Linear Attention Notes

Huwan Peng

January, 2026

Abstract

These notes introduce linear attention and its variants. We cover:

1. How linear attention compresses key-value history into a fixed-size state, reducing complexity from $O(L^2)$ to $O(L)$;
2. The Delta Rule and gated variants for improved memory fidelity;
3. Chunkwise algorithms that enable parallel training via the WY representation and UT transformation. Complete derivations and algorithms are provided.

Contents

1 Linear Attention and Variants

1.1 Original Linear Attention

1.1.1 Limitations of Standard Attention

The standard softmax attention is defined as

$$\mathbf{o}_t = \sum_{i=1}^{t} \frac{\exp(\mathbf{q}_t^\top \mathbf{k}_i)}{\sum_{j=1}^{t} \exp(\mathbf{q}_t^\top \mathbf{k}_j)} \mathbf{v}_i$$

where $\mathbf{o}_i, \mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i \in \mathbb{R}^d$ are the i-th output, query, key, and value vector, respectively. d represents the head dimension. In matrix form (used for training and prefill), we have

$$\mathbf{O} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$$

where $\mathbf{O}, \mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times d}$. This formulation runs efficiently on GPUs with high throughput. Techniques like FlashAttention [1] further reduce memory I/O for intermediate results.

However, standard attention suffers from quadratic complexity in sequence length. Each output $\mathbf{o}_t$ requires computing the query $\mathbf{q}_t$ against all previous keys and values vectors $\mathbf{k}_j, \mathbf{v}_j, j \in [1, t]$. This requires storing all key-value pairs (KV-cache). Since head dimension d is constant, we have

- **Computational Complexity**: $O(L^2 d) = O(L^2)$
- **Memory Complexity**: $O(Ld) = O(L)$

With the emergence of multimodal and reasoning models, LLM context lengths keep increasing, making both compute and memory costs prohibitively expensive.

1.1.2 Intuition Behind Linear Attention

The attention mechanism can be interpreted in many ways. One perspective views it as computing similarity scores between a query and all keys.

$$\mathbf{o}_t = \sum_{i=1}^{t} \frac{\text{sim}(\mathbf{q}_t, \mathbf{k}_i)}{\sum_{j=1}^{t} \text{sim}(\mathbf{q}_t, \mathbf{k}_j)} \mathbf{v}_i$$

Standard attention uses the exponential function for similarity function *sim*.

Linear Attention [2] defines a kernel-based similarity $\text{sim}(\mathbf{q}, \mathbf{k}) = \phi(\mathbf{q})^\top \phi(\mathbf{k})$, where $\phi : \mathbb{R}^d \to \mathbb{R}^d$ is a feature map. The attention function becomes

$$\mathbf{o}_t = \sum_{i=1}^{t} \frac{\phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_i)}{\sum_{j=1}^{t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_j)} \mathbf{v}_i$$

Using the associativity of matrix multiplication, this simplifies to

$$\mathbf{o}_t = \frac{\phi(\mathbf{q}_t)^\top \sum_{i=1}^{t} \phi(\mathbf{k}_i)\mathbf{v}_i^\top}{\phi(\mathbf{q}_t)^\top \sum_{j=1}^{t} \phi(\mathbf{k}_j)} = \frac{\mathbf{S}_t \phi(\mathbf{q}_t)}{\mathbf{z}_t^\top \phi(\mathbf{q}_t)}$$

where $\mathbf{S}_t = \sum_{i=1}^{t} \mathbf{v}_i \phi(\mathbf{k}_i)^\top \in \mathbb{R}^{d \times d}$, and $\mathbf{z}_t = \sum_{j=1}^{t} \phi(\mathbf{k}_j) \in \mathbb{R}^d$. The numerator is a $\mathbb{R}^D$ vector, the denominator is a scalar.

Later research [3] shows that removing the denominator improves numerical stability, and that the identity mapping works well for ϕ. This simplifies the equation to

$$\mathbf{o}_t = \mathbf{S}_t \mathbf{q}_t \in \mathbb{R}^d$$

with the state update rules of

$$\mathbf{S}_t = \sum_{i=1}^{t} \mathbf{v}_i \mathbf{k}_i^\top = \mathbf{S}_{t-1} + \mathbf{v}_t \mathbf{k}_t^\top \in \mathbb{R}^{d \times d}$$

It is essentially a linear RNN with matrix-valued hidden states.

Why it matters: It effectively "compresses" the history of all keys and values $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times d}$ into this fixed-size state matrix $\mathbf{S} \in \mathbb{R}^{d \times d}$. This reduces the computational complexity to $O(L)$ and the memory complexity to $O(1)$.

1.2 The Delta Rule (DeltaNet)

Essentially, the state $\mathbf{S}$ in Linear Attention is a key-value associative memory. The problem with the original state update rule is that it accumulates additively without bound. Values grow unbounded, and old information persists indefinitely. This leads to poor *Associative Recall*, i.e., the ability to retrieve a specific value given its key. It cannot overwrite or update specific facts efficiently; it just accumulates noise.

The **Delta Rule** [6] introduces a "correction" mechanism. Rather than simply adding new data, it computes the difference (delta) between the state's current prediction and the target value, then updates the state to minimize this error

$$\mathbf{S}_t = \mathbf{S}_{t-1} - \beta_t (\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top$$

where β_t is the learning rate, $\mathbf{S}_{t-1}\mathbf{k}_t$ is the prediction, and $\mathbf{v}_t$ is the actual value.

We have two interpretations of the Delta Rule in the context of Linear Attention.

1.2.1 The Optimization Perspective: Online SGD on Regression

This perspective treats matrix $\mathbf{S}$ not just as a state, but as a set of **weights** for a linear model. At each step, the model trains on the current token to better predict values from keys.

At step t, we want the matrix $\mathbf{S}$ to map the current key $\mathbf{k}_t$ to the current value $\mathbf{v}_t$. We define the loss using mean squared error.

$$\mathcal{L}_t(\mathbf{S}) = \frac{1}{2} \|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2$$

To minimize this loss, we compute the gradient with respect to $\mathbf{S}$.

$$\nabla_{\mathbf{S}} \mathcal{L}_t = (\mathbf{S}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top$$

Applying one step of Stochastic Gradient Descent (SGD) with learning rate β_i.

$$\begin{aligned}\mathbf{S}_t &= \mathbf{S}_{t-1} - \beta_t \nabla_{\mathbf{S}} \mathcal{L}_t(\mathbf{S}_{t-1}) \\ &= \mathbf{S}_{t-1} - \beta_t (\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top\end{aligned}$$

This is precisely the Delta Rule.

