^{-4} 26min	7.3% 7.3% η: 5×10^{-4} 34min	100.0% 100.0% η: 10^{-3} 2min	44.3% 41.6% η: 5×10^{-4} 32min	62.0% 56.2% η: 5×10^{-4} 37min	99.1% 99.0% η: 5×10^{-4} 10min	99.3% 7.3% η: 5×10^{-3} 34min
NCA 8×8 2 200 $\lvert V\rvert = 16$ $T = 128$ 275k	87.1% 86.2% η: 5×10^{-3} 27min	86.9% 83.4% η: 5×10^{-3} 35min	78.6% 77.4% η: 10^{-2} 27min	70.7% 70.5% η: 5×10^{-3} 34min	71.3% 71.1% η: 10^{-3} 35min	77.0% 75.5% η: 10^{-2} 40min	78.9% 76.3% η: 10^{-2} 39min
STATE TRACKING Group: A5 $\lvert V\rvert = 61$ $T = 128$ 280k	100.0% 100.0% η: 5×10^{-3} 22min	100.0% 8.7% η: 5×10^{-3} 10min	6.3% 5.5% η: 10^{-3} 27min	6.5% 5.2% η: 10^{-2} 31min	6.2% 5.4% η: 10^{-3} 36min	100.0% 100.0% η: 10^{-3} 8min	100.0% 100.0% η: 5×10^{-3} 8min

KEY: xx.x% Best result xx.x% Best result for linear transformers Stopped early Minimum accuracy η Learning rate Wall time to completion T Sequence length $\lvert V\rvert$ Vocabulary size Num. parameters

D.2 Training Details

For every synthetic task, we train each model with three different training seeds. For each task, we randomly generate 100,000 training samples (with a fixed seed, different from the model seed), and 3,000 test samples (again with a fixed seed different from the training samples, and the model seed).

We employ the AdamW optimizer with weight decay set to 10^{-1}, sweep the following learning rates: $\{10^{-4}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-2}, 5 \times 10^{-2}\}$, and use a cosine scheduler with linear warm-up for 10% of the training steps. We set the batch size to 256, and train for 500 epochs, resulting in a total of 195,000 gradient updates. For each model, we select the learning rate that yields best accuracy for a particular task.

After every epoch the training script verifies the performance on the task test set, and if test accuracy is higher than 99%, training stops early. This implies that all results greater than 99% in Table D.1 should be considered equivalent.

Overall, we train 126 models per task, for a grand-total of 1,134 models trained for Table D.1. Each model is trained on a single node with 24 CPUs, 250GiB of RAM, and an NVIDIA H100 accelerator for up to one hour.

D.3 Synthetic Tasks Definitions

Below we briefly describe each of the synthetic tasks we used for comparison[2]:

- **Multiple-Query Associative Recall (MQAR)** Given an input sequence of key-value pairs $(k, v) \in \mathcal{K} \times \mathcal{V}$, followed by noise interwoven with the keys, output for each key its associated value. The task vocabulary is defined as $V = \mathcal{K} \cup \mathcal{V}$. We report accuracy across the set of responses to key inputs. Random chance is $\frac{1}{|\mathcal{V}|}$.
- **k-hops** Given an input sequence of values from a dictionary V, for each element in the sequence "hop back" and report the value following its previous occurrence in the sequence. Keep on hopping back k times. The special case of 1-hop is also known as *induction heads*. For accuracy computation, we consider the model output at every position. Random chance is $\frac{1}{|V|}$.
- **Cycle Navigation** Given a sequence of steps $x_t \in \{-s, \dots, 0, \dots, s\}$ on a cycle of size c, output the final position $\mod (\sum_{t=1}^{T} x_t, c)$. Each sample in this task has a different length, where T is uniformly sampled in a pre-determined range. The task vocabulary is composed of $(2 \times s + 1)$ step tokens, c output tokens, and an $=$ token. We compute accuracy on solely the output of the $=$ tokens. Random chance is $\frac{1}{c}$.
- **Modular Arithmetic (without brackets)** Given a sequence of digits in N and operators $\{+, -, \times\}$ forming an arithmetic expression, output the result modulo $|N|$, respecting standard order of operations. Each sample in this task has a different length with the $=$ token denoting the end of the expression. The task vocabulary is $V = N \cup \{+, -, \times, =\}$. For accuracy computation we solely consider the output of the $=$ tokens. Random chance is $\frac{1}{|N|} = \frac{1}{|V|-4}$.
- **Modular Arithmetic (with brackets)** An extension of the above task where operators are augmented with brackets $\{(,)\}$. As above, the output respects the standard order of operations, with the brackets having priority. Accuracy and vocabulary are built as described above. Random chance is at $\frac{1}{|N|} = \frac{1}{|V|-6}$.
- **Copy Memory** Given a sequence $[x_t]_{t=1}^{M}$ of values from a dictionary D, followed by `pad` tokens and a `trigger` at x_{T-M}, output the original sequence. Accuracy is computed on the M output digits. The task vocabulary is defined as $V = D \cup \{\texttt{pad}, \texttt{trigger}\}$. Random chance is at $\frac{1}{|D|} = \frac{1}{|V|-2}$.
- **State-Tracking** A_5 Given a sequence of elements from the alternating group A_5 (even permutations), output the cumulative group products at each position. The vocabulary size is $|V| = 61$, accounting for all even permutations plus a padding token. Accuracy is averaged across all element of the sequence. Random chance is at $\frac{1}{|V|-1}$.
- **State-Tracking** S_5 An extension of the above task with both odd and even permutations. The vocabulary size is $|V| = 121$, representing *all* permutations and a padding token. As with A_5,

[2]Task descriptions have been adapted from Danieli et al. (2025)

random chance is at $\frac{1}{|V|-1}$. Results for S_5 are shown in Sec. 4.2 and Fig. 6, and hence not covered in Tab. D.1.

- **Neural Cellular Automata (NCA)** Given a sequence of tokenized grids, determine how the sequence would continue according to a transition function (determined by a 3×3 convolution, a non-linearity, and another 3×3 convolution). NCA has several hyper-parameters: the grid size, the number of colors allowed in the grid, and the number of transition functions. We follow the NCA formulation from Lee et al. (2026), but apply two simplifications to make this task apt for single layer training: we limit the number of transition functions available in the training set, and we populate the test set only with previously-seen transition functions (albeit from unseen starting states). The vocabulary size is determined by the tokenizer, which encodes 2×2 sub-blocks in a single token resulting in $|V| = 4^\gamma$ with γ being the number of colors. We average the accuracy across all positions except the tokens corresponding to the first grid. Random chance is at $\frac{1}{|V|}$.

D.4 Results & Analysis

Looking at Tab. D.1, we observe that LSTM/GRU represent a performance upper-bound for the following synthetic tasks: k-Hops $k = 1$, Cycle Navigation, Modular Arithmetic with brackets, NCA, and State-Tracking A_5. This is expected, as both LSTM and GRU are non-linear recurrent neural networks, and are expected to have higher expressivity than linear transformers. However, their poor performance on MQAR (67.4% & 75.9%), k-hops $k = 4$ (54.7% & 50.1%), and Copy Memory (11.5% & 7.3%) shows that both LSTM and GRU are brittle to train under the model architecture and training dynamics we are employing. On the other hand, Transformers (equipped with short-convolutions) are able to solve MQAR, k-hops ($k = 1$), Cycle Navigation, and Copy Memory, but struggle with k-Hops $k = 4$ (48.7%), modular arithmetic without brackets (43.8%), and State Tracking A_5 (6.0%). This is also in line with expectations, given the superior performance of softmax Transformer layers on recall tasks, but their limited ability to apply complex manipulations to the input sequence.

Considering only linear transformers, we observe that ΔBGLT has the highest performance across all tasks except Modular Arithmetic without brackets. For this task, BGLT shows +2.5pp but also higher variability (the lowest accuracy across seeds is 77.0% for BGLT, and 89.8% for ΔBGLT). ΔBGLT even beats the LSTM/GRU baselines in MQAR, k-Hops $k = 4$, Modular Arithmetic without brackets. We also notice that for Cycle Navigation –though all models solve the task– both BGLT and ΔBGLT reach a solution in less than half the wall time of the next fastest method, indicating BGLT is able to quickly converge on a solution.

We remark that only KDA and ΔBGLT achieve good performance on k-hops $k = 4$, and hypothesize that their incorporation of the Delta-Rule correction allows both models to better control erasure of information, which would be particularly relevant for k-hops $k = 4$.

D.5 NCA: Performance under different transition functions

Unlike most other synthetic tasks, NCA involves tracking multiple states (that is 2D grids in a sequence) with complex transitions functions. Moreover, NCA provides *explicit control* over the algorithmic complexity of these transitions. This tunability makes this benchmark ideal to systematically verify our core hypothesis: that block-diagonal gates unlock the capacity to track richer, more complex state dynamics compared to other linear transformers.

We characterize the behavior of *single-layer* models under two lenses: firstly, we increase the difficulty of each of the NCA hyper-parameters (grid size, number of colors, number of transition functions) and sweep across learning rates and models (mirroring the analysis in the previous section) to quantify to which degree models are able to solve NCA (cf. Tab. D.2). Secondly, we select the best-performing LSTM, Transformer, KDA and ΔBGLT models from the baseline in Tab. D.2 and quantify their behavior with respect to rollout complexity, which we compute as the compression factor of a rollout under gzip's DEFLATE implementation (following Lee et al. (2026)). The results of this rollout complexity analysis are shown in Fig. D.1 and D.2.

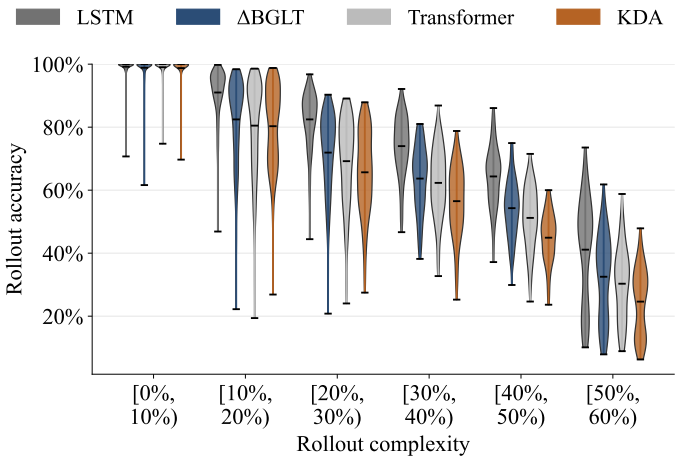


Figure D.1: **NCA acccuracy violin plots binned by rollout complexity** Models shown are best-performing LSTM, ΔBGLT, Transformer and KDA models from Tab. D.2 with 8×8, 2, 200, test sequence length $T = 512$.

Notice that, in contrast with the results in App. D.4 (where the testing sequence length matches the training sequence length) these results are computed on $4\times$ the training sequence length[3].

Table D.2 lists maximum and minimum performance across different NCA settings. These results show a mostly stable model ranking: LSTM > GRU > ΔBGLT ≈ Transformer ≈ BGLT > KDA > GLA (with the exception of increased transition functions). Looking at the different axes of difficulty, we find that increasing the grid size has a small effect on performance: specifically ΔBGLT sees a -0.6pp drop in accuracy when going from the baseline to the 12×12. In this same condition, Transformer increases accuracy by +2.6pp, we hypothesize this is due to the additional compute afforded to the Transformer by the longer sequence length. Increasing the number of colors has a much more marked effect, with an average -30.0pp drop in accuracy when using four colors, and -46.7pp average drop when using eight colors across all models. Increasing the number of transition functions has a limited effect for recurrent models (both linear transformers and nonlinear RNNs): between -0.6pp and +0.7pp change in accuracy with respect to the baseline setting at 2000. In contrast, Transformer which has -4.0pp accuracy drop, which suggests the self-attention mechanism has a lower ceiling of transition functions they can track. Under increased number of NCA transition functions the model ranking changes to LSTM > GRU > ΔBGLT > BGLT > Transformer > KDA > GLA.

For both rollout complexity analyses, we select the best-performing LSTM, Transformer, KDA, and ΔBGLT models from the NCA baseline (8×8, 2, 200, $T = 512$) and compute the gzip DEFLATE complexity of each of the 3,000 test rollouts. Figure D.1 presents the overall accuracy of each model across six rollout complexity bins. Unsurprisingly, accuracy is negatively correlated with rollout complexity[4]. In the figure, we observe that performance differences sharpen with higher rollout complexities.

Figure D.2 shows the average accuracy at every rollout step (that is, the average accuracy over the 16 tokens that compose each grid) except the first one which does not participate in accuracy computation as per NCA task definition) for rollouts with three different levels of complexity. Across all complexity levels, but especially at medium complexity, we observe that all methods increase accuracy beyond the training regime; this is not surprising, the more steps a model sees the easier it is to determine the underlying transition function. Overall the performance of the ranking of the models is LSTM > ΔBGLT ≈ Transformer > KDA. However, if we focus on the higher complexity subplot

[3]Across grid sizes we keep the number of training/test grids constant: as the grid increases in size, so does the sequence length.

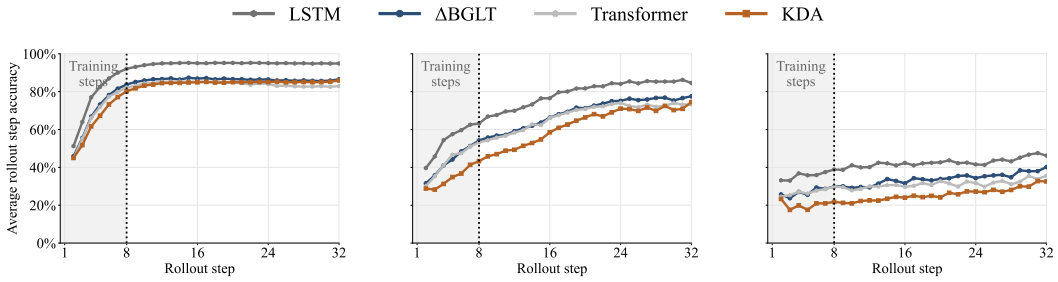
[4]We found -0.90, -0.90, -0.89, -0.92 Pearson correlations coefficients for LSTM, ΔBGLT, Transformer, and KDA respectively.

Table D.2: **NCA Difficulty Analysis** We train multiple *single-layer* models described in App. D.1 on eight NCA tasks with different hyper-parameters. The training recipe described in App. D.2. We report the highest/lowest accuracy across three seed for the highest-performing of all six swept learning rates. All models are trained on rollouts of eight grids and tested on rollouts of 32 grids.

	LSTM	GRU	TRANSF.	GLA	KDA	BGLT	ΔBGLT
			8×8 2 200				
BASELINE $\|V\| = 16$ $T = 512$ 275k	93.7% 65.7% Δ_B =+0.4pp	93.6% 78.9% Δ_B =+0.1pp	86.0% 66.2% Δ_B =+3.5pp	80.4% 65.7% Δ_B =+0.9pp	81.0% 66.7% Δ_B =+0.7pp	85.8% 65.7% Δ_B =+2.1pp	87.2% 65.7% Δ_B =+0.5pp
INCREASING GRID SIZE							
10×10 $\|V\| = 16$ $T = 800$ 275k	94.1% 35.1% Δ_B =+0.4pp	93.7% 70.2% Δ_B =+0.1pp	89.5% 69.9% Δ_B =+3.5pp	81.3% 69.7% Δ_B =+0.9pp	81.7% 68.9% Δ_B =+0.7pp	87.9% 70.0% Δ_B =+2.1pp	87.7% 69.7% Δ_B =+0.5pp
12×12 $\|V\| = 16$ $T = 1,152$ 275k	91.0% 67.3% Δ_B =-2.7pp	89.6% 34.1% Δ_B =-3.9pp	88.6% 67.3% Δ_B =+2.6pp	79.5% 67.3% Δ_B =-0.9pp	80.4% 67.3% Δ_B =-0.7pp	86.1% 67.3% Δ_B =+0.3pp	86.6% 67.2% Δ_B =-0.6pp
INCREASING NUMBER OF COLORS							
4 $\|V\| = 256$ $T = 512$ 306k	62.6% 36.4% Δ_B =-31.1pp	62.6% 47.9% Δ_B =-31.0pp	57.6% 42.3% Δ_B =-28.4pp	49.5% 36.3% Δ_B =-30.9pp	50.3% 34.5% Δ_B =-30.7pp	57.5% 36.3% Δ_B =-28.2pp	57.5% 37.8% Δ_B =-29.8pp
8 $\|V\| = 4,096$ $T = 512$ 797k	43.0% 27.3% Δ_B =-50.6pp	43.1% 27.3% Δ_B =-50.4pp	42.1% 33.2% Δ_B =-43.8pp	34.3% 27.3% Δ_B =-46.1pp	34.4% 29.5% Δ_B =-46.6pp	42.3% 27.3% Δ_B =-43.5pp	41.3% 30.0% Δ_B =-45.9pp
INCREASING NUMBER OF TRANSITION FUNCTIONS							
500 $\|V\| = 16$ $T = 512$ 275k	93.7% 66.3% Δ_B =+0.1pp	93.5% 78.6% Δ_B =-0.0pp	84.7% 63.4% Δ_B =-1.3pp	80.9% 66.3% Δ_B =+0.5pp	81.9% 68.5% Δ_B =+0.9pp	87.2% 66.3% Δ_B =+1.4pp	86.9% 66.3% Δ_B =-0.3pp
1000 $\|V\| = 16$ $T = 512$ 275k	93.0% 66.4% Δ_B =-0.7pp	92.9% 66.4% Δ_B =-0.6pp	85.6% 62.3% Δ_B =-0.4pp	80.7% 66.4% Δ_B =+0.3pp	81.5% 66.4% Δ_B =+0.5pp	85.7% 66.4% Δ_B =-0.0pp	86.8% 57.9% Δ_B =-0.5pp
2000 $\|V\| = 16$ $T = 512$ 275k	93.0% 66.4% Δ_B =-0.6pp	92.9% 66.4% Δ_B =-0.6pp	81.9% 47.6% Δ_B =-4.0pp	81.1% 66.4% Δ_B =+0.7pp	81.5% 66.4% Δ_B =+0.5pp	86.0% 66.4% Δ_B =+0.2pp	87.0% 66.4% Δ_B =-0.2pp

KEY: xx.x% Best result xx.x% Best result for linear transformers Minimum accuracy Δ_B Maximum accuracy difference with baseline Grid size Num. colors Num. transition functions $|V|$ Vocabulary size T Sequence length Num. parameters

we notice the same sharpening we reported in Fig. D.1 with ΔBGLT obtaining higher performance than Transformer.



(a) Low complexity [10%, 20%) (b) Medium complexity [30%, 40%) (c) High complexity [50%, 60%)

Figure D.2: **NCA rollout step accuracy binned by rollout complexity** Each line point is an average of 16 tokens (corresponding to a rollout step) for best LSTM, ΔBGLT, Transformer, and KDA models binned by rollout complexity, trained with 8×8, 2, 200, and evaluated at test sequence length $T = 512$.

E LANGUAGE MODEL TRAINING AND SCALING

Summary

This appendix details the training setup for our language models. Hyperparameters were tuned at the 400M parameter scale and transferred to 1.5B, 3B, and 7B with width and depth scaling rules; we check the depth rule empirically with coordinate checks. We also fully describe the Transformer++ baseline and the evaluation benchmarks.
Take-home: The Muon optimizer reaches AdamW's final loss in 15–20% fewer steps for only a $\sim 5\%$ per-step compute overhead, justifying its use for all large-scale runs. While Muon's RMS-normalized update yields lower loss, it shrinks the viable learning rate window. Furthermore, assigning the correct optimizer to the gate's low-rank projections is critical for stability: KDA diverges at high learning rates, whereas BGLT remains stable. We also find that weight decay impacts performance far less than the learning rate. Finally, we scale Muon across width with the spectral rule of Zheng et al. (2026) rather than the Kimi recipe, whose per-coordinate update grows with width.

E.1 HYPERPARAMETER SELECTION

As a preparation for large-scale training of our models, in this section we conduct a rigorous hyperparameter search at the proxy scale of 400M trainable parameters. The goal is to map out their training dynamics, stability properties and ideal optimizer configurations, and to identify the optimal set of base hyperparameters for each architecture considered. In App. E.2, we then proceed to combine these with scaling rules to define our large-scale training setup.

Setup We conduct our analysis on models at the 400M parameter scale. Architecturally, all models comprise 24 blocks, 8 attention heads, a hidden dimension of 1024, and an MLP inner dimension of $\sim$2800, slightly adjusted per-cell, to maintain parameter parity. Under these constraints, our models are nearly parameter-matched: ΔBGLT has 3.6% more non-embedding trainable parameters than the KDA baseline. We train with a sequence length of $T = 4096$ and a batch size of 256 for $\sim$36B tokens, equivalent to a $5\times$ Chinchilla token budget (Hoffmann et al., 2022). The final checkpoint of each run is evaluated on 1,000 batches (1B tokens), drawn from the training distribution with a different sampling seed than the one used during training.

Muon optimizer Following recent work (Kimi Team et al., 2025; Kimi Team, 2026), we consider the Muon optimizer (Jordan et al., 2024). Muon updates weights by orthogonalizing their momentum through a fixed number of Newton-Schulz iterations and only applies to two-dimensional maps. Because one-dimensional parameters and embeddings fall outside its scope, training requires a hybrid setup where Muon updates hidden linear maps while AdamW (Loshchilov & Hutter, 2019) handles the rest. We evaluate this hybrid configuration against training all parameters with AdamW. For our implementation, we use `Dion` (Ahn et al., 2025) with 5 Newton-Schulz iterations and a momentum of 0.95. We test both standard Muon and Moonlight's RMS-scaled update (Liu et al., 2025). Across all hybrid runs, we tune learning rates for AdamW and Muon independently.

AdamW vs. Muon Muon consistently outperforms pure AdamW (Fig. E.1): over a $5\times$ Chinchilla budget at 400M, it matches AdamW's final loss in 15–20% fewer steps, which easily offsets its $\sim 5\%$ per-step runtime overhead on identical hardware. We therefore adopt Muon for all large-scale models. Moonlight's RMS-normalized update reaches lower cross-entropy than the standard formulation at equivalent token budgets, at the cost of a narrower window of stable learning rates (Fig. E.2a–b).

