A Survey of Efficient Attention Methods: Hardware-efficient, Sparse, Compact, and Linear Attention

Jintao Zhang[1], Rundong Su[*1], Chunyu Liu[*1], Jia Wei[*1], Ziteng Wang[*1], Haoxu Wang[*1], Pengle Zhang[1], Huiqiang Jiang[1], Haofeng Huang[1], Chendong Xiang[1], Haocheng Xi[2], Shuo Yang[2], Xingyang Li[3], Yuezhou Hu[2], Tianyu Fu[1], Tianchen Zhao[1], Yicheng Zhang[1], Boqun Cao[1], Youhe Jiang[1], Chang Chen[1], Kai Jiang[1], Huayu Chen[1], Min Zhao[1], Xiaoming Xu[1], Yi Wu[4], Fan Bao[4], Jun Zhu[1], Jianfei Chen[1]

Abstract

In modern transformers, the attention operation is the only component with a time complexity of $\mathcal{O}(N^2)$, whereas all other operations scale linearly as $\mathcal{O}(N)$, where N denotes the sequence length. As sequence lengths in generative models (e.g., language and video generation) continue to increase, improving the efficiency of attention has become increasingly critical. Recently, numerous excellent works have been proposed to enhance the computational efficiency of attention operation. Broadly, these works can be classified into four categories: (1) Hardware-efficient attention: Optimizing attention computation efficiency by leveraging hardware characteristics. (2) Sparse attention: Selectively performing a subset of computations in attention while omitting others. (3) Compact attention: Compressing the KV cache of attention by weight sharing or low rank decomposition while keeping computational cost unchanged, as with a full-sized KV cache. (4) Linear attention: Redesigning the computational formulation of attention to achieve $\mathcal{O}(N)$ time complexity. In this paper, we present a comprehensive survey of these efficient attention methods.

Contents

[*]Co-second authorship. [1]Tsinghua University. [2]UC Berkeley. [3]MIT. [4]ShengShu.

1 Introduction

Attention is the core of transformers (Vaswani et al., 2017) that powers applications from language modeling to vision understanding and multimodal generation. Despite their success, a fundamental computational bottleneck exists: the time complexity of attention is $\mathcal{O}(N^2)$, which grows quadratically with sequence length N. As models scale to handle increasingly long contexts, this quadratic cost becomes prohibitive, limiting deployment in applications with low latency requirements or on resource-constrained hardware. Addressing this challenge has led to a surge of research efforts toward efficient attention.

In general, efficient attention methods aim to reduce time or memory costs while preserving the effectiveness of the standard attention. These approaches can be divided into four main categories: (1) **Hardware-efficient attention** optimizes the implementation efficiency of the original attention without changing its computation logic. By better utilizing modern GPU features, techniques such as matrix tiling, kernel fusion (Dao et al., 2022; Dao, 2024), and quantization to leverage the low-bit Tensor Core (Zhang et al., a; 2024a; 2025b;d; Shah et al., 2024) are introduced to accelerate attention. (2) **Compact attention** compresses the KV cache (Zhao et al., 2023) to reduce memory overhead during inference. This is typically achieved through weight sharing (Ainslie et al., 2023) and low-rank decomposition (Liu et al., 2024a), enabling memory-efficient caching while keeping computational cost unchanged compared with a full-sized KV cache. (3) **Sparse attention** reduces computational cost by skipping calculations for non-critical parts of the attention matrix. This approach is viable because the attention matrix is commonly observed to be sparse, with a large number of its values being close to zero. Typically, sparse attention is implemented by applying a fixed or dynamic sparse mask to the matrix, which directs the model to perform attention operations only on the unmasked positions, i.e., the crucial positions. (4) **Linear attention** removes the softmax operation, which enables reordering the matrix multiplications in the attention and avoids the $\mathcal{O}(N^2)$ time complexity. Specifically, it first computes the key–value product $K^\top V$ and then multiplies the result with the queries Q, reducing the overall computational complexity to linear $\mathcal{O}(N)$.

In this survey, we systematically and comprehensively review efficient attention, summarizing existing works, their motivations, and core principles. First, in Section 2, we introduce the preliminaries of attention, including FlashAttention and the differences in attention methods across various tasks. Next, in Section 3, we present the motivation, fundamental idea, and common practices of the four categories, i.e., hardware-efficient attention, sparse attention, compact attention, and linear attention. Subsequently, in Sections 4, 5, 6, and 7, we provide a detailed discussion of each category, including their unified frameworks and formalized formulations, a summary of the methods along with their features, and dedicated descriptions of the implementation details for individual methods.

2 Preliminary

2.1 Standard Attention Computation

The core idea of attention (Bahdanau et al., 2016) is to dynamically retrieve information by computing a weighted sum of values, where the weights are determined by the relevance of each value's corresponding key to a given query. A standard implementation is the Scaled Dot-Product Attention (Vaswani et al., 2017). It operates on matrices of queries (Q), keys (K), and values (V). The relevance score is calculated as the dot product of a query with a key. To allow the model

Table 1: Notations.

Notation	Shape	Meaning
N, n, h, d	1×1	sequence length, num heads and head dim of attention
D_m, D	1×1	hidden dim of a model, total dim of all attention heads ($D = hd$)
Q, K, V, O	$N \times d$	qurey, key, value, and output of attention
S, P	$N \times N$	$S = QK^\top, P = \text{Softmax}(S)$
M	$N \times N$	the mask needed to be added to P
b_q, b_{kv}	1×1	flashattention block size for Q and K, V
$\mathbf{M}$	$\frac{N}{b_q} \times \frac{N}{b_{kv}}$	the mask needed to be added to each flashattention block of P
$\mathbf{Q}_i, \mathbf{O}_i$	$b_q \times d$	flashattention block for Q, O, i.e., $Q[i \times b_q : (i+1) \times b_q]$
$\mathbf{K}_j$, $\mathbf{V}_j$	$b_{kv} \times d$	flashattention block for K, V, i.e., $K[j \times b_{kv} : (j+1) \times b_{kv}]$
$\mathbf{S}_{i,j}, \widetilde{\mathbf{P}}_{i,j}$	$b_q \times b_{kv}$	flashattention block for S and P
$m_{i,j}, l_{i,j}$	$b_q \times 1$	prefix rowmax and expsum statistics for online softmax
q_t, k_t, v_t	$1 \times d$	query, key, value tokens of attention inputs
H_t, h_t	$d \times d, 1 \times d$	hidden state formed by summing $k_{t-1}^T v_{t-1}$ up to the current state
G	$d \times d$	forget gate in linear attention
β	1×1	select gate in linear attention
ϕ	-	kernel functions in linear attention

to jointly attend to information from different representation subspaces, Multi-Head Attention (MHA) (Vaswani et al., 2017) is operated as follows: The queries, keys, and values each undergo h parallel projections using different learned weight matrices to produce independent inputs for h heads. Then, the scaled dot-product attention is applied in parallel to each head, yielding h distinct output matrices. This final linear projection acts to synthesize the information from all attention heads, learning to optimally combine their outputs into a single final output. For each individual attention head, the computation is formulated as:

$$S = \frac{QK^\top}{\sqrt{d}}, \quad P = \text{softmax}(S), \quad O = PV. \tag{1}$$

Here, $Q, K, V \in \mathbb{R}^{N \times d}$ are the input matrices for a single head, $S \in \mathbb{R}^{N \times N}$ is the matrix of attention scores, $O \in \mathbb{R}^{N \times d}$ is the output of the head. The attention probability matrix $P \in \mathbb{R}^{N \times N}$, also known as the attention map, represents the normalized importance of each query-key pair. For numerical stability, the softmax function, $\text{softmax}(S)_{i,j} = \frac{\exp(S_{i,j})}{\sum_k \exp(S_{i,k})}$, is implemented by subtracting the row-wise maximum $m_i = \max(S_i)$ before exponentiation to prevent overflow: $P_{i,j} = \frac{\exp(S_{i,j} - m_i)}{\sum_k \exp(S_{i,k} - m_i)}$.

2.2 Background of GPU

Modern GPUs are highly parallel processors featuring a hierarchical memory architecture. To explain the hardware, we use NVIDIA's CUDA terminology, as its computational and memory abstractions are common to most modern GPUs. A GPU's computational power is derived from its multiple streaming multiprocessors (**SM**s). Each SM integrates numerous **CUDA Cores** for general-purpose workloads and specialized **Tensor Cores** to accelerate matrix multiplication (NVIDIA Corporation, 2025). The aggregate performance of these cores determines the GPU's overall computational throughput, $\underline{C_t}$ (operations/second), measured in floating-point operations per second (`FLOPS`). To supply data to these computational units, the GPU employs a tiered memory hierarchy. Each SM is coupled with a small, high-bandwidth on-chip **Shared Memory**. Concurrently, all SMs share

access to a large-capacity **Global Memory**, typically High Bandwidth Memory (HBM), which is characterized by lower bandwidth (Jia et al., 2018), denoted as B_w (bytes/second).

To better illustrate these concepts, consider a GPU performing matrix multiplication $C = AB$, where $A \in \mathbb{R}^{M\times K}$, $B \in \mathbb{R}^{K\times N}$, and $C \in \mathbb{R}^{M\times N}$ are all stored in float data type. The computation can be executed by transferring A and B from global memory to shared memory for computing, and then writing C back from shared memory to global memory progressively. The total I/O (read and write) volume is $4(MK + KN + MN)$ Bytes, while the total computational workload is $2MKN$ operations (including both multiplications and additions). The total I/O time is $T_{\text{I/O}} = \frac{4(MK+KN+MN)}{B_w}$ seconds, and the total computation time is $T_{\text{compute}} = \frac{2MKN}{C_t}$ seconds. Since computation and I/O can be overlapped via asynchronous pipelining (NVIDIA, 2025; Shah et al., 2024) the overall latency is determined by $\max(T_{\text{I/O}}, T_{\text{compute}})$. If $T_{\text{I/O}}$ dominates, the process is memory-bound; if T_{compute} dominates, it is compute-bound.

