A. Related Work

Since the introduction of Self-Attention [61], significant research has been conducted to reduce its quadratic cost in processing long input sequences. As models and systems scale to million-token contexts, Attention's bottlenecks have become critical blockers to frontier agentic applications in coding, information gathering, and scientific discovery [6, 7, 29]. Prior works have proposed various approximation schemes to overcome these limitations. For example, Reformer [32] uses locality-sensitive hashing to group tokens with similar embeddings. This enables the model to attend only to a subset of tokens rather than the entire sequence. Other works equip Transformer models with "compressed" memory tokens that are updated dynamically and causally over sliding windows on entire sequence chunks [8, 41, 42]. While a lot of prior work have focused on reducing the quadratic complexity of Attention with sparse approximations [44, 73], this work focuses on linear approximations of Attention.

A.1. Linear Attention

Linear Attention methods approximate the Attention mechanism with constant-size recurrent dynamical systems [3, 9, 68, 71]. Numerous State-Space Model (SSM) variations have been proposed, ranging from those closely resembling Linear Attention [60] or Linear Time-Invariant dynamical systems [23, 74], to those introducing novel adaptive or gated state updates [9, 45, 68].

Despite their differences, all SSMs follow the same basic working principle inspired by classical state-space models [30]: they process the input sequence by maintaining a *fixed-size* state that acts as a compressed (lossy) representation of all processed tokens. Moreover, when implemented in hardware, the state must have finite precision and "fades away the past" as more samples are processed. Successful SSM layers typically employ hardware-aware implementations that efficiently utilize modern matrix multiplication accelerators through highly parallelizable and scalable primitives, including associative scans [11, 21], chunking mechanisms [9, 68], and techniques that avoid materializing the entire state in slow high-bandwidth memory [21].

From a modeling perspective, most Linear Attention implementations introduce data-dependent gating factors to control the speed of their "fading" memory, balancing expressivity with scalability. For example, the transition from Mamba to Mamba2 replaced channel-wise data-dependent gating with head-wise gating for better scalability and Tensor Cores utilization. Input-dependent Gating has been shown to empirically improve training stability [1, 71] and has driven the development of Linear Attention models (e.g., from S4 [22] to Mamba [21] and from DeltaNet [69] to Gated DeltaNet [71]). In our work, we demonstrate that gating emerges naturally as a consequence of solving a weighted least squares objective function, establishing a connection to the favorable numerical properties classically described in the adaptive filtering literature [39, 54, 55].

A.2. Hybrid State Space Attention Models

While extending the recurrent state in SSM layers has yielded performant models, they typically underperform on tasks requiring recall of information from the distant past [28, 64]. Hybrid State-Space Models address this limitation by complementing SSMs' "fading" state with Attention layers [10, 11, 18, 36]. Early architectures simply stacked SSMs and Attention layers with different blending ratios [9, 21, 64] or replaced full Attention layers with Sliding Window Attention [11]. More sophisticated designs have recently emerged [18, 75].

Notably, B'MOJO [75] complements SSMs' fading state with "eidetic" memory by combining SSMs with Sliding Window Attention (SWA) in a single layer. Within the window, tokens can attend to a selected set of past tokens that were deemed difficult to predict using an asynchronous causal selection mechanism. B'MOJO was the first hybrid model to propose a parallel fusion of SSM and SWA at the layer level. Subsequent works [2, 13] have shown this parallel fusion approach to be more performant (at equivalent compute) than the stacked approach of earlier works.

Thanks to their lower memory footprint and test-time scalability over long sequences, Hybrid architectures are expanding into long-range agentic tasks and have recently been trained with Reinforcement Learning at scale [5]. When coupled with system-level optimizations like prefix caching [46] and specialized inference engines [33], Hybrid models can increase the number of rollouts (exploration), thereby improving end-to-end performance in Reinforcement Learning loops.

B. Discussions and Limitations

Thanks to its expressive test-time ridge regression objective, Gated KalmaNet extends previous fading memory layers like Mamba2, LongHorn and Gated DeltaNet, all of which only depend on an instantaneous test-time objective. However, GKA is only optimal among linear memory layers, solving our test-time objective leveraging non-linear updates while still maintaining hardware efficiency and numerical stability is an interesting area for future research. Despite the efficient kernels we implemented, we believe even faster implementations of our idea are possible, e.g., via *sketching* (see Section L for

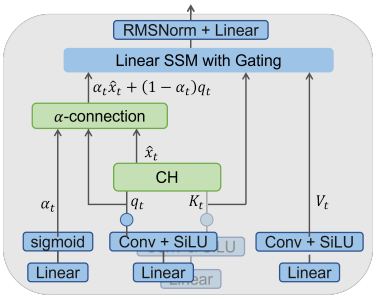


Figure 4. **Our GKA block**. Blue refers to established practices in the literature with the solid circles denote ℓ_2 normalization. Green components (CH and α-connection) are our proposals.

preliminary results). Finally, while we have showed promising results in combining GKA with Attention layers into Hybrid models (Section M), further scaling beyond 3B parameters models is required to validate GKA on more challenging real world problems.

C. Architectural Consideration

Our GKA layer in Fig. 4 includes two components (in green) on top of established practices (in blue). The CH component is described in Section 4.1, thus here we introduce the *α-connection*. First, the sigmoid activation ensures $\alpha_t \in [0, 1]$, so the output of the α-connection is a convex combination of the original query q_t and the output $\hat{x}_t$ of CH. Second, it plays a similar role to residual connection, which establishes a direct path that facilitates the gradient flow and improves training; we show this is indeed the case in Section J.3. Finally, the full architecture for GKA is the standard Transformer, with its attention layer replaced by the GKA layer.

D. Forward and Backward Passes of Chebyshev Iteration (Details)

In Section 4.2 we described our chunk-wise implementation of the CH method with adaptive regularization and gating. We now give full details omitted there.

D.1. Forward Pass

CH in Detail. We begin with describing the CH method (Algorithm 1) in more detail. Assume we have a linear system of equations $H\xi = q$ where H is a $D \times D$ positive definite matrix. We assume H has its all eigenvalues lie in the interval $[\mu, L]$ and the values of μ and L is known. Note that solving this system is equivalent to solving the following quadratic problem:

$$\min_{\xi \in \mathbb{R}^D} \frac{1}{2}\xi^\top H\xi - \xi^\top q. \tag{11}$$

The classic Chebyshev Iteration in its standard form is presented in Algorithm 1. In the initialization phase, we set $\rho = \frac{L-\mu}{L+\mu}$, which is the typical convergence rate of gradient descent applied to the above quadratic problem with stepsize $\frac{2}{L+\mu}$; vaguely speaking, in this setting, this stepsize choice is optimal (e.g., that allows gradient descent to converge the fastest possible). Algorithm 1 initializes two points, ξ_{-1} and ξ_0. Here ξ_{-1} is zero, and ξ_0 is a gradient step for (11) starting at ξ_{-1} and with stepsize $\frac{2}{L+\mu}$. The final component in initialization is the weight $\omega_0 = 2$. This is the starting point for the weight schedule recursion of ω_i in (weight schedule). Similarly, the initialization of ξ_{-1}, ξ_0 is where we start to compute ξ_i, whose update consists of (grad descent) and (momentum). Note that (grad descent) is with stepsize $2 \cdot \omega_i/(L+\mu)$. Since $\omega_i > 1$, this stepsize is strictly larger than $2/(L+\mu)$, the latter being the optimal stepsize for vanilla gradient descent. Such a large stepsize alone might not guarantee convergence, but it is balanced by the (momentum) term $\xi_{i-1} - \xi_{i-2}$ with positive weight $\omega_i - 1$ so that the convergence of the Chebyshev iterative method is ensured.

Numerical Stability Considerations. Now we analyze the numerical properties of the Chebyshev Iteration. The major computation consists of matrix-vector multiplication; in a batched parallel implementation, this turns out to be matrix-matrix multiplication. For this, the numerical accuracy is well controlled (e.g., in Triton we could specify the accuracy in `tl.dot`). The update of ω_i in (weight schedule) might raise numerical concerns as it involves division. That said, we show this division operates in a numerically well-behaved range as ω_i is decreasing with i yet lower bounded by 1:

Lemma 3. *For any r, we have $2 = \omega_0 \geq \cdots \geq \omega_r \geq \omega_1^* > 1$, where ω_1^* is defined as*

$$\omega_1^* := \frac{2(1-\sqrt{1-\rho^2})}{\rho^2}.$$

As a consequence, we have $4 - \rho^2\omega_i \in [2, 4]$ for all $i = 0, \ldots, r$.