Optimizer assignment Figure E.1 shows that KDA cells become increasingly unstable as the learning rate grows. This instability occurs under both AdamW and Muon with the RMS update rule of Kimi Team et al. (2025), whereas ΔBGLT remains stable. This behavior is not an implementation artifact, as we use KDA's original kernel without modification. Following prior work showing that low-rank projections benefit from AdamW optimization (Qiu et al., 2026), we therefore study the choice of optimizer for the low-rank projections of KDA and ΔBGLT. We focus on the layers parameterizing the diagonal decay (`f_proj` for KDA and `sigma_proj` for ΔBGLT, see App. C),

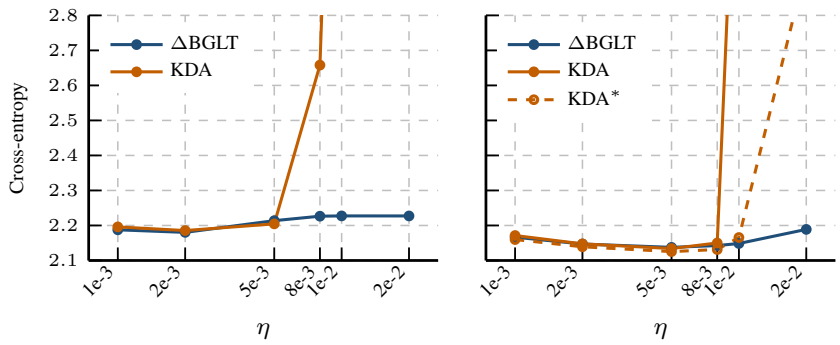


Figure E.1: Cross-entropy achieved with different learning rates for models trained with AdamW (left) and Muon (right) optimizers. Muon accelerates convergence relative to AdamW at the 400M scale for both KDA and ΔBGLT, at a 5× Chinchilla token budget, reaching a lower final cross-entropy. KDA* denotes KDA with the up-projection of the gate-parametrising layer trained with AdamW rather than Muon. We discuss KDA's stability in App. E.1.

which we find is a critical parameter affecting the basin width and training stability at this scale. We find in Fig. E.2c that assigning the down-projection to Muon and the up-projection to AdamW lowers the cross-entropy of both cells relative to Fig. E.2b while preserving the optimal learning rates, and widens the stable basin of KDA. For ΔBGLT, (c) also uses the low-rank `sigma_proj` bottleneck, whereas (a) and (b) use dense forget-gate projections.

Hyperparameters and Stability We tuned learning rates and weight decay separately for each cell. For Muon and AdamW respectively, optimal learning rates are $5 \times 10^{-3}/5 \times 10^{-3}$ for KDA and $5 \times 10^{-3}/2 \times 10^{-3}$ for ΔBGLT (Fig. E.2b–c). Weight decay had a weaker impact on loss (Fig. E.4). We set Muon and AdamW decay to $0.12/0.05$ for KDA and $0.12/0.08$ for ΔBGLT. Under these settings, training is stable for all models (Fig. E.3). However, the cells show different gradient dynamics. While KDA pre-clip gradient norms stay below 0.81 throughout training, ΔBGLT

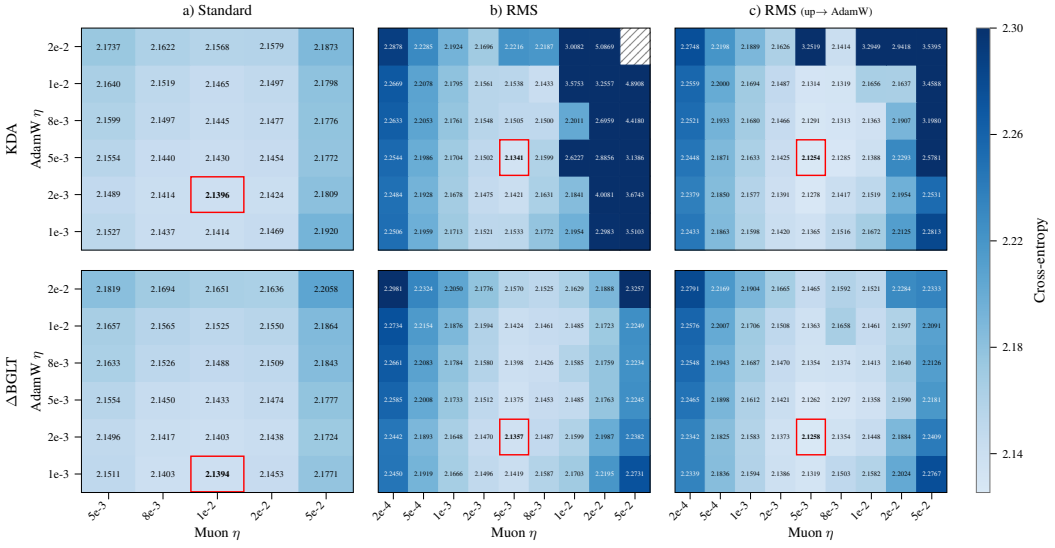


Figure E.2: Muon update rule and forget gate placement ablation. Sweeps over learning rate for parameters under AdamW and Muon for KDA (top) and ΔBGLT (bottom) for Muon-optimized training. (a) Original Muon update rule with both forget gate layers on AdamW for ΔBGLT and on Muon for KDA. (b) RMS-normalized update rule, with both forget gate layers on AdamW for ΔBGLT and on Muon for KDA. (c) RMS-normalized update placing the gate's down-projection on Muon and its up-projection on AdamW for both cells.

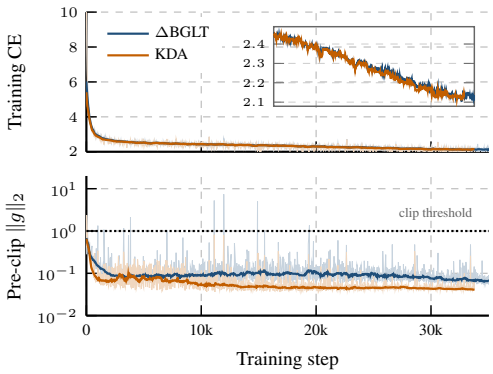


Figure E.3: **Training stability at 400M.** Cross-entropy and pre-clip gradient norm over the full training run (35B tokens for KDA, 36B for ΔBGLT) at optimal hyperparameters.

produces nine spikes above the 1.0 threshold, peaking at 7.4 near step 12k. Past step 10k, ΔBGLT also has roughly twice the median gradient norm of KDA (0.088 vs. 0.046). Gradient clipping absorbs these spikes effectively, and none cause lasting damage to cross-entropy.

E.2 Scaling

We extensively tune hyperparameters at the 400M parameter proxy scale (App. E.1) and use scaling rules to project these optimal values to the 1.5B, 3B, and 7B models. While these scaling rules are designed to keep the larger models in the feature-learning regime, empirical noise at the proxy scale and finite-width effects can cause the exact loss minimum to shift slightly. Therefore, rather than transferring the values blindly, we used the rule-projected hyperparameters as a highly accurate starting point for a localized, fine-grained search at scale. The final values reported in Tab. E.1 reflect the exact optima found through this localized tuning. This section explains the scaling rules we used to anchor this search and why we chose them.

How the models grow From 400M to 7B, the models mostly get wider. The width goes from $d = 1024$ to $d = 2048$, 3072 and 4096. The depth stays at $L = 24$ layers up to 3B, and only the 7B model is deeper, with $L = 32$. We measure growth with two multipliers relative to the 400M model:

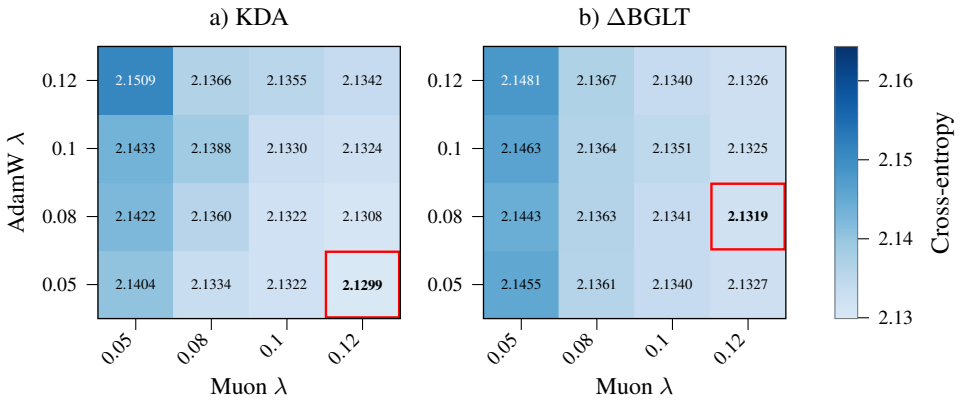


Figure E.4: Weight decay sweep for AdamW and Muon using the Muon with RMS update rule at 400M scale for a) KDA and b) ΔBGLT cells. We find that weight decay has a weaker impact on cross-entropy than learning rate.

$m_N = d/1024$ for width and $m_L = L/24$ for depth. For example, the 3B model has $m_N = 3$ and $m_L = 1$.

What the rules must achieve The standard rule is to rescale the learning rate, the weight decay and the AdamW ϵ, the residual branch outputs, and the weight initialization, so that each activation changes by about the same amount per training step at every model size. In this regime, known as the feature learning regime (Yang et al., 2022), every layer keeps learning at the same pace as in the 400M model. This ensures that the projected hyperparameters land close to the true optimum, drastically reducing the search space required for our final localized tuning.

Width scaling for Muon and AdamW Our models use two optimizers (App. E.1), and each needs its own rule. For the weights trained with Muon, we follow the spectral rule of Zheng et al. (2026): we divide the learning rate by $\sqrt{m_N}$ and multiply the weight decay by $\sqrt{m_N}$. For the weights trained with AdamW, we follow Complete(d)P (Mlodozeniec et al., 2026): we divide the learning rate and the AdamW ϵ by m_N, and multiply the weight decay by m_N. For example, the Muon learning rate of KDA and BGLT goes from 5×10^{-3} at 400M to 3.54×10^{-3}, 2.89×10^{-3} and 2.50×10^{-3} at 1.5B, 3B and 7B. For Transformer++ we also scale the initialization of the hidden weights by $1/\sqrt{m_N}$, the standard practice for Transformers (Yang et al., 2022). For KDA and BGLT we keep the initialization of the 400M models: how the scale of the initialization interacts with the gates of a linear recurrence is not established, and we leave a study of initialization scaling for linear Transformers to future work.

Deviation from Kimi-Muon update Kimi K2 and K3 (Kimi Team et al., 2025; Kimi Team, 2026) scale Muon differently (Liu et al., 2025). Muon orthogonalizes its momentum (Jordan et al., 2024), so the only way it can adapt to width is through a single scalar that multiplies the update. Kimi-Muon sets this number proportional to $\sqrt{\max(d_{\text{in}}, d_{\text{out}})}$. As a result, the update of each coordinate in the hidden layers grows like $\sqrt{d}$ instead of staying constant (Zheng et al., 2026). Hyperparameters tuned at 400M then become too aggressive as the model gets wider, and the model drifts out of the feature learning regime. The problem comes from this scaling factor, not from Muon itself: the original Muon of Jordan et al. (2024) does not have it. The spectral rule restores a constant per-coordinate update (Zheng et al., 2026).

Depth scaling Only the 7B model is deeper than the 400M model, so this rule matters only there. Following CompleteP (Dey et al., 2026) and Zheng et al. (2026), we multiply the output of each residual branch by $1/m_L$, and do the same for the AdamW ϵ of the parameters inside the blocks. Learning rates and weight decays stay unchanged. At 7B we have $m_L = 32/24$, so each residual branch is scaled by 0.75. We verify this rule with coordinate checks (Yang et al., 2022): we take models of increasing depth at width $d = 1024$, where the width rules are inactive, train them for 20 Muon steps on random tokens, and measure how far each activation moves from its initial value. If the rule works, this change is the same at every depth, and the log-log plots are flat. Without depth scaling the residual stream grows with depth (Fig. E.5(a)). With the rule, the branch outputs are flat and the residual stream tilts only slightly (Fig. E.5(b)).

Batch size The batch also grows, from 256 sequences (1M tokens) at 400M to 512 sequences (2M tokens) for all larger models. Complete(d)P (Mlodozeniec et al., 2026) gives a correction for AdamW, but we know of no equivalent for Muon. We apply no batch-size correction to either optimizer, so our rules cover width and depth only. When we tried the correction on both optimizers for Transformer++ at 1.5B, it raised the training loss by 0.09 nats (App. E.3), which suggests that leaving the batch size uncorrected is a safe choice.

Optimizer assignment at scale At 400M, the choice of optimizer for the small low-rank projections inside the gates had a large effect on stability (App. E.1). Moving the up-projection to AdamW widened the range of stable learning rates for both KDA and ΔBGLT. At scale we go one step further and train all low-rank projections, as well as the output gates, with AdamW (Tab. E.2), as recommended by Qiu et al. (2026). We did not test this exact assignment at 400M. We chose it to lower the risk of the instabilities we observed for KDA at high learning rates.

Table E.1: **Training hyperparameters at 1.5B / 3B / 7B.** Cells with three values list them per scale. Non-embedding parameters exclude the input embeddings but include the output embeddings (Porian et al., 2025). The Chinchilla multiple is the token budget of the cosine schedule.

Model	Transformer++	KDA	BGLT
d_{model}	2048 / 3072 / 4096	2048 / 3072 / 4096	2048 / 3072 / 4096
Layers L	24 / 24 / 32	24 / 24 / 32	24 / 24 / 32
Heads	16 / 24 / 32	16 / 24 / 32	16 / 24 / 32
Head dim	128	128	128
MLP inner dim	6552 / 8584 / 11672	6368 / 8400 / 11488	6176 / 8200 / 11280
Non-emb. params (M)	1434 / 2903 / 6868	1434 / 2903 / 6869	1435 / 2904 / 6868
Training			
Tokens (B)	143 / 290 / 275	143 / 290 / 275	143 / 290 / 275
Chinchilla multiple	5× / 5× / 2×	5× / 5× / 2×	5× / 5× / 2×
Batch size (tokens)	2M	2M	2M
Steps (10^3)	68.4 / 138.4 / 131.0	68.4 / 138.4 / 131.0	68.4 / 138.4 / 131.0
Warmup steps	684 / 1384 / 3275	683 / 1384 / 3275	684 / 1384 / 3274
LR schedule	cosine	cosine	cosine
Final / Max-LR (10^{-4})	1.25 / 1.25 / 1.25	5 / 2 / 2	5 / 5 / 5
Gradient clipping	1.0	1.0	1.0
z-loss	10^{-4}	10^{-4}	10^{-4}
Initialization			
Hidden std (10^{-3})	14.1 / 11.5 / 10.0	20 / 20 / 20	20 / 20 / 20
Residual branch scale	1 / 1 / 0.75	1 / 1 / 0.75	1 / 1 / 0.75
Muon			
Max-LR (10^{-3})	3.54 / 2.89 / 2.50	3.54 / 2.89 / 2.50	3.54 / 2.89 / 2.50
WD	0.14 / 0.17 / 0.20	0.17 / 0.21 / 0.24	0.17 / 0.21 / 0.24
Momentum	0.95	0.95	0.95
AdamW			
Max-LR, width-scaled (10^{-3})	4.00 / 2.67 / 2.00	1.00 / 1.67 / 1.25	1.00 / 0.67 / 0.50
Max-LR, unscaled (10^{-3})	8 / 8 / 8	2 / 5 / 5	2 / 2 / 2
WD, width-scaled	0.20 / 0.30 / 0.40	0.10 / 0.15 / 0.20	0.16 / 0.24 / 0.32
WD, unscaled	–	0.05 / 0.05 / 0.05	0.08 / 0.08 / 0.08
β_1, β_2	0.9, 0.95	0.9, 0.95	0.9, 0.95
ϵ, blocks (10^{-9})	5.0 / 3.3 / 1.9	5.0 / 3.3 / 1.9	5.0 / 3.3 / 1.9
ϵ, QK norms (10^{-9})	10 / 10 / 7.5	–	–
ϵ, input embedding (10^{-9})	5.0 / 3.3 / 2.5	5.0 / 3.3 / 2.5	5.0 / 3.3 / 2.5
ϵ, final norm (10^{-9})	10 / 10 / 10	5.0 / 3.3 / 2.5	5.0 / 3.3 / 2.5
ϵ, readout (10^{-9})	10 / 10 / 10	10 / 10 / 10	10 / 10 / 10

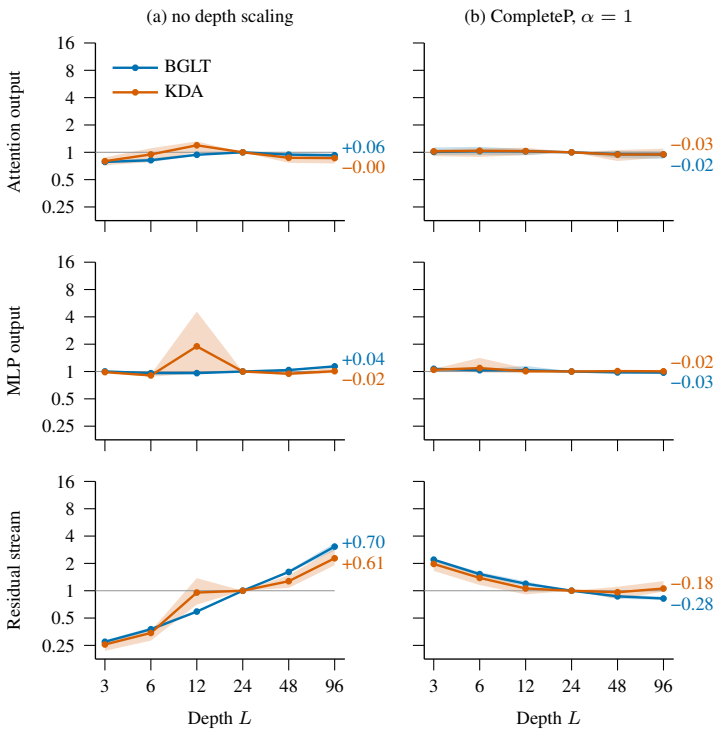


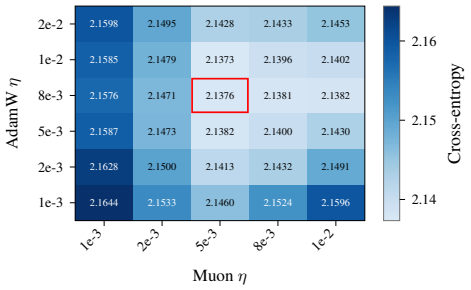
Figure E.5: **Coordinate check (Yang et al., 2022) for depth scaling.** RMS change of each activation from its initial value after 20 Muon updates at the peak learning rate on random tokens, divided per seed by its value at the base depth $L = 24$; lines are the geometric mean over 5 seeds and bands the range across seeds. Width is fixed at $d = 1024$. Columns: (a) no depth scaling, (b) residual branches and the AdamW ϵ of in-block parameters scaled by $24/L$ following CompleteP (Dey et al., 2026) and Zheng et al. (2026), with learning rates and weight decays unchanged.

E.3 TRANSFORMER++ BASELINE

Architecture The Transformer++ baseline (Vaswani et al., 2017) uses the same backbone as the recurrent models (Sec. 4.1), that is, pre-norm blocks with a SwiGLU MLP (Shazeer, 2020), with causal softmax attention as the sequence mixer, using rotary position embeddings (Su et al., 2024) at base frequency 5×10^5, RMSNorm on queries and keys (Henry et al., 2020), and heads of dimension 128. Parameters are matched to the recurrent models by widening the MLP (Tab. E.1). At 400M, Transformer++ has $d_{\text{model}} = 1024$, 24 layers, 8 heads and an MLP inner dimension of 2992, for 354M non-embedding parameters, and trains with a batch of 256 sequences.

Hyperparameter selection We follow the protocol in App. E.1. Muon updates all attention and MLP weight matrices, while AdamW updates the input embedding, the readout and the normalization gains. Unlike the recurrent models, Transformer++ has no low-rank gate projections. Figure E.6 reports the learning-rate (top) and weight-decay (bottom) sweeps.

We choose a Muon learning rate of 5×10^{-3} and an AdamW learning rate of 8×10^{-3} because each minimizes the loss when averaged over the other optimizer's learning rates. This makes the choice robust to hyperparameter sensitivity in a region where five configurations fall within 10^{-3} nats of the minimum (well below seed-to-seed spread), while remaining within 2×10^{-4} nats of the single best cell. For weight decay, measured at the adopted learning rates, values from 0.08 to 0.15 lie within 0.0016 nats of each other, below the seed-to-seed spread, so we adopt 0.10, 9×10^{-4} nats from the best value.



Weight decay	0.08	0.10	0.12	0.15
Held-out CE	2.1376	2.1369	2.1360	2.1363

Figure E.6: **Transformer++ hyperparameter sweeps at 400M**, using Muon with the RMS update rule at 5× Chinchilla. Top: learning-rate sweep at weight decay 0.08. Cells show the cross-entropy of the final checkpoint on 1B held-out tokens from the training distribution, and the red box marks, for each optimizer, the learning rate minimizing the loss averaged over the other optimizer's learning rates. Bottom: weight-decay sweep at the adopted Muon/AdamW learning rates $5 \times 10^{-3}/8 \times 10^{-3}$. Values from 0.08 to 0.15 lie within 0.0016 nats of each other, and we adopt 0.10.

Scaling We apply the width and depth rules of App. E.2, including the scaling of the hidden-weight initialization by $1/\sqrt{m_N}$. We apply no batch-size correction: at 1.5B, the SDE correction to batch size 512 on both optimizers raised the training loss by 0.09 nats over the uncorrected run at matched steps. With $m_N = d_{\text{model}}/1024$, the Muon learning rate and weight decay scale as $5 \times 10^{-3}\, m_N^{-1/2}$ and $0.10\, m_N^{1/2}$. Parameters not on Muon use AdamW with a learning rate of 8×10^{-3} and no weight decay, except the readout, whose learning rate and weight decay scale as $8 \times 10^{-3}\, m_N^{-1}$ and $0.10\, m_N$. Table E.1 lists the model sizes and training budgets.

E.4 EVALUATION

We use `lm-eval-harness` (Gao et al., 2024) to evaluate all of our language models. We consider general language modeling benchmarks: LAMBADA (Paperno et al., 2016), WikiText-103 (Merity et al., 2016), PIQA (Bisk et al., 2019), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2019), ARC-easy, ARC-challenge (Clark et al., 2018). We also evaluate on recall-intensive tasks such as multi-hop birth-year retrieval (Allen-Zhu, 2025), SWDE (Lockard et al., 2019; Arora et al., 2023; 2025), FDA (Arora et al., 2023; 2025), SQuAD v2 (Rajpurkar et al., 2018), TriviaQA (Joshi et al., 2017) and cloze-formatted variants of some of these tasks (Arora et al., 2024b).