1.2.2 The Retrieval Perspective: Associative Memory with Correction

This perspective views the state $\mathbf{S}$ as a database that stores pairs $(\mathbf{k}, \mathbf{v})$. We can retrieve $\mathbf{v}$ by querying with $\mathbf{k}$.

Original Linear Attention uses simple addition $\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top$. When we query the database with $\mathbf{k}_t$ to retrieve $\mathbf{v}_t$, we get:

$$\begin{aligned}\mathbf{v}_t' &= \mathbf{S}_t\mathbf{k}_t \\ &= (\mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top)\mathbf{k}_t \\ &= (\mathbf{S}_{t-2} + \mathbf{v}_{t-1}\mathbf{k}_{t-1}^\top + \mathbf{v}_t\mathbf{k}_t^\top)\mathbf{k}_t \\ &= \mathbf{v}_t(\mathbf{k}_t^\top\mathbf{k}_t) + \left(\sum_{i=1}^{t-1} \mathbf{v}_i\mathbf{k}_i^\top\right)\mathbf{k}_t \\ &= \underbrace{\mathbf{v}_t(\mathbf{k}_t^\top\mathbf{k}_t)}_{\text{Desired term}} + \underbrace{\sum_{i=1}^{t-1} \mathbf{v}_i(\mathbf{k}_i^\top\mathbf{k}_t)}_{\text{Interference}}\end{aligned}$$

Interference: Every single item from the beginning ($i = 1$) to the last step ($i = t - 1$) contributes to the output, weighted by the dot product ($\mathbf{k}_i^\top\mathbf{k}_t$).

- If previous keys were all orthogonal ($\mathbf{k}_i^\top\mathbf{k}_t = 0$ for all $i \neq t$), the interference term would vanish, giving $\mathbf{v}_t' = \mathbf{v}_t$.
- In high dimensions, random vectors are rarely orthogonal. As context length grows, even small correlations accumulate into significant noise.

The Delta Rule first queries the memory's current prediction $\mathbf{S}_{old}\mathbf{k}_{new}$, then subtracts it from the target value before updating,

$$\mathbf{S}_{new} = \mathbf{S}_{old} + \beta_t(\mathbf{v}_{target} - \underbrace{\mathbf{S}_{old}\mathbf{k}_{new}}_{\mathbf{v}_{predict}})\mathbf{k}_{new}^\top$$

so we have

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \beta_t(\mathbf{v}_t - \mathbf{S}_{t-1}\mathbf{k}_t)\mathbf{k}_t^\top$$

If we query the matrix immediately after the update (assuming normalized keys $\mathbf{k}_t^\top \mathbf{k}_t = 1$ and step size $\beta_t = 1$ for simplicity):

$$\begin{aligned}\mathbf{S}_t \mathbf{k}_t &= (\mathbf{S}_{t-1} + \beta_t(\mathbf{v}_t - \mathbf{S}_{t-1}\mathbf{k}_t)\mathbf{k}_t^\top)\mathbf{k}_t \\ &= \mathbf{S}_{t-1}\mathbf{k}_t + (\mathbf{v}_t - \mathbf{S}_{t-1}\mathbf{k}_t)(\mathbf{k}_t^\top \mathbf{k}_t) \\ &= \mathbf{S}_{t-1}\mathbf{k}_t + \mathbf{v}_t - \mathbf{S}_{t-1}\mathbf{k}_t \\ &= \mathbf{v}_t\end{aligned}$$

In practice, the learning rate $\beta_t \in (0, 1)$ is set dynamically. When $\beta_t = 0$, the memory content remains unchanged; and when $\beta_t = 1$, the old value is completely replaced.

1.3 Gated Delta Rule

Even with Delta Rule correction, it can be useful to decay old context (e.g., forgetting the start of a sentence). Gated DeltaNet (GDN) [5] introduces a scalar forget gate, $\alpha_t \in (0, 1)$, to decay the old state.

$$\begin{aligned}\mathbf{S}_t &= \alpha_t \mathbf{S}_{t-1} + \beta_t(\mathbf{v}_t - \alpha_t \mathbf{S}_{t-1}\mathbf{k}_t)\mathbf{k}_t^\top \\ &= \alpha_t \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top\end{aligned}$$

The state matrix $\mathbf{S}$ has a fixed size. It can only store limited information before saturating. Pure DeltaNet optimizes the matrix without any forgetting mechanism. For very long sequences, the matrix becomes overloaded with competing associations. Gated DeltaNet decays old memories to free capacity for new information.

1.3.1 Kimi Delta Attention (KDA)

In most GDN models, α is a scalar, which means everything in the state fades equally. Kimi Delta Attention (KDA) [4] introduces a diagonalized gate $\mathrm{diag}(\boldsymbol{\alpha}_t)$, enabling per-dimension control over memory decay and positional awareness.

$$\mathbf{S}_t = (\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top)\mathrm{diag}(\boldsymbol{\alpha}_t)\mathbf{S}_{t-1} + \beta_t \mathbf{k}_t \mathbf{v}_t^\top$$

Unlike scalar gating, the diagonal matrix $\mathrm{diag}(\boldsymbol{\alpha}_t)$ allows each row of the state to decay at a different rate.

2 Hardware-Efficient Chunkwise Algorithm

Linear Attention transforms the parallelizable attention computation $(\mathbf{QK}^\top)\mathbf{V}$ into a recurrent form. While this reduces memory and computational costs, it can hurt throughput, especially during training and prefill.

For multi-token input, using only the recurrent formula ($\mathbf{S}_t = \mathbf{S}_{t-1} + \ldots$) requires waiting for each token to complete before processing the next. Since per-token computation is small, this results in poor hardware utilization and low throughput.

2.1 Chunkwise Parallelism

Chunkwise Parallelism [6] is a hybrid approach combining parallel and sequential computation. Instead of calculating intermediate hidden states for every token sequentially, we update states at regular intervals of size C (the chunk size). This allows us to exploit parallel matrix multiplication to generate all C outputs in a chunk. We illustrate this using original Linear Attention on the first chunk.