Proof. If $L = \mu$, then H is a scaled identity matrix, and the algorithm is simplified a lot. So we assume $L > \mu$ in what follows. With $L > \mu > 0$ we have $\rho \in (0, 1)$. Since $\omega_0 = 2$, we have $4 - \rho^2\omega_0 \geq 2$ and therefore $0 < \omega_1 \leq 2$. Repeating this argument and we see $\omega_i \in (0, 2]$ for all i. By the definition of ω_i, to show $\omega_i \leq \omega_{i-1}$ is to show

$$\frac{4}{4 - \rho^2\omega_{i-1}} \leq \omega_{i-1} \Leftrightarrow g(\omega_i) \leq 0$$

where g is defined as $g(\omega) = \rho^2\omega^2 - 4\omega + 4$. Note that $g(\omega)$ has two roots, ω_1^*, as defined earlier, and $\omega_2^* = \frac{2(1+\sqrt{1-\rho^2})}{\rho^2}$; ω_1^*, ω_2^* are the two fixed points of the update (weight schedule). Observing that $\omega_0 = 2$ lies in the interval (ω_1^*, ω_2^*), and moreover, for any $i \geq 1$, if $\omega_{i-1} > \omega_1^*$ we must have

$$\omega_i = \frac{4}{4 - \rho^2\omega_{i-1}} > \frac{4}{4 - \rho^2\omega_1^*} = \omega_1^*.$$

This proves $\omega_i > \omega_1^*$ for all $i = 1, \ldots, r$. Next, since $\omega_0 = 2$ lies in the interval (ω_1^*, ω_2^*) where $g(\omega)$ decreases, therefore we have $\omega_1 \leq \omega_0$. Thus ω_1 lies in (ω_1^*, ω_2^*) again. We could then conclude inductively that $\omega_1^* < \omega_i \leq \omega_{i-1}$ for all $i = 1, \ldots, r$. □

From Lemma 3 we know that the update of ω_i in (weight schedule) would not create much numerical concern in a forward pass, as we have $\omega_i \in [1, 2]$ for all i. Furthermore, we can bound the rate at which ω_i converges to ω_1^*:

Lemma 4. *Define* $\kappa := \frac{L}{\mu}$. *For any* $i = 1, \dots, r$, *we have*

$$(\omega_i - \omega_1^*) \le R^i \cdot (\omega_0 - \omega_1^*),$$

where R *is defined as*

$$R := \frac{\kappa - 1}{\kappa + 1} \cdot \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}.$$

Proof. From the update rule of ω_i in (weight schedule) and the fixed point property of ω_1^*, we have

$$\begin{aligned}
\omega_i - \omega_1^* &= \frac{4}{4 - \rho^2 \omega_{i-1}} - \frac{4}{4 - \rho^2 \omega_1^*} \\
&= \frac{4\rho^2}{(4 - \rho^2 \omega_{i-1})(4 - \rho^2 \omega_1^*)} \cdot (\omega_{i-1} - \omega_1^*) \\
&\overset{\text{(i)}}{=} \frac{\rho^2 \omega_1^*}{4 - \rho^2 \omega_{i-1}} \cdot (\omega_{i-1} - \omega_1^*) \\
&\overset{\text{(ii)}}{\le} \frac{\rho^2 w_{i-1} \omega_1^*}{4} \cdot (\omega_{i-1} - \omega_1^*) \\
&\overset{\text{(iii)}}{\le} \left(1 - \sqrt{1 - \rho^2}\right) \cdot (\omega_{i-1} - \omega_1^*) \\
&\overset{\text{(iv)}}{=} \left(\frac{\kappa - 1}{\kappa + 1} \cdot \frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right) \cdot (\omega_{i-1} - \omega_1^*)
\end{aligned}$$

Here, (i) follows from the fact that ω_1^* is a fixed point, (ii) follows from Lemma 3 that $\omega_i \le \omega_{i-1}$, (iii) follows from the definition of ω_1^* and the fact $w_{i-1} \le 2$, and (iv) follows from the definitions of κ and ρ. The proof is concluded by unrolling the above recurrence. □

Remark 1. Here, we call R the linear convergence rate (or *contraction factor*) of ω_i to ω_1^*. First-order methods for solving $H\xi = q$ converge at most at a rate $R_a := \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$, and we see ω_i converges at an even faster rate. Numerically, assuming $\kappa = \frac{L}{\mu} = \frac{1.02}{0.02} = 51$, we then have:

$$\begin{aligned}
R \approx 0.7253, \quad R^5 \approx 0.2, \quad R^{10} \approx 0.04, \quad R^{20} \approx 0.0016, \quad R^{30} \approx 6 \times 10^{-5} \\
R_a \approx 0.7543, \quad R_a^5 \approx 0.244, \quad R_a^{10} \approx 0.0597, \quad R_a^{20} \approx 0.0036, \quad R_a^{30} \approx 0.0002.
\end{aligned}$$

Thus, with $\kappa = 51$, the update of ω_i in (weight schedule) converges in at most 20 iterations up to the bfloat16 precision.

D.2. Backward Pass

We now give details for backpropagation through the Chebyshev Iteration (Algorithm 1) via implicit differentiation.

Computing $\frac{dL}{dq_t}$ and $\frac{dL}{dk_t}$. First, we follow Table 1 and Lemma 2, and compute dq_t. Then, given the equation $(H_t + \lambda_t I)dq_t = dx_t^*$, we have that

$$d(H_t + \lambda_t I) = -dq_t (x_t^*)^\top. \tag{12}$$

Therefore $d\lambda_t = \mathrm{tr}(d(H_t + \lambda_t I)) = -(x_t^*)^\top dq_t$. Since we set $\lambda_t = a \cdot \|H_t\|_{\mathrm{F}}$, this indicates

$$dH_t = -dq_t (x_t^*)^\top - a \cdot \frac{H_t}{\|H_t\|_{\mathrm{F}}} \cdot \left((x_t^*)^\top dq_t\right). \tag{13}$$

Note that this expression of dH_t is *partial*: It accounts for the upstream gradient from dq_t only and one might think of the subsequent states all depend on H_t. We will accumulate the gradients later when needed.

Now, the recursion of H_t in (CH) implies

$$dk_i = \sum_{t \ge i} \left(dH_t + (dH_t)^\top\right) k_i \cdot \frac{\zeta_t}{\zeta_i} \tag{14}$$

$$= \sum_{t \ge i} \frac{\zeta_t}{\zeta_i} \left(-dq_t \otimes (x_t^*)^\top - x_t^* \otimes (dq_t)^\top + w_t H_t\right) k_i, \tag{15}$$

which proves Lemma 1. We refer the reader to Section D.2.1 for more detailed derivations of dq_t and dk_t.

Derivatives for Gating. In practice we often parameterize γ_t in the log space to ensure numerical stability. Thus, let us first revise our notations for this case. Let $g_t = \log \gamma_t$ and $G_t := \sum_{i=1}^{t} g_i = \log\left(\prod_{i=1}^{t} \gamma_i\right)$. Then the mask matrix M is

$$M_{i,j} = \begin{cases} \exp(G_j - G_i) & j \geq i; \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

Now, since for any $c = 1, \ldots, C$ we have

$$H_c = \exp(G_c) \cdot H_0 + \sum_{j=1}^{c} \exp(G_c - G_j) \cdot k_j k_j^\top, \tag{17}$$

for any G_i we have the following basic derivatives:

$$\frac{dH_c}{dG_i} = \begin{cases} 0 & c < i; \\ \exp(G_i) \cdot H_0 + \sum_{j=1}^{i-1} \exp(G_i - G_j) \cdot k_j k_j^\top & c = i; \\ -\exp(G_c - G_i) k_i k_i^\top & c > i, \end{cases} \qquad i = 1, \ldots, C, \quad c = 1, \ldots, C; \tag{18}$$

$$\frac{dH_{C+1}}{dG_i} = -\exp(G_{C+1} - G_i) k_i k_i^\top \qquad i = 1, \ldots, C \tag{19}$$

With dH_{C+1} being the aggregated gradient from the future, we have for $i = 1, \ldots, C$ that

$$dG_i = \sum_{c=i}^{C+1} \langle dH_c, \frac{dH_c}{dG_i} \rangle \tag{20}$$

$$= e^{G_i} \langle dH_i, H_0 \rangle + \sum_{j=1}^{i-1} e^{G_i - G_j} \langle dH_i, k_j k_j^\top \rangle - \sum_{c=i+1}^{C} e^{G_c - G_i} \langle dH_c, k_i k_i^\top \rangle - e^{G_{C+1} - G_i} \langle dH_{C+1}, k_i k_i^\top \rangle \tag{21}$$

$$= e^{G_i} \langle dH_i, H_0 \rangle + \sum_{j=1}^{i} e^{G_i - G_j} \langle dH_i, k_j k_j^\top \rangle - \sum_{c=i}^{C} e^{G_c - G_i} \langle dH_c, k_i k_i^\top \rangle - e^{G_{C+1} - G_i} \langle dH_{C+1}, k_i k_i^\top \rangle \tag{22}$$

$$dG_{C+1} = e^{G_{C+1}} \langle dH_{C+1}, H_0 \rangle + \sum_{j=1}^{C} e^{G_{C+1} - G_j} \langle dH_{C+1}, k_j k_j^\top \rangle \tag{23}$$

Note that in one of the above equations we add and subtract the term $\langle dH_i, k_i k_i^\top \rangle$, which will simplify the implementation.