Table E.2: Assignment of each trainable parameter to the Muon (group 1), AdamW with weight decay (group 2), AdamW without weight decay (group 3) optimizers for Muon-optimized training. $\star$ represents low-rank projections for which the assignment is discussed in App. E.1.

parameter	shape @400M	cell	*Group 1* Muon	*Group 2* AdamW w/ wd	*Group 3* AdamW w/o wd	% of total (@400M)
input embedding						
`in_embedding.weight`	$(32k, 1024)$	all			×	8.37
output unembeddings						
`out_embedding.weight`	$(32k, 1024)$	all		×		8.37
`out_embedding.bias`	$(32k)$	all			×	0.01
`out_embedding.norm`	(1024)	all			×	0.00
attn QKV + out proj						
`attn.q_proj.weight`	$(1024, 1024)$	all	×			6.43
`attn.k_proj.weight`	$(1024, 1024)$	all	×			6.43
`attn.v_proj.weight`	$(1024, 1024)$	all	×			6.43
`attn.o_proj.weight`	$(1024, 1024)$	all	×			6.43
short convs						
`attn.q_conv1d.weight`	$(1024, 1, 4)$	all		×		0.03
`attn.k_conv1d.weight`	$(1024, 1, 4)$	all		×		0.03
`attn.v_conv1d.weight`	$(1024, 1, 4)$	all		×		0.03
attn norms						
`attn.o_norm.weight`	(128)	all			×	0.00
block norms						
`norm1.weight`	(1024)	all			×	0.01
`norm2.weight`	(1024)	all			×	0.01
MLP						
`mlp.fc1.weight`	$(*, 1024)$	all	×			17.82
`mlp.fc2.weight`	$(1024, *)$	all	×			17.82
`mlp.gate.weight`	$(*, 1024)$	all	×			17.82
gate / transition						
`attn.b_proj.weight`	$(8, 1024)$	KDA		×		0.05
`attn.dt_bias`	$(*)$	KDA, BGLT			×	0.00
`attn.A_log`	(8)	KDA, BGLT			×	0.00
`attn.beta_proj.weight`	$(8, 1024)$	BGLT		×		0.05
`attn.f_proj.weight`	$(8, 1024)$	BGLT		×		0.05
`attn.ab_proj.1.bias`	(1536)	BGLT			×	0.01
`attn.sigma_dt_bias`	$(8, 8, 3, 2)$	BGLT			×	0.00
`attn.sigma_A_log`	$(8, 8, 2)$	BGLT			×	0.00
`attn.alpha_u`	$(8, 8, 2)$	BGLT			×	0.00
`attn.alpha_v`	$(8, 8, 2)$	BGLT			×	0.00
gate / transition: low-rank projections*						
`attn.f_proj.0.weight`	$(128, 1024)$	KDA		×		0.81
`attn.f_proj.1.weight`	$(1024, 128)$	KDA		×		0.81
`attn.ab_proj.1.weight`	$(1536, 256)$	BGLT		×		2.36
`attn.ab_proj.0.weight`	$(256, 1024)$	BGLT		×		1.57
`attn.sigma_proj.0.weight`	$(128, 1024)$	BGLT		×		0.79
`attn.sigma_proj.1.weight`	$(384, 128)$	BGLT		×		0.30
output gate*						
`attn.g_proj.0.weight`	$(128, 1024)$	all		×		0.80
`attn.g_proj.1.weight`	$(1024, 128)$	all		×		0.80
`attn.g_proj.1.bias`	(1024)	all			×	0.01
% of total parameters @400M			79.20	12.39	8.41	

F CHUNK-WISE PARALLEL FORMULATION OF BGLT

Summary

This section details the step-by-step derivation of the chunk-wise parallel formulation for BGLT, providing the mathematical foundation for the kernels in App. G. Its blueprint is similar to other Linear Transformers, and consists of four steps: (1) building the chunk-local attention and key-Gram matrices; (2) computing and applying the Delta-rule correction by solving a unit lower-triangular system; (3) updating the state, sequentially across chunks; (4) computing the outputs. The additional complications introduced by BGLT pertain mainly to step (1). The backward pass follows the same logical steps, in reverse.

A Linear RNN can be applied to an input sequence either in parallel or sequentially, trading off number of sequential step against the complexity of operations within each step, as illustrated in Fig. F.1. The chunk-wise parallel formulation allows us to freely interpolate between these two extrema. In this section we describe in detail the chunk-wise parallel formulation for BGLT, providing mathematical derivations for the relevant formulas implemented by the kernels in App. G. We refer to App. B for a reference list of the notation we rely on, together with several properties and lemmas we will use in our derivations. In App. F.1 we outline the operations involved in the forward pass application of BGLT, while in App. F.2 we detail the corresponding backward pass.

F.1 FORWARD PASS

The target recurrence considered in the most general form of BGLT, including the Delta Rule correction, is given by:

$$\boldsymbol{S}_t = (\boldsymbol{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top)\boldsymbol{G}_t \boldsymbol{S}_{t-1} + \beta_t \boldsymbol{k}_t \boldsymbol{v}_t^\top, \qquad \boldsymbol{o}_t = \boldsymbol{S}_t^\top \boldsymbol{q}_t, \tag{41}$$

where $(\boldsymbol{I} - \beta_t \boldsymbol{k}_t \boldsymbol{k}_t^\top)$ is a Householder factor, responsible for implementing the Delta Rule correction, while $\boldsymbol{G}_t$ is the novel gate proposed in BGLT, and presents a block-diagonal structure, $\boldsymbol{G}_t := \alpha_t \bigoplus_{b=1}^{n_B} \boldsymbol{G}_t^{(b)}$, with each block further factorized as a Kronecker product $\boldsymbol{G}_t^{(b)} := \boldsymbol{G}_t^{(b,A)} \otimes \boldsymbol{G}_t^{(b,B)}$. As in Gated DeltaNet (Yang et al., 2025a), note that we can incorporate the Delta Rule contribution into the values, and rewrite the recurrence in the canonical form:

$$\boldsymbol{S}_t = \boldsymbol{G}_t \boldsymbol{S}_{t-1} + \boldsymbol{k}_t \underbrace{\beta_t \left(\boldsymbol{v}_t - (\boldsymbol{G}_t \boldsymbol{S}_{t-1})^\top \boldsymbol{k}_t\right)^\top}_{=:\boldsymbol{\delta}_t^\top} = \boldsymbol{G}_t \boldsymbol{S}_{t-1} + \boldsymbol{k}_t \boldsymbol{\delta}_t^\top = \sum_{s=1}^{t} \boldsymbol{G}_{s+1\to t} \boldsymbol{k}_s \boldsymbol{\delta}_s^\top, \tag{42}$$

where we assumed $\boldsymbol{S}_0 \equiv \boldsymbol{0}$ for simplicity, although this is not a strict requirement. The quantities

$$\boldsymbol{\delta}_t := \beta_t \left(\boldsymbol{v}_t - (\boldsymbol{G}_t \boldsymbol{S}_{t-1})^\top \boldsymbol{k}_t\right) \in \mathbb{R}^{d_v} \tag{43}$$

represent the Delta-Rule-corrected updates. We point out that, if the Delta Rule correction is not included in the RNN definition, all following derivations still hold: it suffices to collapse $\boldsymbol{\delta}_t \equiv \boldsymbol{v}_t$ instead.

To recover the chunk-wise formulation, we first partition the input sequence into T/C *chunks* of size C. Each chunk $n = 1, \ldots, T/C$ starts at index $n_S := (n-1)C + 1$ and ends at index $n_E := nC$,

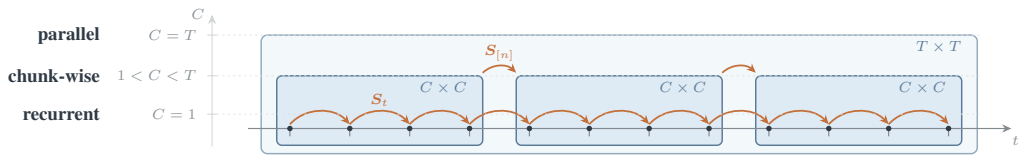


Figure F.1: **Linear RNN application modes.** Blue regions are computed in parallel, orange arrows represent sequential dependencies. The recurrent, chunk-wise, and parallel applications of a Linear RNN trade-off sequentiality with complexity: at $C = 1$ every step covers a single token and sequence mixing is purely sequential, while at $C = T$ the whole sequence is covered at once, at quadratic cost in T. Intermediate values of C require T/C sequential steps and perform parallel operations at quadratic cost in C.

extrema included. In the following, with some abuse of notation, we denote with t both the chunk-local index and its corresponding global index $n_S + t - 1$, and similarly for s. Our goal is to express the recurrence output as a function of only chunk-local variables, and of the state at the end of the previous chunk, denoted as $\boldsymbol{S}_{[n-1]} \coloneqq \boldsymbol{S}_{n_S-1} = \boldsymbol{S}_{(n-1)C} \in \mathbb{R}^{d_k \times d_v}$. To this end, we unroll the recurrence (42) across the chunk, to obtain:

$$\begin{aligned}\boldsymbol{S}_t &= \sum_{s=1}^{t} \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top = \boldsymbol{G}_{n_S\to t}\sum_{s=1}^{n_S-1} \boldsymbol{G}_{s+1\to n_S-1}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top + \sum_{s=n_S}^{t} \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top \\ &= \boldsymbol{G}_{n_S\to t}\boldsymbol{S}_{[n-1]} + \sum_{s=n_S}^{t} \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top, \qquad \text{for} \quad t = n_S, \dots, n_E.\end{aligned} \tag{44}$$

The state $\boldsymbol{S}_t$ in (44) still depends on the Delta Rule corrections $\boldsymbol{\delta}_s$. These too can be expressed as a function of only chunk-local variables, substituting (44) into their definition in (43):

$$\begin{aligned}\boldsymbol{\delta}_t &= \beta_t(\boldsymbol{v}_t - (\boldsymbol{G}_t\boldsymbol{S}_{t-1})^\top\boldsymbol{k}_t) = \beta_t\boldsymbol{v}_t - \beta_t\left(\boldsymbol{G}_{n_S\to t}\boldsymbol{S}_{[n-1]} + \sum_{s=n_S}^{t-1} \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top\right)^\top\boldsymbol{k}_t \\ &= \beta_t(\boldsymbol{v}_t - \boldsymbol{S}_{[n-1]}^\top\boldsymbol{G}_{n_S\to t}^\top\boldsymbol{k}_t) - \sum_{s=n_S}^{t-1} \beta_t((\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s)^\top\boldsymbol{k}_t)\boldsymbol{\delta}_s, \qquad \text{for} \quad t = n_S, \dots, n_E.\end{aligned} \tag{45}$$

From this, we can recover the output read-out for any given index in the n-th chunk:

$$\begin{aligned}\boldsymbol{o}_t &= \boldsymbol{S}_t^\top\boldsymbol{q}_t = \left(\boldsymbol{G}_{n_S\to t}\boldsymbol{S}_{[n-1]} + \sum_{s=n_S}^{t} \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\boldsymbol{\delta}_s^\top\right)^\top\boldsymbol{q}_t \\ &= \boldsymbol{S}_{[n-1]}^\top\boldsymbol{G}_{n_S\to t}^\top\boldsymbol{q}_t + \sum_{s=n_S}^{t} ((\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s)^\top\boldsymbol{q}_t)\boldsymbol{\delta}_s, \qquad \text{for} \quad t = n_S, \dots, n_E.\end{aligned} \tag{46}$$

Having expressed all required quantities as a function of the previous-chunk state and chunk-local quantities, we are now left with reformulating their computation as tensor operations. We do so in the following paragraphs, mirroring the order of operations performed in the actual forward pass for BGLT. This is also reported in the schematics in Fig. F.2.

Attention and Gram matrix assembly The first step in the chunk-wise application consists in defining and assembling the actual tensors appearing in eqs. (44) to (46). Here and in the following, we denote them with a $(*)_{[n]}$ subscript to highlight their chunk-local nature, and, with some abuse of notation, we use $t, s \in [1, \dots, C]$ as chunk-local indices to identify their entries. We can group these tensors according to their shape. Two of them have dimensions $C \times C$: these are the Attention and key-Gram matrices appearing in eqs. (45) and (46), and respectively defined as

$$\begin{aligned}[\boldsymbol{A}_{[n]}^{qk}]_{t,s} &\coloneqq \boldsymbol{q}_t^\top\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s, \qquad \text{and} \\ [\boldsymbol{A}_{[n]}^{kk}]_{t,s} &\coloneqq \boldsymbol{k}_t^\top\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s.\end{aligned} \tag{47}$$

Note that the two matrices share the same structure, with $\boldsymbol{k}_t$ playing the role of the query in $\boldsymbol{A}_{[n]}^{kk}$. The efficient assembly of Attention and

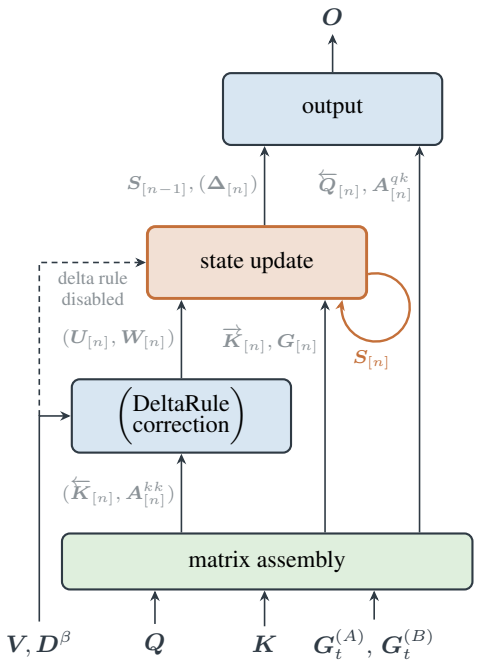


Figure F.2: Forward pass through the BGLT cell. Operations and quantities in (brackets) are optional, computed if the Delta Rule correction is included.

Gram matrices when block-diagonal gates are involved is one of the core contribution of this work, and the kernels responsible for this are described in detail in App. G.1. The remaining relevant tensors are of shape $C \times d_k$, and collect *pushed* queries and keys, i.e., quantities to which specific sequences of gates has been applied. These are defined as:

$$\begin{aligned} [\overleftarrow{\boldsymbol{Q}}_{[n]}]_{t,-}^\top &:= \overleftarrow{\boldsymbol{q}}_t := \boldsymbol{G}_{n_S \to t}^\top \boldsymbol{q}_t, & [\overleftarrow{\boldsymbol{K}}_{[n]}]_{t,-}^\top &:= \overleftarrow{\boldsymbol{k}}_t := \boldsymbol{G}_{n_S \to t}^\top \boldsymbol{k}_t, \\ [\overrightarrow{\boldsymbol{K}}_{[n]}]_{s,-}^\top &:= \overrightarrow{\boldsymbol{k}}_s := \boldsymbol{G}_{s+1 \to n_E} \boldsymbol{k}_s. \end{aligned} \tag{48}$$

The first two contain pushed queries and keys to the start of the chunk, while the last is composed of keys pushed to its end.

Delta Rule correction As a second step, we must recover the Delta Rule corrections $\boldsymbol{\delta}_t$ appearing in (45). To illustrate this, let us first introduce the tensor quantities

$$[\boldsymbol{\Delta}_{[n]}]_{t,-}^\top := \boldsymbol{\delta}_t, \qquad [\boldsymbol{V}_{[n]}]_{t,-}^\top := \boldsymbol{v}_t, \qquad \text{and} \qquad \boldsymbol{D}_{[n]}^\beta := \mathrm{Diag}[\beta_{n_S}, \dots, \beta_{n_E}] \in \mathbb{R}^{C \times C}, \tag{49}$$

with $\boldsymbol{\Delta}_{[n]}, \boldsymbol{V}_{[n]} \in \mathbb{R}^{C \times d_v}$. This allows us to rewrite (45) in tensor form, as such:

$$\boldsymbol{\Delta}_{[n]} = \tilde{\boldsymbol{V}}_{[n]} - \left((\boldsymbol{D}_{[n]}^\beta \boldsymbol{A}_{[n]}^{kk}) \odot \boldsymbol{M}_{-1} \right) \boldsymbol{\Delta}_{[n]}, \quad \text{with } \tilde{\boldsymbol{V}}_{[n]} := \boldsymbol{D}_{[n]}^\beta \left(\boldsymbol{V}_{[n]} - \overleftarrow{\boldsymbol{K}}_{[n]} \boldsymbol{S}_{[n-1]} \right) \in \mathbb{R}^{C \times d_v}, \tag{50}$$

where the previously-computed Gram matrix $\boldsymbol{A}_{[n]}^{kk}$ appears, element-wise multiplied by the strictly causal mask $[\boldsymbol{M}_{-1}]_{t,s} := \delta_{t>s}$. Each $\boldsymbol{\delta}_t$ correction thus depends only on those preceding it, meaning corrections over the whole chunk are recovered by solving the following *unit lower-triangular* system

$$\boldsymbol{T}_{[n]} \boldsymbol{\Delta}_{[n]} = \tilde{\boldsymbol{V}}_{[n]}, \qquad \text{with} \qquad \boldsymbol{T}_{[n]} := \boldsymbol{I} + (\boldsymbol{D}_{[n]}^\beta \boldsymbol{A}_{[n]}^{kk}) \odot \boldsymbol{M}_{-1}. \tag{51}$$

While this can be trivially solved via forward-substitution in C steps, doing so would forgo sequence-parallelism. Instead, as described in App. G.2, we compute the inverse explicitly via preconditioned Newton-Schulz, which requires only matrix products and thus maintains parallelism. Moreover, by separating the terms in (50) that do not depend on the incoming state, we get

$$\boldsymbol{\Delta}_{[n]} = \underbrace{\boldsymbol{T}_{[n]}^{-1} \boldsymbol{D}_{[n]}^\beta \boldsymbol{V}_{[n]}}_{=:\boldsymbol{U}_{[n]}} - \underbrace{\boldsymbol{T}_{[n]}^{-1} \boldsymbol{D}_{[n]}^\beta \overleftarrow{\boldsymbol{K}}_{[n]}}_{=:\boldsymbol{W}_{[n]}} \boldsymbol{S}_{[n-1]} = \boldsymbol{U}_{[n]} - \boldsymbol{W}_{[n]} \boldsymbol{S}_{[n-1]}. \tag{52}$$

Pre-computing the chunk-local $\boldsymbol{U}_{[n]}, \boldsymbol{W}_{[n]}$ terms is useful in reducing the amount of operations required in the sequential path, as described below.

State update The state update is the only sequential operation in the chunk-wise formulation: this is responsible for propagating the state across chunks, recovering its chunk-end value. Denoting as $\boldsymbol{G}_{[n]} := \boldsymbol{G}_{n_S \to n_E}$ the whole-chunk gate products, we get from evaluating (44) at the chunk end and substituting (52)

$$\boldsymbol{S}_{[n]} = \boldsymbol{G}_{[n]} \boldsymbol{S}_{[n-1]} + \overrightarrow{\boldsymbol{K}}_{[n]}^\top \boldsymbol{\Delta}_{[n]} = \boldsymbol{G}_{[n]} \boldsymbol{S}_{[n-1]} + \overrightarrow{\boldsymbol{K}}_{[n]}^\top \left(\boldsymbol{U}_{[n]} - \boldsymbol{W}_{[n]} \boldsymbol{S}_{[n-1]} \right). \tag{53}$$

This is where having pre-computed $\boldsymbol{U}_{[n]}$ and $\boldsymbol{W}_{[n]}$ pays off: no triangular inverse $\boldsymbol{T}_{[n]}^{-1}$ appears, reducing the slower sequential path to two matrix products and one application of the block-diagonal $\boldsymbol{G}_{[n]}$.

Output The final step is again parallel across chunks, and computes the final output of the linear RNN. Stacking $\boldsymbol{o}_t$ as the rows of $\boldsymbol{O}_{[n]} \in \mathbb{R}^{C \times d_v}$, we can rewrite (46) as

$$\boldsymbol{O}_{[n]} = \overleftarrow{\boldsymbol{Q}}_{[n]} \boldsymbol{S}_{[n-1]} + (\boldsymbol{A}_{[n]}^{qk} \odot \boldsymbol{M}) \boldsymbol{\Delta}_{[n]}, \tag{54}$$

where the previously-computed linear Attention matrix $\boldsymbol{A}_{[n]}^{qk}$ appears, element-wise multiplied by the causal mask $[\boldsymbol{M}]_{t,s} := \delta_{t \geq s}$. The two terms above are respectively the inter- and intra-chunk output contributions.

This completes the mathematical description of the chunk-wise parallel application of the forward pass through BGLT. In the following section, we illustrate instead the backward pass.

F.2 BACKWARD PASS

In this section we provide the mathematical formulation for the chunk-wise backward pass through the cell application. We compactly denote as $\delta \boldsymbol{X}$ the gradient of the loss function $\mathcal{L}$ with respect to the parameter $\boldsymbol{X}$, and refer to App. B for useful properties we rely on in the derivations to follow.