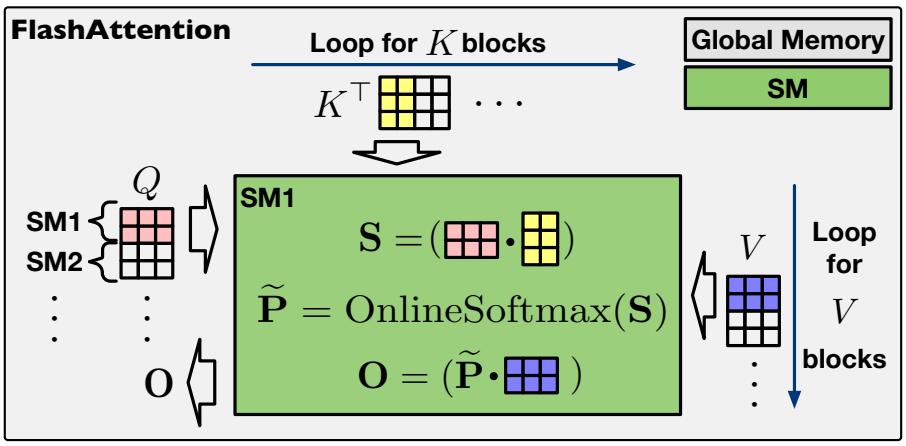

Figure 1: Illustration of FlashAttention. Different Q blocks are processed on different SMs, where each SM iteratively loads K and V blocks and finally outputs an O block.

2.3 FlashAttention

The query, key, and value matrices Q, K, and V have dimensions $N \times d$, while the score and probability matrices S and P are $N \times N$. Although d is relatively small (e.g., 64 or 128), N can reach thousands or even millions. Consequently, the $N \times N$ matrices (S, P) are far larger than (Q, K, V), and a naive implementation suffers from heavy global memory I/O when reading and writing (S, P). As shown in Figure 1, FlashAttention (Dao et al., 2022; Dao, 2024) addresses this by partitioning Q, K, and V along the token dimension into blocks $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ of sizes b_q, b_{kv}, and b_{kv}, respectively. To avoid memory I/O for (S, P), it uses online softmax (Milakov & Gimelshein, 2018) to compute each output block $\mathbf{O}_i$ progressively. First, for each block of $\mathbf{K}_i, \mathbf{V}_i$, it computes iteratively as follows:

$$\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^\top / \sqrt{d}, \quad (m_{i,j}, \widetilde{\mathbf{P}}_{ij}) = \tilde{\sigma}(m_{i,j-1}, \mathbf{S}_{ij}). \tag{2}$$

$$l_{ij} = \exp(m_{i,j-1} - m_{i,j}) l_{i,j-1} + \text{rowsum}(\widetilde{\mathbf{P}}_{ij}). \tag{3}$$

$$\mathbf{O}_{ij} = \text{diag}\left(\exp(m_{i,j-1} - m_{ij})\right) \mathbf{O}_{i,j-1} + \widetilde{\mathbf{P}}_{ij} \mathbf{V}_j. \tag{4}$$

Here, m_{ij} and l_{ij} are $b_q \times 1$ vectors initialized to $-\infty$ and 0, respectively. The operator $\tilde{\sigma}()$ denotes the online softmax, which updates according to $m_{ij} = \max\{m_{i,j-1}, \text{rowmax}(\mathbf{S}_{ij})\}$ and $\widetilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - m_{ij})$. After completing all iterations, i.e., $j = N/b_{kv}$, a final output block is given by $\mathbf{O}_i = \text{diag}(l_{ij})^{-1}\mathbf{O}_{ij}$.

2.4 Attention in Different Tasks

2.4.1 Attention in Non-Autoregressive Models

In non-autoregressive models such as BERT (Koroteev, 2021) and Vision Transformer (Han et al., 2022), attention is computed in parallel over a sequence of N input tokens , with each token attending to every other:

$$O = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V. \tag{5}$$

For N input tokens, the query Q and key K matrices have dimensions $N \times d$, and the value matrix V has dimensions $N \times d$. The computational cost comes from the matrix multiplications $QK^\top$ and PV, which have a time complexity of $\mathcal{O}(N^2d)$. Most models today use FlashAttention (Dao, 2024) to efficiently read the input matrices, with $\mathcal{O}(Nd)$ I/O complexity. However, for a large number of tokens N, the quadratic growth in arithmetic operations significantly outweighs the $\mathcal{O}(Nd)$ memory access cost. Thus, this operation is typically **compute-bound** for non-autoregressive models.

2.4.2 Attention in LLM Training

To preserve the autoregressive property of LLMs (Zhao et al., 2023), a lower-triangular causal mask is applied during training, ensuring that each token can only attend to itself and preceding tokens:

$$O = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right)V. \tag{6}$$

where the mask matrix M is a lower-triangular mask defined as:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \le i. \\ -\infty & \text{if } j > i. \end{cases} \tag{7}$$

The matrix dimensions are identical to the non-autoregressive case, and the addition of the $N \times N$ causal mask M does not change the dominant computational complexity of $\mathcal{O}(N^2d)$. Similar to attention without a causal mask in non-autoaggressive tasks, attention in LLM training is also **compute-bound**.

2.4.3 Attention in LLM Inference

The inference process of modern LLMs can be divided into two main stages: (1) the **prefilling phase**, which processes the initial prompt and produces the first output token; and (2) the **decoding phase**, which generates the remaining tokens autoregressively. In the prefilling phase, the model processes all input tokens in parallel, computing the complete Q, K, V matrices for the prompt in the same manner as in training (Section 2.4.2), which is typically compute-bound. The resulting K and V matrices are stored in the GPU's global memory as the KV cache (Zhao et al., 2023). In the decoding phase, the model generates one token at a time. For each new token, it computes the

corresponding q, k, v vectors, and the query q attends to all keys and values. The attention output is computed as:

$$o = \text{softmax}\left(\frac{q(K\|k)^\top}{\sqrt{d}}\right)(V\|v). \tag{8}$$

Here, the query q, new key k, and new value v are $1 \times d$ vectors, while the cached key matrix K and value matrix V are of size $N \times d$, where N is the current sequence length. The symbol $\|$ denotes the vertical concatenation operation. The term $K\|k$ represents appending the new key vector k as a new row to the existing key matrix K. Similarly, $V\|v$ represents appending the new value vector v as a new row to the value matrix V. Both resulting matrices have a new size of $(N+1) \times d$. This operation involves computation complexity of $\mathcal{O}(Nd)$ for matrix-vector multiplication and memory complexity of $\mathcal{O}(Nd)$ for loading the KV cache. As observed in FlashAttention (Dao et al., 2022), on modern GPUs, I/O operations are often significantly slower than computation, even when both have equivalent complexity. Thus, the performance of decoding is typically limited by memory bandwidth, not compute speed, making this operation memory-bound.

3 Overview

In this section, we outline the motivations, fundamental ideas, and common practices of four major categories: hardware-efficient attention, sparse attention, compact attention, and linear attention.

3.1 Hardware-Efficient Attention

To design efficient operations on modern GPUs, it is crucial to understand their memory hierarchy, as mentioned in Section 2.2. A typical workflow involves loading data from global memory to shared memory, performing computations, and writing the results back to global memory. To hide memory access latency, computation and memory I/O can be fully overlapped and executed in parallel through pipelining (NVIDIA, 2025; Shah et al., 2024). Consequently, the total execution time is dominated by the longer of the two, which becomes the performance bottleneck. Specifically, when the computation time is greater than the memory I/O time, the operation is compute-bound; when the memory I/O time is greater than the computation time, the operation is memory-bound.

Naive attention is inefficient under heavy I/O. As described in Section 2.1, naive attention is memory-bound because of the materialization of two large intermediate matrices, the attention scores $S = QK^\top$ and the probability matrix $P = \text{softmax}(S)$, which are both of size $\mathcal{O}(N^2)$ for a sequence of length N. Storing these matrices requires writing them to and reading them from the slow global memory, creating a significant I/O bottleneck. Let's consider an example on an NVIDIA A100 GPU (NVIDIA Corporation, 2020), which has a global memory bandwidth of up to 2.0 TB/s and a peak throughput of 312 `TFLOPS` for `FP16` operations (NVIDIA). Suppose we have input matrices Q, K, and V with dimensions $[100\text{K}, 64]$. The intermediate matrices S and P will have dimensions $[100\mathbf{K}, 100\mathbf{K}]$. The size of just one of these intermediate matrices in 16-bit precision is: $100\text{K} \times 100\text{K} \times 2$ bytes $= 2e10$ bytes $\approx 19\texttt{GB}$ ($1\text{ K} = 1024$). The total number of floating-point operations (`FLOPs`) for the two matrix multiplications ($QK^\top$ and PV) is $4N^2d$. The computation time is: $T_{\text{compute}} = \frac{2\times(2N^2d)}{C_t} = \frac{4\times 1e10\times 64}{312\times 1e12} \approx 8.2$ ms, where C_t quantifies the GPU's raw computational speed in operations per second. The memory access time, dominated by writing and reading S and P from global memory, is: $T_{\text{I/O}} = \frac{2\times(\text{size of } S + \text{size of } P)}{B_w} = \frac{8\times 1e10 \text{ bytes}}{2\times 1e12 \text{ bytes/s}} \approx 40$ ms, where B_w measures GPU's memory data transfer rate in Bytes per second. The memory access time