Recall that $dH_t = -dq_t (x_t^*)^\top - \frac{1}{2} \cdot w_t H_t$ with $w_t = \frac{2a \cdot (x_t^*)^\top dq_t}{\|H_t\|_F}$. In computing the derivatives of G_i the first term $dq_t (x_t^*)^\top$ is the standard term that arises in that of (Linear-SSM), which we omit here. We now focus on the second term $\frac{1}{2} \cdot w_t H_t$. This implies the gradients dG_i and dG_{C+1} are partly given respectively by (using the notations in Section 4.2.2 and omitting some algebraic operations)

$$\frac{1}{2} \cdot \langle w_i H_i, H_i \rangle - \frac{1}{2} \cdot k_i^\top B_i k_i \qquad \text{and} \qquad \frac{1}{2} \cdot e^{G_C} \langle \widetilde{B}_{C+1}, H_0 \rangle + \frac{1}{2} \cdot \sum_{j=1}^{C} e^{G_C - G_j} \cdot k_j^\top \widetilde{B}_{C+1} k_j. \tag{24}$$

Computing the first term $\langle w_i H_i, H_i \rangle$ in parallel is easy by invoking the definition of w_i and the Frobenius norm of H_i we stored during the forward pass. Computing the quadratic terms $k_i^\top B_i k_i$ and $k_j^\top \widetilde{B}_{C+1} k_j$ in parallel is easy and follows from our computation of $B_i k_i$ and $\widetilde{B}_{C+1} k_i$ for dk_i in Section 4.2.2. Computing $\langle \widetilde{B}_{C+1}, H_0 \rangle$ is easy since we recompute the initial states H_0 of each chunk and have them available during the backward pass, while $\widetilde{B}_{C+1}$ is updated backwards in a for loop.

D.2.1. Computing $\frac{dL}{dq_t}$ and $\frac{dL}{dk_t}$.

In forward pass we solve

$$(H_t + \lambda_t I) x_t = q_t$$

$$\begin{aligned} x_t &= (H_t + \lambda_t I)^{-1} q_t \\ \implies dx_t &= \underbrace{(H_t + \lambda_t I)^{-1}}_{J_{q_t \to x_t}} dq_t \end{aligned} \tag{25}$$

Recall that the gradient is transpose of the Jacobian, thus we obtain

$$\frac{dL}{dq_t} = (H_t + \lambda_t I)^{-1} \frac{dL}{dx_t}. \tag{26}$$

Thus, we can obtain $\frac{dL}{dq_t}$ by running a Chebyshev iteration to solve (for z) the linear system of equations

$$(H_t + \lambda_t I) z = \frac{dL}{dx_t}.$$

Now we have

$$\begin{aligned} dx_t &= d(H_t + \lambda_t I)^{-1} q_t \\ dx_t &= -(H_t + \lambda_t I)^{-1} d(H_t + \lambda_t I)(H_t + \lambda_t I)^{-1} q_t \\ dx_t &= -(H_t + \lambda_t I)^{-1} d(H_t + \lambda_t I) x_t \\ &= (x_t^\top \otimes -(H_t + \lambda_t I)^{-1}) \text{vec}(d(H_t + \lambda_t I)) \end{aligned} \tag{27}$$

In the last equality we have used the identity $\text{vec}(ABC) = (C^\top \otimes A)\text{vec}(B)$.

Now we will compute the Jacobian of λ with respect to H_t:

$$\begin{aligned} \lambda_t &= a||H_t||_F \\ &= a\sqrt{\text{Tr}(H_t^\top H_t)} \\ \implies d\lambda &= ad\Big(\sqrt{\text{Tr}(H_t^\top H_t)}\Big) \\ &= a\frac{1}{2||H_t||_F} \text{Tr}((dH_t)^\top H_t + H_t^\top (dH_t)) \\ &= a\frac{1}{||H_t||_F} \text{vec}(H_t)^\top d\text{vec}(H_t) \end{aligned} \tag{28}$$

Substituting (28) in (27).

$$dx_t = (x_t^\top \otimes -(H_t + \lambda_t I)^{-1}) \Big(\text{vec}(dH_t) + \frac{a}{||H_t||_F} \text{vec}(I)\text{vec}(H_t)^\top \text{vec}(dH_t))\Big) \tag{29}$$

Thus, we can obtain $\text{vec}(\frac{dL}{dH_t})$ as,

$$\text{vec}(\frac{dL}{dH_t}) = (x_t \otimes -(H_t + \lambda_t I)^{-1}) \frac{dL}{dx_t} + \frac{a}{||H_t||_F} \text{vec}(H_t)\text{vec}(I)^\top (x_t \otimes -(H_t + \lambda_t I)^{-1}) \frac{dL}{dx_t} \tag{30}$$

Substituting from (26),

$$\begin{aligned} \text{vec}(\frac{dL}{dH_t}) &= -(x_t \otimes \frac{dL}{dq_t}) - \frac{a}{||H_t||_F} \text{vec}(H_t)\text{vec}(I)^\top (x_t \otimes \frac{dL}{dq_t}) \\ &= -(x_t \otimes \frac{dL}{dq_t}) - \frac{a}{||H_t||_F} \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \end{aligned} \tag{31}$$

Now, with gating, we have $H_t = \gamma_t H_{t-1} + k_t k_t^\top$. Which can be unrolled as

$$H_l = \sum_{i=0}^{l} \Big(\prod_{k=i}^{l-1} \gamma_k \Big) k_i k_i^\top \tag{32}$$

We will compute $\frac{dL}{dk_l}$ for some $l \leq t$,

$$\frac{dL}{dk_l} = \sum_{t \geq l} \frac{d\text{vec}(H_t)}{dk_l} \text{vec}(\frac{dL}{dH_t}) \tag{33}$$

Computing $\frac{d\text{vec}(H_t)}{dk_l}$ for some $t \geq l$

$$H_t = \prod_{i=l}^{t-1} \gamma_i k_l k_l^\top + \text{terms indep. of } k_l. \tag{34}$$

Taking differentials on both sides,

$$\begin{aligned} dH_t &= \prod_{i=l}^{t-1} \gamma_i \Big[dk_l k_l^\top + k_l (dk_l)^\top \Big] \\ d(\text{vec}(H_t)) &= \prod_{i=l}^{t-1} \gamma_i \Big[\text{vec}(dk_l k_l^\top) + \text{vec}(k_l (dk_l)^\top) \Big] \\ &= \underbrace{\Big(\prod_{i=l}^{t-1} \gamma_i \Big) \Big[(k_l \otimes I) + (I \otimes k_l) \Big]}_{J_{k_l \to \text{vec}(H)}} dk_l \end{aligned} \tag{35}$$

where in the last equality we used the identity $\text{vec}(k_l dk_l^\top) = dk_l \otimes k_l = (I \otimes k_l) dk_l$.