Output. Walking backwards through App. F.1, the first operation we encounter is the output computation (54). The gradient flow from $\delta \boldsymbol{O}_{[n]}$ produces

$$\begin{aligned} \delta^{\text{out}} \boldsymbol{S}_{[n-1]} &= \overleftarrow{\boldsymbol{Q}}_{[n]}^{\top} \delta \boldsymbol{O}_{[n]} \\ \delta^{\text{out}} \boldsymbol{\Delta}_{[n]} &= (\boldsymbol{A}_{[n]}^{qk} \odot \boldsymbol{M})^{\top} \delta \boldsymbol{O}_{[n]} \\ \delta \boldsymbol{A}_{[n]}^{qk} &= \left(\delta \boldsymbol{O}_{[n]} \boldsymbol{\Delta}_{[n]}^{\top} \right) \odot \boldsymbol{M} \\ \delta \overleftarrow{\boldsymbol{Q}}_{[n]} &= \delta \boldsymbol{O}_{[n]} \boldsymbol{S}_{[n-1]}^{\top}. \end{aligned} \tag{55}$$

The last quantity is itself a derived one: following the definition of $\overleftarrow{\boldsymbol{Q}}_{[n]}$ in (48), we recover the first contribution to the gradient for the queries,

$$[\delta^{\text{out}} \boldsymbol{Q}_{[n]}]_{t,-} = [\delta \overleftarrow{\boldsymbol{Q}}_{[n]}]_{t,-} \boldsymbol{G}_{n_S \to n_S+t-1}^{\top} \qquad \text{for } t = 1, \dots, C, \tag{56}$$

as well as the first contribution to the gradient for the gates. To compute the latter, we directly leverage the block-diagonal and Kronecker structures we impose on each gate, in order to simplify calculations. In particular, denoting as $(*)\big|_{(b)}$ the sub-vector of $(*)$ with components i corresponding to the b-th block, $i = (b-1)m + 1, \dots, mb$, we have:

$$\begin{aligned} \overleftarrow{\boldsymbol{q}}_t \big|_{(b)} &= (\boldsymbol{G}_{n_S \to n_S+t-1}^{(b)})^{\top} \boldsymbol{q}_{n_S+t-1} \big|_{(b)} = \left(\boldsymbol{G}_{n_S \to n_S+t-1}^{(b,A)} \otimes \boldsymbol{G}_{n_S \to n_S+t-1}^{(b,B)} \right)^{\top} \boldsymbol{q}_{n_S+t-1} \big|_{(b)} \\ &= \text{vec} \left((\boldsymbol{G}_{n_S \to n_S+t-1}^{(b,A)})^{\top} \text{mat} \left(\boldsymbol{q}_{n_S+t-1} \big|_{(b)} \right) \boldsymbol{G}_{n_S \to n_S+t-1}^{(b,B)} \right) \qquad \text{for } t = 1, \dots, C, \end{aligned} \tag{57}$$

where $\text{mat}\,(*)$ is the reshaping of vector $(*) \in \mathbb{R}^{\hat{m}^2}$ into a matrix of size $\mathbb{R}^{\hat{m} \times \hat{m}}$, row-wise, while $\text{vec}\,(*)$ is its inverse operation. Given this, we can recover the output contribution to the gradient with respect to any single gate factor within the chunk $\delta^{\text{out}} \boldsymbol{G}_i^{(b,j)}$, $i = n_S, \dots, n_E$ employing Lem. 1, to get:

$$\begin{aligned} \delta^{\text{out}} \boldsymbol{G}_i^{(b,A)} &= \sum_{t=i}^{n_E} (\boldsymbol{G}_{i+1 \to t}^{(b,A)})^{\top} \text{mat} \left(\boldsymbol{q}_{n_S+t-1} \big|_{(b)} \right) \boldsymbol{G}_{n_S \to t}^{(b,B)} \text{mat} \left(\delta^{\text{out}} \overleftarrow{\boldsymbol{q}}_t \big|_{(b)} \right)^{\top} (\boldsymbol{G}_{n_S \to i-1}^{(b,A)})^{\top}, \\ \delta^{\text{out}} \boldsymbol{G}_i^{(b,B)} &= \sum_{t=i}^{n_E} (\boldsymbol{G}_{i+1 \to t}^{(b,B)})^{\top} \text{mat} \left(\boldsymbol{q}_{n_S+t-1} \big|_{(b)} \right)^{\top} \boldsymbol{G}_{n_S \to t}^{(b,A)} \text{mat} \left(\delta^{\text{out}} \overleftarrow{\boldsymbol{q}}_t \big|_{(b)} \right) (\boldsymbol{G}_{n_S \to i-1}^{(b,B)})^{\top}. \end{aligned} \tag{58}$$

State update. The output gradients produce the seeds for the backward pass through the sequential state update (53). Here gradients flow across chunk boundaries, and must hence be computed sequentially across chunks, in the reverse direction. The most pressing concern at this stage is to correctly propagate gradients from $\delta \boldsymbol{S}_{[n]}$ to $\delta \boldsymbol{S}_{[n-1]}$, but note that the flow follows two more paths on top of the read-out contribution from (55): through the state update itself, and through the Delta Rule correction $\delta \boldsymbol{\Delta}_{[n]}$. The backward recurrence needs then to compute, in order, starting from $\delta \boldsymbol{S}_{[T/C]} \equiv \boldsymbol{0}$:

$$\begin{aligned} \delta \boldsymbol{\Delta}_{[n]} &= \delta^{\text{out}} \boldsymbol{\Delta}_{[n]} + \overrightarrow{\boldsymbol{K}}_{[n]} \delta \boldsymbol{S}_{[n]} \\ \delta \boldsymbol{S}_{[n-1]} &= \delta^{\text{out}} \boldsymbol{S}_{[n-1]} - \boldsymbol{W}_{[n]}^{\top} \delta \boldsymbol{\Delta}_{[n]} + \boldsymbol{G}_{[n]}^{\top} \delta \boldsymbol{S}_{[n]}. \end{aligned} \tag{59}$$

The first completes the gradient for the Delta Rule correction, the second correctly propagates it to the previous chunk. These operations occupy the *hot* sequential path for the recursion, but the state update actually produces two additional gradients, all computable post-hoc, in parallel across chunks. These are:

$$\begin{aligned} \delta \overrightarrow{\boldsymbol{K}}_{[n]} &= \boldsymbol{\Delta}_{[n]} \delta \boldsymbol{S}_{[n]}^{\top}, \\ \delta \boldsymbol{G}_{[n]} &= \boldsymbol{M}_B \odot \left(\delta \boldsymbol{S}_{[n]} \boldsymbol{S}_{[n-1]}^{\top} \right) = \boldsymbol{M}_B \odot \left(\sum_{i=1}^{d_v} [\delta \boldsymbol{S}_{[n]}]_{|,i} [\boldsymbol{S}_{[n-1]}]_{|,i}^{\top} \right), \end{aligned} \tag{60}$$

where $[\boldsymbol{M}_B]_{i,j} \coloneqq \delta_{\lfloor (i-1)/m \rfloor = \lfloor (j-1)/m \rfloor}$ is the block-diagonal mask with blocks of size d_k/n_B. The first gradient pertains to keys pushed to the end of the chunk: we treat this quantity similarly to $\delta\overleftarrow{\boldsymbol{Q}}_{[n]}$ in (55), and obtain gradient contributions for keys and gate block factors:

$$
\begin{aligned}
[\delta^{\text{upd}}\boldsymbol{K}_{[n]}]_{s,-} &= [\delta\overrightarrow{\boldsymbol{K}}_{[n]}]_{s,-}\boldsymbol{G}_{n_S+s\to n_E} \qquad \text{for } s = 1,\dots,C,\\
\delta^{\text{upd}^1}\boldsymbol{G}_i^{(b,A)} &= \sum_{s=n_S}^{i-1} (\boldsymbol{G}_{i+1\to n_E}^{(b,A)})^\top \operatorname{mat}\left(\delta\overrightarrow{\boldsymbol{k}}_s\big|_{(b)}\right) \boldsymbol{G}_{s+1\to n_E}^{(b,B)} \operatorname{mat}\left(\boldsymbol{k}_s\big|_{(b)}\right)^\top (\boldsymbol{G}_{s+1\to i-1}^{(b,A)})^\top,\\
\delta^{\text{upd}^1}\boldsymbol{G}_i^{(b,B)} &= \sum_{s=n_S}^{i-1} (\boldsymbol{G}_{i+1\to n_E}^{(b,B)})^\top \operatorname{mat}\left(\delta\overrightarrow{\boldsymbol{k}}_s\big|_{(b)}\right)^\top \boldsymbol{G}_{s+1\to n_E}^{(b,A)} \operatorname{mat}\left(\boldsymbol{k}_s\big|_{(b)}\right) (\boldsymbol{G}_{s+1\to i-1}^{(b,B)})^\top,
\end{aligned}
\tag{61}
$$

for $b = 1,\dots,n_B$ and $i = n_S,\dots,n_E$. The second gradient pertains the cumulative products of gates along the whole chunk. We can extract from this an additional gradient contribution to the individual gate block factors using Lem. 2:

$$
\begin{aligned}
\delta^{\text{upd}^2}\boldsymbol{G}_i^{(b,A)} &= \sum_{j=1}^{d_v} \Big((\boldsymbol{G}_{i+1\to n_E}^{(b,A)})^\top \operatorname{mat}\left([\delta\boldsymbol{S}_{[n]}]_{|,j}\big|_{(b)}\right)\\
&\qquad \cdot \boldsymbol{G}_{n_S\to n_E}^{(b,B)} \operatorname{mat}\left([\boldsymbol{S}_{[n-1]}]_{|,j}\big|_{(b)}\right)^\top \Big) (\boldsymbol{G}_{n_S\to i-1}^{(b,A)})^\top,\\
\delta^{\text{upd}^2}\boldsymbol{G}_i^{(b,B)} &= \sum_{j=1}^{d_v} \Big((\boldsymbol{G}_{i+1\to n_E}^{(b,B)})^\top \operatorname{mat}\left([\delta\boldsymbol{S}_{[n]}]_{|,j}\big|_{(b)}\right)^\top\\
&\qquad \cdot \boldsymbol{G}_{n_S\to n_E}^{(b,A)} \operatorname{mat}\left([\boldsymbol{S}_{[n-1]}]_{|,j}\big|_{(b)}\right) \Big) (\boldsymbol{G}_{n_S\to i-1}^{(b,B)})^\top,
\end{aligned}
\tag{62}
$$

for $b = 1,\dots,n_B$ and $i = n_S,\dots,n_E$.

Delta Rule correction. In (59) we considered only part of the gradient flow through the Delta Rule correction (that is, the one corresponding to the state $\delta\boldsymbol{S}_{[n-1]}$). The remainder is tracked in the following:

$$
\begin{aligned}
&\delta(\boldsymbol{T}_{[n]}^{-1}) = \delta\boldsymbol{\Delta}_{[n]}(\boldsymbol{V}_{[n]} - \overleftarrow{\boldsymbol{K}}_{[n]}\boldsymbol{S}_{[n-1]})^\top \boldsymbol{D}_{[n]}^\beta = \delta\boldsymbol{\Delta}_{[n]}\tilde{\boldsymbol{V}}_{[n]}^\top\\
\implies\quad &\delta\boldsymbol{T}_{[n]} = -\boldsymbol{T}_{[n]}^{-\top}(\delta\boldsymbol{\Delta}_{[n]}(\boldsymbol{V}_{[n]} - \overleftarrow{\boldsymbol{K}}_{[n]}\boldsymbol{S}_{[n-1]})^\top \boldsymbol{D}_{[n]}^\beta)\boldsymbol{T}_{[n]}^{-\top} = -(\boldsymbol{T}_{[n]}^{-\top}\delta\boldsymbol{\Delta}_{[n]})\boldsymbol{\Delta}_{[n]}^\top\\
\implies\quad &\delta\boldsymbol{A}_{[n]}^{kk} = \boldsymbol{D}_{[n]}^\beta\left(\delta\boldsymbol{T}_{[n]} \odot \boldsymbol{M}_{-1}\right)\\
\implies\quad &\delta\boldsymbol{\beta}_{[n]} = \left((\boldsymbol{T}_{[n]}^{-\top}\delta\boldsymbol{\Delta}_{[n]}) \odot (\boldsymbol{V}_{[n]} - \overleftarrow{\boldsymbol{K}}_{[n]}\boldsymbol{S}_{[n-1]}) + (\delta\boldsymbol{T}_{[n]} \odot \boldsymbol{M}_{-1}) \odot \boldsymbol{A}_{[n]}^{kk}\right)\mathbf{1}\\
&\delta\boldsymbol{V}_{[n]} = \boldsymbol{D}_{[n]}^\beta(\boldsymbol{T}_{[n]}^{-\top}\delta\boldsymbol{\Delta}_{[n]})\\
&\delta\overleftarrow{\boldsymbol{K}}_{[n]} = -\boldsymbol{D}_{[n]}^\beta(\boldsymbol{T}_{[n]}^{-\top}\delta\boldsymbol{\Delta}_{[n]})\boldsymbol{S}_{[n-1]}^\top,
\end{aligned}
\tag{63}
$$

where we used the formula for the gradient through matrix inversion $\delta\boldsymbol{X} = -\boldsymbol{X}^{-\top}\delta(\boldsymbol{X}^{-1})\boldsymbol{X}^{-\top}$. Note in particular the presence of the quantity $\boldsymbol{T}_{[n]}^{-\top}\delta\boldsymbol{\Delta}_{[n]}$: this is the adjoint of the Delta Rule unit lower-triangular solve, requiring solving an upper-triangular system instead. The last gradient $\delta\overleftarrow{\boldsymbol{K}}_{[n]}$ splits analogously to $\delta\overleftarrow{\boldsymbol{Q}}_{[n]}$, and produces additional gradient contributions to keys and gate blocks:

$$
\begin{aligned}
[\delta^{\text{corr}}\boldsymbol{K}_{[n]}]_{t,-} &= [\delta\overleftarrow{\boldsymbol{K}}_{[n]}]_{t,-}\boldsymbol{G}_{n_S\to n_S+t-1}^\top \qquad \text{for } t = 1,\dots,C,\\
\delta^{\text{corr}}\boldsymbol{G}_i^{(b,A)} &= \sum_{t=i}^{n_E} (\boldsymbol{G}_{i+1\to t}^{(b,A)})^\top \operatorname{mat}\left(\boldsymbol{k}_{n_S+t-1}\big|_{(b)}\right) \boldsymbol{G}_{n_S\to t}^{(b,B)} \operatorname{mat}\left(\delta\overleftarrow{\boldsymbol{k}}_t\big|_{(b)}\right)^\top (\boldsymbol{G}_{n_S\to i-1}^{(b,A)})^\top,\\
\delta^{\text{corr}}\boldsymbol{G}_i^{(b,B)} &= \sum_{t=i}^{n_E} (\boldsymbol{G}_{i+1\to t}^{(b,B)})^\top \operatorname{mat}\left(\boldsymbol{k}_{n_S+t-1}\big|_{(b)}\right)^\top \boldsymbol{G}_{n_S\to t}^{(b,A)} \operatorname{mat}\left(\delta\overleftarrow{\boldsymbol{k}}_t\big|_{(b)}\right) (\boldsymbol{G}_{n_S\to i-1}^{(b,B)})^\top.
\end{aligned}
\tag{64}
$$

Attention and Gram matrix assembly. The final derived quantities the gradients must flow through are the Attention and Gram matrices in (47). These produce:

$$
\begin{aligned}
[\delta^{\text{k-side}}\boldsymbol{K}_{[n]}]^\top_{s,-} &= \sum_{t>s} \boldsymbol{G}^\top_{s+1\to t}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}_{[n]}]_{t,s} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}_{[n]}]_{t,s}\right)\\
[\delta^{\text{q-side}}\boldsymbol{Q}_{[n]}]^\top_{t,-} &= \sum_{s<t} [\delta\boldsymbol{A}^{qk}_{[n]}]_{t,s}\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\\
[\delta^{\text{q-side}}\boldsymbol{K}_{[n]}]^\top_{t,-} &= \sum_{s<t} [\delta\boldsymbol{A}^{kk}_{[n]}]_{t,s}\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s\\
\delta\boldsymbol{G}^{\text{Attn}}_i &= \boldsymbol{M}_B \odot \sum_{s<i}\sum_{t\geq i}\left(\boldsymbol{G}^\top_{i+1\to t}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}_{[n]}]_{t,s} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}_{[n]}]_{t,s}\right)\right)\left(\boldsymbol{G}_{s+1\to i-1}\boldsymbol{k}_s\right)^\top.
\end{aligned}
\tag{65}
$$

The last gradient comes in the same sum of outer-products formulation as $\delta^{\text{upd}}\boldsymbol{G}_{[n]}$ in (60), and we can backpropagate through it to the single gate factors in a similar manner. To this end, we first define, for $t, s = n_S, \dots, n_E$ and $b = 1, \dots, n_B$,

$$
\delta\tilde{\boldsymbol{Q}}^{(b)}_{t,s} \coloneqq \text{mat}\left([\delta\boldsymbol{A}^{qk}_{[n]}]_{t,s}\boldsymbol{q}_t\big|_{(b)} + [\delta\boldsymbol{A}^{kk}_{[n]}]_{t,s}\boldsymbol{k}_t\big|_{(b)}\right), \tag{66}
$$

which allows us to more compactly write

$$
\begin{aligned}
\delta^{\text{Attn}}\boldsymbol{G}^{(b,A)}_i &= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}(\boldsymbol{G}^{(b,A)}_{i+1\to t})^\top\delta\tilde{\boldsymbol{Q}}^{(b)}_{t,s}\boldsymbol{G}^{(b,B)}_{s+1\to t}\,\text{mat}\left(\boldsymbol{k}_s\big|_{(b)}\right)^\top(\boldsymbol{G}^{(b,A)}_{s+1\to i-1})^\top,\\
\delta^{\text{Attn}}\boldsymbol{G}^{(b,B)}_i &= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}(\boldsymbol{G}^{(b,B)}_{i+1\to t})^\top(\delta\tilde{\boldsymbol{Q}}^{(b)}_{t,s})^\top\boldsymbol{G}^{(b,A)}_{s+1\to t}\,\text{mat}\left(\boldsymbol{k}_s\big|_{(b)}\right)(\boldsymbol{G}^{(b,B)}_{s+1\to i-1})^\top.
\end{aligned}
\tag{67}
$$

The kernel responsible for efficiently backpropagating through the Attention and Gram matrices assembly is another core contribution of this work, which we describe more in detail in App. G.3.

Combining everything together The final step in the backward pass consists in collecting the various gradient contributions to the kernel inputs. These are:

$$
\begin{aligned}
\delta\boldsymbol{G}^{(b,A)}_i &= \delta^{\text{out}}\boldsymbol{G}^{(b,A)}_i + \delta^{\text{upd}^1}\boldsymbol{G}^{(b,A)}_i + \delta^{\text{upd}^2}\boldsymbol{G}^{(b,A)}_i + \delta^{\text{corr}}\boldsymbol{G}^{(b,A)}_i + \delta^{\text{Attn}}\boldsymbol{G}^{(b,A)}_i\\
\delta\boldsymbol{G}^{(b,B)}_i &= \delta^{\text{out}}\boldsymbol{G}^{(b,B)}_i + \delta^{\text{upd}^1}\boldsymbol{G}^{(b,B)}_i + \delta^{\text{upd}^2}\boldsymbol{G}^{(b,B)}_i + \delta^{\text{corr}}\boldsymbol{G}^{(b,B)}_i + \delta^{\text{Attn}}\boldsymbol{G}^{(b,B)}_i\\
\delta\boldsymbol{K}_{[n]} &= \delta^{\text{upd}}\boldsymbol{K}_{[n]} + \delta^{\text{corr}}\boldsymbol{K}_{[n]} + \delta^{\text{k-side}}\boldsymbol{K}_{[n]} + \delta^{\text{q-side}}\boldsymbol{K}_{[n]}\\
\delta\boldsymbol{Q}_{[n]} &= \delta^{\text{out}}\boldsymbol{Q}_{[n]} + \delta^{\text{q-side}}\boldsymbol{Q}_{[n]},
\end{aligned}
\tag{68}
$$

as well as the already-complete $\delta\boldsymbol{\beta}_{[n]}$ from (63).

G KERNELS

Summary

In this section we detail the custom GPU kernels developed to render BGLT trainable at scale, and measure their hardware performance. The main contributions pertain to the assembly of the chunk-local attention and Gram matrices via hierarchical local factorization. The structure is similar for all kernels, in that target matrices are filled tile-wise, and separate strategies are employed for diagonal and off-diagonal tiles: while the latter can utilize Tensor Core efficiently, the former often falls back to a intra-tile sequential approach. A minor contribution consists in stabilizing the unit lower-triangular solver for the Delta-Rule correction, using bi-diagonally preconditioned Newton-Schulz, at no extra cost.

In App. F, we outlined the mathematical steps required for the chunk-wise formulation of BGLT. This section describes the GPU kernels responsible for efficiently computing those quantities. Our implementation builds on Flash Linear Attention (FLA) (Yang & Zhang, 2024): we follow its decomposition of the chunk-wise pipeline into state propagation, output computation and WY-representation kernels, and reuse its utilities for chunk indexing, gating, ℓ_2-normalization and short convolutions. We therefore focus strictly on the novel modifications introduced in this work. Specifically, we detail the inverse-free assembly of the Attention and Gram matrices in the forward pass (App. G.1) and their corresponding gradient flows in the backward pass (App. G.3). We also introduce an improved, Tensor Core-optimized algorithm for computing the lower-triangular inverse required by the Delta Rule correction (App. G.2).

G.1 FORWARD PASS: ATTENTION AND GRAM ASSEMBLY

The kernel presented in this section represents the main novelty, implementation-wise, introduced with BGLT: an efficient way to assemble the chunk-local Attention and Gram matrices, without having to invert gates. The purpose of this kernel is to compute the quantities

$$[\boldsymbol{A}^{qk}_{[n]}]_{t,s} = \boldsymbol{q}_t^\top \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s, \quad \text{and} \quad [\boldsymbol{A}^{kk}_{[n]}]_{t,s} = \boldsymbol{k}_t^\top \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s, \quad \text{for } t = 1,\dots,C, \quad s = 1,\dots,t. \tag{69}$$

By our definition in (6), BGLT uses as transition matrix $\boldsymbol{G}_t$ a direct sum of independent $m \times m$ blocks. The chunk-local matrices then decouple across the blocks, and we can write

$$[\boldsymbol{A}^{qk}_{[n]}]_{t,s} = \sum_{b=1}^{n_B} \boldsymbol{q}_t\big|_{(b)}^\top \boldsymbol{G}^{(b)}_{s+1\to t}\boldsymbol{k}_s\big|_{(b)}, \qquad \text{and} \qquad [\boldsymbol{A}^{kk}_{[n]}]_{t,s} = \sum_{b=1}^{n_B} \boldsymbol{k}_t\big|_{(b)}^\top \boldsymbol{G}^{(b)}_{s+1\to t}\boldsymbol{k}_s\big|_{(b)}, \tag{70}$$

where $(*)\big|_{(b)}$ denotes the b-th m-dimensional slice of $(*)$. The kernel then simply executes a loop over the $n_B = d/m$ blocks to accumulate the full matrices. To avoid cluttering notation in the remainder of this section we drop references to chunks n and blocks b, but the involved quantities should still be interpreted as chunk- and block-local.