The State Update (Inter-chunk): First, we express the recurrent state update for a full chunk. For a chunk from time 1 to C, the final state is the sum of the previous state ($\mathbf{S}_0$) and all key-value updates within the chunk,

$$\begin{aligned}\mathbf{S}_C &= \mathbf{S}_{C-1} + \mathbf{v}_C\mathbf{k}_C^\top \\ &= \mathbf{S}_{C-2} + \mathbf{v}_{C-1}\mathbf{k}_{C-1}^\top + \mathbf{v}_C\mathbf{k}_C^\top \\ &= \mathbf{S}_0 + \sum_{i=1}^{C} \mathbf{v}_i\mathbf{k}_i^\top \\ &= \mathbf{S}_0 + \mathbf{V}_{1:C}^\top\mathbf{K}_{1:C}\end{aligned}$$

where $\mathbf{V}_{1:C}, \mathbf{K}_{1:C} \in \mathbb{R}^{C\times d}$. The term $\mathbf{V}_{1:C}^\top\mathbf{K}_{1:C}$ results in a $d \times d$ matrix. Unlike the sequential sum, this matrix multiplication can be efficiently parallelized on hardware.

The Output Calculation (Intra-chunk): Next, we derive the output for any specific token r inside the chunk (where $1 \le r \le C$). The output vector $\mathbf{o}_r$ depends on the current state $\mathbf{S}_r$ and the query $\mathbf{q}_r$.

$$\begin{aligned}\mathbf{o}_r &= \mathbf{S}_r\mathbf{q}_r \\ &= \left(\mathbf{S}_0 + \sum_{i=1}^{r} \mathbf{v}_i\mathbf{k}_i^\top\right)\mathbf{q}_r \\ &= \mathbf{S}_0\mathbf{q}_r + \sum_{i=1}^{r} \mathbf{v}_i(\mathbf{k}_i^\top\mathbf{q}_r)\end{aligned}$$

By stacking the vectors into matrices for the entire chunk, we can compute all outputs $O_{1:C} \in \mathbb{R}^{C\times d}$ in parallel,

$$\mathbf{O}_{1:C} = \underbrace{\mathbf{Q}_{1:C}\mathbf{S}_0^\top}_{\text{First Term}} + \underbrace{\text{Mask}(\mathbf{Q}_{1:C}\mathbf{K}_{1:C}^\top)\mathbf{V}_{1:C}}_{\text{Second Term}}$$

where the first term applies the historical context (previous state $\mathbf{S}_0$) to all queries in the chunk simultaneously. The second term resembles standard causal self-attention within the chunk, but without softmax.

This shows that we can effectively parallelize computation *within* chunks while maintaining linear complexity over the full sequence.

2.2 Chunkwise DeltaNet

2.2.1 WY Representation for Delta Rule

However, the chunkwise algorithm does not directly apply to DeltaNet, which introduces decay on $\mathbf{S_{t-1}}$.

$$\begin{aligned}\mathbf{S}_t &= \mathbf{S}_{t-1} - \beta_t(\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top \\ &= \mathbf{S}_{t-1}\underbrace{(\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top)}_{\text{Decay } D_t} + \beta_t\mathbf{v}_t\mathbf{k}_t^\top\end{aligned}$$

Define the decay matrix $\mathbf{D}_t := \mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top$. Computing the end-of-chunk state requires applying these decay matrices to both the state and each update.

$$\begin{aligned}\mathbf{S}_C &= \mathbf{S}_{C-1}\mathbf{D}_C + \beta_C\mathbf{v}_C\mathbf{k}_C^\top \\ &= \mathbf{S}_{C-2}\mathbf{D}_{C-1}\mathbf{D}_C + \beta_{C-1}\mathbf{v}_{C-1}\mathbf{k}_{C-1}^\top\mathbf{D}_C + \beta_C\mathbf{v}_C\mathbf{k}_C^\top \\ &\vdots \\ &= \mathbf{S}_0\prod_{t=1}^{C}\mathbf{D}_t + \sum_{t=1}^{C}(\beta_t\mathbf{v}_t\mathbf{k}_t^\top\prod_{j=t+1}^{C}\mathbf{D}_j)\end{aligned}$$

The issue is that we cannot simply factor out the $\mathbf{D}$ terms or compute them in parallel. Each term in the summation requires a different partial product of the $\mathbf{D}$ matrices, creating sequential dependencies.

To address this, we split the state update into two components,

$$\mathbf{S}_C = \underbrace{\mathbf{S}_0 \cdot \mathbf{P}_C}_{\text{Decaying the Past}} + \underbrace{\mathbf{H}_C}_{\text{Accumulating the Update}}$$

where $\mathbf{P}_C = \prod_{t=1}^{C}\mathbf{D}_t$, $\mathbf{H}_C = \sum_{t=1}^{C}(\beta_t\mathbf{v}_t\mathbf{k}_t^\top\prod_{j=t+1}^{C}\mathbf{D}_j)$.

Our goal is to express $\mathbf{P}_C$ and $\mathbf{H}_C$ as sums of rank-1 terms. This enables computation via matrix multiplications and elementwise operations, replacing the sequential loop over C.

a. The Transformation of P

P follows the classic **WY Representation** for Householder-like matrices. The theorem states that we can collapse the product of C rank-1 updates into a single summation of rank-1 updates,

$$\mathbf{P}_C = \prod_{t=1}^{C}(\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top) = \mathbf{I} - \sum_{t=1}^{C}\mathbf{w}_t\mathbf{k}_t^\top$$

where $\mathbf{w}_t$ is a new *decay weight* vector to be calculated. The theorem can be proved by mathematical induction as follows.