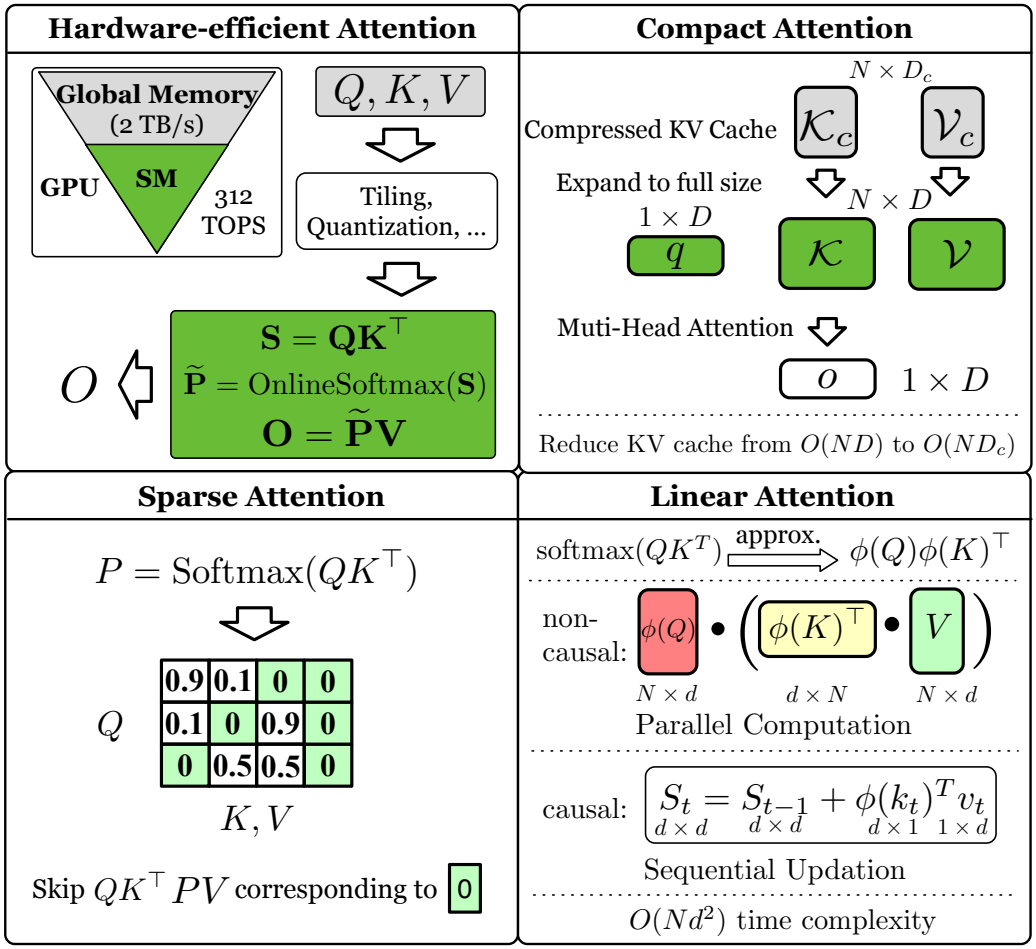

Figure 2: Overview of efficient attention methods.

is nearly five times greater than the computation time, making standard attention an inefficient, memory-bound operation.

To relieve the I/O burden, FlashAttention introduces a tiling strategy that partitions the Q, K, and V matrices into smaller tiles ($\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$) that fit entirely within the shared memory of one SM. This design integrates with online softmax (Milakov & Gimelshein, 2018), eliminating the need to write and read the $N \times N$ matrices S and P between global and shared memory. This transformation turns standard softmax attention from memory-bound to compute-bound, significantly increasing the attention efficiency.

As discussed in Section 2.4.3, after using FlashAttention to optimize I/O, LLM inference is characterized by two distinct phases: compute-bound prefilling phase and memory-bound decoding phase. Hardware-efficient attention methods are therefore engineered to address the specific bottlenecks of each phase. For the compute-bound prefilling stage, hardware-efficient methods focus on maximizing parallel processing and computational throughput. For the memory-bound decoding stage, hardware-efficient methods focus on accelerating the I/O for the KV cache.

3.2 Compact Attention

Exploding KV cache memory in LLM inference. During inference of LLMs, vanilla Multi-Head Attention (MHA) (Vaswani et al., 2017) introduces $\mathcal{O}(Nhd)$ memory complexity for KV cache. This consumes a lot of GPU memory and can even dominate VRAM usage when the context is long. For example, with $N = 128$K context length, $h = 32$ attention heads and $d = 128$ head dimensionality in a 48-layer Transfomer with MHA, the KV cache accounts for 96`GB` memory when saved in `bfloat16` format, exceeding the VRAM needed for parameters of the entire model and most GPUs including A100, H100, etc. The problem becomes even worse in batch decoding systems where KVs of multiple sequences need to be cached simultaneously.

KV cache compression. To mitigate the high memory cost of the KV cache, a common approach is to decouple the KV entries stored in memory ("storage KV") from those used in attention computation ("computation KV"), and then compress the storage KV while preserving the size of computation KV, which we call "Compact Attention". Let $D = hd$ denote the total dimensionality for attention computation, and $\mathcal{K} = [K^{(1)}, \ldots, K^{(h)}]$ denotes the concatenation of h attention heads with $K^{(i)} \in \mathbb{R}^{N\times d}$ being the key matrix of head i, likewise for $\mathcal{V}$. To reduce KV cache memory, these methods first store compact key and value tensors, $\mathcal{K}_c, \mathcal{V}_c \in \mathbb{R}^{N\times D_c}$, where $D_c < D$ is the compressed KV size per token, instead of the larger tensors $\mathcal{K}, \mathcal{V} \in \mathbb{R}^{N\times D}$. Before attention computation, the compact tensors are expanded to full size via a function from D_c to D, typically implemented by replication. In this way, the KV cache becomes much smaller than MHA to save memory, while the computation KV remains the same size to avoid degraded performance.

The formulation at a high level is presented as follows.

$$\mathcal{K}_c, \mathcal{V}_c \in \mathbb{R}^{N\times D_c} \tag{9}$$

$$\mathcal{K}, \mathcal{V} = \text{Expand}_{\mathcal{K}}(\mathcal{K}_c), \text{Expand}_{\mathcal{V}}(\mathcal{V}_c) \in \mathbb{R}^{N\times D} \tag{10}$$

$$o = \text{MHA}(q, \mathcal{K}, \mathcal{V}) \tag{11}$$

where $\text{MHA}(\cdot)$ denotes the multi-head attention operation as stated in Section 2.1. Compact attention methods reduce the size of KV cache $\mathcal{K}_c$ and $\mathcal{V}_c$ from $O(ND)$ to $O(ND_c)$ to save memory, while the computation KV states $\mathcal{K}, \mathcal{V}$ remain the same size as MHA with the $\text{Expand}(\cdot)$ function to avoid degraded performance.

3.3 Sparse Attention

Sparsity for matrix multiplication efficiency. A matrix is considered sparse when it contains a large number of zero elements. Such sparsity can be exploited to accelerate matrix multiplication (MatMul) $C = AB$ in two main cases: (1) if $C[i, j] = 0$, the computation involving the entire row $A[i, :]$ and column $B[:, j]$ can be skipped; (2) if $A[i, j] = 0$, the product $A[i, j] \cdot B[j, :]$ contributes nothing to $C[i, :]$ and can be omitted; similarly, if $B[i, j] = 0$, the computation of $A[:, i] \cdot B[i, j]$ can be avoided.

The Softmax operation applies an exponential transformation to the inputs and normalizes them such that the outputs sum to 1. This operation significantly compresses smaller input values toward near-zero values. Consequently, the attention map in attention $P = \text{Softmax}(QK^\top/\sqrt{d})$ exhibits inherent sparsity (Child et al., 2019; Zhang et al., 2025c), as the softmax operation often creates many values approaching zero. Then, when some values in P are equal to zero, the corresponding computation in matrix multiplication of $QK^\top$ and PV can be skipped. Sparse attention exploits such sparsity to accelerate attention by two steps. First, it constructs a *sparse mask* M, which

determines which elements of the attention map P should be computed. Second, it computes attention only for the parts corresponding to the *sparse mask* M:

$$P = \text{Softmax}(M + QK^\top/\sqrt{d}). \tag{12}$$

$$O = PV. \tag{13}$$

where M is an $N \times N$ matrix whose elements are either 0 or $-\infty$. $M_{ij} = 0$ specifies that both the attention score $q_i k_j^\top$ and its corresponding output $P[i,j]v_j$ should be computed, while $M_{ij} = -\infty$ indicates these computations should be skipped.

Sparse FlashAttention. Typically, the actual computation process of sparse attention needs to be combined with FlashAttention to achieve efficient performance. This is because FlashAttention-based methods improve efficiency by obviating the need to write the intermediate S and P matrices to global memory and subsequently read them back into shared memory. FlashAttention performs block-wise matrix multiplication for $QK^\top$, and PV, which requires that the granularity of sparse attention must at least match the block size used in FlashAttention. Since FlashAttention computes $QK^\top$ and PV in a block-wise manner, it imposes a fundamental constraint: the granularity of any applied sparsity cannot be finer than the operator's block size. Implementing sparse FlashAttention is intuitive. By skipping certain block matrix multiplications of $\mathbf{Q}_i\mathbf{K}_j^\top$ and $\widetilde{\mathbf{P}}_{ij}\mathbf{V}_j$ according to the M, we can accelerate the attention computation.

3.4 Linear Attention

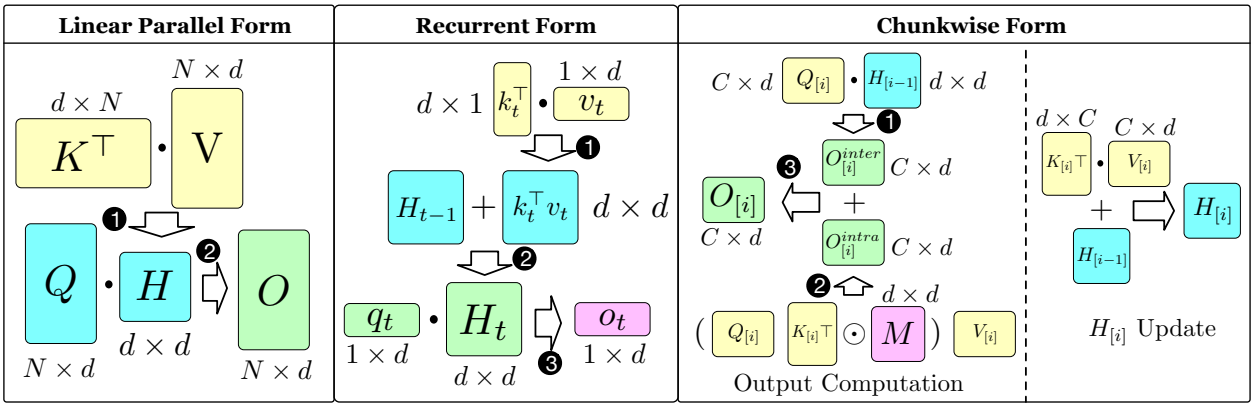

Figure 3: Linear attention computation forms.