Note that both matrices in (69) share the same structure: a query tested against a key, which has been pushed by the gates in the gap spanning the two; the only difference lies in the fact that, for the Gram matrix, $\boldsymbol{k}$ covers simultaneously the role of key and query. As discussed in Sec. 3, the transport $\boldsymbol{G}_{s+1\to t}$ ties the two t, s indices together, and the target matrices cannot be factorized into a single product of two tensors (one spanning t, the other s). Nonetheless, *local* factorization is still feasible: for any reference token f in between t and s, we can in fact split

$$\boldsymbol{q}_t^\top \boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s = (\boldsymbol{G}^\top_{f\to t}\boldsymbol{q}_t)^\top(\boldsymbol{G}_{s+1\to f-1}\boldsymbol{k}_s) = ({}^{f}\overleftarrow{\boldsymbol{q}}_t)^\top\, \overrightarrow{\boldsymbol{k}}^{f-1}_s, \tag{71}$$

so by pushing keys and queries to a common frame of reference, we can still rely on tensor products to efficiently compute entries for all (t, s) pairs in $[f, \dots, C] \times [1, \dots, f)$. To leverage this, we further divide tokens into multiple non-overlapping tiles I which together cover the whole chunk, and treat their boundaries as frames of reference. The kernel is then organized in two phases: the first phase computes entries for (t, s) pairs within a *same* tile $(t, s) \in I \times I$, too close to admit a useful frame split; the second phase computes entries for pairs within *different* tiles $(t, s) \in I \times J$, $I > J$, for which we can rely on matrix multiplications, by splitting gates products across tiles boundaries.

Algorithm 1: Matrix assembly forward – Phase 1: diagonal tiles

Input: Queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$, transition gates $\boldsymbol{G}_t$

// Thread t owns token t: its query, its key and its gate

for $b = 1$ **to** n_B **do**

  Read block-local quantities $\boldsymbol{G}_t \leftarrow \boldsymbol{G}_t^{(b)}, \boldsymbol{q}_t \leftarrow \boldsymbol{q}_t\big|_{(b)}, \boldsymbol{k}_t \leftarrow \boldsymbol{k}_t\big|_{(b)}$;

  // Phase 1 ———————————————————————————————————————

  // - Diagonal walk: assemble diagonals step-by-step

  Accumulate $[(\boldsymbol{A}^{qk}, \boldsymbol{A}^{kk})]_{t,t} \mathrel{+}= (\boldsymbol{q}_t, \boldsymbol{k}_t)^\top \boldsymbol{k}_t$; // main diag

  $\boldsymbol{x}_t \leftarrow \boldsymbol{k}_t$; // init traveling state: un-pushed key

  for $\delta = 1$ **to** $|I|$ **do**

    $\boldsymbol{x}_t \leftarrow \texttt{shuffle}(\boldsymbol{x}_{t-1})$; // systolic shift: only x_t travels

    $\boldsymbol{x}_t \leftarrow \boldsymbol{G}_t \boldsymbol{x}_t$; // each lane applies its own gate

    if $t - \delta$ within tile **then**

      Accumulate $[(\boldsymbol{A}^{qk}, \boldsymbol{A}^{kk})]_{t,t-\delta} \mathrel{+}= (\boldsymbol{q}_t, \boldsymbol{k}_t)^\top \boldsymbol{x}_t$;

  $\overrightarrow{\boldsymbol{k}}_t^{I_E}\big|_{(b)} \leftarrow \texttt{shuffle}(\boldsymbol{x}_{t-1})$; // align pushed keys to their own thread

  // - Epilogue: align queries, prepare whole-tile cumprods

  $\boldsymbol{G}_{I_S \to t} \leftarrow \texttt{HSbwd}(\boldsymbol{G})$; // Hillis-Steele parallel scan

  $(\overleftarrow{\boldsymbol{q}}_t^{I_S}\big|_{(b)}, \overleftarrow{\boldsymbol{k}}_t^{I_S}\big|_{(b)}) \leftarrow \boldsymbol{G}_{I_S \to t}^\top (\boldsymbol{q}_t, \boldsymbol{k}_t)$; // push queries back to tile start

  $\boldsymbol{G}_{[I]}^{(b)} \leftarrow \boldsymbol{G}_{I_S \to I_E}$; // store whole-tile product for Phase 2

This ordering is also reflected in the way the Attention and Gram matrices are assembled: in Phase 1, their diagonal tiles $[I, I]$ are filled, followed by the off-diagonal ones in Phase 2. The kernel work for the two separate phases is outlined next.

Phase 1: diagonal tiles In the first phase of the algorithm, we take both queries and keys from the same tile $(t, s) \in I \times I$. We compute and accumulate matrix contributions block-by-block, processing queries and keys in slices of size m. The algorithm proceeds by pushing keys forward along t, progressively multiplying them by gates from subsequent positions. The Attention and Gram matrices contributions are then assembled one diagonal at a time: at step $\delta = 0, \ldots, |I|$ we produce in fact a block's contributions for

$$[\boldsymbol{A}^{qk}]_{t,t-\delta} = \boldsymbol{q}_t^\top \boldsymbol{G}_{t-\delta+1 \to t} \boldsymbol{k}_{t-\delta}, \qquad \text{and} \qquad [\boldsymbol{A}^{kk}]_{t,t-\delta} = \boldsymbol{k}_t^\top \boldsymbol{G}_{t-\delta+1 \to t} \boldsymbol{k}_{t-\delta}. \tag{72}$$

To minimize data movement, we set up a *systolic shift*: only one key travels from one token to the next, incorporating the local gate at each step, so that the key gets progressively pushed as it moves through tokens, and can get tested with the local query on-the-fly. This procedure terminates once every key has hit the tile end I_E, at which point all entries for the diagonal tiles have been computed.

To set up the stage for Phase 2, we need the queries, too, to be pushed to the tile boundary. This means pushing queries *back* to the tile start I_S, i.e., computing $\boldsymbol{G}_{I_S \to t}^\top \boldsymbol{q}_t$. In the epilogue of Phase 1, then, the (inclusive) cumulative gate products are computed efficiently via Hillis-Steele parallel scan running tile-wise. Pseudocode for the first part of the algorithm is provided in Alg. 1, while the second phase, covering off-diagonal tiles, is illustrated next.

Phase 2: off-diagonal tiles For the second phase of the algorithm, we take queries and keys from different tiles $(t, s) \in I \times J$, $I > J$ and use the row tile start I_S as a common frame of reference for local factorization. By the end of Phase 1, adjacent tiles already have queries and keys pushed to the same frame of reference, but tiles further apart must first close the gap: this is done by pushing keys by the whole-tile cumulative products for tiles in-between I and J:

$$\overrightarrow{\boldsymbol{k}}_s^{I_S - 1} = \boldsymbol{G}_{J_E+1 \to I_S - 1} \overrightarrow{\boldsymbol{k}}_s^{J_E}, \qquad \text{with} \qquad \boldsymbol{G}_{J_E+1 \to I_S - 1} \coloneqq \prod_{J < K < I} \boldsymbol{G}_{[K]}, \tag{73}$$

Algorithm 2: Matrix assembly forward – Phase 2: off-diagonal tiles ($I > J$)

Input: Pushed queries and keys ${}^{I_S\leftarrow}\boldsymbol{q}_t$, ${}^{I_S\leftarrow}\boldsymbol{k}_t$, $\overrightarrow{\boldsymbol{k}}_s^{J_E}$ and whole-tile products $\boldsymbol{G}_{[I]}$ from Phase 1

// Lane i covers both tile positions: query $t = I_S + i$, key $s = J_S + i$

for $b = 1$ **to** n_B **do**

  Read block-local $\boldsymbol{G}_{[I]} \leftarrow \boldsymbol{G}^{(b)}_{[I]}, ({}^{I_S\leftarrow}\boldsymbol{q}_t, {}^{I_S\leftarrow}\boldsymbol{k}_t) \leftarrow ({}^{I_S\leftarrow}\boldsymbol{q}_t, {}^{I_S\leftarrow}\boldsymbol{k}_t)\big|_{(b)}$, $\overrightarrow{\boldsymbol{k}}_s^{J_E} \leftarrow \overrightarrow{\boldsymbol{k}}_s^{J_E}\big|_{(b)}$;

  // Phase 2

  // - Push keys and queries to common frame I_S

  $\boldsymbol{G}_{J_E+1\to I_S-1} \leftarrow \prod_{J<K<I} \boldsymbol{G}_{[K]}$;

  $\overrightarrow{\boldsymbol{k}}_s^{I_S-1} \leftarrow \boldsymbol{G}_{J_E+1\to I_S-1}\overrightarrow{\boldsymbol{k}}_s^{J_E}$; // push across in-between tiles

  // - compute matrix tiles via matrix multiplication

  Accumulate $[\boldsymbol{A}^{qk}]_{I,J}$ += MMA$\big([{}^{I_S\leftarrow}\boldsymbol{Q}]_I, [\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J^\top\big)$;

  Accumulate $[\boldsymbol{A}^{kk}]_{I,J}$ += MMA$\big([{}^{I_S\leftarrow}\boldsymbol{K}]_I, [\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J^\top\big)$;

At this stage, both operands share the same frame I_S, and a whole tile of the Attention and Gram matrices can be recovered via a single tensor product:

$$[\boldsymbol{A}^{qk}]_{I,J} = [{}^{I_S\leftarrow}\boldsymbol{Q}]_I \left([\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J\right)^\top, \qquad [\boldsymbol{A}^{kk}]_{I,J} = [{}^{I_S\leftarrow}\boldsymbol{K}]_I \left([\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J\right)^\top, \tag{74}$$

where $[(*)]_{I,J}$ denotes the (I, J)-th sub-tile of matrix $(*)$. Even in this case we proceed block-wise, accumulating contributions to tiles of both matrices one block at a time. This justifies our choice of targeting blocks of size $m = 16$: this way, the dimensions of the involved tensors properly map to Tensor Cores. Moreover, our Kronecker factorization allows us to process contributions from a whole block while only manipulating two 4×4 factors, reducing register pressure and computation overhead. For pseudocode corresponding to this phase of the algorithm, see Alg. 2.

G.2 Forward pass: improving the Unit Triangular Solver for Delta Rule

As outlined in (51), including the Delta Rule correction (Yang et al., 2025a) in the chunk-wise parallel application of BGLT requires inverting the $C \times C$ unit lower-triangular matrix

$$\boldsymbol{T}_{[n]} \coloneqq \boldsymbol{I} + (\boldsymbol{D}^{\beta}_{[n]}\boldsymbol{A}^{kk}_{[n]}) \odot \boldsymbol{M}_{-1}, \tag{75}$$

with the strictly lower-triangular part composed by the β-scaled Gram matrix

$$\boldsymbol{L}_{[n]} \coloneqq (\boldsymbol{D}^{\beta}_{[n]}\boldsymbol{A}^{kk}_{[n]}) \odot \boldsymbol{M}_{-1}. \tag{76}$$

The current FLA implementation (Yang & Zhang, 2024), computes its inverse by tiling $\boldsymbol{L}_{[n]}$ in blocks of size 16×16. The diagonal tiles of $\boldsymbol{T}^{-1}_{[n]}$ are computed using vanilla forward substitution in `fp32`. For the off-diagonal tiles, the implementation leverages the property

$$\begin{aligned}
[\boldsymbol{T}_{[n]}\boldsymbol{T}^{-1}_{[n]}]_{I,J} &= \sum_K [\boldsymbol{T}_{[n]}]_{I,K}[\boldsymbol{T}^{-1}_{[n]}]_{K,J} \\
&= \sum_{K<I} [\boldsymbol{T}_{[n]}]_{I,K}[\boldsymbol{T}^{-1}_{[n]}]_{K,J} + [\boldsymbol{T}_{[n]}]_{I,I}[\boldsymbol{T}^{-1}_{[n]}]_{I,J} + \cancelto{\mathbf{0}}{\sum_{K>I} [\boldsymbol{T}_{[n]}]_{I,K}[\boldsymbol{T}^{-1}_{[n]}]_{K,J}} \\
\implies [\boldsymbol{T}^{-1}_{[n]}]_{I,J} &= -[\boldsymbol{T}_{[n]}]^{-1}_{I,I} \sum_{K<I} [\boldsymbol{T}_{[n]}]_{I,K}[\boldsymbol{T}^{-1}_{[n]}]_{K,J}
\end{aligned} \tag{77}$$

where $[(*)]_{I,J}$ denotes the (I, J)-th sub-tile of matrix $(*)$. Following this, the off-diagonal tiles are computed progressively, subdiagonal-by-subdiagonal, via tile-wise matrix multiplications. The precision of this matrix multiplication operation is tunable (either `tf32`, `ieee` or `tf32x3`).

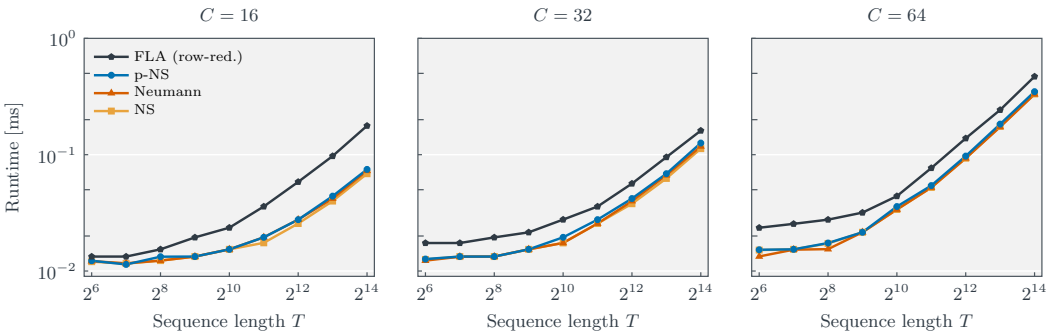


Figure G.1: Runtime to compute $\boldsymbol{T}_{[n]}^{-1}$ using FLA's forward substitution, Neumann series, Newton-Schulz, or our preconditioned Newton-Schulz, for matrices of different sizes C.

Tensor-Core only triangular solve: diagonal tile inverse via Newton-Schulz Using forward substitution for the diagonal tiles $[\boldsymbol{T}_{[n]}^{-1}]_{I,I}$ is, however, suboptimal: it is a purely sequential operation in the tile size, which does not leverage at all the advantages of Tensor Cores. To address this, we employ instead Newton-Schulz (NS) for the calculation of the inverses of the diagonal tiles. NS is simply Newton method applied to the function $\boldsymbol{F}(\boldsymbol{X}) = \boldsymbol{X}^{-1} - \boldsymbol{T} = \boldsymbol{0}$ (where we dropped the chunk-wise and tile-wise $[(*)_{[n]}]_{I,I}$ notation for clarity). The method generates the iterates:

$$\begin{aligned}\boldsymbol{X}_k &= \boldsymbol{X}_{k-1} - [d\boldsymbol{F}(\boldsymbol{X}_{k-1})]^{-1}\boldsymbol{F}(\boldsymbol{X}_{k-1}) = \boldsymbol{X}_{k-1} + \boldsymbol{X}_{k-1}(\boldsymbol{X}_{k-1}^{-1} - \boldsymbol{T})\boldsymbol{X}_{k-1} \\ &= 2\boldsymbol{X}_{k-1} - \boldsymbol{X}_{k-1}\boldsymbol{T}\boldsymbol{X}_{k-1} = \boldsymbol{X}_{k-1}(2\boldsymbol{I} - \boldsymbol{T}\boldsymbol{X}_{k-1}) :\end{aligned} \tag{78}$$

each update can then be computed with two matrix multiplications. Note that starting from $\boldsymbol{X}_0 = \boldsymbol{I} - \boldsymbol{L}$ produces a truncated Neumann series, whose terms double at each iteration:

$$\begin{aligned}\boldsymbol{X}_0 &= \boldsymbol{I} - \boldsymbol{L} = \textstyle\sum_{i=0}^{1}(-\boldsymbol{L})^i \\ \boldsymbol{X}_1 &= (\boldsymbol{I} - \boldsymbol{L})(2\boldsymbol{I} - (\boldsymbol{I} + \boldsymbol{L})(\boldsymbol{I} - \boldsymbol{L})) \\ &= \boldsymbol{I} - \boldsymbol{L} + \boldsymbol{L}^2 - \boldsymbol{L}^3 = \textstyle\sum_{i=0}^{3}(-\boldsymbol{L})^i \\ &\vdots \\ \boldsymbol{X}_3 &= \boldsymbol{I} - \boldsymbol{L} + \boldsymbol{L}^2 - \ldots - \boldsymbol{L}^{15} = \textstyle\sum_{i=0}^{15}(-\boldsymbol{L})^i,\end{aligned} \tag{79}$$

and, since the strictly lower-triangular term $\boldsymbol{L}$ is nilpotent, *three* iterations of NS are always sufficient to recover the inverse: $\boldsymbol{T}^{-1} = \sum_{i=0}^{\infty}(-\boldsymbol{L})^i = \sum_{i=0}^{15}(-\boldsymbol{L})^i \equiv \boldsymbol{X}_3$. In summary, we can compute the inverse of each diagonal tile of $\boldsymbol{T}^{-1}$ with 6 matrix multiplications, using only Tensor Cores. The resulting runtime speedup over FLA's sequential row-reduction is reported in Fig. G.1: for $C = 16$ we are $\approx 3\times$ faster than FLA, reducing to $\approx 1.5 - 2.0\times$ for $C = 32, 64$, since the off-diagonal tiles computation rely on Tensor Cores for both our and FLA's implementations.

We point out that a Tensor Core-only implementation for the diagonal tile inverse is also employed in FlashKDA (Chen et al., 2026): there, the authors realize the Neumann series directly via repeated squaring. While *analytically* both algorithms produce the same output, we opt for NS because *numerically* it offers better precision over vanilla repeated squaring, at no additional cost. In fact, the error $\boldsymbol{E}_k$ for NS satisfies the following property:

$$\boldsymbol{E}_{k+1} \coloneqq \boldsymbol{I} - \boldsymbol{T}\boldsymbol{X}_{k+1} = \boldsymbol{I} - \boldsymbol{T}\boldsymbol{X}_k(2\boldsymbol{I} - \boldsymbol{T}\boldsymbol{X}_k) = (\boldsymbol{I} - \boldsymbol{T}\boldsymbol{X}_k)^2 = \boldsymbol{E}_k^2, \tag{80}$$

implying it gets quickly eliminated if it starts close enough to 0. Moreover, concurrently to our work, a similar NS-base approach has also been proposed in Sobczyk et al. (2026); in our work, however, we further stabilize the method, as described next.

Stabilizing Newton-Schulz via bidiagonal preconditioning Despite its fast convergence, and its advantageous use of Tensor Cores, the fact that NS essentially relies on repeated squaring still exposes it to large transient growths: as they accumulate powers of $\boldsymbol{L}^i$, the iterates $\boldsymbol{X}_k$ in (79) might grow significantly in magnitude before converging, which, since the operations are performed in

Algorithm 3: Bidiagonally-preconditioned Newton–Schulz inverse of unit triangular matrix

Input: Strictly lower-triangular, β-scaled Gram matrix $\boldsymbol{L}_{[n]}$

Output: The inverse $\boldsymbol{T}_{[n]}^{-1} = (\boldsymbol{I} + \boldsymbol{L}_{[n]})^{-1}$

parfor Each diagonal tile I **do**
  $\boldsymbol{X}_I \leftarrow \boldsymbol{I} - [\tilde{\boldsymbol{L}}_{[n]}]_{I,I}$; // Apply preconditioner, 1 matmul
  for $k \leftarrow 1$ **to** 2 **do**
    $\boldsymbol{X}_I \leftarrow \boldsymbol{X}_I \left(2\boldsymbol{I} - (\boldsymbol{I} + [\tilde{\boldsymbol{L}}_{[n]}]_{I,I})\boldsymbol{X}_I\right)$; // NS step, 2 matmuls ($\times 2$ steps)
  $[\boldsymbol{T}_{[n]}^{-1}]_{I,I} \leftarrow \boldsymbol{X}_I[\boldsymbol{P}_\Delta]_I$; // Undo preconditioning, 1 matmul
for Each block sub-diagonal D **do**
  parfor Each tile I in the D-th subdiagonal **do**
    $[\boldsymbol{T}_{[n]}^{-1}]_{I,I-D} \leftarrow -[\boldsymbol{T}_{[n]}]_{I,I}^{-1} \sum_{K<I} [\boldsymbol{T}_{[n]}]_{I,K} [\boldsymbol{T}_{[n]}^{-1}]_{K,I-D}$
return $\boldsymbol{T}_{[n]}^{-1}$;

`tf32`, could result in impactful round-off errors or even NaNs. The most pathological case occurs when $\boldsymbol{L}$ is composed entirely of unit entries, i.e. $\boldsymbol{L} = \boldsymbol{M}_{-1}$: its powers contain binomial entries growing like $[\boldsymbol{M}_{-1}^k]_{ij} = \binom{i-j-1}{k-1}$, and peaking at $\binom{14}{7} = 3432$; the partial sums then explode by three orders of magnitude before canceling back to entries in $\{0, \pm 1\}$, which `tf32` cannot properly track. Most importantly, *this case is realized in practice*: it appears whenever we encounter sequences of repeated tokens, with gates close to identity, which is not entirely uncommon during training. Fortunately, we can counteract this via *preconditioning*. Preconditioning amounts to re-casting the target problem into one with better numerical properties, often by priming it with the solution of a simpler problem of the same type. Since the solution to the pathological case is a bi-diagonal matrix[5], the natural choice is to use as a preconditioner the unit bi-diagonal matrix obtained by taking the sub-diagonal from $\boldsymbol{L}$ itself:

$$\boldsymbol{P}_\Delta \coloneqq \boldsymbol{I} - \boldsymbol{L}_{-1}, \qquad \text{with} \qquad [\boldsymbol{L}_{-1}]_{i,j} \coloneqq \delta_{i-1=j}[\boldsymbol{L}]_{i,j}. \tag{81}$$

The target problem then becomes the preconditioned one, i.e., we need to invert the quantity:

$$\boldsymbol{P}_\Delta(\boldsymbol{I} + \boldsymbol{L}) = (\boldsymbol{I} - \boldsymbol{L}_{-1})(\boldsymbol{I} + \boldsymbol{L}) = \boldsymbol{I} + \underbrace{\boldsymbol{L} - \boldsymbol{L}_{-1} - \boldsymbol{L}_{-1}\boldsymbol{L}}_{=:\tilde{\boldsymbol{L}}}. \tag{82}$$

The preconditioned problem is of the same type as the original, in that we still need to invert a unit lower-triangular matrix, except it is built from $\tilde{\boldsymbol{L}}$ rather than $\boldsymbol{L}$: as such, we can still use NS. Once

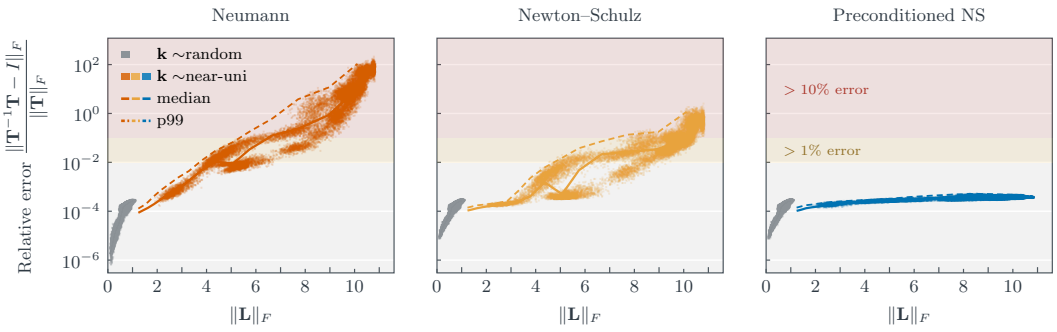


Figure G.2: Relative error in Frobenius norm from inverting $\boldsymbol{T}_{[n]}$ via Neumann series (left), Newton-Schulz (middle), and our preconditioned Newton-Schulz (right), for key-Gram matrices $\boldsymbol{L}_{[n]}$ deriving from different key distributions (20k+ realizations). While random $\boldsymbol{k}$ distributions (gray) pose no problems for numerical stability, for more aligned distributions of keys (colored) non-preconditioned algorithms result in markedly higher numerical errors.