$$
\begin{aligned}
\prod_{t=1}^{C} \mathbf{D}_t &= \prod_{t=1}^{C-1} \mathbf{D}_t(\mathbf{I} - \beta_C \mathbf{k}_C \mathbf{k}_C^\top) \\
&= (\mathbf{I} - \sum_{t=1}^{C-1} \mathbf{w}_t \mathbf{k}_t^\top)(\mathbf{I} - \beta_C \mathbf{k}_C \mathbf{k}_C^\top) \\
&= \mathbf{I} - \sum_{t=1}^{C-1} \mathbf{w}_t \mathbf{k}_t^\top - \beta_C \mathbf{k}_C \mathbf{k}_C^\top + (\sum_{t=1}^{C-1} \mathbf{w}_t \mathbf{k}_t^\top)\beta_C \mathbf{k}_C \mathbf{k}_C^\top \\
&= \mathbf{I} - \sum_{t=1}^{C-1} \mathbf{w}_t \mathbf{k}_t^\top - \underbrace{\beta_C \left(\mathbf{k}_C - \sum_{t=1}^{C-1} \mathbf{w}_t \mathbf{k}_t^\top \mathbf{k}_C \right)}_{\mathbf{w}_C} \mathbf{k}_C^\top \\
&= \mathbf{I} - \sum_{t=1}^{C} \mathbf{w}_t \mathbf{k}_t^\top
\end{aligned}
$$

The proof also shows the recursive formula for computing the $\mathbf{w}$ vector.

b. The Transformation of H

Now consider $\mathbf{H}_C$. $\mathbf{H}_C$ represents the state contribution from updates within the current chunk alone. In other words, it equals the final state $\mathbf{S}_C$ when the initial state is zero, $\mathbf{S}_0 = 0$.

$$
\begin{aligned}
\mathbf{S}_C &= \mathbf{S}_0 \cdot \mathbf{P}_C + \mathbf{H}_C, \\
\mathbf{H}_C &= \mathbf{S}_C \big|_{\mathbf{S}_0 = 0}.
\end{aligned}
$$

We show that $\mathbf{S}_C$ (with $\mathbf{S}_0 = 0$) can also be written as a single sum of rank-1 terms, $\mathbf{S}_C = \sum_{t=1}^{C} \mathbf{u}_t \mathbf{k}_t^\top$, for some vectors $\mathbf{u}_t$. The derivation follows a similar induction as for $\mathbf{P}_C$.

$$
\begin{aligned}
\mathbf{S}_C &= \mathbf{S}_{C-1}(\mathbf{I} - \beta_C \mathbf{k}_C \mathbf{k}_C^\top) + \beta_n \mathbf{v}_C \mathbf{k}_C^\top \\
&= (\sum_{t=1}^{C-1} \mathbf{u}_t \mathbf{k}_t^\top)(\mathbf{I} - \beta_C \mathbf{k}_C \mathbf{k}_C^\top) + \beta_C \mathbf{v}_C \mathbf{k}_C^\top \\
&= \sum_{t=1}^{C-1} \mathbf{u}_t \mathbf{k}_t^\top - (\sum_{t=1}^{C-1} \mathbf{u}_t \mathbf{k}_t^\top)\beta_C \mathbf{k}_C \mathbf{k}_C^\top + \beta_C \mathbf{v}_C \mathbf{k}_C^\top \\
&= \sum_{t=1}^{C-1} \mathbf{u}_t \mathbf{k}_t^\top + \underbrace{\beta_C \left(\mathbf{v}_C - \sum_{t=1}^{C-1} \mathbf{u}_t \mathbf{k}_t^\top \mathbf{k}_C \right)}_{\mathbf{u}_C} \mathbf{k}_C^\top \\
&= \sum_{t=1}^{C} \mathbf{u}_t \mathbf{k}_t^\top
\end{aligned}
$$

c. Matrix Form

Substituting the simplified $\mathbf{P}_C$ and $\mathbf{H}_C$ into the state equation,

$$\begin{aligned}\mathbf{S}_C &= \mathbf{S}_0 \cdot \mathbf{P}_C + \mathbf{H}_C \\ &= \mathbf{S}_0\big(\mathbf{I} - \sum_{t=1}^{C} \mathbf{w}_t \mathbf{k}_t^\top\big) + \sum_{t=1}^{C} \mathbf{u}_t \mathbf{k}_t^\top \\ &= \mathbf{S}_0 + \sum_{t=1}^{C} (\mathbf{u}_t - \mathbf{S}_0 \mathbf{w}_t)\mathbf{k}_t^\top,\end{aligned}$$

where the vectors $\mathbf{w}_t$ and $\mathbf{u}_t$ are defined recursively as

$$\begin{aligned}\mathbf{w}_t &= \beta_t \left(\mathbf{k}_t - \sum_{i=1}^{t-1} \mathbf{w}_i \mathbf{k}_i^\top \mathbf{k}_t\right), \\ \mathbf{u}_t &= \beta_t \left(\mathbf{v}_t - \sum_{i=1}^{t-1} \mathbf{u}_i \mathbf{k}_i^\top \mathbf{k}_t\right).\end{aligned}$$

We can then compute the output vector $\mathbf{o}_t$ for any time step t within the chunk as

$$\begin{aligned}\mathbf{o}_t &= \mathbf{S}_t \mathbf{q}_t \\ &= \mathbf{S}_0 \mathbf{q}_t + \sum_{j=1}^{t} (\mathbf{u}_j - \mathbf{S}_0 \mathbf{w}_j)(\mathbf{k}_j^\top \mathbf{q}_t).\end{aligned}$$

By stacking all time steps in the chunk into matrices of shape $C \times d$, we obtain the compact matrix forms

$$\begin{aligned}\mathbf{S}_C &= \mathbf{S}_0 + (\mathbf{U} - \mathbf{W}\mathbf{S}_0^\top)^\top \mathbf{K}, \\ \mathbf{O}_{1:C} &= \mathbf{Q}\mathbf{S}_0^\top + \mathrm{Mask}(\mathbf{Q}\mathbf{K}^\top)\,(\mathbf{U} - \mathbf{W}\mathbf{S}_0^\top),\end{aligned}$$

where Mask$(\cdot)$ applies a causal (strictly lower-triangular) mask over the chunk.

This derivation applies recursively across chunks by treating the final state of the previous chunk as the initial state $\mathbf{S}_0$ for the current chunk.