The core idea of linear attention is to reduce computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ with respect to sequence length N. This is achieved by replacing the softmax function with a kernel function ϕ, which allows the standard attention computation to be reordered as:

$$O = \phi(Q)(\phi(K)^\top V). \tag{14}$$

By first calculating the term $(\phi(K)^\top V)$, this method avoids the explicit construction of the massive $N \times N$ matrix $QK^\top$, thus achieving linear complexity.

Computational forms for training and inference. The need to reconcile the $\mathcal{O}(N)$ efficiency of linear attention with the causality required by autoregressive (AR) models leads to three distinct computational forms.

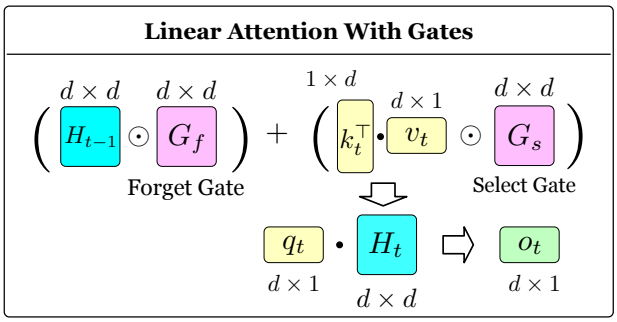

Figure 4: Linear attention with forget and select gates.

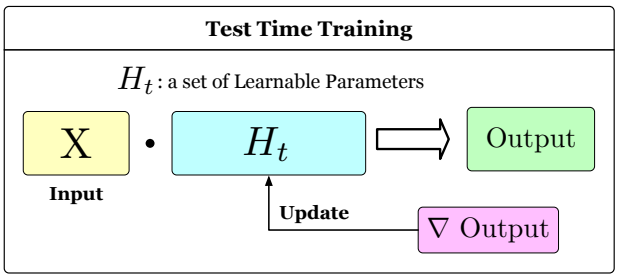

Figure 5: Overview of test time training.

1. `Linear Parallel Form.` This is the direct application of the core idea, defined as $O = \phi(Q)(\phi(K)^\top V)$. It is highly efficient for the training and inference of non-autoregressive (NAR) tasks, where the entire sequence is processed simultaneously. However, this form is unsuitable for AR models for two reasons: (1) For autoregressive inference, its simultaneous processing violates step-by-step causality. (2) For autoregressive training, forcing causality with a mask $(O = (\phi(Q)\phi(K)^\top \odot M)V)$ creates a `Quadratic Parallel Form`, reverting complexity to an inefficient $\mathcal{O}(N^2)$.

2. `Recurrent Form.` This form is designed for efficient autoregressive inference. It introduces a fixed-size state $H_t = \sum_{i=1}^{t} \phi(k_i)^\top v_i$ that is updated recurrently: $H_t = H_{t-1} + \phi(k_t)^\top v_t$. The output is then computed as $o_t = \phi(q_t)H_t$. This approach eliminates the growing KV cache of standard attention, making each generation step a constant-time $\mathcal{O}(1)$ operation. While ideal for inference, its inherently sequential nature is hardware-inefficient for training.

3. `Chunkwise Form.` This form is a hybrid solution designed for autoregressive training, resolving the issues of the previous forms. It divides the sequence into fixed-size chunks and uses a dual strategy: Attention is computed in quadratic parallel form within each chunk to maximize parallelization. Causality is maintained by passing a recurrent state between chunks. This gives a practical complexity of $\mathcal{O}(NCd + Nd^2)$, where C is the chunk size.

Forget and Select Gates. To enable the fixed-size hidden state H_t to dynamically hold the most relevant information, the forget and select gates were introduced. Then the H_t update can be formulated as:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot \phi(k_t)^\top v_t. \tag{15}$$

Here, inspired by gates of RNNs(Medsker et al., 2001), $G_f^{(t)}$ acts as a forget gate, deciding how much historical information (H_{t-1}) to keep, and $G_s^{(t)}$ serves as a select gate, determining how much current information to hold.

Test Time Training. Test-Time Training (TTT) (Sun et al., 2024) views the hidden state H_t as a set of learnable parameters, also called 'fast weights' (Schlag et al., 2021b). TTT will continuously update the hidden state via gradient descent (Robbins & Monro, 1951) during both training and inference.

4 Hardware-efficient Attention

In this section, we present unified formulations for hardware-efficient attention, provide a comprehensive summary of representative methods and their key features, and elaborate on the implementation details of individual approaches.

Table 2: Summary of hardware-efficient attention methods. − means no additional preprocessing.

Method	Category	$\Psi(\cdot)$ Type	$\Theta(\cdot)$ Type
FlashAttention (Dao et al., 2022)	Prefilling	−	−
FlashAttention2 (Dao, 2024)	Prefilling	−	−
SageAttention (Zhang et al., a)	Prefilling	`INT8` quantizer	−
SageAttention2 (Zhang et al., 2024a)	Prefilling	`INT4` quantizer	`FP8` quantizer
SageAttention2++ (Zhang et al., 2025d)	Prefilling	`INT4` quantizer	`FP8` quantizer
SageAttention3 (Zhang et al., 2025b)	Prefilling	`FP4` quantizer	`FP4` quantizer
FlashAttention3 (Shah et al., 2024)	Prefilling	− or `FP8` quantizer	− or `FP8` quantizer
FlashDecoding (Dao et al., 2023)	Decoding	split KV cache	split KV cache
KVQuant (Hooper et al., 2024)	Decoding	`INT2/3/4` quantizer	`INT2/3/4` quantizer
KiVi (Liu et al., 2024d)	Decoding	`INT2` quantizer	`INT2` quantizer
PagedAttention (Kwon et al., 2023)	Decoding	reallocate KV cache	reallocate KV cache
FlashInfer (Ye et al., 2025)	Decoding	reallocate KV cache	reallocate KV cache

4.1 Framework

Corresponding to two stages in LLMs inference as introduced in Section. 2.4.3, *Hardware-efficient Attention* can be divided into two categories, i.e., prefilling and decoding. Inspired by FlashAttention, prefilling methods also partition Q, K and V into blocks $\mathbf{Q}_i$, $\mathbf{K}_i$, $\mathbf{V}_i$. They compute each output block $\mathbf{O}_i$ iteratively as follows:

$$\hat{\mathbf{Q}}, \hat{\mathbf{K}}, \hat{\mathbf{V}} = \Psi(\mathbf{Q}), \Psi(\mathbf{K}), \Theta(\mathbf{V}). \tag{16}$$

$$\mathbf{S} = \hat{\mathbf{Q}}\hat{\mathbf{K}}^\top, \quad \hat{\mathbf{P}} = \Theta(\text{softmax}(\mathbf{S})), \quad \mathbf{O} = \hat{\mathbf{P}}\hat{\mathbf{V}}, \tag{17}$$

where $\Psi(\cdot), \Theta(\cdot)$ are preprocess functions to accelerate computation, e.g., quantization functions. For simplification, we omitted the division by $\sqrt{d}$ and the details of online softmax. Decoding methods also partition K and V into blocks, but their input $\mathbf{q}$ is a vector, so the output vector $\mathbf{o}$ is computed as follows:

$$\hat{\mathbf{K}}, \hat{\mathbf{V}} = \Psi(\mathbf{K}), \Theta(\mathbf{V}). \tag{18}$$

$$\mathbf{s} = \mathbf{q}\hat{\mathbf{K}}^\top, \quad \mathbf{p} = \text{softmax}(\mathbf{s}), \quad \mathbf{o} = \mathbf{p}\hat{\mathbf{V}}. \tag{19}$$

where $\Psi(\cdot), \Theta(\cdot)$ are KV cache preprocess functions. In Table 2, we summarize these two categories of hardware-efficient attention methods. The $\Phi(\cdot)$ Type and $\Theta(\cdot)$ Type refer to different pre-processing functions. Splitting the KV cache means splitting the KV cache along sequence length to be loaded by different SMs. Reallocating the KV cache means organizing the KV cache memory into different storage formats. For example, PagedAttention allocates KV cache into fixed-size pages to reduce memory fragments, boosting the I/O speed.

4.2 Methods

4.2.1 Prefilling

Hardware-efficient attention methods for optimizing the prefilling phase typically aim to accelerate computation by fully exploiting hardware capabilities.

► **FlashAttention1 & 2.** Building on the online softmax (Milakov & Gimelshein, 2018) technique, FlashAttention (Dao et al., 2022) pioneered the fusion of scaled dot-product attention into a single kernel by partitioning Q, K, V into blocks and computing $\mathbf{S}_{ij} = \mathbf{Q}_i\mathbf{K}_j^\top$ with on-the-fly softmax computation. Due to the linearity of matrix multiplication PV, the output blocks $\mathbf{O}_{ij}$ can be incrementally updated alongside the online softmax statistics with rescaling factor $\exp(m_{i,j-1} - m_{i,j})$ (as detailed in Section 2.3). FlashAttention (Dao et al., 2022) keeps $\mathbf{K}_j, \mathbf{V}_j$ in shared memory while iterating over $\mathbf{Q}_i$ blocks, requiring each SM to read/write intermediate results through global memory to update $\mathbf{O}_{ij} \leftarrow \mathbf{O}_{i,j-1}$, resulting in $O(N^2d/b_{kv})$ memory transfers. To reduce this quadratic I/O complexity, FlashAttention2 (Dao, 2024) inverts the loop order by keeping $\mathbf{Q}_i$ in shared memory and iterating over $\mathbf{K}_j, \mathbf{V}_j$ blocks, updating $l_{ij}, m_{ij}, \mathbf{O}_{ij}$ locally within each SM, achieving $O(Nd)$ global memory transfers in total. For backpropagation, both versions recompute $\mathbf{S}_{ij} = \mathbf{Q}_i\mathbf{K}_j^\top$ to calculate gradients, leveraging the stored l_i, m_i global statistics from the forward pass to directly obtain exact probabilities $\mathbf{P}_{ij} = \exp(\mathbf{S}_{ij} - m_i)/l_i$ without additional softmax operation. Consequently, FlashAttention and FlashAttention2 achieve approximately 3× and 10× speedup, respectively, in training (forward + backward) compared to standard PyTorch implementations, and have become the de facto baseline for long-context training.