[5]Note in fact that $[(\boldsymbol{I} + \boldsymbol{M}_{-1})^{-1}]_{i,j} = \delta_{i=j} - \delta_{i-1=j}$, that is, the inverse of the all-unit lower triangular matrix is a bi-diagonal matrix with 1 on the main diagonal, and -1 on the sub-diagonal.

we get the solution to the preconditioned problem $(\boldsymbol{I}+\tilde{\boldsymbol{L}})^{-1}$, the solution to the original problem can be recovered by simply rescaling by $\boldsymbol{P}_\Delta$:

$$\boldsymbol{P}_\Delta(\boldsymbol{I}+\boldsymbol{L})=(\boldsymbol{I}+\tilde{\boldsymbol{L}}) \quad\Longrightarrow\quad \boldsymbol{T}^{-1}=(\boldsymbol{I}+\boldsymbol{L})^{-1}=(\boldsymbol{I}+\tilde{\boldsymbol{L}})^{-1}\boldsymbol{P}_\Delta. \tag{83}$$

Note that both the preconditioning and the un-preconditioning operations eqs. (82) and (83) require one extra matrix multiplication each. Crucially, however, the strictly lower-triangular component of the preconditioned system $\tilde{\boldsymbol{L}}$ presents an *all-0 first subdiagonal*. We can convince ourselves of this by looking at its definition in (82): $\boldsymbol{L}-\boldsymbol{L}_{-1}$ removes the first subdiagonal from $\boldsymbol{L}$, while $\boldsymbol{L}_{-1}\boldsymbol{L}$ is a product of two matrices whose diagonal is 0, which also produces a zero subdiagonal. As a consequence, its Neumann series truncates after the 7th power of $\tilde{\boldsymbol{L}}$, and NS requires only 2 iterations to convergence, bringing the total of matrix multiplications required for inversion to $1+2\cdot 2+1=6$, just like in the vanilla NS case.

The resulting algorithm, outlined in Alg. 3, allows for the recovery of the inverse of the unit lower-triangular matrix $\boldsymbol{T}_{[n]}$ using only matrix multiplication, with better stability than computing Neumann series for its diagonal tiles, and at an overall equivalent cost. See Fig. G.1 for a comparison of runtimes, and Fig. G.2 for the benefits in numerical precision.

We note that our preconditioner is designed for $\beta\in[0,1]$ which is case for large hybrid LLMs using the Delta Rule such as Kimi K3 (Kimi Team, 2026). Letting $\beta\in[0,2]$ can still cause transient growth and subsequently large round-off error under repeated tokens with $\beta>1$.

G.3 Backward pass: Attention and Gram assembly

Similarly to the forward pass, the main novelty introduced by BGLT in the backward pass lies in the dual of the matrix assembly kernel from App. G.1. Its objective is to propagate gradients from the individual entries of the chunk-local Attention and Gram matrices back to queries, keys, and gates. Crucially, while the forward assembly required accumulating contributions across all gate blocks and running them concurrently incurred the risk of racing conditions, the backward operations are instead *perfectly parallel* across blocks. In fact, matrix gradients are simply broadcast to each block, and the kernel can thus process quantities independently and concurrently in m-dimensional slices, effectively benefiting from an additional axis of parallelism. To streamline our notation in the following derivations, we drop all explicit references to block (b) and chunk $[n]$ indices, with the understanding that all operations described next are restricted to a single block and chunk.

For convenience, we recall the gradient contributions for queries and keys derived in (65):

$$\begin{aligned}
[\delta^{\text{k-side}}\boldsymbol{K}]^\top_{s,-} &= \sum_{t>s}\underbrace{\boldsymbol{G}^\top_{s+1\to t}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s}+\boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)}_{=:\delta^k\overleftarrow{\boldsymbol{q}}_{s,s\to t}}\\
[\delta^{\text{q-side}}\boldsymbol{Q}]^\top_{t,-} &= \sum_{s<t}[\delta\boldsymbol{A}^{qk}]_{t,s}\underbrace{\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s}_{=:\overrightarrow{\boldsymbol{k}}_{s\to t}}\\
[\delta^{\text{q-side}}\boldsymbol{K}]^\top_{t,-} &= \sum_{s<t}[\delta\boldsymbol{A}^{kk}]_{t,s}\underbrace{\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s}_{=:\overrightarrow{\boldsymbol{k}}_{s\to t}}.
\end{aligned} \tag{84}$$

We point out that the expressions above all share a similar structure: a sum of queries and/or keys weighted according to the Attention and Gram matrices gradients, each pushed to a given index by a cumulative product of gates. To organize the description of our kernel, we distinguish these expressions depending on the index that survives the summation operation: we denote as *key-side* contributions the ones indexed by a key s and obtained summing over queries t, while the *query-side* contributions are indexed by a query t and obtained summing over keys s. Noticeably, the keys $\boldsymbol{K}$ receive gradient contributions from both sides, since they play the role of queries in the assembly of the Gram matrix in (47). A similar structure is maintained also in the definition of the gate gradients

from (67):

$$\begin{aligned}
\delta^{\text{Attn}}\boldsymbol{G}_i^{(A)} &= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}(\boldsymbol{G}_{i+1\to t}^{(A)})^\top \delta\tilde{\boldsymbol{Q}}_{t,s}\boldsymbol{G}_{s+1\to t}^{(B)}\,\mathrm{mat}\,(\boldsymbol{k}_s)^\top\,(\boldsymbol{G}_{s+1\to i-1}^{(A)})^\top \\
&= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}\underbrace{(\boldsymbol{G}_{i+1\to t}^{(A)})^\top \delta\tilde{\boldsymbol{Q}}_{t,s}\boldsymbol{G}_{i+1\to t}^{(B)}}_{=:\mathrm{mat}\left(\delta^k\overleftarrow{\boldsymbol{q}}_{s,i\to t}\right)}\boldsymbol{G}_i^{(B)}\underbrace{\boldsymbol{G}_{s+1\to i-1}^{(B)}\,\mathrm{mat}\,(\boldsymbol{k}_s)^\top\,(\boldsymbol{G}_{s+1\to i-1}^{(A)})^\top}_{=:\mathrm{mat}\left(\overrightarrow{\boldsymbol{k}}_{s\to i-1}\right)^\top} \\
\delta^{\text{Attn}}\boldsymbol{G}_i^{(B)} &= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}(\boldsymbol{G}_{i+1\to t}^{(B)})^\top \delta\tilde{\boldsymbol{Q}}_{t,s}^\top\boldsymbol{G}_{s+1\to t}^{(A)}\,\mathrm{mat}\,(\boldsymbol{k}_s)\,(\boldsymbol{G}_{s+1\to i-1}^{(B)})^\top \\
&= \sum_{s=n_S}^{i-1}\sum_{t=i}^{n_E}\underbrace{(\boldsymbol{G}_{i+1\to t}^{(B)})^\top \delta\tilde{\boldsymbol{Q}}_{t,s}^\top\boldsymbol{G}_{i+1\to t}^{(A)}}_{=:\mathrm{mat}\left(\delta^k\overleftarrow{\boldsymbol{q}}_{s,i\to t}\right)^\top}\boldsymbol{G}_i^{(A)}\underbrace{\boldsymbol{G}_{s+1\to i-1}^{(A)}\,\mathrm{mat}\,(\boldsymbol{k}_s)\,(\boldsymbol{G}_{s+1\to i-1}^{(B)})^\top}_{=:\mathrm{mat}\left(\overrightarrow{\boldsymbol{k}}_{s\to i-1}\right)}.
\end{aligned} \tag{85}$$

Even in this case we have weighted sums of pushed queries and keys, but with two relevant distinctions: (i) the pushed quantities are further combined among themselves to recover the gradient contributions (with the gate's other Kronecker factor acting as an additional weight), and (ii) the sums span simultaneously all keys before and all queries after a given gate, implying that gate gradients must collect both key-side and query-side contributions.

While the double sum for the gate gradients introduces additional complexity, the main challenge for the backward kernel is still represented by the gates cumulative products appearing in the expressions above. Like for the forward pass in App. G.1, in fact, these products inextricably tie together both t and s indices, making chunk-wise factorization between keys and queries impossible (unless we rely on matrix inversions, which can result in numerical instabilities). Factorizing *locally*, however, is still possible, for those (t, s) pairs which are spread apart enough to admit a common frame of reference in-between them. Once we can introduce such a frame, we can split transitions before and after it: this allows us to reduce the sums to contractions along a single axis, that is, a single tensor product.

The resulting backward kernel follows the same tile-wise structure as the forward one, whereby diagonal tiles are handled by sequential sweeps and off-diagonal ones by tile-wide matrix multiplications. Since gate gradients require two reductions, however, the work in each phase of the algorithm is also doubled. On a diagonal tile, this translates into performing two sweeps (one forward to push keys, the other backward to push queries); on off-diagonal tiles, we contract via matrix multiplications on both axes (once against keys, once against queries, to recover two independent reductions); either way, the other axis survives, and requires a final tile-wise parallel scan to properly reduce over. More details for each of these operations are reported next.

Phase 1: diagonal tiles In the first phase of the backward assembly algorithm, we collect contributions for gradients of keys, queries and gates in I, stemming from quantities within the I-th tile itself. Following eqs. (84) and (85), these reduce to:

$$\begin{aligned}
[\delta_{I,I}^{\text{k-side}}\boldsymbol{K}]_{s,-}^\top &= \sum_{t=s}^{I_E}\boldsymbol{G}_{s+1\to t}^\top\underbrace{\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s}+\boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)}_{=:\delta^k\boldsymbol{q}_{t,s}} \\
[\delta_{I,I}^{\text{q-side}}(\boldsymbol{Q},\boldsymbol{K})]_{t,-}^\top &= \sum_{s=I_S}^{t-1}[\delta(\boldsymbol{A}^{qk},\boldsymbol{A}^{kk})]_{t,s}\boldsymbol{G}_{s+1\to t}\boldsymbol{k}_s
\end{aligned} \tag{86}$$

for queries and keys, and

$$\begin{aligned}
\delta_{I,I}^{\text{Attn}}\boldsymbol{G}_i^{(A)} &= \sum_{s=I_S}^{i-1}\left[\sum_{t=i}^{I_E}\mathrm{mat}\left(\boldsymbol{G}_{i+1\to t}^\top\delta^k\boldsymbol{q}_{t,s}\right)\right]\boldsymbol{G}_i^{(B)}\,\mathrm{mat}\,(\boldsymbol{G}_{s+1\to i-1}\boldsymbol{k}_s)^\top \\
\delta_{I,I}^{\text{Attn}}\boldsymbol{G}_i^{(B)} &= \sum_{s=I_S}^{i-1}\left[\sum_{t=i}^{I_E}\mathrm{mat}\left(\boldsymbol{G}_{i+1\to t}^\top\delta^k\boldsymbol{q}_{t,s}\right)^\top\right]\boldsymbol{G}_i^{(A)}\,\mathrm{mat}\,(\boldsymbol{G}_{s+1\to i-1}\boldsymbol{k}_s)
\end{aligned} \tag{87}$$

for the gate factors, with I_S, I_E denoting the tile's start and end indices, respectively. We point out that every term is assembled from just two families: keys $\boldsymbol{k}_s$ pushed forward, and adjoints $\delta^k \boldsymbol{q}_{t,s}$ pushed backwards (with gate gradients requiring both, meeting at a common offset i). The algorithm works by building the two families in two separate sweeps.

DOWN-SWEEP $s \to I_E$. The first sweep is the same systolic shift used in the forward: keys travel forward across lanes, each lane applying its own gate, so that at each step d lane s holds the key from d tokens back, pushed to token s. This pushed key can be stored for the up-sweep as well, as we will need it for the gate gradients. Each lane can then include the new term in $\boldsymbol{G}_{s-d+1\to s}\boldsymbol{k}_{s-d}$ into the query-side contributions in (86) by fetching the correct Attention and Gram matrix gradients, before passing the key to the next lane and continue the systolic shift. At the end of this sweep, the query-side contributions are correctly accounted for, and we have computed all pushed keys from any s to any $t \in (s, I_E]$.

UP-SWEEP $I_S \leftarrow t$. The second sweep runs a similar systolic shift, but backwards, and it directly transports a running sum of partially-pushed adjoints (the bracketed term in (87)). At step d, each lane t adds its own contribution to the running sum (including its own adjoint $\delta^k \boldsymbol{q}_{t,t-d}$) so that it holds the full t-wide reduction it needs for gate t: after fetching the correct pushed key from the down-sweep, it can then compute an extra term in the gate gradient contribution, before applying its own gate to the running sum and handing it over to the previous lane. Most importantly, at the end of the up-sweep, the running sum contains exactly the key-side contributions for (86), which can then be directly incorporated into the key gradients.

Pseudocode for the resulting algorithm is provided in Alg. 4. The next phase accounts for contributions from off-diagonal tiles, and can start employing matrix multiplications on tensor cores.

Phase 2: off-diagonal tiles For an off-diagonal tile (I, J), the cumulative gate product separating a key $s \in J$ from a query $t \in I$ splits into three pieces: one internal to the key's tile (running from s to J_E, J-tile's end), one internal to the query's tile (running from I-tile's start, I_S to t), and the whole-tile products of any tiles lying in between. We can then split dependencies on s from those on t, and once keys and queries have been pushed to a common frame of reference (aligned with a tile's boundary), we can replace the step-by-step traversal performed within the diagonal tiles with more efficient matrix multiplications over whole tiles.

Note that each off-diagonal tile, in general, must produce both families of contributions: key-side ones for each gate $i \in I$ and query-side ones for each gate $i \in J$. The type of contribution also determines what quantities the Attention and Gram matrix gradients are multiplied by: queries for key-side contributions, keys for query-side ones. In practice, this matrix multiplication performs the *first* of the two reductions necessary to recover the full gradient contributions, deferring the remaining one to Phase 3. In the following, we describe how each family of quantities is computed in practice, and refer to Alg. 5 for the corresponding pseudocode.

KEY-SIDE $i \in J$. For key-side quantities, this phase of the algorithm is responsible for reducing across queries $t \in I$: it does so efficiently multiplying them by the (I, J)-th tile of the Attention and Gram matrices gradients. The first quantity to recover is this tile's contribution to the key-side gradients in (84):

$$
\begin{aligned}
[\delta^{\text{k-side}}_{I,J}\boldsymbol{K}]^\top_{s,-} &= \sum_{t=I_S}^{I_E} \boldsymbol{G}^\top_{s+1\to t}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right) \\
&= \boldsymbol{G}^\top_{s+1\to I_S-1}\left[\sum_{t=I_S}^{I_E}\Big(\underbrace{\boldsymbol{G}^\top_{I_S\to t}\boldsymbol{q}_t}_{=:{}^{I_S\leftarrow}\boldsymbol{q}_t}[\delta\boldsymbol{A}^{qk}]_{t,s} + \underbrace{\boldsymbol{G}^\top_{I_S\to t}\boldsymbol{k}_t}_{=:{}^{I_S\leftarrow}\boldsymbol{k}_t}[\delta\boldsymbol{A}^{kk}]_{t,s}\Big)\right] \\
&= \boldsymbol{G}^\top_{s+1\to I_S-1}\Big[\underbrace{[\delta\boldsymbol{A}^{qk}]^\top_{I,J}[{}^{I_S\leftarrow}\boldsymbol{Q}]_I + [\delta\boldsymbol{A}^{kk}]^\top_{I,J}[{}^{I_S\leftarrow}\boldsymbol{K}]_I}_{=:[\delta^k\,{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_{I,J}}\Big]_{s,-} \\
&= \boldsymbol{G}^\top_{s+1\to I_S-1}\Big[[\delta^k\,{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_{I,J}\Big]_{s,-}.
\end{aligned}
\tag{88}
$$

Algorithm 4: Matrix assembly backward – Phase 1: diagonal tiles

Input: Queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$, transition factors $\boldsymbol{G}_t^{(A)}, \boldsymbol{G}_t^{(B)}$, cotangents $\delta\boldsymbol{A}^{qk}, \delta\boldsymbol{A}^{kk}$

// Thread t loads its info

Read block-local quantities $\boldsymbol{G}_t^{(A)} \leftarrow \boldsymbol{G}_t^{(bA)}, \boldsymbol{G}_t^{(B)} \leftarrow \boldsymbol{G}_t^{(bB)}, \boldsymbol{q}_t \leftarrow \boldsymbol{q}_t\big|_{(b)}, \boldsymbol{k}_t \leftarrow \boldsymbol{k}_t\big|_{(b)}$;

// Phase 1 – Diagonal walk

// - Down-sweep

$\boldsymbol{x}_t \leftarrow \boldsymbol{k}_t$; // init traveling state: un-pushed key

for $d = 1$ **to** $|I|$ **do**

  $\boldsymbol{x}_t \leftarrow \texttt{shuffle}(\boldsymbol{x}_{t-1})$; // systolic shift: only x_t travels

  $\boldsymbol{x}_t \leftarrow \mathrm{vec}\left(\boldsymbol{G}_t^{(A)}\,\mathrm{mat}\,(\boldsymbol{x}_t)\,(\boldsymbol{G}_t^{(B)})^\top\right)$; // each lane applies its own gate

  if $t - d$ within tile **then**

    Accumulate $[\delta^{\text{q-side}}(\boldsymbol{Q}, \boldsymbol{K})]_{t,-} \mathrel{+}= [\delta(\boldsymbol{A}^{qk}, \boldsymbol{A}^{kk})]_{t,t-d}\boldsymbol{x}_t$;

    Store $\overrightarrow{\boldsymbol{k}}_{t-d\to t} = \boldsymbol{x}_t$; // store partially-pushed keys for up-sweep

// - Up-sweep

$\boldsymbol{y}_t \leftarrow \boldsymbol{0}$; // init traveling state: adjoint

for $d = |I|$ **down to** 1 **do**

  $\boldsymbol{y}_t \mathrel{+}= \boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,t-d} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,t-d}$; // include adjoint contribution

  if $t - d$ within tile **then**

    Read $\overrightarrow{\boldsymbol{k}}_{t-d\to t-1}$; // stored during down-sweep

    Accumulate $\delta^{\text{Attn}}\boldsymbol{G}_t^{(A)} \mathrel{+}= \mathrm{mat}\,(\boldsymbol{y}_t)\,\boldsymbol{G}_t^{(B)}\,\mathrm{mat}\left(\overrightarrow{\boldsymbol{k}}_{t-d\to t-1}\right)^\top$;

    Accumulate $\delta^{\text{Attn}}\boldsymbol{G}_t^{(B)} \mathrel{+}= \mathrm{mat}\,(\boldsymbol{y}_t)^\top\,\boldsymbol{G}_t^{(A)}\,\mathrm{mat}\left(\overrightarrow{\boldsymbol{k}}_{t-d\to t-1}\right)$;

  $\boldsymbol{y}_t \leftarrow \mathrm{vec}\left((\boldsymbol{G}_t^{(A)})^\top\,\mathrm{mat}\,(\boldsymbol{y}_t)\,\boldsymbol{G}_t^{(B)}\right)$; // each lane applies its own gate

  $\boldsymbol{y}_t \leftarrow \texttt{shuffle}(\boldsymbol{y}_{t+1})$; $(\boldsymbol{y}_{15} \leftarrow \boldsymbol{0})$; // systolic shift: only y_t travels

Accumulate $[\delta^{\text{k-side}}\boldsymbol{K}]_{t,-} \mathrel{+}= \boldsymbol{y}_t$; // terminal state: sum over $s > t$

The terms in $[\delta^{k}\overleftarrow{\tilde{\boldsymbol{Q}}}^{I_S}]_{I,J}$ are computed by first pushing the I-th tile's queries and keys to its start I_S, and then multiplying the resulting tensors by the (I, J)-th tile of the Attention and Gram gradient matrices, respectively. A final push to the corresponding index in the J-th tile provides the required gradient contributions.