Subsituting the Jacobian (transposed for gradients) from (35) to (33) we obtain.

$$\frac{dL}{dk_l} = \sum_{t \geq l} \Big(\prod_{i=l}^{t-1} \gamma_i \Big) \Big[(k_l^\top \otimes I) + (I \otimes k_l^\top) \Big] \text{vec}(\frac{dL}{dH_t}) \tag{36}$$

Substituting the expression for $\text{vec}(\frac{dL}{dH_t})$ into equation (36) we get:

$$\begin{aligned} \frac{dL}{dk_l} &= -\sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \quad \Big[(k_l^\top \otimes I) + (I \otimes k_l^\top) \Big] \Big[(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{||H_t||_F} \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \Big] = \\ &= -\sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \qquad \Big[\Big((k_l^\top \otimes I)(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{||H_t||_F} (k_l^\top \otimes I) \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \Big) + \\ &\qquad\qquad (I \otimes k_l^\top)(x_t \otimes \frac{dL}{dq_t}) + \frac{a}{||H_t||_F} (I \otimes k_l^\top) \text{vec}(H_t) \langle \frac{dL}{dq_t}, x_t \rangle \Big] \end{aligned} \tag{37}$$

Note that the following equations hold:

$$\begin{aligned} (k_l^\top \otimes I)(x_t \otimes \frac{dL}{dq_t}) &= (k_l^\top x_t \otimes \frac{dL}{dq_t}) = \langle k_l, x_t \rangle \frac{dL}{dq_t} \\ (I \otimes k_l^\top)(x_t \otimes \frac{dL}{dq_t}) &= x_t \otimes k_l^\top \frac{dL}{dq_t} = \langle k_l, \frac{dL}{dq_t} \rangle x_t \end{aligned} \tag{38}$$

since $(A \otimes B)(C \otimes D) = AC \otimes BD$ and the fact that the Kronecker products after the simplification is a scalar times a vector.

For the other terms is holds:

$$\begin{aligned} (k_l^\top \otimes I) \text{vec}(H_t) &= \text{vec}(H_t k_l) \\ (I \otimes k_l^\top) \text{vec}(H_t) &= \text{vec}(k_l^\top H_t) = \text{vec}(H_t^\top k_l) \end{aligned} \tag{39}$$

where we used the fact $\text{vec}(AXB) = (B^\top \otimes A)\text{vec}(X)$ and the fact that the vec operator applied to a row vector returns the same result as applying it on its transpose (so we go from $k_l^\top H_t$ to $H_t^\top k_l$). Since H_t is symmetric we can sum both contributions and get twice that amount.

Eventually we get:

$$\boxed{\frac{dL}{dk_l} = -\sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[\langle k_l, x_t \rangle \frac{dL}{dq_t} + \langle k_l, \frac{dL}{dq_t} \rangle x_t + \frac{2a}{||H_t||_F} \langle x_t, \frac{dL}{dq_t} \rangle H_t k_l \right]} \tag{40}$$

Or equivalently, collecting the terms that are linear in the gradient:

$$\boxed{\frac{dL}{dk_l} = -\sum_{t \geq l} \left(\prod_{i=l}^{t-1} \gamma_i \right) \left[\langle k_l, x_t \rangle \frac{dL}{dq_t} + \langle k_l, \frac{dL}{dq_t} \rangle x_t + \frac{2a \langle x_t, \frac{dL}{dq_t} \rangle}{||H_t||_F} H_t k_l \right]} \tag{41}$$

Note: The last term creates a dependence on k_l through $H_t k_l$, which is expected since the regularization λ_t couples the gradient computation.

E. Proof of Lemma 2

Lemma 2 describes an interesting phenomenon where, for the CH method (Algorithm 1), the gradient $d\hat{q}$ obtained from implicit differentiation coincides with the exact gradient dq obtained via backpropagation (chain rule). To prove this result, one way is to derive an analytic expression for dq (Section E.1) and then inspect the recursions. However, this can be algebraically involved. Here, we present a clear proof based on some simple observations.

First, note that the output ξ_r is linear in q and moreover there is a matrix function $p_r(H) \in \mathbb{R}^{D\times D}$ such that

$$\xi_r = p_r(H) \cdot q. \tag{42}$$

Here $p_r(H)$ is a polynomial function of H that encodes the Chebyshev iteration (Algorithm 1). Conversely, we understand that $p_r(H) \cdot q$ can be computed by applying the Chebyshev iteration with H, q for r iterations (together with other parameters such as μ, L). Then, given the output gradient $d\xi_r$, we have

$$dq = p_r(H)^\top \cdot d\xi_r = p_r(H) \cdot d\xi_r, \tag{43}$$

where the last equality follows, since H is symmetric, which implies $p_r(H)$ is symmetric. The proof is finished by observing that $p_r(H) \cdot d\xi_r$ can be computed via Algorithm 1 with $H = H, q = d\xi_r$ and other parameters, which gives us dq.

E.1. The Exact Backward Pass for dq and dH

Here we show how to obtain the exact gradients of dH and dq in Algorithm 1 given the output gradient $d\xi_r$, which might be of independent interests. The key insight here is that the Chebyshev iteration can be *reversed*.

Backward Pass for dq. Let I be the identity matrix of suitable size. To derive a backward pass of Algorithm 1, we first write down the update of ξ_i concisely in the following recursion

$$\xi_i = A_i\xi_{i-1} + b_i\xi_{i-2} + c_iq, \tag{44}$$

where A_i, b_i, c_i are defined as

$$A_i = \omega_i I - \frac{2\cdot\omega_i}{L+\mu}H, \quad b_i = -(\omega_i - 1), \quad c_i = \frac{2\cdot\omega_i}{L+\mu}. \tag{45}$$

Note that A_i is symmetric. Define $d\xi_i := \frac{d\mathcal{L}}{d\xi_i}$ for every i. With some loss function $\mathcal{L}$, assume we are now given $d\xi_r$, and our goal is to compute $dq := \frac{d\mathcal{L}}{dq}$. Since q appears in (44) for every i, we know $\xi_0, \xi_1, \ldots, \xi_r$ all depend on q. Therefore, with $c_0 := \frac{2}{L+\mu}$, we have

$$dq = \sum_{i=0}^{r} c_i \cdot d\xi_i.$$

It remains to compute $d\xi_i$ for every i. Applying the chain rule to (44), we obtain

$$\begin{aligned} d\xi_{r-1} &= A_r \cdot d\xi_r \\ d\xi_{i-2} &= A_{i-1} \cdot d\xi_{i-1} + b_i \cdot d\xi_i, \quad \forall i = r, \ldots, 2. \end{aligned} \tag{46}$$

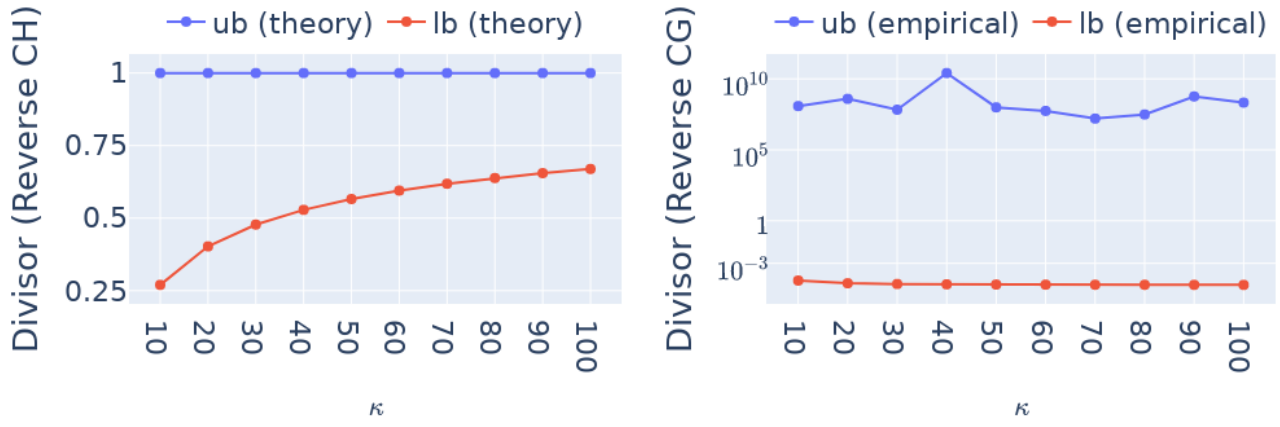
Note that A_i, b_i, c_i depend on some constant terms and ω_i. Thus, to compute them backward we assume access to ω_r and these constants. By reversing (weight schedule) we derive the following recursion:

$$\begin{aligned} \nu_r &\leftarrow \omega_r \\ \nu_{i-1} &\leftarrow \frac{4}{\rho^2}\left(1 - \frac{1}{\nu_i}\right), \quad \forall i = r, \ldots, 1. \end{aligned} \tag{47}$$

Similarly to how ω_i decreases with i and converges to ω_1^*, we may prove ν_i is convergent to the other fixed point, ω_2^*, as i decreases (and the iterate does not stop at $i = 1$).