► **SageAttention.** SageAttention (Zhang et al., a) builds on the block-tiling strategy of FlashAttention (Dao et al., 2022) and quantizes Q and K to `INT8` on a per-block basis, where each block $\mathbf{Q}_i$ and $\mathbf{K}_j$ has its own quantization scale $\delta_{\mathbf{Q}_i} = \max(|\mathbf{Q}_i|)/127$ and $\delta_{\mathbf{K}_j} = \max(|\mathbf{K}_j|)/127$. Using these scales, the attention score is approximated as $S_{i,j} \approx \hat{\mathbf{Q}}_i\hat{\mathbf{K}}_j^\top(\delta_{\mathbf{Q}_i} \times \delta_{\mathbf{K}_j}/\sqrt{d})$. To mitigate quantization error, SageAttention applies a preprocessing step that subtracts the token-wise mean from K, and it keeps $\widetilde{\mathbf{P}}_{i,j}$ and $\mathbf{V}_j$ in `FP16` while using an `FP16` accumulator, instead of `FP32`, to compute $\widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j$. With these optimizations, SageAttention achieves a 2.1× speedup over FlashAttention while preserving end-to-end performance across language, image generation, and video generation models.

► **SageAttention2 & 2++.** SageAttention2 & 2++ (Zhang et al., 2024a; 2025d) further accelerate attention by quantizing $QK^\top$ directly to `INT4`, leveraging the speed advantage of `INT4` tensor cores. To reduce the quantization error introduced by low bit-width and outliers, SageAttention2 exploits the layout of Tensor Core and proposes a per-thread quantization granularity. It also applies the preprocessing step of Sageattention to both K and Q, removing outliers from both matrices before quantization. For the PV computation, SageAttention2 utilizes `FP8` tensor cores and adopts a two-level accumulation strategy to address the FP22 accumulator limitations in NVIDIA's Ada and Hopper architectures. SageAttention2++ further adopts `FP16` accumulators for PV computation to achieve additional speedup on consumer-level GPUs. Together, SageAttention2 and 2++ achieve up to a 3× and 3.9× speedup over FlashAttention while preserving end-to-end performance across language, image, and video generation tasks.

► **FlashAttention3.** While recent GPUs like H100 have massively increased TensorCore throughput, computation is bottlenecked by softmax operations on CUDA cores. FlashAttention 3 (Shah et al., 2024) addresses this by leveraging Hopper architecture (NVIDIA, 2022) features.

First, it adopts the warp-specialized programming paradigm where different warp groups (128 threads) execute load and compute tasks separately to fully hide data transfer latency. Second, it uses asynchronous TensorCores to overlap computation: while CUDA cores compute online softmax for the current block $(m_{ij}, l_{ij}, \widetilde{\mathbf{P}}_{ij}, \mathbf{O}_{ij})$, TensorCores concurrently perform $\widetilde{\mathbf{P}}_{i,j-1}\mathbf{V}_{j-1}$ multiplication from the previous iteration, enabling parallel operation of different computational stages. FlashAttention3 further exploits `FP8` TensorCores by quantizing Q, K, V to `FP8` precision for matrix multiplications, providing 2× theoretical speedup over `FP16/BF16`. These optimizations achieve up to 75% TensorCore utilization, yielding 1.5-2× speedup over FlashAttention2 and reaching 1.2 `PFLOPS` throughput with `FP8` precision, demonstrating the possibility of low-precision attention.

► **SageAttention3.** SageAttention3 (Zhang et al., 2025b) extends low-bit attention to both inference and training. For inference, it applies `FP8` microscaling quantization to the $QK^\top$ and PV matrix multiplications using a fine-grained 1×16 group size, which mitigates outlier effects and improves `FP8` accuracy. It also introduces a two-level quantization for P, first normalizing each token's range to $[0, 448 \times 6]$ via per-token scaling and then applying FP4 microscaling, maximizing the representational capacity of `FP8` quantization scales of P. For efficient training, the SageAttention3 paper also proposes `SageBwd`. It preserves the most accuracy-sensitive matrix multiplication in the backward pass at `FP16` while quantizing the others to `INT8`. SageAttention3 achieves 1038 TOPS on RTX5090 (a 5× speedup over FlashAttention) for inference, end-to-end losslessly accelerating various large models. `SageBwd` delivers lossless accuracy in fine-tuning tasks.

4.2.2 Decoding

As discussed in Section 2.4.1, the decoding phase is memory-bound due to the large I/O overhead for the KV cache, and hardware-efficient attention methods for decoding primarily aim to accelerate I/O for the KV cache.

► **FlashDecoding.** Though FlashAttention2 (Dao, 2024) works well in training and prefilling, partitioning by $\mathbf{Q}_i$ blocks across SMs leads to poor SM utilization due to the limited number of query tokens in the decoding phase. FlashDecoding (Dao et al., 2023) addresses this by further splitting the KV cache: it partitions K, V along the sequence dimension into multiple sub-sequences $\{K^{(j)}, V^{(j)}\}$, with each SM processing all query tokens against one KV sub-sequence using FlashAttention2 $\text{FA2}(Q, K^{(j)}, V^{(j)})$, then performing reduction of $l^{(j)}$ and $m^{(j)}$ online softmax statistics of each sub-sequence in global memory to rescale and aggregate the final results O. This KV-split approach enables full SM utilization during decoding, achieving 8× speedup over FlashAttention2 on long sequences with CodeLlama-34B.

► **PagedAttention.** PagedAttention (Kwon et al., 2023) proposes an attention mechanism designed to reduce KV cache memory overhead in LLM serving by introducing a paging system inspired by virtual memory. PagedAttention breaks the KV cache into fixed-size pages that can be shared, reused, and efficiently allocated across requests. This avoids redundant copies and reduces memory fragmentation. Integrated into the open-source system vLLM, this approach enables near-zero memory waste and supports high-throughput decoding. PagedAttention is well-suited for real-time LLM applications such as chatbots and inference APIs, where memory efficiency and scalability are critical. Its design allows more concurrent requests with less GPU memory, making LLM serving more cost-effective.

► **KIVI.** To reduce KV cache memory usage, KIVI (Liu et al., 2024d) analyzes K, V distributions in popular LLMs, revealing that keys should be quantized per-channel while values should be

quantized per-token to preserve accuracy. To integrate with decoding, KIVI maintains the most recent R key-value pairs in `FP16` precision and applies group quantization to the previous KV cache, where every G consecutive elements share a single scaling factor. Since keys are quantized per-channel (Zhang et al., a), unquantized keys exceeding R tokens are only quantized when reaching the size of G and appended to the quantized cache blocks. During computation, quantized portions are dequantized and concatenated with `FP16` KV cache for decoding. This tuning-free 2-bit quantization achieves 2.6× peak memory reduction while maintaining model quality, enabling 4× larger batch sizes and 2.35-3.47× throughput improvements on Llama (Touvron et al., 2023), Falcon (Almazrouei et al., 2023), and Mistral (Jiang et al., 2023) models.

▶ **KVQuant.** KVQuant (Hooper et al., 2024) proposes a novel quantization framework for compressing the key-value (KV) cache in low precision to reduce memory access. KVQuant analysis KV cache in LLM, revealing that K before RoPE (Su et al., 2024a) exhibits channel-wise outliers, whereas V is more uniformly distributed. Based on this, KVQuant introduces three techniques: (1) pre-RoPE key quantization to exploit smoother distributions before positional encoding, (2) sensitivity-weighted non-uniform quantization that allocates more precision to sensitive tokens, and (3) a dense-plus-sparse outlier handling scheme that isolates and stores large-magnitude values separately using 8-bit precision while compressing the rest to 3-bit. This method is efficient in LLM inference in long-context scenarios, such as retrieval-augmented generation and code completion, where `FP16` KV caches would quickly exceed GPU memory limits.

▶ **FlashInfer.** FlashInfer (Ye et al., 2025) combines multiple strategies, including memory-efficient KV-cache storage, attention code generation, and dynamic scheduling. It stores key–value caches in a compact way, supporting different formats such as paged attention (Kwon et al., 2023) and radix-tree layouts (Zheng et al., 2024). These are unified into a single block-sparse matrix format, which can adjust block sizes based on the amount of shared context. Then, FlashInfer allows users to specify attention variants, which can be Just-In-Time (JIT) compiled into high-performance CUDA kernels. When input lengths vary, FlashInfer uses a load-balanced scheduler that still works with CUDA Graph's static execution mode, avoiding runtime overhead. FlashInfer is integrated into production frameworks like SGLang (Zheng et al., 2024) and vLLM (Kwon et al., 2023), giving significant speedups, especially for long-context inference and when generating multiple sequences in parallel.

5 Compact Attention

In this section, we first present unified formulations for compact attention, then review representative methods, summarizing their key characteristics, and then formulate individual solutions.

5.1 Overall Framework

Compact attention methods are designed to reduce the memory consumption of the KV cache during LLM inference. In MHA, we store the full-resolution KV matrices exactly as used in computation, causing the KV cache size to grow rapidly. Compact attention methods decouple storage KV from computation KV, storing compressed KV states and expanding them for computation. This approach significantly reduces storage KV size compared to MHA, lowering memory usage, while preserving the computation KV size to prevent significant performance degradation.

Table 3: Summary of compact attention methods.