Next we must recover the gate gradients. For illustrating this part, we reference (67) and consider only the first Kronecker factor: its gradient contributions pertaining to the (I, J) tile considered

satisfy:

$$\begin{aligned}
\delta^{\text{k-side}}_{I,J}\boldsymbol{G}^{(A)}_i &= \left(\boldsymbol{G}^{(A)}_{i+1\to I_S-1}\right)^\top \left[\sum_{t=I_S}^{I_E}\sum_{s=J_S}^{i-1} \underbrace{\left(\boldsymbol{G}^{(A)}_{I_S\to t}\right)^\top \operatorname{mat}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)\boldsymbol{G}^{(B)}_{I_S\to t}}_{=\operatorname{mat}\left({}^{I_S\leftarrow}\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s} + {}^{I_S\leftarrow}\boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)} \right. \\
&\qquad \left. \underbrace{\boldsymbol{G}^{(B)}_{s+1\to I_S-1}\operatorname{mat}\left(\boldsymbol{k}_s\right)^\top}_{=\operatorname{mat}\left(\overrightarrow{\boldsymbol{k}}^{(B),I_S-1}_s\right)^\top}\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top\right] \\
&= \left(\boldsymbol{G}^{(A)}_{i+1\to I_S-1}\right)^\top \left[\sum_{s=J_S}^{i-1} \underbrace{\operatorname{mat}\left(\sum_{t=I_S}^{I_E} {}^{I_S\leftarrow}\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s} + \sum_{t=I_S}^{I_E} {}^{I_S\leftarrow}\boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)}_{=\operatorname{mat}\left(\left[[\delta^k\,{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_{I,J}\right]_{s,-}\right)} \right. \\
&\qquad \left. \operatorname{mat}\left(\overrightarrow{\boldsymbol{k}}^{(B),I_S-1}_s\right)^\top\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top\right] \\
&= \left(\boldsymbol{G}^{(A)}_{i+1\to J_E}\right)^\top \left[\sum_{s=J_S}^{i-1}\left(\boldsymbol{G}^{(A)}_{J_E+1\to I_S-1}\right)^\top \operatorname{mat}\left(\left[[\delta^k\,{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_{I,J}\right]_{s,-}\right) \right. \\
&\qquad \left. \operatorname{mat}\left(\left[[\overrightarrow{\boldsymbol{K}}^{(B),I_S-1}]_J\right]_{s,-}\right)^\top\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top\right].
\end{aligned} \tag{89}$$

Note that the same quantity $[\delta^k\,{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_{I,J}$ we computed for (88) appears here as well, and can effectively be recycled. In this case, however, this quantity is tested against the corresponding $\boldsymbol{k}_s$, partially pushed to right before the I-tile start, by applying *only the second Kronecker factor* cumulative product $\boldsymbol{G}^{(B)}_{s+1\to I_S-1}$. This produces this tile's adjoints for $\boldsymbol{G}^{(A)}_i$:

$$\begin{aligned}
[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{I,J} \coloneqq &\left(\boldsymbol{G}^{(A)}_{J_E+1\to I_S-1}\right)^\top * \\
&\operatorname{mat}\left([\delta\boldsymbol{A}^{qk}]^\top_{I,J}[{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_I + [\delta\boldsymbol{A}^{kk}]^\top_{I,J}[{}^{I_S\leftarrow}\overrightarrow{\boldsymbol{K}}]_I\right) * \operatorname{mat}\left([\overrightarrow{\boldsymbol{K}}^{(B),I_S-1}]_J\right)^\top,
\end{aligned} \tag{90}$$

where $*$ denotes (broadcast) batched matrix multiplication. The resulting adjoint $[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{I,J}$ is a 3D tensor, containing one $\hat{m}\times\hat{m}$ matrix (the same dimension of a Kronecker factor) per each element s in the tile. With these quantities defined, we can compactly write the gate gradient contributions as

$$\delta^{\text{k-side}}_{I,J}\boldsymbol{G}^{(A)}_i = \left(\boldsymbol{G}^{(A)}_{i+1\to J_E}\right)^\top \sum_{s=J_S}^{i-1}\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{I,J}\right]_{s,-,-}\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top. \tag{91}$$

Following an analogous procedure, we can rewrite also the gradients for the second Kronecker factor:

$$\delta^{\text{k-side}}_{I,J}\boldsymbol{G}^{(B)}_i = \left(\boldsymbol{G}^{(B)}_{i+1\to J_E}\right)^\top \sum_{s=J_S}^{i-1}\left[[\boldsymbol{F}^{(B)}_{\text{k-side}}]_{I,J}\right]_{s,-,-}\left(\boldsymbol{G}^{(B)}_{s+1\to i-1}\right)^\top, \tag{92}$$

with the corresponding adjoint $[\boldsymbol{F}^{(B)}_{\text{k-side}}]_{I,J}$ this time being given by

$$\begin{aligned}
[\boldsymbol{F}^{(B)}_{\text{k-side}}]_{I,J} \coloneqq &\left(\boldsymbol{G}^{(B)}_{J_E+1\to I_S-1}\right)^\top * \\
&\operatorname{mat}\left([\delta\boldsymbol{A}^{qk}]^\top_{I,J}[{}^{I_S\leftarrow}\tilde{\boldsymbol{Q}}]_I + [\delta\boldsymbol{A}^{kk}]^\top_{I,J}[{}^{I_S\leftarrow}\overrightarrow{\boldsymbol{K}}]_I\right)^\top * \operatorname{mat}\left([\overrightarrow{\boldsymbol{K}}^{(A),I_S-1}]_J\right).
\end{aligned} \tag{93}$$

With these manipulations, we manage to reduce the sum over t to a sequence of efficient tensor multiplications, leaving only the sum over s still missing before we can recover the full gradient. This will be efficiently computed in Phase 3 of the algorithm, but first we provide the analogous derivations for the query-side contributions.

Algorithm 5: Matrix assembly backward – Phase 2: off-diagonal tiles ($I > J$)

Input: Keys $\boldsymbol{K}$, queries $\boldsymbol{Q}$, gates products $\boldsymbol{G}_{I_S \to t}, \boldsymbol{G}_{s+1 \to J_E}$, gradients $\delta\boldsymbol{A}^{qk}, \delta\boldsymbol{A}^{kk}$

// Lane i covers both tile positions: queries $t = I_S + i$, key $s = J_S + i$

Read block-local $(\boldsymbol{q}_t, \boldsymbol{k}_t) \leftarrow (\boldsymbol{q}_t, \boldsymbol{k}_t)\big|_{(b)}$ from row tile I and $\boldsymbol{k}_s \leftarrow \boldsymbol{k}_s\big|_{(b)}$ from column tile J;

// Phase 2 – per-tile key-side and query-side adjoint computations

// - Push queries and keys to common frame I_S

$({}^{I_S\leftarrow}\boldsymbol{q}_t, {}^{I_S\leftarrow}\boldsymbol{k}_t) \leftarrow \mathrm{vec}\left((\boldsymbol{G}^{(A)}_{I_S \to t})^\top \mathrm{mat}\left((\boldsymbol{q}_t, \boldsymbol{k}_t)\right) \boldsymbol{G}^{(B)}_{I_S \to t}\right)$; // keys/queries backward to I_S

$\overrightarrow{\boldsymbol{k}}^{I_S-1}_s \leftarrow \mathrm{vec}\left(\boldsymbol{G}^{(A)}_{s+1 \to I_S-1} \mathrm{mat}\left(\boldsymbol{k}_s\right) (\boldsymbol{G}^{(B)}_{s+1 \to I_S-1})^\top\right)$; // keys forward to I_S

// - Key-side: apply Attention and Gram matrices gradients to queries, reduce over t

$[\delta^k {}^{I_S\leftarrow}\boldsymbol{Q}]_{I,J} \leftarrow \mathtt{MMA}([\delta\boldsymbol{A}^{qk}]^\top_{I,J}, [{}^{I_S\leftarrow}\boldsymbol{Q}]_I) + \mathtt{MMA}([\delta\boldsymbol{A}^{kk}]^\top_{I,J}, [{}^{I_S\leftarrow}\boldsymbol{K}]_I)$;

Accumulate $[\delta^{\text{k-side}}\boldsymbol{K}]^\top_{s,-}$ += $\mathrm{vec}\left((\boldsymbol{G}^{(A)}_{s+1 \to I_S-1})^\top \mathrm{mat}\left(\left[[\delta^k {}^{I_S\leftarrow}\boldsymbol{Q}]_{I,J}\right]_{s,-}\right) \boldsymbol{G}^{(B)}_{s+1 \to I_S-1}\right)$;

Accumulate $[\boldsymbol{F}^{(A)}_{\text{k-side}}]_J$ += $(\boldsymbol{G}^{(B)}_{J_E+1 \to I_S-1})^\top * \mathrm{mat}\left([\delta^k {}^{I_S\leftarrow}\boldsymbol{Q}]_{I,J}\right) * \mathrm{mat}\left([\overrightarrow{\boldsymbol{K}}^{(A),I_S-1}]_J\right)^\top$;

Accumulate $[\boldsymbol{F}^{(B)}_{\text{k-side}}]_J$ += $(\boldsymbol{G}^{(B)}_{J_E+1 \to I_S-1})^\top * \mathrm{mat}\left([\delta^k {}^{I_S\leftarrow}\boldsymbol{Q}]_{I,J}\right)^\top * \mathrm{mat}\left([\overrightarrow{\boldsymbol{K}}^{(A),I_S-1}]_J\right)$;

// - Query-side: apply Attention and Gram matrices gradients to keys, reduce over s

$[(\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}, \delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1})]_{I,J} \leftarrow \mathtt{MMA}([(\delta\boldsymbol{A}^{qk}, \delta\boldsymbol{A}^{kk})]_{I,J}, [\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J)$;

Accumulate $[(\delta^{\text{q-side}}\boldsymbol{Q}, \delta^{\text{q-side}}\boldsymbol{K})]^\top_{t,-}$ += $\boldsymbol{G}_{I_S \to t}[(\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}, \delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1})]^\top_{t,-}$;

Accumulate $[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I$ += $\mathrm{mat}\left([{}^{I_S,(B)\leftarrow}\boldsymbol{Q}]_I\right) * \mathrm{mat}\left([\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top$
$+ \mathrm{mat}\left([{}^{I_S,(B)\leftarrow}\boldsymbol{K}]_I\right) * \mathrm{mat}\left([\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top$;

Accumulate $[\boldsymbol{F}^{(B)}_{\text{q-side}}]_I$ += $\mathrm{mat}\left([{}^{I_S,(A)\leftarrow}\boldsymbol{Q}]_I\right)^\top * \mathrm{mat}\left([\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)$
$+ \mathrm{mat}\left([{}^{I_S,(A)\leftarrow}\boldsymbol{K}]_I\right)^\top * \mathrm{mat}\left([\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)$;

QUERY-SIDE $i \in I$. This represents the dual case to the key-side illustrated above: here, we reduce across keys, multiplying them by tiles of the Attention and Gram gradient matrices. In this case, the first quantity to recover is this tile's contribution to the query-side gradients in (84):

$$
\begin{aligned}
[\delta^{\text{q-side}}_{I,J}\boldsymbol{Q}]^\top_{t,-} &= \sum_{s=J_S}^{J_E} [\delta\boldsymbol{A}^{qk}]_{t,s}\boldsymbol{G}_{s+1 \to t}\boldsymbol{k}_s = \boldsymbol{G}_{I_S \to t}\sum_{s=J_S}^{J_E} [\delta\boldsymbol{A}^{qk}]_{t,s}\underbrace{\boldsymbol{G}_{s+1 \to I_S-1}\boldsymbol{k}_s}_{=\overrightarrow{\boldsymbol{k}}^{I_S-1}_s} \\
&= \boldsymbol{G}_{I_S \to t}\Big[\underbrace{[\delta\boldsymbol{A}^{qk}]_{I,J}[\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J}_{=:[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}}\Big]^\top_{t,-} = \boldsymbol{G}_{I_S \to t}\Big[[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\Big]^\top_{t,-} \\
[\delta^{\text{q-side}}_{I,J}\boldsymbol{K}]^\top_{t,-} &= \sum_{s=J_S}^{J_E} [\delta\boldsymbol{A}^{kk}]_{t,s}\boldsymbol{G}_{s+1 \to t}\boldsymbol{k}_s = \boldsymbol{G}_{I_S \to t}\sum_{s=J_S}^{J_E} [\delta\boldsymbol{A}^{kk}]_{t,s}\boldsymbol{G}_{s+1 \to I_S-1}\boldsymbol{k}_s \\
&= \boldsymbol{G}_{I_S \to t}\Big[\underbrace{[\delta\boldsymbol{A}^{kk}]_{I,J}[\overrightarrow{\boldsymbol{K}}^{I_S-1}]_J}_{=:[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}}\Big]^\top_{t,-} = \boldsymbol{G}_{I_S \to t}\Big[[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\Big]^\top_{t,-}
\end{aligned}
\tag{94}
$$

Similarly to the key-side contributions, computing the terms in $[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}$ and $[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}$ requires first pushing the J-th tile's keys to right before the I-th tile start, and then multiplying the

resulting tensor by the (I, J)-th tile of the Attention and Gram gradient matrices, respectively. To recover the actual gradient contributions, we finally push each resulting vector to the correct index in the I-th tile.

Also the gate gradients follow a similar derivation as for the key-side: considering again only the first Kronecker factor in (67), its gradient contributions from (I, J) tile are given by

$$\begin{aligned}
\delta^{\text{q-side}}_{I,J}\boldsymbol{G}^{(A)}_i &= \Bigg[\sum_{t=i}^{I_E}\sum_{s=J_S}^{J_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top \operatorname{mat}\left(\boldsymbol{q}_t[\delta\boldsymbol{A}^{qk}]_{t,s} + \boldsymbol{k}_t[\delta\boldsymbol{A}^{kk}]_{t,s}\right)\underbrace{\boldsymbol{G}^{(B)}_{i+1\to t}\boldsymbol{G}^{(B)}_i\boldsymbol{G}^{(B)}_{I_S\to i-1}}_{=\boldsymbol{G}^{(B)}_{I_S\to t}} \\
&\qquad \underbrace{\boldsymbol{G}^{(B)}_{s+1\to I_S-1}\operatorname{mat}\left(\boldsymbol{k}_s\right)^\top\left(\boldsymbol{G}^{(A)}_{s+1\to I_S-1}\right)^\top}_{=\operatorname{mat}\left(\overrightarrow{\boldsymbol{k}}^{I_S-1}_s\right)^\top}\Bigg]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top \\
&= \sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\Bigg[\underbrace{\operatorname{mat}\left(\boldsymbol{q}_t\right)\boldsymbol{G}^{(B)}_{I_S\to t}}_{=\operatorname{mat}\left({}^{I_S,(B)}\overleftarrow{\boldsymbol{q}}_t\right)}\underbrace{\operatorname{mat}\left(\sum_{s=J_S}^{J_E}[\delta\boldsymbol{A}^{qk}]_{t,s}\,\overrightarrow{\boldsymbol{k}}^{I_S-1}_s\right)^\top}_{=\operatorname{mat}\left(\left[[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right]_{t,-}\right)^\top} \\
&\qquad + \underbrace{\operatorname{mat}\left(\boldsymbol{k}_t\right)\boldsymbol{G}^{(B)}_{I_S\to t}}_{\operatorname{mat}\left({}^{I_S,(B)}\overleftarrow{\boldsymbol{k}}_t\right)}\underbrace{\operatorname{mat}\left(\sum_{s=J_S}^{J_E}[\delta\boldsymbol{A}^{kk}]_{t,s}\,\overrightarrow{\boldsymbol{k}}^{I_S-1}_s\right)^\top}_{=\operatorname{mat}\left(\left[[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right]_{t,-}\right)^\top}\Bigg]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top \\
&= \sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\Bigg[\operatorname{mat}\left(\left[{}^{I_S,(B)}\overleftarrow{\boldsymbol{Q}}]_I\right]_{t,-}\right)\operatorname{mat}\left(\left[[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right]_{t,-}\right)^\top \\
&\qquad + \operatorname{mat}\left(\left[{}^{I_S,(B)}\overleftarrow{\boldsymbol{K}}]_I\right]_{t,-}\right)\operatorname{mat}\left(\left[[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right]_{t,-}\right)^\top\Bigg]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top.
\end{aligned} \tag{95}$$

Even in this case, we can reuse the $[\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}$ and $[\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}$ quantities computed for (94). Unlike for the key-side, though, the Attention and Gram matrix terms carry two different left factors, ${}^{I_S,(B)}\overleftarrow{\boldsymbol{q}}_t$ and ${}^{I_S,(B)}\overleftarrow{\boldsymbol{k}}_t$, and cannot be collapsed into a single matrix product: as a consequence, we need two matrix multiplications per tile, rather than one. This notwithstanding, we can still collect the resulting terms to recover this tile's adjoints,

$$\begin{aligned}
[\boldsymbol{F}^{(A)}_{\text{q-side}}]_{I,J} = \operatorname{mat}\left(\left[{}^{I_S,(B)}\overleftarrow{\boldsymbol{Q}}]_I\right)\ *\ \operatorname{mat}\left([\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top \\
+\operatorname{mat}\left(\left[{}^{I_S,(B)}\overleftarrow{\boldsymbol{K}}]_I\right)\ *\ \operatorname{mat}\left([\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top,
\end{aligned} \tag{96}$$

where $\boldsymbol{F}^{(A)}_{\text{q-side}}$ is again a 3D tensor holding one $\hat{m}\times\hat{m}$ matrix for each element t of this tile, so that

$$\delta^{\text{q-side}}_{I,J}\boldsymbol{G}^{(A)}_i = \left[\sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_{I,J}\right]_{t,-,-}\right]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top. \tag{97}$$

The second Kronecker factor follows a similar derivation, with adjoint given by:

$$\begin{aligned}
[\boldsymbol{F}^{(B)}_{\text{q-side}}]_{I,J} = \operatorname{mat}\left(\left[{}^{I_S,(A)}\overleftarrow{\boldsymbol{Q}}]_I\right)^\top\ *\ \operatorname{mat}\left([\delta^q\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top \\
+\operatorname{mat}\left(\left[{}^{I_S,(A)}\overleftarrow{\boldsymbol{K}}]_I\right)^\top\ *\ \operatorname{mat}\left([\delta^k\overrightarrow{\boldsymbol{K}}^{I_S-1}]_{I,J}\right)^\top,
\end{aligned} \tag{98}$$

which allows to write the gradient contributions compactly as

$$\delta^{\text{q-side}}_{I,J}\boldsymbol{G}^{(B)}_i = \left[\sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(B)}_{i+1\to t}\right)^\top\left[[\boldsymbol{F}^{(B)}_{\text{q-side}}]_{I,J}\right]_{t,-,-}\right]\left(\boldsymbol{G}^{(B)}_{I_S\to i-1}\right)^\top. \tag{99}$$

Mirroring the key-side derivations, we are left with only sums over t. These are completed in the last phase of the algorithm, described next.

Phase 3: epilogue To recover the final gradients with respect to a gate $i \in I$, we must combine terms from tiles K both *after* ($K > I$) and *before* it ($K < I$): the first provide key-side contributions, the latter query-side ones. In either case, this requires performing one last reduction over the missing index from Phase 2, that is, over s for key-side and over t for query-side. To outline this procedure, let us consider once again gradients for the first Kronecker factor of a gate $i \in I$. These are given by:

$$\begin{aligned}
\delta^{\text{Attn}}\boldsymbol{G}^{(A)}_i &= \sum_{K>I}\delta^{\text{k-side}}_{K,I}\boldsymbol{G}^{(A)}_i + \sum_{K<I}\delta^{\text{q-side}}_{I,K}\boldsymbol{G}^{(A)}_i \\
&= \sum_{K>I}\left(\boldsymbol{G}^{(A)}_{i+1\to I_E}\right)^\top\sum_{s=I_S}^{i-1}\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{K,I}\right]_{s,-,-}\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top \\
&+ \sum_{K<I}\left[\sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_{I,K}\right]_{t,-,-}\right]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top \\
&= \left(\boldsymbol{G}^{(A)}_{i+1\to I_E}\right)^\top\sum_{s=I_S}^{i-1}\left[\sum_{K>I}\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{K,I}\right]_{s,-,-}\right]\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top \\
&+ \left[\sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\sum_{K<I}\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_{I,K}\right]_{t,-,-}\right]\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top.
\end{aligned} \tag{100}$$

Note that the innermost sum from the key-side and query-side contributions accumulate adjoints, respectively, over a *column* $[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I \coloneqq \sum_{K>I}[\boldsymbol{F}^{(A)}_{\text{k-side}}]_{K,I}$, or over a *row* of tiles $[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I \coloneqq$

Algorithm 6: Matrix assembly backward – Phase 3: epilogue

Input: Adjoints $[\boldsymbol{F}^{(A,B)}_{\text{k-side}}]_I$, $[\boldsymbol{F}^{(A,B)}_{\text{q-side}}]_I$ from Phase 2, gates $\boldsymbol{G}^{(A)}_i, \boldsymbol{G}^{(B)}_i$, cumulative products $\boldsymbol{G}_{i+1\to I_E}, \boldsymbol{G}_{I_S\to i-1}$

// Lane i owns gate token $i = I_S + i$; all sweeps are independent

// Phase 3 – affine sweeps within the tile

for $F \in \{A, B\}$ **do**

- // - Key-side: forward Hillis-Steele, gate right-multiplied
- $\boldsymbol{M}_i \leftarrow (\boldsymbol{G}^{(F)}_i)^\top$; $\boldsymbol{R}_i \leftarrow \left[[\boldsymbol{F}^{(F)}_{\text{k-side}}]_I\right]_{i-1}$; // init quantities for affine recursion
- $\boldsymbol{R} \leftarrow \texttt{HSfwd}(\boldsymbol{M}, \boldsymbol{R})$;
- Accumulate $\delta^{\text{Attn}}\boldsymbol{G}^{(F)}_i \mathrel{+}= \left(\boldsymbol{G}^{(F)}_{i+1\to I_E}\right)^\top \boldsymbol{R}_{i-1}$;
- // - Query-side: backward Hillis-Steele, gate left-multiplied
- $\boldsymbol{M}_i \leftarrow (\boldsymbol{G}^{(F)}_{i+1})^\top$; $\boldsymbol{R}_i \leftarrow \left[[\boldsymbol{F}^{(F)}_{\text{q-side}}]_I\right]_i$; // (re-)init quantities for affine recursion
- $\boldsymbol{R} \leftarrow \texttt{HSbwd}(\boldsymbol{N}, \boldsymbol{R})$;
- Accumulate $\delta^{\text{Attn}}\boldsymbol{G}^{(F)}_i \mathrel{+}= \boldsymbol{R}_i\left(\boldsymbol{G}^{(F)}_{I_S\to i-1}\right)^\top$;

$\sum_{K<I}[\boldsymbol{F}^{(A)}_{\text{q-side}}]_{I,K}$. In either case, the outermost sums obey a specific structure:

$$\begin{aligned}
&\sum_{s=I_S}^{i-1}\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I\right]_s\left(\boldsymbol{G}^{(A)}_{s+1\to i-1}\right)^\top\\
&=\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I\right]_{i-1}+\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I\right]_{i-2}\left(\boldsymbol{G}^{(A)}_{i-1}\right)^\top+\ldots+\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I\right]_{I_S}\left(\boldsymbol{G}^{(A)}_{I_S+1\to i-1}\right)^\top,\quad\text{and}\\
&\sum_{t=i}^{I_E}\left(\boldsymbol{G}^{(A)}_{i+1\to t}\right)^\top\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I\right]_t\\
&=\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I\right]_i+\left(\boldsymbol{G}^{(A)}_{i+1}\right)^\top\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I\right]_{i+1}+\ldots+\left(\boldsymbol{G}^{(A)}_{i+1\to I_E}\right)^\top\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I\right]_{I_E},
\end{aligned}\tag{101}$$

implying that they can be obtained as the result of affine recursions, the one for the key-side running *forward* across the I-th tile, the one for the query-side running *backward*. These recursions are given by, respectively

$$\begin{aligned}
\boldsymbol{R}^{(A)}_{\text{k-side},r}&=\boldsymbol{R}^{(A)}_{\text{k-side},r-1}(\boldsymbol{G}^{(A)}_r)^\top+\left[[\boldsymbol{F}^{(A)}_{\text{k-side}}]_I\right]_r,&&\text{with}\quad\boldsymbol{R}^{(A)}_{\text{k-side},I_S-1}=\boldsymbol{0},\quad\text{and}\\
\boldsymbol{R}^{(A)}_{\text{q-side},r}&=(\boldsymbol{G}^{(A)}_{r+1})^\top\boldsymbol{R}^{(A)}_{\text{q-side},r+1}+\left[[\boldsymbol{F}^{(A)}_{\text{q-side}}]_I\right]_r,&&\text{with}\quad\boldsymbol{R}^{(A)}_{\text{q-side},I_E+1}=\boldsymbol{0},
\end{aligned}\tag{102}$$

and can both be efficiently solved in parallel via Hillis-Steele. This allows us to finally write

$$\delta^{\text{Attn}}\boldsymbol{G}^{(A)}_i=\left(\boldsymbol{G}^{(A)}_{i+1\to I_E}\right)^\top\boldsymbol{R}^{(A)}_{\text{k-side},i-1}+\boldsymbol{R}^{(A)}_{\text{q-side},i}\left(\boldsymbol{G}^{(A)}_{I_S\to i-1}\right)^\top.\tag{103}$$

The procedure for recovering the gradient contributions to the *second* Kronecker factor is exactly equivalent: it suffices to swap the A superscript with B into the formulas above. We report pseudocode for the full Phase 3 in Alg. 6.