Backward Pass for dA. From (44) and (45) we see that

$$dA_i = d\xi_i \otimes \xi_{i-1}^\top, \quad dH = -\frac{2\cdot\omega_i}{L+\mu} \cdot dA_i \tag{48}$$



(a) (b)

Figure 5. (a) The theoretical lower and upper bounds for the values of the divisor b_i that arise in reversing Chebyshev (49); (b) The empirical lower and upper bounds for the divisor that arises in reversing CG.

where $\otimes$ denotes the Kronecker product; this is the out product of $d\xi_i$ and $\xi_{i-1}^\top$, as $d\xi_i$ and ξ_{i-1} are vectors.

Reverse Chebyshev Iteration. At first glance, computing $dA_i = d\xi_i \otimes \xi_{i-1}^\top$ requires storing ξ_{i-1} in the forward pass, and the actual calculation of dA_i is done after we run the backward pass for $d\xi_i$ in (46). However, storing all ξ_i's would be memory-inefficient. To address this issue, a main insight here is that we can reverse (44) and write

$$\xi_{i-2} = \frac{1}{b_i}(\xi_i - A_i\xi_{i-1} + c_i q). \tag{49}$$

This implies that we can recover all the iterates $\xi_r, \dots, \xi_0$ as soon as we have access to the last two, ξ_r, ξ_{r-1}. Therefore, to obtain dA_i, we can run two iteration schemes in (46) and (49) simultaneously.

Remark 2. We find that being able to run the iterative update *backward* in a numerically stable fashion is a main feature of the Chebyshev iterative method (or more generally, gradient descent variants with momentum). Vanilla gradient descent can not efficiently reverse its iterate $\xi_i = \xi_{i-1} - \gamma_i(H\xi_{i-1} - q)$ with stepsize γ_i, as it requires inverting $(I - \gamma_i H)$. Moreover, reversing (49) can be done stably, as b_i is often in a good numerical range, which means division by b_i in (49) is not an issue. To see this, first note that by Lemma 3 we have

$$1 \geq -b_i = \omega_i - 1 \geq \omega_1^* - 1.$$

Note that ω_1^* defined in Lemma 3 is an increasing function of ρ and therefore of κ. We then have that $-b_i \in [0.25, 1]$ for any $\kappa \geq 10$ (we will not consider the case $\kappa < 10$ as this means we need to add a very large regularization strength which might harm the minimization of the regression loss). In comparison, if we were to reverse the CG iteration, we would need to divide a quantity that is often numerically as small as 10^{-3} or as large as 10^{10} (see Fig. 5). This is why it is numerically unstable to reverse CG.

F. Experimental Setup

Model Configurations. We consider models of 3 different sizes: 440M, 1B, and 2.8B. This is summarized in Table 4. All models are with the GPT2 tokenizer similarly to [63].

Training Configurations. All models are trained with the AdamW optimizer with initial learning rate 10^{-3}, 5% warm-up steps, cosine schedule, gradient clipping with maximum norm 1.

Models of the same scale use the same training configurations. Specifically (see also Table 5):

- For 440M models, we use sequence length 2048 and 8B DCLM tokens.
- For 1B models, we use sequence length 2048 and 20B DCLM tokens.
- For 2.8B models, we use sequence length 4096 and 100B DCLM tokens.

Algorithm 2: Backward Pass of Chebyshev Iteration

Input: $H, d\xi_r, L, \mu, ,$ number of iterations r, the final weight ω_r;

Initialize $\rho \leftarrow \frac{L-\mu}{L+\mu}, d\xi_{r+1} \leftarrow 0, \nu_r \leftarrow \omega_r, \nu_{r+1} \leftarrow 0, \nu_0 \leftarrow 1$, $dq \leftarrow \frac{2\nu_r}{L+\mu} \cdot d\xi_r$, $dH \leftarrow -\frac{2\nu_r}{L+\mu} d\xi_r \otimes \xi_{r-1}^\top$;

For $i = r, \ldots, 1$:

$$\xi_{i-2} \leftarrow -\frac{1}{\nu_i - 1}\left(\xi_i - \left(\nu_i I - \frac{2 \cdot \nu_i}{L+\mu} H\right)\xi_{i-1} + \frac{2 \cdot \nu_i}{L+\mu} q\right) \tag{50}$$

$$d\xi_{i-1} \leftarrow \left(\nu_i \cdot d\xi_i - \frac{2 \cdot \nu_i}{L+\mu} \cdot H \cdot d\xi_i\right) - (\nu_{i+1} - 1) \cdot d\xi_{i+1} \tag{51}$$

$$\nu_{i-1} \leftarrow \frac{4}{\rho^2}\left(1 - \frac{1}{\nu_i}\right) \tag{52}$$

$$dq \leftarrow dq + \frac{2\nu_{i-1}}{L+\mu} \cdot d\xi_{i-1} \tag{53}$$

$$dH \leftarrow dH - \frac{2\nu_{i-1}}{L+\mu} \cdot d\xi_{i-1} \otimes \xi_{i-2}^\top \tag{54}$$

Output: dq, dH;

Table 4. Model sizes and the corresponding architectural configurations.

Model Size	Number of Layers	Number of Heads	Hidden Dimension
440M	28	8	1024
1B	28	12	1536
2.8B	32	20	2560

Table 5. Model sizes and the corresponding architectural configurations.

Model Size	Global Batch Size	Total Number of Training Tokens	Sequence Length
440M	1M	8B	2048
1B	2M	20B	2048
2.8B	2M	100B	4096

Model Hyperparameters. We use default parameters for all other models as given in the Flash-Linear-Attention v0.4.0 library (except the ones mentioned in Table 4). For our approach, we use $\lambda_t = 0.02 \cdot \|H_t\|_{\mathrm{F}}$, with gating and α-connection enabled by default, unless otherwise specified. We also run CH for 30 iterations for all experiments.

Individual Experiments. We now describe the setups for each individual experiment.

In Fig. 1a, we randomly generate tensors $k \in \mathbb{R}^{B\times T\times H\times D}$ and q and normalize them along the last dimension (D). Here B, T, H, D simulate the batch size, sequence length, number of heads, and head dimension, respectively. Then we compute the covariance matrices $H \in \mathbb{R}^{B\times T\times H\times D\times D}$ of k, normalize its every $D \times D$ slice by its Frobenius norm. The code to generate data is shown below.

```
k = torch.randn(B, T, H, D).to(dtype).to('cuda')
q = torch.randn(B, T, H, D).to(dtype).to('cuda')

q = q / torch.linalg.vector_norm(q, dim=-1, keepdim=True).to(q)
k = k / torch.linalg.vector_norm(k, dim=-1, keepdim=True).to(k)

kk = torch.einsum('...i,...j->...ij', k, k).cumsum(1)
kk = kk / torch.linalg.matrix_norm(kk, ord='fro')[..., None, None]
```

```
kk.diagonal(dim1=-2, dim2=-1).add_(ridge_strength)
```

For Fig. 1c and Fig. 1e, we generate random input ids with vocabulary size 5000, sequence length 2048 within a 5-layer LLAMA; we set 2 heads and head dimension 128 for this architecture.

In the MQAR experiments of Fig. 2a, we follow the standard experimental setting but consider a strictly harder setting with smaller model dimension (or hidden dimension). Indeed, in the setting of [1], the model dimension is always larger than or equal to the number of KV pairs, while in the setting here, in some cases the model dimension is smaller than the number of KV pairs, in which case linear SSMs could not perfectly memorize all KV pairs.

In the main paper, Fig. 2a is without any gating or α-connection.

In Fig. 3 we considered the following tasks for long context evaluations. Reported results for each task is average over the score obtained for individual datasets in that task.