Method	KV Cache	Parameters	Expand Method
MHA (Vaswani et al., 2017)	$2hd$	$4D_m hd$	None
MQA (Shazeer, 2019)	$2d$	$2D_m hd + 2D_m d$	Repeat
GQA (Ainslie et al., 2023)	$2h_{kv}d$	$2D_m hd + 2D_m h_{kv} d$	Repeat
MLA (Liu et al., 2024a)	$r_{kv} + d_R$	$D_m(r_q + r_{kv} + d^R + hd)$ $+r_q h(d^N + d^R) + r_{kv} h(d^N + d)$	Low Rank Projection + Repeat
MFA (Hu et al., 2024)	$2d$	$D_m(3d + hd) + hd^2$	Repeat
TPA (Zhang et al., 2025h)	$(r_k + r_v)(h + d)$	$D_m(r_q + r_k + r_v)(h + d) + D_m hd$	Tensor Product

The general formulation can be expressed as follows.

$$q, \mathcal{K}_c, \mathcal{V}_c = \text{Proj}_{\mathcal{Q}}(x), \text{Proj}_{\mathcal{K}_c}(X), \text{Proj}_{\mathcal{V}_c}(X). \tag{20}$$

$$\mathcal{K}, \mathcal{V} = \text{Expand}_{\mathcal{K}}(\mathcal{K}_c), \text{Expand}_{\mathcal{V}}(\mathcal{V}_c). \tag{21}$$

$$o = \text{MHA}(q, \mathcal{K}, \mathcal{V}). \tag{22}$$

where $\mathcal{K} = [K^{(1)}, \ldots, K^{(h)}] \in \mathbb{R}^{N \times D}$ denotes the concatenation of h attention head key matrices, with $K^{(i)} \in \mathbb{R}^{N \times d}$ representing the key matrix of head i and $D = hd$. The same notation applies to q and $\mathcal{V}$. Here, $x \in \mathbb{R}^{D_m}$ is the hidden state of the current token, $X \in \mathbb{R}^{n \times D_m}$ is the matrix of hidden states for the context tokens, Proj$(\cdot)$ and Expand$(\cdot)$ denote the projection and expansion functions, respectively, and MHA$(\cdot)$ denotes the multi-head attention operation. Vanilla MHA (Vaswani et al., 2017) can be expressed within this formulation by setting $\text{Proj}_{\mathcal{K}_c}(X) = xW_{\mathcal{K}_c} \in \mathbb{R}^{N \times D}$, where $W_{\mathcal{K}_c} \in \mathbb{R}^{D_m \times D}$, and similarly for Proj_Q and Proj_{V_c}. The expansion function is simply $\text{Expand}_{\mathcal{K}}(\cdot) = \text{Expand}_{\mathcal{V}}(\cdot) = \text{id}(\cdot)$, i.e., the identity projection. Compact attention methods reduce the size of the cached key–value tensors $\mathcal{K}_c$ and $\mathcal{V}_c$ by modifying the projection function and utilizing a non-trivial expansion function. These methods do not alter the procedure or computational cost of scaled dot-product attention, but they can substantially lower the memory consumption of the KV cache, enabling memory-efficient scaling, particularly for long sequences.

Table 3 summarizes the KV cache size for each token, total parameters for attention, and expansion function type for compact attention methods, with the notations detailed in the sections below.

5.2 Methods

► **MQA.** The core innovation of Multi-Query Attention (MQA) (Shazeer, 2019) lies in its asymmetric head architecture: while MQA maintains h distinct query heads, it uses only a single shared head for both keys and values. Specifically, the input sequence X is first projected into a single key–value pair using two weight matrices $W_{\mathcal{K}_c}, W_{\mathcal{V}_c} \in \mathbb{R}^{D \times d}$: $\mathcal{K}_c, \mathcal{V}_c = XW_{\mathcal{K}_c}, XW_{\mathcal{V}_c}$ The resulting single KV pair is then replicated across all h query heads to form the final key and value matrices $\mathcal{K}$ and $\mathcal{V}$ for computation. This replication can be expressed using the Kronecker product $\otimes$ (Broxson, 2006): $\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1 \times h}, \mathcal{V}_c \otimes \mathbf{1}_{1 \times h}$. where $\mathbf{1}_{1 \times h}$ is a row vector of ones that

tiles the shared projections to all query heads. Another perspective is to view MQA as weight sharing: $\mathcal{K} = (XW_{\mathcal{K}_c}) \otimes \mathbf{1}_{1\times h} = X(W_{\mathcal{K}_c} \otimes \mathbf{1}_{1\times h})$, indicating that MQA essentially shares a single projection matrix $W_{\mathcal{K}_c}$ across all h heads. This design reduces the KV cache's memory by a factor of h compared to standard MHA. However, such aggressive weight sharing limits representational capacity, as the model operates with only one KV head, which reduces performance relative to MHA.

$$q, \mathcal{K}_c, \mathcal{V}_c = x \overset{D_m \times hd}{W_Q}, X \overset{D_m \times d}{W_{\mathcal{K}_c}}, X \overset{D_m \times d}{W_{\mathcal{V}_c}}. \tag{23}$$

$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h}. \tag{24}$$

► **GQA.** Grouped-Query Attention (GQA) (Ainslie et al., 2023) provides an interpolation between MHA and MQA. The key idea is to partition the h total query heads into h_{kv} distinct groups, where h_{kv} divides h. Within each group, the h/h_{kv} query heads share a single key–value pair. First, a compact set of h_{kv} key and value heads, denoted $\mathcal{K}_c$ and $\mathcal{V}_c$, is obtained by projecting the input X: $\mathcal{K}_c = XW_{\mathcal{K}_c}, \mathcal{V}_c = XW_{\mathcal{V}_c}$, where $W_{\mathcal{K}_c}, W_{\mathcal{V}_c} \in \mathbb{R}^{D\times h_{kv}d}$. To enable attention across all h query heads, these h_{kv} key–value heads are replicated within their respective groups h/h_{kv} times: $\mathcal{K} = \mathcal{K}_c \otimes \mathbf{1}_{1\times(h/h_{kv})}, \mathcal{V} = \mathcal{V}_c \otimes \mathbf{1}_{1\times(h/h_{kv})}$. GQA reduces the KV cache size by a factor of h/h_{kv} compared to MHA, delivering an effective balance between model quality and efficiency, and has become the mainstream design choice in numerous large-scale language models, including LLaMA (Dubey et al., 2024), Gemma (Team et al., 2024), Qwen (Bai et al., 2023), and many others.

$$q, \mathcal{K}_c, \mathcal{V}_c = x \overset{D_m \times hd}{W_Q}, X \overset{D_m \times h_{kv}d}{W_{\mathcal{K}_c}}, X \overset{D_m \times h_{kv}d}{W_{\mathcal{V}_c}}. \tag{25}$$

$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h/h_{kv}}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h/h_{kv}}. \tag{26}$$

► **MLA.** Multi-Head Latent Attention (MLA) (Liu et al., 2024a) reduces KV cache size primarily through low-rank projection. Specifically, it caches a shared, down-projected low-rank KV state: $\mathcal{K}_c^N = \mathcal{V}_c = XW_{\mathcal{KV}}^{\text{down}} \in \mathbb{R}^{N\times r_{kv}}$, and then up-projects it for computation: $\mathcal{K}^N = \mathcal{K}_c^N W_{\mathcal{K}}^{\text{up}} \in \mathbb{R}^{N\times d^N}, \mathcal{V} = \mathcal{V}_c^N W_{\mathcal{V}}^{\text{up}} \in \mathbb{R}^{N\times d}$. The Rotary Position Encoding (RoPE) (Su et al., 2024b) can not be applied to the low-rank compression because RoPE requires storing the up-projected KV states instead of their compressed form, eliminating the intended memory savings. To incorporate positional information alongside the aforementioned NoPE (no positional encoding) branch (Kazemnejad et al., 2023), MLA adds an additional RoPE branch with a single-head key: $\mathcal{K}_c^R = \text{RoPE}(XW_{\mathcal{K}_c^R}) \in \mathbb{R}^{N\times d^R}$, which is replicated across h attention heads: $\mathcal{K}^R = \mathcal{K}_c^R \otimes \mathbf{1}_{1\times h}$, and then concatenated with the NoPE branch within each head to form the final key state: $\mathcal{K} = [\mathcal{K}^N, \mathcal{K}^R] \in \mathbb{R}^{N\times h(d^N+d^R)}$. The query is also constructed via a low-rank projection to both reduce parameter count and match the dimensionality of the keys. With these two branches, the KV cache size per token is reduced from $2hd$ to $r_{kv} + d^R$.

$$q = [x \overset{D_m \times r_q}{W_{\mathcal{Q}}^{\text{down}}} \overset{r_q \times hd^N}{W_{\mathcal{Q}}^{\text{up},N}}, \text{RoPE}(x \overset{D_m \times r_q}{W_{\mathcal{Q}}^{\text{down}}} \overset{r_q \times hd^R}{W_{\mathcal{Q}}^{\text{up},R}})]. \tag{27}$$

$$\mathcal{K}_c^N = \mathcal{V}_c = X \overset{D_m \times r_{kv}}{W_{\mathcal{KV}}^{\text{down}}}. \tag{28}$$

$$\mathcal{K}_c^R = \text{RoPE}(X \overset{D_m \times d^R}{W_{\mathcal{K}_c^R}}). \tag{29}$$

$$\mathcal{K} = [\mathcal{K}_c^N \overset{r_{kv} \times hd^N}{W_{\mathcal{K}}^{\text{up}}}, \mathcal{K}_c^R \otimes 1_{1\times h}]. \tag{30}$$

$$\mathcal{V} = \mathcal{V}_c \overset{r_{kv} \times hd}{W_{\mathcal{V}}^{\text{up}}}. \tag{31}$$