2.2.2 UT Transformation

While the WY representation enables parallelization of the state update, the computation of $\mathbf{W}$ and $\mathbf{U}$ remains a bottleneck due to their recursive definitions. Specifically, the formulas for $\mathbf{w}_t$ and $\mathbf{u}_t$ are inherently sequential.

$$\mathbf{w}_t = \beta_t \left(\mathbf{k}_t - \sum_{i=1}^{t-1} \mathbf{w}_i (\mathbf{k}_i^\top \mathbf{k}_t)\right) = \beta_t \mathbf{k}_t + \sum_{i=1}^{t-1} \mathbf{w}_i (\beta_t \mathbf{k}_i^\top \mathbf{k}_t)$$

This recursion requires a sequential loop over $t = 1 \to C$, which limits parallelism.

To address this, the **UT Transformation** rewrites the recursion as a triangular matrix inversion, enabling efficient parallel computation.

Define a strictly lower-triangular matrix $\mathbf{L}$ that encodes the interaction terms between steps i and t.

$$\mathbf{L}_{t,i} = \begin{cases} \beta_t \mathbf{k}_i^\top \mathbf{k}_t & \text{if } i < t \text{ (past affects future)} \\ 0 & \text{otherwise} \end{cases}$$

In matrix notation, this is the lower triangle of the Gram matrix,

$$\mathbf{L} = \text{tril}(\text{diag}(\boldsymbol{\beta})\mathbf{K}\mathbf{K}^\top, -1)$$

where $\mathbf{K} \in \mathbb{R}^{C \times d}$ contains all keys, $\boldsymbol{\beta} \in \mathbb{R}^C$ contains all learning rates of the block, and -1 denotes the strict lower triangle (excluding the diagonal).

Stacking all $\mathbf{w}_t$ as rows of matrix $\mathbf{W}$. We can write the recursion for all time steps at once.

$$\mathbf{W} = \text{diag}(\boldsymbol{\beta})\mathbf{K} - \mathbf{L}\mathbf{W}$$
$$\mathbf{W} = (\mathbf{I} + \mathbf{L})^{-1}(\text{diag}(\beta)\mathbf{K})$$

2.2.3 Matrix Inversion via Forward Substitution

The UT transformation requires computing $\mathbf{T} = (\mathbf{I} + \mathbf{L})^{-1}\text{diag}(\boldsymbol{\beta}) \in \mathbb{R}^{C \times C}$, where $\mathbf{L}$ is strictly lower triangular. Since $\mathbf{I} + \mathbf{L}$ is unit lower triangular (lower triangular with ones on the diagonal), we can efficiently compute $\mathbf{T}$ using **forward substitution**.

We solve the following linear system $(\mathbf{I} + \mathbf{L})\mathbf{T} = \text{diag}(\boldsymbol{\beta})$ column by column.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{21} & 1 & 0 & \cdots & 0 \\ L_{31} & L_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{C1} & L_{C2} & L_{C3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1C} \\ T_{21} & T_{22} & \cdots & T_{2C} \\ T_{31} & T_{32} & \cdots & T_{3C} \\ \vdots & \vdots & \ddots & \vdots \\ T_{C1} & T_{C2} & \cdots & T_{CC} \end{bmatrix} = \begin{bmatrix} \beta_1 & 0 & 0 & \cdots & 0 \\ 0 & \beta_2 & 0 & \cdots & 0 \\ 0 & 0 & \beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \beta_C \end{bmatrix}$$

Let $\mathbf{t} \in \mathbb{R}^C$ be a column of $\mathbf{T}$, and let $\mathbf{b} \in \mathbb{R}^C$ be a column of $\text{diag}(\boldsymbol{\beta})$. We solve $(\mathbf{I} + \mathbf{L})\mathbf{t} = \mathbf{b}$ by expanding the system.

$$\begin{aligned} t_1 &= b_1 \\ L_{21}t_1 + t_2 &= b_2 \\ L_{31}t_1 + L_{32}t_2 + t_3 &= b_3 \\ &\vdots \\ L_{C1}t_1 + L_{C2}t_2 + \cdots + L_{C,C-1}t_{C-1} + t_C &= b_C \end{aligned}$$

Rearranging each equation to solve for t_i.

$$
\begin{aligned}
t_1 &= b_1 \\
t_2 &= b_2 - L_{21} t_1 \\
t_3 &= b_3 - L_{31} t_1 - L_{32} t_2 \\
&\vdots \\
t_C &= b_C - \sum_{j=1}^{C-1} L_{Cj} t_j
\end{aligned}
$$

Each component t_i depends only on previously computed components $t_1, \ldots, t_{i-1}$, enabling sequential computation from $i = 1$ to $i = C$. We repeat this procedure for $k = 1, \ldots, C$ to obtain all columns of $\mathbf{T}$.

Complexity. Forward substitution for a single column requires $O(C^2)$ operations. Computing the full matrix $\mathbf{T}$ requires C such solves, giving $O(C^3)$ total operations. Since C is a small constant (typical chunk size), this cost is negligible compared to the $O(Cd^2)$ matrix multiplications in the main algorithm.

2.2.4 Complete Algorithm

Together, these techniques enable hardware-efficient execution of DeltaNet.

We process the sequence in blocks of size C, maintaining a fixed-size recurrent state $\mathbf{S} \in \mathbb{R}^{d \times d}$ between blocks.

Inputs:

- Current block: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{C \times d}, \boldsymbol{\beta} \in \mathbb{R}^{C}$
- Previous state: $\mathbf{S}_0 \in \mathbb{R}^{d \times d}$

Algorithm Steps:

1. **Compute Interaction Matrix**:

$$\mathbf{L} = \mathrm{tril}(\mathrm{diag}(\boldsymbol{\beta})\mathbf{K}\mathbf{K}^\top, -1) \in \mathbb{R}^{C \times C}$$

2. **Invert Matrix**:

$$\mathbf{T} = (\mathbf{I} + \mathbf{L})^{-1}\mathrm{diag}(\boldsymbol{\beta}) \in \mathbb{R}^{C \times C}$$

3. **Generate the W and U**:

$$\mathbf{W} = \mathbf{T}\mathbf{K} \in \mathbb{R}^{C \times d}, \quad \mathbf{U} = \mathbf{T}\mathbf{V} \in \mathbb{R}^{C \times d}$$

4. **Update State (Inter-Chunk)**:

$$\mathbf{S}_C = \mathbf{S}_0 + (\mathbf{U} - \mathbf{W}\mathbf{S}_0^\top)^\top \mathbf{K} \in \mathbb{R}^{d \times d}$$