- *Retrieval-Augmented Generation (RAG)*: These tasks consist of open-domain question answer where the model is given a gold passage (passage containing the answer) interspersed between many other retrieved passages from a corpus [53, Wikipedia dump split into 100-word passages]. The model is tasked with answering the question based on the obtained passages. We consider the following datasets from HELMET [72] for this task: Natural Questions, TriviaQA, PopQA, HotpotQA.
- *Many-shot In-Context Learning (ICL)*: ICL tests LLMs ability to learn new skills from a few examples. Here the task is to learn to classify between different concepts based on several in-context examples of the said concept. We consider the following datasets from HELMET [72] for this task: TREC Coarse, TREC Fine, NLU, BANKING77, CLINIC150.
- *Synthetic Recall*: These tasks are variations of the “Needle-in-a-Haystack" task [31] where the goal is to retrieve an important piece of information, the “needle" from a long context of distractor tokens, the “haystack". These variations also test multi-hop tracing and aggregation capabilities of the model. We consider the following datasets from RULER [27] for this task: S-NIAH-1/2/3, MK-NIAH-1,2,3, MV-NIAH, MQ-NIAH, VT, CWE, FWE.
- *LongQA*: These are long document based question-answering tasks. The documents are typically made long by randomly sampling different paragraphs from the same dataset along with the paragraph that contains the answer. We consider the following datasets from RULER [27] for this task: SQuAD, HotpotQA.

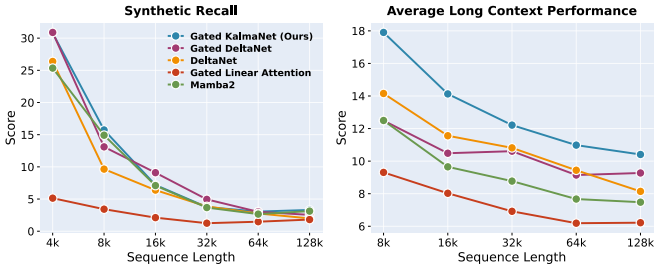


Figure 6. Recall and average performance without *S-NIAH-1*.

G. Synthetic Recall Without S-NIAH-1

As noted in the main paper, synthetic recall tasks differ significantly from natural text. E.g., in *S-NIAH-1*, 19 of 20 words can repeat 10000+ times while ground-truth appears once. GKA's regression objective then overweights irrelevant tokens, hindering retrieval. Excluding this task, GKA is competitive on synthetic recall (10 remaining RULER tasks) with better average (over RAG, ICL, Recall & LongQA) performance, as shown in Fig. 6.

H. Throughput for Long Sequences

Table 6 profiles GKA's throughput across model scales and context lengths. Due to its linear-time complexity, GKA maintains nearly constant throughput as sequence length increases, given a fixed batch size of 4M tokens.

Table 6. Throughput in tokens/GPU/sec. GKA achieves up to 2× higher throughput than Transformers at 64K context length.

Scale	Model	16K	32K	64K
8B	Transformer	6096.37	4766.25	3382.50
	GKA	6898.53	6808.94	6721.64
14B	Transformer	2833.99	2221.56	1485.24
	GKA	3318.28	3256.45	3256.45

I. How Does The Performance GKA Scale with Compute?

We consider models at three different scales: 440M, 1B and 2.8B. For training configurations and architecture refer to Section F. We use prototypical tasks from LM-Harness (see Section 5.3.1 for list of tasks) to evaluate language modeling capabilities of GKA and compare with baseline SSM/fading memory layers. Table 7 shows that at 440M scale, GKA is competitive with GDN and Deltanet. However, differences emerge at larger scales, with GKA showing increasing benefits. In particular, the retrieval capabilities of our model, as measured by FDA and SWDE consistently outperform all SSM baselines at 1B and 2.8B scale. We also report the results of equal-sized Transformer for completeness, which serves as a performance ceiling at each scale.

Table 7. **GKA shows stronger scaling with compute that other SSM baseline models.** LM-Harness results for models at different scales: 440M, 1B and 2.8B. All models were trained from scratch. 440M and 1B models were trained on 8B and 20B tokens respectively in accordance to the Chinchila scaling laws [26]. For the 2.8B model we trained on 100B tokens.

Model	**ARC-C** acc_n ↑	**ARC-E** acc_n ↑	**BoolQ** acc ↑	**COPA** acc ↑	**HellaSWAG** acc_n ↑	**PIQA** acc_n ↑	**SciQ** acc_n ↑	**Winogrande** acc ↑	**FDA** contains ↑	**SWDE** contains ↑	**Avg**
440M Models											
Transformer	24.40	42.26	59.88	70.00	36.19	64.15	61.50	**51.70**	**5.17**	**35.64**	**45.09**
Gated Linear Attention	24.06	40.28	56.57	71.00	32.70	62.24	57.80	50.67	1.00	9.18	40.55
Gated DeltaNet	**25.17**	41.96	58.23	**72.00**	36.96	**64.69**	63.6	**51.7**	1.91	11.88	42.81
DeltaNet	25.09	41.92	**61.13**	65.00	37.20	64.47	**64.00**	49.49	2.81	14.31	42.54
Gated KalmaNet (Ours)	24.57	**43.22**	56.94	71.00	**37.22**	64.47	62.8	50.83	1.45	14.04	42.65
1B Models											
Transformer	26.62	46.42	59.94	**77.00**	44.01	67.14	**68.30**	54.06	**8.35**	**45.18**	**49.70**
Mamba2	**28.07**	46.63	60.21	70.00	44.57	67.57	65.50	54.30	1.45	15.75	45.40
Gated Linear Attention	25.94	42.00	58.84	70.00	36.34	63.60	58.20	51.85	1.45	10.53	41.88
Gated DeltaNet	27.05	**47.98**	59.54	74.00	44.27	67.36	66.2	53.83	2.18	17.82	46.02
DeltaNet	27.56	46.25	59.97	71.00	43.18	67.74	65.90	**55.41**	3.09	20.61	46.07
Gated KalmaNet (Ours)	25.43	46.55	**60.73**	74.00	**44.59**	**68.88**	67.60	52.41	6.17	21.87	46.82
2.8B Models											
Transformer	32.25	56.10	**64.28**	80.00	60.96	73.56	79.50	61.72	**58.53**	**72.28**	**63.92**
Mamba2	32.24	59.64	58.72	82.00	62.23	73.78	79.80	62.19	7.71	41.13	55.94
Gated Linear Attention	27.82	50.80	52.57	78.00	48.83	70.13	69.60	54.54	2.81	20.43	47.55
Gated DeltaNet	32.59	**60.02**	62.75	82.00	62.8	74.32	80.6	62.35	8.26	44.28	57.00
DeltaNet	**32.85**	58.16	42.51	81.00	61.13	73.78	43.90	61.72	11.80	46.08	51.29
Gated KalmaNet (Ours)	32.51	59.89	61.68	**85.00**	**63.84**	**74.81**	**83.2**	**64.17**	12.89	50.95	58.89

J. Ablations

In this section we consider ablations for various modeling choices made in arriving at our final GKA model. For all ablations, we consider 2.8B models trained on 100B tokens on DCLM at 4K context length (unless mentioned otherwise). We use the same architecture and training configurations for these ablations as mentioned in Section F.

J.1. Does Adaptive Regularization Help?

As discussed in Section 4.1, we introduced adaptive regularization to control the condition number of $H_T + \lambda_t I$ for numerical stability. Here we ablate this choice, specifically we compare the following runs.

1. *Adaptive regularization*. We train a model with $\lambda_t = a||H_t||_F$. We report results for $a = 0.02$ for this run.
2. *Constant regularization* We train same model architecture (as above) with $\lambda_t = 0.25$ (a constant). This choice of 0.25 is motivated from concurrent work [63] which explored a similar ridge regression objective for LLM training.

As shown in Fig. 7, without strict condition number control, gradient norms spike during training, leading to increased cross entropy loss (compared to the run with adaptive regularization).

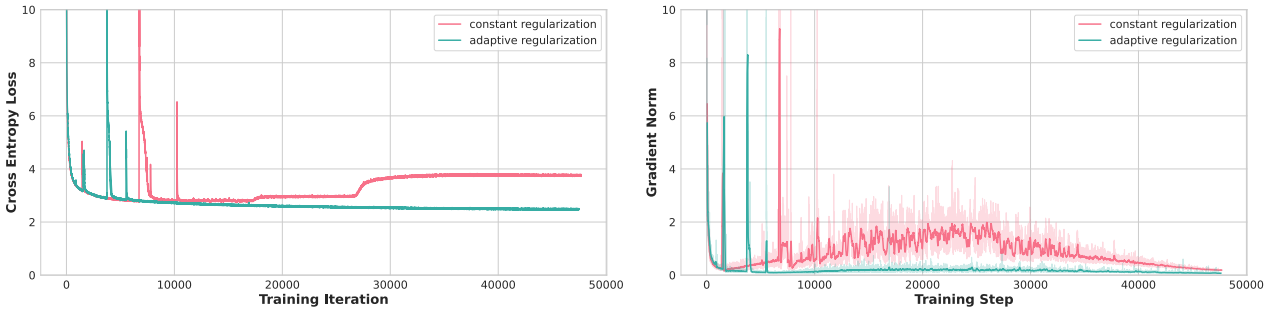


Figure 7. **Adaptive regularization results in smoother and better training curves.** (a) Plots the training curve for 2.8B models on 100B tokens from DCLM. (b) Plots the corresponding gradient norm. The model with constant regularization (red curve) results in a higher loss that can be attributed to its non-smooth trajectory over the course of its training run (spiky gradient norms).