► **MFA.** Multi-matrix Factorization Attention (MFA) (Hu et al., 2024) builds on MQA by caching a single KV head shared across all query heads. It introduces two enhancements to narrow the performance gap between MQA and MHA under the same parameter budget. First, it increases the per-head dimensionality d to improve the representational capacity of each head. Second, it applies a low-rank factorization to the query projection: the initial down-projection weight $W_{\mathcal{Q}}^{\text{down}} \in \mathbb{R}^{D_m \times d}$ is shared across heads, followed by an up-projection weight $W_{\mathcal{Q}}^{\text{up}} \in \mathbb{R}^{d \times hd}$ that generates multiple query heads. This design reduces the parameter cost of scaling h from roughly $2D_m d$ (for both query and output projections) to approximately $D_m d$ for the output projection alone.

$$q = x \overset{D_m \times d}{W_{\mathcal{Q}}^{\text{down}}} \overset{d \times hd}{W_{\mathcal{Q}}^{\text{up}}}. \tag{32}$$

$$\mathcal{K}_c, \mathcal{V}_c = X \overset{D_m \times d}{W_{\mathcal{K}_c}^{\text{down}}}, X \overset{D_m \times d}{W_{\mathcal{V}_c}^{\text{down}}}. \tag{33}$$

$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h}. \tag{34}$$

► **TPA.** Tensor-Product Attention (TPA) (Zhang et al., 2025h), originating from low-rank decomposition of matrices, generalizes the "repeating" operation used in MQA and GQA through a learnable tensor product. In MQA and GQA, the repetition follows a fixed pattern: $\mathcal{K} = \mathcal{K}_c \otimes \mathbf{1}$, where $\mathbf{1}$ is a constant. TPA replaces this with a learnable formulation: $\mathcal{K} = \mathcal{K}_c^A \otimes \mathcal{K}_c^B$, where $\mathcal{K}_c^A \in \mathbb{R}^{N\times h\times 1}$ encodes head-specific scaling factors and $\mathcal{K}_c^B \in \mathbb{R}^{N\times 1\times d}$ encodes shared feature components. Here, $\otimes$ denotes the tensor (Kronecker) product along the last two axes. In this basic form, the resulting $\mathcal{K}$ is a rank-1 approximation of the full-size key state in MHA. To increase capacity, TPA introduces r such factorized components and averages them, producing a storage KV of rank-r: $\mathcal{K} = \frac{1}{r}\sum_{i=1}^{r} \mathcal{K}_i$ with each $\mathcal{K}_i$ factorized as above. The same construction applies to $\mathcal{V}$, reducing the KV cache size to $(r_k + r_v)(h + d)$ per token. The query projection is also formulated via the tensor product, preserving symmetry in the overall structure.

$$q^A, q^B = x \overset{D_m \times r_q h}{W_{\mathcal{Q}}^A}, x \overset{D_m \times r_q d}{W_{\mathcal{Q}}^B} \tag{35}$$

$$= [q^{A,1}, \ldots, q^{A,r_q}], [q^{B,1}, \ldots, q^{B,r_q}] \tag{36}$$

$$q = \frac{1}{r_q}\sum_{i=1}^{r_q} \overset{h\times 1}{q^{A,i}} \otimes \overset{1\times d}{q^{B,i}} \tag{37}$$

$$\mathcal{K}_c^A, \mathcal{K}_c^B = X \overset{D_m \times r_k h}{W_{\mathcal{K}_c}^A}, X \overset{D_m \times r_k d}{W_{\mathcal{K}_c}^B} \tag{38}$$

$$\mathcal{V}_c^A, \mathcal{V}_c^B = X \overset{D_m \times r_v h}{W_{\mathcal{V}_c}^A}, X \overset{D_m \times r_v d}{W_{\mathcal{V}_c}^A} \tag{39}$$

$$\mathcal{K}, \mathcal{V} = \frac{1}{r_k}\sum_{i=1}^{r_k} \mathcal{K}_c^{A,i} \otimes \mathcal{K}_c^{B,i}, \frac{1}{r_v}\sum_{i=1}^{r_v} \mathcal{V}_c^{A,i} \otimes \mathcal{V}_c^{B,i} \tag{40}$$

6 Sparse Attention

In this section, we present unified formulations for sparse attention, summarize representative methods along with their key properties, and discuss the implementation details of specific approaches.

Table 4: Summary of sparse attention methods.

Category	Method	LLM	Reduce KV Storage	DIT	Training Free
Pattern Based	StreamingLLM (Xiao et al., 2024b)	✓	✓	✗	✓
	DiTFastAttn (Yuan et al., 2024)	✗	✗	✓	✓
	SampleAttn (Zhu et al., 2024)	✓	✗	✗	✓
	MoA (Fu et al., 2024)	✓	✓	✗	✓
	DuoAttention (Xiao et al., 2025)	✓	✓	✗	○
	SparseVideoGen (Xi et al., 2025)	✗	✗	✓	✓
	Radial Attention (Li et al., 2025)	✗	✗	✓	✓
	STA (Zhang et al., 2025e)	✗	✗	✓	✓
	NeighborAttn (Hassani et al., 2023a)	✗	✗	✓	✓
	PAROAttn (Zhao et al., 2025)	✗	✗	✓	✓
Dynamic Sparse	SLA (Zhang et al., 2025a)	✗	✗	✓	✗
	SpargeAttn (Zhang et al., 2025c)	✓	✗	✓	✓
	H2O (Zhang et al., 2023)	✓	✓	✗	✓
	InfLLM (Xiao et al., 2024a)	✓	✗	✗	✓
	MInference (Jiang et al., 2024)	✓	✗	✗	✓
	FlexPrefill (Lai et al., 2025)	✓	✗	✗	✓
	SparQAttn (Ribar et al., 2024)	✓	✓	✗	✓
	LokiAttn (Singhania et al., 2024)	✓	✗	✗	✓
	SeerAttention (Gao et al., 2024)	✓	✗	✗	○
	RetrievalAttn (Liu et al., 2024b)	✓	✓	✗	✓
	FPSAttention (Liu et al., 2025)	✗	✗	✓	✗
	NSA (Yuan et al., 2025)	✓	✓	✗	✗
	MoBA (Lu et al., 2025)	✓	✗	✗	✗ [1]
	VMoBA (Wu et al., 2025)	✓	✗	✓	✗ [1]
	CHAI (Agarwal et al., 2024)	✓	✓	✗	✓
	Quest (Tang et al., 2024)	✓	✗	✗	✓
	MagicPig (Chen et al., 2024b)	✓	✗	✗	✓
	DraftAttn (Shen et al., 2025)	✗	✗	✓	✓
	HashAttn (Desai et al., 2025)	✓	✗	✗	○
	XAttention (Xu et al., 2025)	✓	✗	✓	✓
	VSA (Zhang et al., 2025f)	✗	✗	✓	✗
	SparseVideoGen2 (Yang et al., 2025b)	✗	✗	✓	✓
	TidalDecode (Yang et al., 2024a)	✓	✗	✗	✓
	LessIsMore (Yang et al., 2025a)	✓	✗	✗	✓
	Twilight (Lin et al., 2025)	✓	✗	✗	✓

° Mark ○ in the training-free column indicates that the method requires training a subset of parameters rather than the entire model.

[1] MoBA and VMoBA can be used without training, though performance may drop.

6.1 Overall Framework

The attention map $P = \text{Softmax}(QK^\top/\sqrt{d})$ exhibits inherent sparsity, as the softmax operation often creates many values approaching zero (Deng et al., 2024). *Sparse attention* methods exploit such sparsity to accelerate attention by two steps. First, it constructs a *sparse mask* M, which determines whether to compute or skip specific elements in the attention map P. Second, it computes attention only for the parts corresponding to the *sparse mask* M.

$$P = \text{Softmax}(M + QK^\top/\sqrt{d}). \tag{41}$$
$$O = PV. \tag{42}$$

Where M is an $N \times N$ matrix whose elements are either 0 or $-\infty$. $M_{i,j} = 0$ specifies that both the attention score $Q_i K_j^T$ and its corresponding output $P_{i,j}V_j$ should be computed, while $M_{i,j} = -\infty$ indicates these computations should be skipped. There are two distinct categories of sparse attention methods based on how the sparse mask is generated:

1. **Pattern-based method** relies on predefined sparsity patterns derived from empirical observations, where the positions of $-\infty$ entries in M follow fixed geometric shapes (e.g., a sliding window shape).

2. **Dynamic sparse attention** computes the *sparse mask* M adaptively during runtime based on some input-dependent functions (e.g., $\mathbf{M}_{i,j} = -\infty$ if $\text{pool}(\mathbf{Q}_i)\text{pool}(\mathbf{K}_j^T) < \tau$ for a threshold τ, where $\text{pool}(\cdot)$ could be mean pooling over tokens).

Sparse FlashAttention. Sparse attention needs FlashAttention for efficiency, with its sparsity pattern matching FlashAttention's block size. Implementing sparse FlashAttention is intuitive: we can just skip certain block matrix multiplications of $Q_i K_j^\top$ and $\widetilde{P}_{i,j}V_j$ according to the sparse mask M, accelerating the attention computation. We formulate sparse attention based on FlashAttention as follows.

Definition 1 (Sparse FlashAttention)**.** *The computation rules for sparse FlashAttention based on the masks are defined as follows:*

$$\mathbf{M}_{i,j} = -\infty \quad \text{if} \quad M[i \times b_q : (i+1) \times b_q][j \times b_{kv} : (j+1) \times b_{kv}] = -\infty. \tag{43}$$
$$\mathbf{Q}_i\mathbf{K}_j^\top, \widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j \ \text{ are skipped if } \quad \mathbf{M}_{i,j} = -\infty. \tag{44}$$

6.2 Preliminaries of Sparse Attention

LLM prefilling and decoding. As discussed in Section 2.4.3, for LLM prefilling, attention computation speed is the primary latency bottleneck. The goal of sparse attention in this context is to omit as many block matrix multiplications between $\mathbf{QK}$ and $\widetilde{\mathbf{P}}\mathbf{V}$ as possible. For LLM decoding, the main bottleneck is the read–write overhead of the KV cache between global memory and shared memory. Here, sparse attention primarily aims to minimize the size of the KV cache, i.e., reducing the I/O of $\mathbf{K}$ and $\mathbf{V}$. Most sparse attention methods designed for language models accelerate both prefilling and decoding, as increasing the number of $-\infty$ entries in M directly improves speed.