5. **Compute Output (Intra-Chunk)**:

$$\mathbf{O}_{1:C} = \mathbf{Q}\mathbf{S}_0^\top + \mathrm{Mask}(\mathbf{Q}\mathbf{K}^\top)(\mathbf{U} - \mathbf{W}\mathbf{S}_0^\top) \in \mathbb{R}^{C \times d}$$

2.3 Chunkwise Gated Delta Network

2.3.1 Gate Formulation

Recall that GDN introduces a scalar forget gate, $\alpha_t \in (0, 1)$, to decay the old state.

$$\mathbf{S}_t = \alpha_t \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$$

We retain the decay matrix $\mathbf{D}_t := \mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$, and define the cumulative decay $\gamma_t = \prod_{i=1}^{t} \alpha_i$. The end-of-chunk state becomes:

$$\begin{aligned}
\mathbf{S}_C &= \alpha_C \mathbf{S}_{C-1} \mathbf{D}_C + \beta_C \mathbf{v}_C \mathbf{k}_C^\top \\
&= \alpha_C \alpha_{C-1} \mathbf{S}_{C-2} \mathbf{D}_{C-1} \mathbf{D}_C + \alpha_C \beta_{C-1} \mathbf{v}_{C-1} \mathbf{k}_{C-1}^\top \mathbf{D}_C + \beta_C \mathbf{v}_C \mathbf{k}_C^\top \\
&\vdots \\
&= \mathbf{S}_0 \gamma_C \mathbf{D}_t + \sum_{t=1}^{C} (\beta_t \mathbf{v}_t \mathbf{k}_t^\top \prod_{j=t+1}^{C} \alpha_j \mathbf{D}_j) \\
&= \gamma_C \mathbf{S}_0 \prod_{t=1}^{C} \mathbf{D}_t + \sum_{t=1}^{C} (\beta_t \mathbf{v}_t \mathbf{k}_t^\top \frac{\gamma_C}{\gamma_t} \mathbf{D}_j)
\end{aligned}$$

The first term $\mathbf{P}_C = \prod_{t=1}^{C} \mathbf{D}_t$ remains unchanged.

However, the second term becomes $\tilde{\mathbf{H}}_C = \sum_{t=1}^{C} (\beta_t \mathbf{v}_t \mathbf{k}_t^\top \frac{\gamma_C}{\gamma_t} \mathbf{D}_j)$. Following a similar derivation:

$$\tilde{\mathbf{H}}_C = \sum_{t=1}^{C} (\frac{\gamma_C}{\gamma_t} \tilde{\mathbf{u}}_t \mathbf{k}_t^\top), \quad \tilde{\mathbf{u}}_t = \beta_t \left(\mathbf{v}_t - \sum_{i=1}^{t-1} (\tilde{\mathbf{u}}_i \frac{\gamma_t}{\gamma_i} \mathbf{k}_i^\top \mathbf{k}_t) \right)$$

By the UT transformation:

$$\tilde{\mathbf{U}} = \left(\mathbf{I} + \mathrm{tril}(\mathrm{diag}(\boldsymbol{\beta})(\boldsymbol{\Gamma} \odot \mathbf{K}\mathbf{K}^\top), -1) \right)^{-1} \mathrm{diag}(\boldsymbol{\beta}) \mathbf{V}$$

where $\boldsymbol{\Gamma} \in \mathbb{R}^{C \times C}$ represents internal relative decay between steps, where $\boldsymbol{\Gamma}_{ij} = \prod_{t=j+1}^{i} \alpha_t = \frac{\gamma_i}{\gamma_j}$, if $i \geq j$ and $\boldsymbol{\Gamma}_{ij} = 0$ otherwise.

And we have the state update equation

$$\begin{aligned}
\mathbf{S}_C &= \gamma_C \mathbf{S}_0 \cdot \mathbf{P}_C + \tilde{\mathbf{H}}_C \\
&= \gamma_C \mathbf{S}_0 \big(\mathbf{I} - \sum_{t=1}^{C} \mathbf{w}_t \mathbf{k}_t^\top\big) + \sum_{t=1}^{C} \frac{\gamma_C}{\gamma_t} \tilde{\mathbf{u}}_t \mathbf{k}_t^\top \\
&= \gamma_C \mathbf{S}_0 + \sum_{t=1}^{C} (\frac{\gamma_C}{\gamma_t} \tilde{\mathbf{u}}_t - \gamma_C \mathbf{S}_0 \mathbf{w}_t) \mathbf{k}_t^\top \\
&= (\gamma_C \mathbf{S}_0) + \sum_{t=1}^{C} (\tilde{\mathbf{u}}_t - \mathbf{S}_0 (\gamma_t \mathbf{w}_t)) (\frac{\gamma_C}{\gamma_t} \mathbf{k}_t^\top) \\
&= \overrightarrow{\mathbf{S}_0} + \sum_{t=1}^{C} (\tilde{\mathbf{u}}_t - \mathbf{S}_0 \overleftarrow{\mathbf{w}_t}) \overrightarrow{\mathbf{k}_t}
\end{aligned}$$

and output vector $\mathbf{o}_t$

$$\begin{aligned}
\mathbf{o}_t &= \mathbf{S}_t \mathbf{q}_t \\
&= \mathbf{S}_0 (\gamma_t \mathbf{q}_t) + \sum_{i=1}^{t} (\tilde{\mathbf{u}}_i - \mathbf{S}_0 (\gamma_i \mathbf{w}_i)) (\frac{\gamma_t}{\gamma_i} \mathbf{k}_i^\top \mathbf{q}_t) \\
&= \mathbf{S}_0 \overleftarrow{\mathbf{q}_t} + \sum_{i=1}^{t} (\tilde{\mathbf{u}}_i - \mathbf{S}_0 \overleftarrow{\mathbf{w}_t}) (\frac{\gamma_t}{\gamma_i} \mathbf{k}_i^\top \mathbf{q}_t)
\end{aligned}$$

where the arrow notation is defined as follows:

- Left Arrow ($\overleftarrow{\mathbf{x}_t} = \gamma_t \mathbf{x}_t$): Decays from the start of the chunk to step t. Applied to variables interacting with history ($\mathbf{Q}, \mathbf{W}$).
- Right Arrow ($\overrightarrow{\mathbf{x}_t} = \frac{\gamma_C}{\gamma_t} \mathbf{x}_t$): Decays from step t to the end of the chunk. Applied to variables that write to the final state ($\mathbf{K}$).
- Decayed History ($\overrightarrow{\mathbf{S}_0} = \gamma_C \mathbf{S}_0$): The initial state decayed across the entire chunk.