J.2. Does Adaptive Weighting Help?

In Section 4.1, we discussed increasing the expressivity of our layer by introducing adaptive weights $\eta_{t,i}$ which re-weigh the past to be exponentially decaying in time. Given constant-sized memory, we hypothesize this adaptive weighting (gating) allows GKA to learn an effective representation by incorporating recency bias into its computation. In this subsection we test this hypothesis. We carry out the following runs.

1. *Adaptive weighting (gating)*. We train a model with adaptive weights. Specifically, for all $t \geq i$, we parameterize the weight for the i^{th} sample at time-step t as $\eta_{t,i} = \prod_{j=i+1}^{t} \gamma_j$, with each $\gamma_j \in [0, 1]$ learnable.
2. *No weighting*. We train the same model architecture as above, but with no weights. This essentially results in an unweighted ridge regression objective obtained by setting $\eta_i = 1$ in (3).

Table 8 shows clear benefits of adapting weighting with improvements across the board on all LM-Harness tasks considered, thereby validating our hypothesis.

Table 8. **Adaptive weighting outperforms across the board on LM-Harness tasks.** Results for 2.8B models trained on 100B tokens from DCLM with and without adaptive weights as introduced in Section 4.1.

Adaptive Weights	ARC-C acc_n ↑	ARC-E acc_n ↑	BoolQ acc ↑	COPA acc ↑	HellaSWAG acc_n ↑	PIQA acc_n ↑	SciQ acc_n ↑	Winogrande acc ↑	FDA contains ↑	SWDE contains ↑	Avg
✗	28.24	51.73	57.68	76	53.87	71.87	71.6	54.38	6.08	33.03	50.45
✓	**32.51**	**59.89**	**61.68**	**85**	**63.84**	**74.81**	**83.2**	**64.17**	**12.89**	**50.95**	**58.89**

J.3. Does α-connection Improve Training of GKA?

In Section C, we introduce the α-connection as a residual connection that establishes a direct path for gradient flow through the GLA solution, improving training stability. This allows the model to fall back on the GLA solution when CH produces

poor-quality results due to non-convergence of the iterative solver within the fixed iteration budget. To validate this design choice, we perform two runs.

R1. *with α-connection*. We train a model with the α-connection as shown in our GKA block in Fig. 4.

R2. *without α-connection*. We train the same model architecture as above, but with no α connection. This can be simply understood as setting $\alpha_t = 1$ for all time-steps t in Fig. 4.

On LM-Harness, both models perform similarly, with R1 and R2 achieving aggregate scores of 58.89 and 58.39, respectively. However, clear differences emerge under long-context evaluation, where we trained both models on an additional 25B tokens from long documents at 128K context length. Fig. 8 shows that GKA without the α-connection exhibits inferior long-context performance on average, with Synthetic Recall and LongQA showing major degradation.

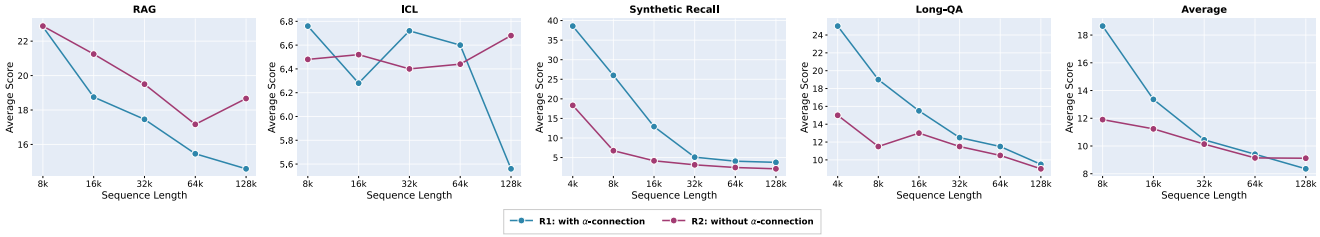
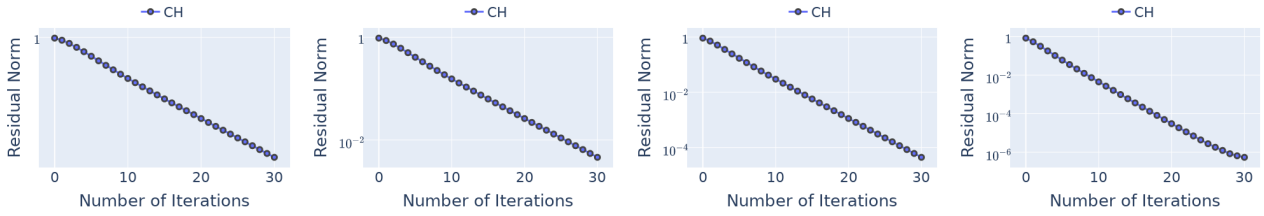


Figure 8. **GKA without the α- connection severaly underperforms on Synthetic Recall and LongQA.** On ICL all SSMs struggle to perform better than random chance (see Fig. 3). Interestingly, although R2 exhibits poorer long-context abilities in aggregate, it outperforms R1 on RAG by a few points.

K. Effects of Different Regularization Strengths

Recall that we proposed setting adaptive regularization $\lambda_t = a \cdot \|H_t\|_{\mathrm{F}}$. We now present experiments validating this choice.
Synthetic Experiments. First, we generate data as per Fig. 1a, where the covariance matrix is normalized by its Frobenius norm. In this case we set $\lambda_t = a$ for a varying in $\{0.01, 0.02, 0.05, 0.1\}$. Fig. 9 shows that the *maximum regularized residual norm* (computed as the maximum of $\|(H_t + \lambda_t I)\xi_i - q\|_2$ over all dimensions where ξ_i is the estimate of CH at iteration i) decreases as we enlarge λ_t. This is because having a large λ_t reduces the condition number. The downside, though, with a large λ_t is that it reduces the memorization capacity, namely, it might enlarge $\|H_t\xi_i - q\|_2$, the true residual of interest.



(a) $\lambda_t = 0.01$ (b) $\lambda_t = 0.02$ (c) $\lambda_t = 0.05$ (d) $\lambda_t = 0.1$

Figure 9. Convergence for varying regularization strengths (batch size 8, sequence length 2048, 8 heads, and head dimension 128).

GKA with different regularization strengths. We train several 2.8B models with varying regularization strength by choosing $a \in [0.01, 0.02, 0.05, 0.1]$. While performance on LM-Harness (Table 9) shows little discrepancy, we observe noticeable differences in long-context performance—where memorization capacity matters most—(Fig. 10). Specifically, the long-context performance of GKA improves initially as we decrease a from $0.1 \to 0.05$. This is expected since this increases the memorization capacity of the model. However, decreasing further from $0.05 \to 0.02 \to 0.01$ causes performance to decrease. This can be attributed to the increasing condition number of the problem, which reduces the quality of the solution computed by CH (Fig. 9).