G.4 Kernel profiling

Having detailed the implementation of our novel chunk-wise parallel algorithms in App. G.1 and G.3, in this section we focus on empirically evaluating their performance on modern accelerators, both in terms of memory consumption and runtime. To anchor our results, we consider as baselines both GLA (Yang et al., 2024a) and KDA (Kimi Team et al., 2025). Since the latter can be loosely interpreted as GLA augmented with the Delta Rule correction, this allows for a closer comparison with both versions of our method, ΔBGLT and BGLT, respectively with and without Delta Rule.

To better contextualize the results in this section, we recall the impact of upgrading gates from *diagonal* matrices, like in the baselines considered, to *block*-diagonal ones. First of all, the memory footprint increases: while BGLT keeps this bounded using Kronecker-factorized blocks, overall the gate parameterization still requires $2\times$ the memory of its diagonal counterpart, as listed in Tab. 1. Secondly, the FLOP count also increases: applying a m-dimensional diagonal block reduces to m element-wise multiplications, while applying an m-dimensional Kronecker-factorized block involves 2 multiplications with matrices of size $\sqrt{m}$ (see App. B), at a FLOP cost of $\sim 2m\sqrt{m}$. On the one hand, as highlighted in Lahoti et al. (2026), this faster scaling of FLOPs with respect to memory is actually desirable: it boosts *arithmetic intensity*, moving the bottleneck away from memory bandwidth and allowing the layer to better saturate modern accelerators. On the other, this fundamental difference in computational density means that the BGLT application cannot realistically hope to match the runtime of GLA or KDA, especially considering the maturity of their kernels. Consequently, our primary goal is to show that the runtime overhead introduced by our block-diagonal gate remains acceptable, and is roughly comparable in magnitude to the slowdown introduced when moving from GLA to KDA.

We structure our profiling at different levels of granularity: first we provide memory allocation and throughput results for the whole model, then consider an individual BGLT layer, and finally zoom into a breakdown of each sub-operation involved with the application of the core BGLT recurrent cell.

G.4.1 MODEL LEVEL

We begin our evaluation at the model level, profiling the full 7B parameter models matching the configuration used for our language modeling experiments in Sec. 4. Alongside Transformer, GLA, and KDA baselines (the latter two using their FLA kernels (Yang & Zhang, 2024)), we benchmark both BGLT and ΔBGLT with the efficient CuTe kernels implementation used to train our models, and include also a naïve, unoptimized Triton implementation as a lower-bound reference to validate the efficacy of our hardware-aware optimizations.

Throughput In Fig. G.3, we report the forward, backward, and combined (fwd+bwd) throughput without gradient checkpointing, across various combinations of batch size B and sequence length T, varied so that $B \times T = 16k$. This is to better isolate the scaling behavior with respect to sequence length. As expected, the standard Transformer suffers throughput degradation at longer T, due to the quadratic complexity of softmax attention. This degradation is just starting at 16k, yet GLA already outpaces the Transformer, which still runs $1.20\times$ faster than ΔBGLT in the forward and backward pass. If we allow for gradient checkpointing, then a single B200 can fit $128k$ tokens, and at that scale the prefill throughput for the Transformer falls well below that of all linear models: we report these results in Fig. 4. In contrast, all Linear Transformers architectures maintain a remarkably stable throughput across T, although some minor degradation is still expected in light of the sequential chunk-wise state-update operation, as described in App. F. Also in line with our expectations, the throughput of BGLT trails behind that of KDA, although the relative slowdown remains highly tractable, being roughly comparable to that between GLA and KDA. Furthermore, the marked performance gap between our optimized kernels and the naïve Triton baseline supports the validity of our implementation.

Memory usage Alongside throughput, in Fig. G.4 we report the peak memory usage measured for our 7B models, corresponding to processing an input of shape $B = 4, T = 4096$. The measurement is partitioned into static allocations (weights and gradients) and dynamic allocations (activations and intermediate tensors). Because the Kronecker-factorized block-diagonal gates involve manipulating denser matrices than the diagonal variants, the raw activation memory footprint for BGLT is inherently heavier. To mitigate this and aid large-scale deployment, our implementation selectively recomputes certain intermediate quantities during the backward pass. Specifically, these are the SVD components of the various Kronecker factors $\boldsymbol{G}_t^{(b,A)}, \boldsymbol{G}_t^{(b,B)}$ (see App. C.2), the pushed queries and keys $\overleftarrow{\boldsymbol{Q}}_{[n]}, \overrightarrow{\boldsymbol{K}}_{[n]}, \overleftarrow{\boldsymbol{K}}_{[n]}$, and, in the case of ΔBGLT, the Delta-correction-related quantities $\boldsymbol{T}_{[n]}^{-1}, \boldsymbol{W}_{[n]}, \boldsymbol{U}_{[n]}$ (see App. F.1). Note that a similar approach is also followed by KDA, which explains their overall lower memory footprint with respect to GLA, even though it involves more complex computations. We report peak memory consumption for both standard and recomputed

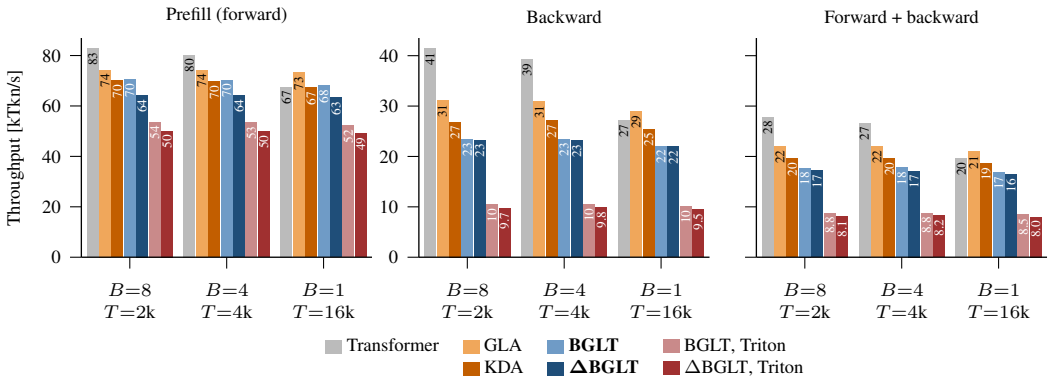


Figure G.3: **Model throughput at 7B** Prefill (left), backward (middle), and forward+backward (right) throughput for the various models used in our language modeling experiments (same configurations as Fig. 4), without activation checkpointing, varying input batch size B and sequence length T so that $B \times T = 16k$. Note the different scale for prefill. Wall-clock time, median of 10 steps after 3 warmup steps, on one B200 GPU.

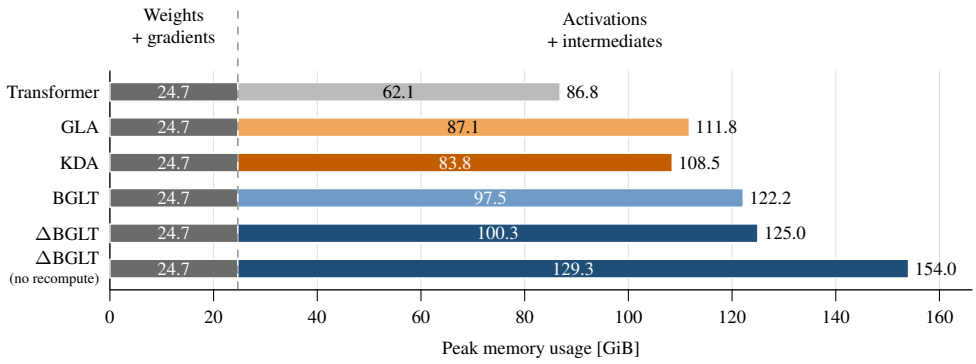


Figure G.4: **Training memory at 7B** Peak allocated memory during one training step of the various models used in our language modeling experiments (same configurations as Fig. G.3), without activation checkpointing, for input of shape $B = 4$, $T = 4096$. We highlight the slice of memory related to hosting the model itself (weights and gradients), separating it from the one required by activations and relevant intermediate quantities. ΔBGLT is also run without recomputation, its recurrence storing the operands its backward otherwise rebuilds. Optimizer state, embeddings and output head are not included. Measured on B200, `bf16` datatypes.

configurations to highlight this trade-off, but all runtime and throughput measurements are evaluated exclusively under the recomputation regime, since it represents the more realistic deployment setting for models of this scale.

Decoding Finally, we compare model runtime during autoregressive generation, profiling the per-token decode latency of the 7B models considered. In Fig. G.5, we report side-by-side runtime measurements, broken down by model component layer type, together with decoding throughput,

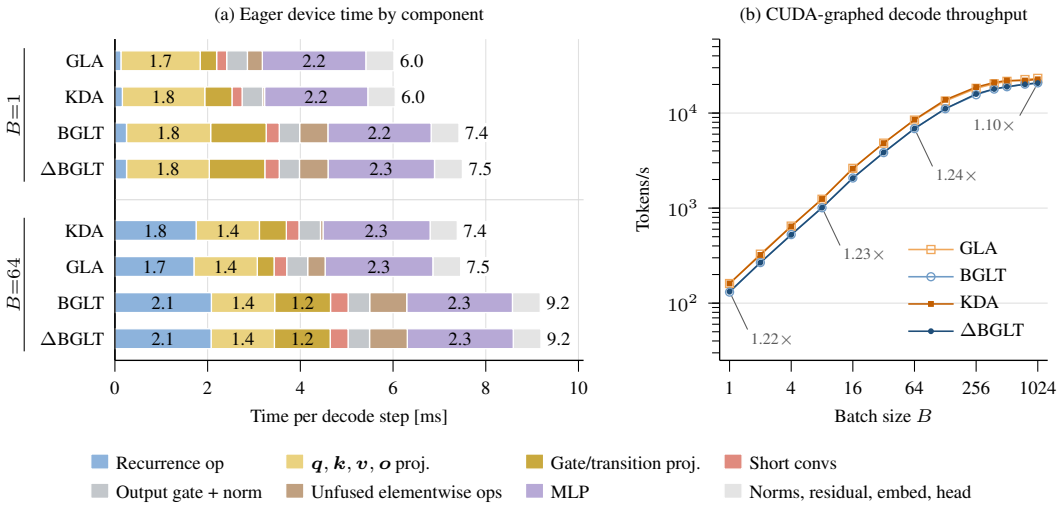


Figure G.5: **Decoding time at 7B** (a) Device time of one greedy decode step of the 7B models by component, compiled eager, at batch sizes 1 and 64, colored as inFig. C.1 (when components coincide). Unfused elementwise ops refer to the separate PyTorch kernels a layer runs outside its fused kernels (for BGLT, these are mainly casts and concatenation operations). (b) Decode throughput against batch size with each step replayed as a CUDA graph, up to the largest batch at which all four models fit in memory, with highlighted the ratio of per-token time of ΔBGLT with respect to KDA. All models run the same short-convolution kernel, and every Triton kernel is autotuned at the batch size it is timed at. One B200, `bf16` weights and activations, `fp32` recurrent state.

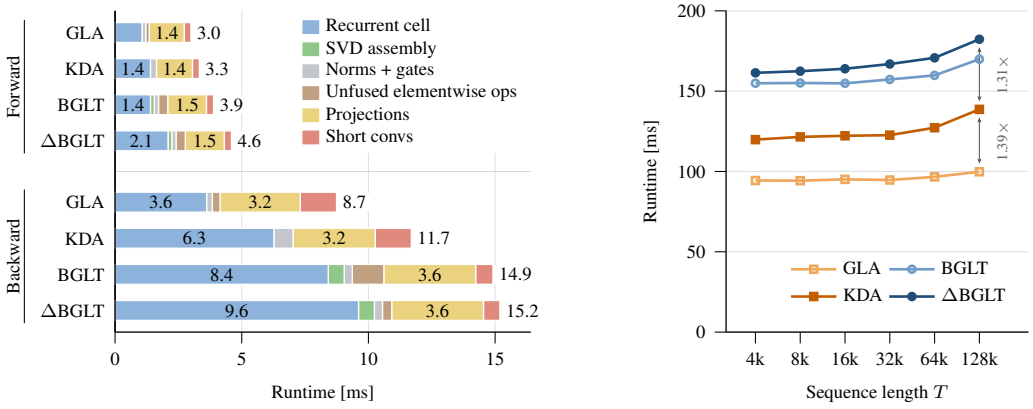
as we increase batch sizes up to hardware saturation ($B \approx 1024$). In the decoding regime, the performance bottleneck shifts further away from arithmetic compute and towards memory bandwidth. Because the memory overhead incurred by BGLT relative to KDA is minimal (only the gate parameter count increases by $2\times$), the additional arithmetic operations required to apply the transition blocks can effectively hide behind the memory fetch latency. This is reflected in our results: at saturation, the decoding time for BGLT essentially converges to that of KDA, achieving highly competitive generation speeds that measure only $\sim 1.10\times$ slower.

G.4.2 Layer level

Zooming in from the full model, we next analyze the execution time of an individual BGLT layer, as described in App. C.1 and illustrated in Fig. C.1.

In Fig. G.6a, we provide runtime comparisons for forward and backward passes through a single layer of the Linear RNNs considered in this work, broken down by architectural component. This allows us to isolate the exact computational overhead of BGLT, separating it from operations that are standard across all Linear RNNs. Unsurprisingly, the largest overhead stems from the actual RNN cell application, whereas execution times for the surrounding components (such as projections and convolutions) are roughly equivalent across all architectures. The larger share of unfused elementwise ops in the backward of BGLT is attributable to the gradient of the input gate β_t, which runs as separate PyTorch kernels when the Delta Rule is off, whereas ΔBGLT absorbs it into its fused kernels. We also highlight the additional cost incurred by BGLT for the SVD assembly of its Kronecker gate factors $\boldsymbol{G}_t^{(b,A)}, \boldsymbol{G}_t^{(b,B)}$, which is necessary for the L_2-normalization procedure outlined in App. C.2.

In Fig. G.6b, we further evaluate the total forward-plus-backward runtime through a single layer as a function of sequence length T. Overall, runtimes remain highly stable across varying sequence lengths, exhibiting only a minimal slope increase dictated by the sequential state update typical of chunk-wise parallel formulations. Most importantly, comparing the performance of the different Linear RNNs confirms our previous model-level results: while employing denser block-diagonal gates increases the execution time of the core cell, the resulting slowdown of BGLT relative to KDA remains contained, and is directly comparable in magnitude to the runtime penalty KDA pays over GLA for introducing the Delta Rule.



(a) Runtime by component

(b) Fwd+bwd runtime

Figure G.6: **Single-layer runtime at 7B** (a) Runtime for input shape $B = 4, T = 4096$, broken down by layer component as highlighted in Fig. C.1, measured via PyTorch profiler; unfused elementwise ops are as in Fig. G.5. (b) Wall-clock forward+backward runtime of back-to-back eager calls, varying sequence length T at a fixed $B \times T = 128k$ tokens ($B = 32$ at $T = 4k$ down to $B = 1$ at $T = 128k$). On a single B200, with `bf16` datatypes.

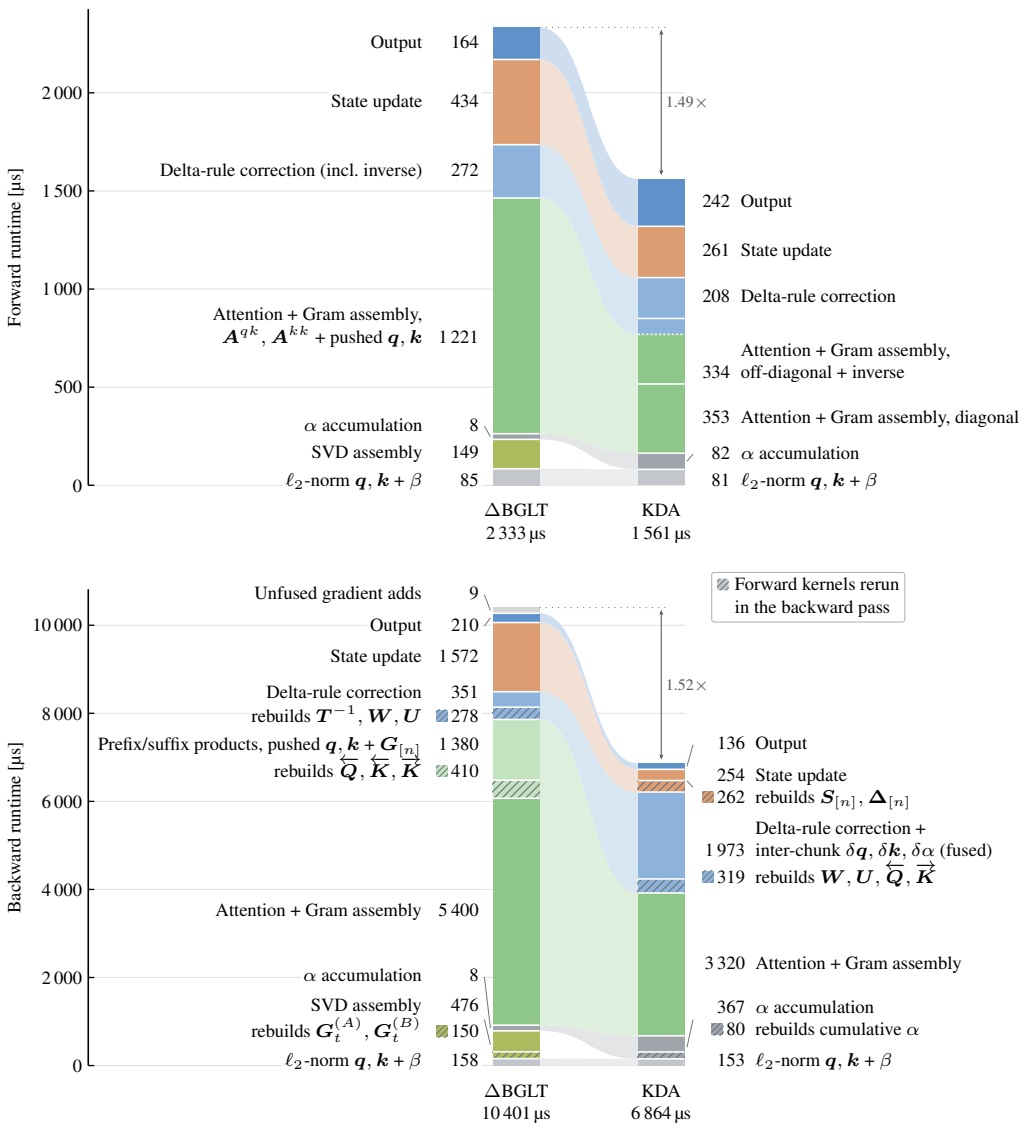


Figure G.7: **Recurrence runtime at 7B** Per-kernel timings breakdown for the forward (top) and backward (bottom) passes through the core Linear RNN cell chunk-wise parallel application, for input $B = 4$, $T = 4096$. Kernels for the two cells are loosely grouped by the step of the chunk-wise formulation they compute (App. F.1), colored as in Fig. F.2. Shaded segments represent quantities already assembled in the forward kernel, which are however discarded and re-assembled in the backward to save memory. Measured via PyTorch profiler, eager, on a single B200, with `bf16` datatypes.

G.4.3 RNN CELL LEVEL

Finally, we complete our analysis by zooming into the core BGLT recurrent cell application, further breaking down the timings from Fig. G.6a to split them across the individual operations conducted during its chunk-wise parallel application, following the steps highlighted in Fig. F.2. We focus on a comparison against KDA, but we point out that a direct 1-to-1 per-kernel match is not possible, due to structural differences in the way operations are fused within the two cells. For example, in ΔBGLT we cleanly decouple the assembly of the key-Gram matrix from its inversion: $\boldsymbol{A}_{[n]}^{qk}$ and $\boldsymbol{A}_{[n]}^{kk}$ are built together in the same kernel (see App. G.1), and $\boldsymbol{A}_{[n]}^{kk}$ is subsequently inverted in a separate kernel (see App. G.2). In contrast, KDA interleaves these operations: one kernel processes only the diagonal

tiles of both matrices, while a second kernel is responsible for both computing the off-diagonal tiles and performing Gram inversion. As an additional confounder, for sub-kernel timing attribution we are forced to run the kernels on eager compilation mode, which introduces further discrepancies with respect to the timings one can reasonably expect at deployment.

To facilitate comparisons while accounting for these discrepancies, we report per-kernel timings of the various operations involved in the application of the core RNN cell for both ΔBGLT and KDA, roughly mapping individual kernel components onto logically equivalent operations. The resulting diagrams in Fig. G.7 help visualize how execution time is distributed across the different operations in the two cells considered, for both the forward and backward passes. Even at this lowest level of analysis, the breakdown confirms our expectations. For operations that are mathematically identical across both architectures, such as the Delta Rule application, the execution times are essentially identical. The main runtime divergences appear in the operations most closely linked to the gate applications, and particularly in the kernel responsible for the Attention and Gram matrix assembly, due to the additional overhead associated with our inverse-free, generalized approach (App. G.1). Nonetheless, the observed overhead remains remarkably contained, especially considering the different complexity associated with our assembly process. Overall, this acts as further validation that the theoretical synergies offered by our Kronecker formulation for BGLT successfully translate into hardware efficiency.