So we have the matrix form:

$$\begin{aligned}
\mathbf{S}_C &= \overrightarrow{\mathbf{S}_0} + \left(\tilde{\mathbf{U}} - \overleftarrow{\mathbf{W}} \mathbf{S}_0^\top\right)^\top \overrightarrow{\mathbf{K}} \\
\mathbf{O}_{1:C} &= \overleftarrow{\mathbf{Q}} \mathbf{S}_0^\top + (\mathbf{Q}\mathbf{K}^\top \odot \mathbf{\Gamma}) \left(\tilde{\mathbf{U}} - \overleftarrow{\mathbf{W}} \mathbf{S}_0^\top\right) \in \mathbb{R}^{C \times d}
\end{aligned}$$

2.3.2 Efficient Computation via Similarity Transformation

In the gated delta network, computing both $\mathbf{W}$ and $\tilde{\mathbf{U}}$ appears to require two separate matrix inversions, $(\mathbf{I} + \mathbf{L})^{-1}$ and $(\mathbf{I} + \tilde{\mathbf{L}})^{-1}$, where $\mathbf{L} = \text{tril}(\text{diag}(\boldsymbol{\beta})\, \mathbf{K}\mathbf{K}^\top, -1)$ and $\tilde{\mathbf{L}} = \text{tril}(\text{diag}(\boldsymbol{\beta})\, (\mathbf{\Gamma} \odot \mathbf{K}\mathbf{K}^\top), -1)$.

However, we show that these matrices are related by the following similarity transformation, allowing us to compute only one inverse.

$$\mathbf{I} + \tilde{\mathbf{L}} = \text{diag}(\boldsymbol{\gamma})(\mathbf{I} + \mathbf{L})\text{diag}(\boldsymbol{\gamma})^{-1}$$

Recall the definitions:

$$L_{ij} = \beta_i(\mathbf{k}_i^\top \mathbf{k}_j), \quad i > j$$
$$\tilde{L}_{ij} = \beta_i \cdot \frac{\gamma_i}{\gamma_j} \cdot (\mathbf{k}_i^\top \mathbf{k}_j), \quad i > j$$

We can apply diagonal matrix scales entries to $\mathbf{L}$ to get $\tilde{\mathbf{L}}$.

$$\left[\text{diag}(\boldsymbol{\gamma})\mathbf{L}\text{diag}(\boldsymbol{\gamma})^{-1}\right]_{ij} = \frac{\gamma_i}{\gamma_j} \cdot \beta_i(\mathbf{k}_i^\top \mathbf{k}_j) = \tilde{L}_{ij}$$

For the identity matrix, the diagonal entries are preserved.

$$\left[\text{diag}(\boldsymbol{\gamma})\mathbf{I}\text{diag}(\boldsymbol{\gamma})^{-1}\right]_{ii} = \frac{\gamma_i}{\gamma_i} = 1$$

Using this property, let $\mathbf{T} = (\mathbf{I} + \mathbf{L})^{-1}$. Since $\mathbf{T}$ is unit lower triangular, we can reuse the decay matrix $\boldsymbol{\Gamma}$ to obtain:

$$\tilde{\mathbf{T}} = (\mathbf{I} + \tilde{\mathbf{L}})^{-1} = \boldsymbol{\Gamma} \odot \mathbf{T}$$

The scaling operation is $O(C^2)$ with fully parallel elementwise operations, which is significantly cheaper than an additional $O(C^3)$ forward substitution with sequential dependencies.

2.3.3 Complete Algorithm

Inputs:

- Current block: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{C\times d}, \boldsymbol{\beta}, \boldsymbol{\alpha} \in \mathbb{R}^C$
- Previous state: $\mathbf{S}_0 \in \mathbb{R}^{d\times d}$

Algorithm Steps:

1. **Precompute Decays**:
   - $\gamma_t = \prod_{i=1}^{t} \alpha_i$ for $t = 1, \ldots, C$
   - $\boldsymbol{\Gamma} \in \mathbb{R}^{C\times C}$ with
   $$\boldsymbol{\Gamma}_{i,j} = \begin{cases} \frac{\gamma_i}{\gamma_j} & \text{if } i \geq j \\ 0 & \text{otherwise} \end{cases}$$
   - Decayed matrices:
   $$\overrightarrow{\mathbf{S}_0} = \gamma_C \, \mathbf{S}_0 \in \mathbb{R}^{d\times d}$$
   $$\overleftarrow{\mathbf{Q}_t} = \gamma_t \, \mathbf{Q}_t \quad \Rightarrow \quad \overleftarrow{\mathbf{Q}} \in \mathbb{R}^{C\times d}$$
   $$\overrightarrow{\mathbf{K}_t} = \frac{\gamma_C}{\gamma_t} \mathbf{K}_t \quad \Rightarrow \quad \overrightarrow{\mathbf{K}} \in \mathbb{R}^{C\times d}$$
2. **Compute Interaction Matrix**:
   $$\mathbf{L} = \text{tril}\big(\text{diag}(\boldsymbol{\beta})\,\mathbf{K}\mathbf{K}^\top, -1\big) \in \mathbb{R}^{C\times C}$$
3. **Invert Matrix via Forward Substitution**:
   $$\mathbf{T} = (\mathbf{I} + \mathbf{L})^{-1}\,\text{diag}(\boldsymbol{\beta}) \in \mathbb{R}^{C\times C}$$

4. **Compute W**:

$$\mathbf{W} = \mathbf{T}\mathbf{K} \in \mathbb{R}^{C\times d}$$
$$\overleftarrow{\mathbf{W}}_t = \gamma_t \, \mathbf{W}_t$$

5. **Compute $\tilde{\mathbf{T}}$ via Similarity Transformation**:

$$\tilde{\mathbf{T}} = \mathbf{\Gamma} \odot \mathbf{T} \in \mathbb{R}^{C\times C}$$