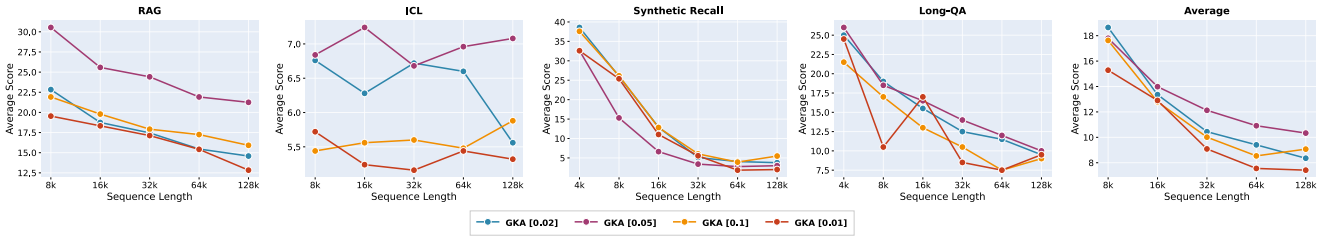


Figure 10. **Long context performance GKA for different regularization strengths.** The long-context performance of GKA improves initially as we decrease a from $0.1 \to 0.05$. This is expected since this increases the memorization capacity of the model. However, decreasing further from $0.05 \to 0.02 \to 0.01$ causes performance to decrease. This can be attributed to the increasing condition number of the problem, which reduces the quality of the solution computed by CH (Fig. 9)

Table 9. **Ablation over different choices of regularization strength $\lambda_t = a \cdot \|H_t\|_{\mathrm{F}}$.** Short-context performance on LM-Harness shows little discrepancy with different regularization strengths.

a	**ARC-C** acc_n ↑	**ARC-E** acc_n ↑	**BoolQ** acc ↑	**COPA** acc ↑	**HellaSWAG** acc_n ↑	**PIQA** acc_n ↑	**SciQ** acc_n ↑	**Winogrande** acc ↑	**FDA** contains ↑	**SWDE** contains ↑	**Avg**
0.01	**33.45**	58.63	62.63	**85.00**	63.36	73.99	81.40	63.14	11.16	**51.49**	58.43
0.02	32.51	59.89	61.68	**85.00**	63.84	74.81	**83.20**	**64.17**	**12.89**	50.95	**58.89**
0.05	32.68	**61.66**	53.57	79.00	63.46	**74.84**	82.60	63.77	11.98	49.68	57.32
0.1	32.76	59.85	**63.52**	84.00	**63.95**	**75.08**	**83.20**	63.54	11.43	51.22	58.86

Table 10. **Latent sketching increases training throughput (by up to 10%) while marginally reducing accuracy (< 1%).** Training throughput is reported in # Billion tokens/day/node. It is measured on a single H200 GPU with a batch size of 1M tokens. Our results indicate minimal regression on LM-harness tasks but up to 10% improvement in training throughput (going from no-sketch to sketch dim 32). However, long context performance is adversely affected with sketching with up to 60% relative drop in performance. Future work will address this by exploring the use of sketching adaptively depending on the "complexity" of the task.

Sketch dimension	LM-Harness avg.	Training throughput
32	57.57	8.37
no-sketch	58.89	7.65

L. Latent Sketching for Approximate Solutions

We introduce the idea of sketching from random matrix theory to further control the amount of FLOPs vs accuracy in GKA. Sketching involves down projecting the normal equations into a low-dimensional subspace, solving the equations in this subspace and finally up-projecting the solution back to the original space. This reduces the worst-case computational complexity of our approach from $\mathcal{O}(D^2r)$ to $\mathcal{O}(d^2r)$, where $d \ll D$ and r is the number of iterations in Algorithm 1. To the best of our knowledge our work is the first one introducing sketching as a viable solution to increase efficiency of neural network layers that are defined implicitly by the solution to an optimization problem. Sketching can be thought of as an analogous to the Multi Latent Attention idea introduced by DeepSeek but applied to fading memory layers. Table 10 shows preliminary results of this idea applied to GKA. Both models (no-sketch and sketch dim 32) are trained from scratch at 2.8B scale on 100B tokens.

M. Hybrid Gated KalmaNet

As discussed in Section A.2, augmenting SSM models with Attention layers has proven to be an effective way of improving performance on tasks that require recalling information from the distant past. In this section, we show that our Gated KalmaNet layer can be interleaved with Attention layers to yield even stronger models. Our Hybrid GKA model is based on the Qwen3 architecture [67]. Namely, our Hybrid model consists of a stack of "decoder" blocks, each of which contains a sequence mixer—either Attention or GKA—followed by an MLP. Similar to Qwen3, our Attention layers use QK normalization layers. Our Hybrid model consists of 30 decoder blocks, 26 of which use GKA as the sequence mixer, and 4 that use Attention. The Attention decoder blocks are at indices 6, 14, 22, and 29. Our Hybrid models follow the same training procedure as our non-Hybrid models. Specifically, we pretrain our Hybrid model on 100B tokens with a 4K context size, followed by fine-tuning on 25B tokens at a 128K context size.

When evaluating our pretrained Hybrid model standard NLP benchmarks, we observe that it improves substantially on recall-oriented tasks (FDA & SWDE) compared to the non-Hybrid model[3], as shown in Table 11. Further, when evaluating our fine-tuned long-context model on tasks that require effective modeling of long-range dependencies, we observe a significant improvement across all context lengths, as shown in Fig. 11.

Table 11. **Our Hybrid GKA + Attention model improves language modeling performance.** When interleaving Attention layers into our GKA models, we observe a significant improvement on recall-oriented tasks, such as FDA and SWDE, while preserving a similar performance on short-context tasks.

Model	**ARC-C** acc_n ↑	**ARC-E** acc_n ↑	**BoolQ** acc ↑	**COPA** acc ↑	**HellaSWAG** acc_n ↑	**PIQA** acc_n ↑	**SciQ** acc_n ↑	**Winogrande** acc ↑	**FDA** contains ↑	**SWDE** contains ↑	**Avg**
Gated KalmaNet (Hybrid)	**33.02**	59.47	**64.07**	80.00	62.74	74.59	81.40	**64.64**	**53.18**	**72.46**	**64.56**
Gated KalmaNet	32.51	**59.89**	61.68	**85.00**	**63.84**	**74.81**	**83.20**	64.17	12.89	50.95	58.89

[3]Note, our non-hybrid model shares the same architecture as the hybrid with the distinction that all 4 Attention layers are replaced with GKA layers.

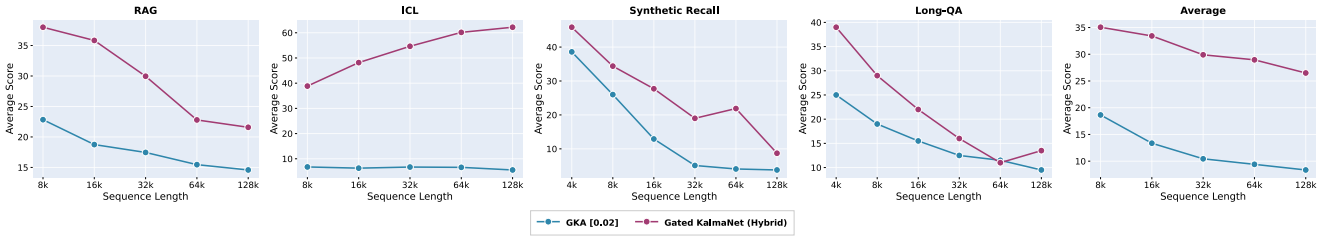


Figure 11. **Our Hybrid GKA + Attention model significantly improves performance across all long-context benchmarks compared to our non-Hybrid model.** Adding a few Attention layers to our GKA model improves long-range dependency modeling, improving performance across all sequence lengths on RAG, ICL, Synthetic Recall, and Long-QA.

N. GKAVision architecture

We obtain GKAVision by replacing the MambaMixer blocks in the MambaVision architecture [24] with GKA. For the GKA layer, we use the following hyperparameters:

1. We employ convolution kernels of size 3 (same as MambaVision).
2. We use an expansion factor of 0.5 (`expand_k` and `expand_v` parameters), which shrinks the input to the layer. This choice was made to ensure comparable model size with MambaVision.
3. We use a head dimension of 16. This was chosen to maximize the number of heads while ensuring the head dimension does not fall below the minimum requirements of Triton kernels.
4. We added an input-selectivity gate, β, to the GKA recurrence which we found to be helpful for boosting performance on ImageNet. Specifically, (CH) is modified to

$$\begin{aligned} H_t &= \gamma_t \cdot H_{t-1} + \beta_t k_t k_t^\top, \;\; U_t = \gamma_t \cdot U_{t-1} + \beta_t v_t k_t^\top, \\ y_t &= U_t \hat{x}_t, \quad \hat{x}_t = \mathrm{CH}(H_t + \lambda_t I, q_t, r), \end{aligned} \tag{55}$$

   where β_t is a per-head scalar modelled as a linear projection of the input followed by a sigmoid activation function. In our ablation study, adding this β gate improved ImageNet Top-1 accuracy from 81.14 $\rightarrow$ 81.27.

Remaining hyperparameters (λ and number of CH iterations) are the same as described in Section F. For training all models on ImageNet, we used the AdamW optimizer with cosine scheduling, a max LR of 5e-4, and weight decay of 0.05. All other training parameters follow the defaults from the official MambaVision repository.