6. **Compute $\tilde{\mathbf{U}}$**:

$$\tilde{\mathbf{U}} = \tilde{\mathbf{T}}\mathbf{V} \in \mathbb{R}^{C\times d}$$

7. **Compute Shared Term**:

$$\mathbf{A} = \tilde{\mathbf{U}} - \overleftarrow{\mathbf{W}}\mathbf{S}_0^\top \in \mathbb{R}^{C\times d}$$

8. **Update State (Inter-Chunk)**:

$$\mathbf{S}_C = \overrightarrow{\mathbf{S}_0} + \mathbf{A}^\top \overrightarrow{\mathbf{K}} \ \in \ \mathbb{R}^{d\times d}$$

9. **Compute Output (Intra-Chunk)**:

$$\mathbf{O}_{1:C} = \overleftarrow{\mathbf{Q}}\,\mathbf{S}_0^\top + (\mathbf{Q}\mathbf{K}^\top \odot \mathbf{\Gamma})\,\mathbf{A} \ \in \ \mathbb{R}^{C\times d}$$

2.4 Chunkwise Kimi Delta Attention

The derivation of the chunkwise KDA algorithm refers to Appendix B of [4].

Inputs:

- Current block: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{C\times d}$, $\boldsymbol{\beta} \in \mathbb{R}^{C}$, $\boldsymbol{\alpha} \in \mathbb{R}^{C\times d}$
- Previous state: $\mathbf{S}_0 \in \mathbb{R}^{d\times d}$

Algorithm Steps:

1. **Precompute Decays and Decayed Inputs**:
   - **Prefix cumulative decays** (decay from position 1 to r):

$$\boldsymbol{\gamma}^r := \boldsymbol{\gamma}^{1\to r} = \prod_{k=1}^{r} \boldsymbol{\alpha}_k \in \mathbb{R}^d, \qquad r = 1, \dots, C,$$

     where the multiplication is element-wise. Stack them as

$$\mathbf{\Gamma}^{1\to C} := \begin{bmatrix} \boldsymbol{\gamma}^1 \\ \vdots \\ \boldsymbol{\gamma}^C \end{bmatrix} = \begin{bmatrix} \boldsymbol{\gamma}^{1\to 1} \\ \boldsymbol{\gamma}^{1\to 2} \\ \vdots \\ \boldsymbol{\gamma}^{1\to C} \end{bmatrix} \in \mathbb{R}^{C\times d}.$$

   - **Suffix-to-end decays** (decay from position r to the end): define $\boldsymbol{\gamma}^0 := \mathbf{1}$ and compute

$$\boldsymbol{\gamma}^{r\to C} := \prod_{k=r}^{C} \boldsymbol{\alpha}_k = \frac{\boldsymbol{\gamma}^C}{\boldsymbol{\gamma}^{r-1}} \in \mathbb{R}^d, \qquad r = 1, \dots, C,$$

where the division is element-wise. Stack them as

$$\boldsymbol{\Gamma}^{i\to C} := \begin{bmatrix} \boldsymbol{\gamma}^{1\to C} \\ \boldsymbol{\gamma}^{2\to C} \\ \vdots \\ \boldsymbol{\gamma}^{C\to C} \end{bmatrix} \in \mathbb{R}^{C\times d}.$$

- **Decayed chunk inputs**:

$$\begin{aligned} \overleftarrow{\mathbf{Q}} &:= \boldsymbol{\Gamma}^{1\to C} \odot \mathbf{Q} \in \mathbb{R}^{C\times d} \\ \overleftarrow{\mathbf{K}} &:= \boldsymbol{\Gamma}^{1\to C} \odot \mathbf{K} \in \mathbb{R}^{C\times d} \\ \overleftarrow{\mathbf{K}}_{inv} &:= \mathbf{K} \oslash \boldsymbol{\Gamma}^{1\to C} \in \mathbb{R}^{C\times d}, \quad \text{(element-wise division)} \\ \overrightarrow{\mathbf{K}} &:= \boldsymbol{\Gamma}^{i\to C} \odot \mathbf{K} \in \mathbb{R}^{C\times d} \end{aligned}$$

- **Decayed initial state** (decayed over the entire chunk):

$$\overrightarrow{\mathbf{S}_0} := \mathrm{diag}(\boldsymbol{\gamma}^{C})\, \mathbf{S}_0 \in \mathbb{R}^{d\times d}.$$

2. **Compute Interaction Matrix**:

$$\mathbf{L} = \mathrm{tril}\Big(\mathrm{diag}(\boldsymbol{\beta})\, \overleftarrow{\mathbf{K}} (\overleftarrow{\mathbf{K}}_{inv})^{\top}, -1\Big) \in \mathbb{R}^{C\times C}.$$

3. **Invert Matrix**: It uses forward substitution to get the inverse.

$$\mathbf{M} = (\mathbf{I} + \mathbf{L})^{-1} \mathrm{diag}(\boldsymbol{\beta}) \in \mathbb{R}^{C\times C}.$$

4. **Compute W and U**:

$$\mathbf{W} = \mathbf{M}\, \overleftarrow{\mathbf{K}} \in \mathbb{R}^{C\times d}, \qquad \mathbf{U} = \mathbf{M}\, \mathbf{V} \in \mathbb{R}^{C\times d}.$$

5. **Compute A**: It combines the update and historical correction terms.

$$\mathbf{A} = \mathbf{U} - \mathbf{W}\mathbf{S}_0 \in \mathbb{R}^{C\times d}$$

6. **Update State (Inter-Chunk)**:

$$\mathbf{S}_C = \overrightarrow{\mathbf{S}_0} + \Big(\overrightarrow{\mathbf{K}}\Big)^{\top} \mathbf{A} \in \mathbb{R}^{d\times d}.$$

7. **Compute Output (Intra-Chunk)**:

$$\mathbf{O}_{1:C} = \overleftarrow{\mathbf{Q}}\, \mathbf{S}_0 + \mathrm{tril}\left(\overleftarrow{\mathbf{Q}}\, (\overleftarrow{\mathbf{K}}_{inv})^{\top}, -1\right) \mathbf{A} \in \mathbb{R}^{C\times d}.$$

References

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[2] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020.

[3] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. 2021.

[4] 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.

[5] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024.

[6] Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length, 2025.