Reduce KV storage. Although most sparse attention methods reduce KV cache I/O in decoding, not all reduce its memory storage. In general, most pattern-based methods can effectively save KV storage because the KV cache requires incremental updates for adjacent queries during decoding. If the sparse mask shape varies significantly across queries, it introduces considerable cache update overhead, making reducing the KV cache difficult.

DiT. Diffusion transformer models (Peebles & Xie, 2023), commonly used for image and video generation, often adopt vision transformers as the backbone. As discussed in Section 2.4.1, attention computation speed is the primary bottleneck.

Training-free property. In Table 4, the *training-free* attribute indicates whether a method requires model training: approaches are marked as not training-free if they involve training model parameters or auxiliary models; otherwise, they are considered training-free.

Table 4 summarizes sparse attention methods based on their *sparse mask M* (pattern-based or dynamic), whether they need to train a model, and applicability to language models and diffusion transformers. The following subsections provide detailed introductions to each method.

6.3 Pattern-based Sparse Attention

► **StreamingLLM.** Standard dense attention struggles with quadratic complexity and fails beyond the pre-training length, while simple window attention (Beltagy et al., 2020) collapses once initial tokens are evicted from the KV cache. StreamingLLM (Xiao et al., 2024b) identifies a key phenomenon called the attention sink, where a few initial tokens consistently receive a significant portion of attention scores, regardless of their semantic content. The failure of window attention is a direct result of evicting these crucial attention sink tokens. StreamingLLM proposes a simple and efficient framework that preserves the KV states of the attention sinks while maintaining a rolling cache of the most recent tokens. This training-free approach enables stable performance on sequences of infinite length while reducing the per-token computational complexity to $O(L)$ for a KV cache of fixed size L.

► **DitFastAttn.** DiTFastAttn (Yuan et al., 2024) is a post-training acceleration method, addressing computational redundancy in DiT models (Peebles & Xie, 2023) through three dimensions. First, it employs window attention (Beltagy et al., 2020) with residual caching to to reduce redundant computation in the spatial dimension. Specifically, at a specific timestep r, both the full attention $O_r = Attention(Q_r, K_r, V_r)$ and the local window attention $W_r = WindowAttention(Q_r, K_r, V_r)$ are computed. The difference between these two outputs is computed as 'residual' $R_r = O_r - W_r$. Then for the following few steps t, it use window attention $W_t = WindowAttention(Q_t, K_t, V_t)$ and the cached "residual" R_r, to computed the output $O_t = W_t + R_r$. Secondly, due to the attention outputs of adjacent timesteps being highly similar, it caches the output of the first step among a sequence of steps with similar attention outputs, and reuses it for the subsequent ones, reducing redundant computations along timesteps. Thirdly, Classifier-Free Guidance (CFG)(Ho & Salimans, 2022) performs two forward passes at each step: one conditional and one unconditional—whose attention outputs are often highly similar. Thus, it reuses the attention output from the conditional pass for the unconditional one, effectively halving the attention computation in these instances.

► **SampleAttn.** SampleAttn (Zhu et al., 2024) proposes a training-free adaptive structured sparse attention for LLM prefilling. SampleAttn identifies two sparsity patterns: (1) a local window pattern

that captures recent context, and (2) a column stripe pattern that represents key global information. The method approximates full attention by dynamically combining these two structured patterns for each attention head. To capture the local window pattern, SampleAttn attends to a fixed percentage of adjacent tokens, allowing the window size to scale with the sequence length. To identify the crucial column of the attention map without computing the entire attention score, SampleAttn employs an two-stage process: it first samples a set of query tokens and calculates a partial attention score matrix S_s with the key tokens; then, based on S_s, it selects the most relevant key and value tokens to perform sparse attention. SampleAttn reduces prefilling latency by up to 2.42× compared with FlashAttention while maintaining over 99% of baseline accuracy across benchmarks.

► **MoA.** MoA (Fu et al., 2024) leverages the inherent heterogeneity and elasticity of attention distribution across attention heads and input lengths in LLMs. Building upon the **uniform** sliding-window attention approach of StreamingLLM (Xiao et al., 2024b), MoA optimizes **different** window-lengths tailored for each attention head and input length. MoA formulates it as an offline search over elastic rules, where each rule defines how window-length scales linearly with input length N via the relation $\alpha + \beta N$, where α and β control the base attention span and its growth rate, respectively. Optimal values for these hyperparameters are identified using a gradient-based profiling method and multi-objective optimization under average density constraints across intended input lengths. With the optimized window lengths, MoA significantly enhances retrieval accuracy by 1.5–7.1× compared to StreamingLLM. It also boosts decoding throughput by 6.6–8.2× over FlashAttention2 while maintaining minimal performance degradation.

► **DuoAttention.** DuoAttention (Xiao et al., 2025) identifies two types of attention heads in LLMs: *retrieval heads* and *streaming heads*. Retrieval heads require full attention to capture globally relevant context, while streaming heads mainly focus on recent and initial tokens (attention sinks), allowing for partial attention and reduced KV cache. To differentiate attention head types and apply appropriate masks, DuoAttention introduces a learnable gate $\alpha \in [0, 1]$ for each head, combining full and streaming-masked attention outputs as $\text{attn} = \alpha \cdot \text{full\_attn} + (1 - \alpha) \cdot \text{streaming\_attn}$. The gate values are optimized by minimizing the mean-squared error between the last hidden states of the full-attention model and the DuoAttention model, with an additional weighted L_1 penalty on α, $(\sum |\alpha|)$, to promote sparsity. At inference time, heads with lower α values are treated as streaming heads, selected based on a specified sparsity quantile. DuoAttention achieves up to 2.55× prefilling and 2.18× decoding speedups, and reduces inference memory by 2.55× in long-context settings.

► **Sparse VideoGen.** Sparse VideoGen (Xi et al., 2025) proposes a training-free sparse attention framework for video diffusion transformers, aiming to reduce the cost of full 3D attention over long video sequences. Given input Q, K, V, Sparse VideoGen aims to classify the heads into either spatial-heads or temporal-heads, which focus on spatially-local tokens and temporally-local tokens, respectively. The corresponding sparse attention mask is defined as M_{spatial} and M_{temporal}, where M_{spatial} consists of a diagonal sliding window and a first-frame sink, and M_{temporal} consists of multiple slanted stripes. The classification is achieved via a lightweight online profiling algorithm. For each attention head h_i, a small subset of tokens (1%) is randomly sampled to compute the MSE between the full attention output O and two sparse approximations. The final mask M_h is set to either M_h^{spatial} or M_h^{temporal} accordingly by comparing the MSE between the golden output and the output after applying the sparse attention mask. After mask assignment, Sparse VideoGen applies static layout transformations to the temporal heads, enabling efficient computation under

the selected sparse pattern. Sparse VideoGen achieves up to 2.3× speedup on video generation tasks.

► **Radial Attention.** Radial Attention (Li et al., 2025) introduces a static sparse attention mask with $\mathcal{O}(N \log N)$ complexity, enhancing both training and inference speeds of video diffusion transformers. The method is motivated by the "Spatiotemporal Energy Decay" phenomenon, which states that both attention compute density and attention scores decrease in the attention map when the spatial and temporal distance between tokens increases. The method partitions the attention map into exponentially widening temporal bands, where the compute density halves with each step from the diagonal. Within each frame-to-frame attention block, a diagonal window with exponentially decreasing width is maintained. Moreover, the minimum unit of the sparse attention map is set to 128×128 block to ensure efficient execution on modern hardware. Radial Attention also guarantees lightweight fine-tuning like LoRA (Hu et al., 2022) for context extension of video diffusion models, as it efficiently preserves the computations of the important token relations. The method accelerates default-length video generation of leading video diffusion transformers (e.g., Wan 2.1, HunyuanVideo) by up to 1.9× without tuning, while bringing up to 4.4× training cost reduction and 3.7× inference speedup for up to 4× longer video generation.

► **STA.** Sliding Tile Attention (Zhang et al., 2025e) accelerates video diffusion transformers by overcoming the computational inefficiencies of conventional 2D and 3D sliding window attention mechanisms. Although sliding window attention reduces FLOPs through locality enforcement, its GPU efficiency is hindered by mixed attention blocks containing both masked and unmasked entries, which disrupt the blockwise computation pattern required by FlashAttention. Sliding Tile Attention addresses this limitation by shifting the attention operation from the token level to the tile level, partitioning the 3D input into fixed-size spatio-temporal tiles and ensuring that each attention block is either fully dense or fully empty. This design eliminates masking overhead and enables highly efficient GPU execution. Implemented atop FlashAttention3 (Shah et al., 2024) and ThunderKittens (Spector et al., 2024), Sliding Tile Attention achieves up to 10.45× acceleration in attention kernel execution and 2.98× end-to-end inference speedup over FlashAttention-3. Sliding Tile Attention supports both training-free and fine-tuned configurations. In the training-free setting, it automatically calibrates per-layer, per-head window sizes using a small prompt set, attaining 58% sparsity and 1.8× end-to-end speedup. In the fine-tuned setting, fixed sparse masks can be optimized to further improve throughput; for example, 91% sparsity yields a 3.5× speedup with negligible degradation in VBench (Huang et al., 2024) scores.

► **NeighborAttn.** Neighborhood Attention (Hassani et al., 2023a) introduces a pixel-wise sliding window attention that localizes each query's attention span to its immediate spatial neighbors. In contrast to Swin transformer (Liu et al., 2021), which employs non-overlapping windowed attention and relies on shifted windows to enlarge the receptive field, Neighborhood Attention preserves translational equivariance and naturally expands the receptive field without manual shifting. This design achieves linear time and space complexity while preserving locality bias, thereby bridging the gap between convolutional networks and self-attention architectures. For practical deployment, Neighborhood Attention is implemented through its NATTEN library (Hassani et al., 2023b) with custom CUDA kernels, achieving up to 40% speedup and 25% memory reduction compared to Swin attention. Based on this mechanism, Neighborhood Attention demonstrates strong performance

across classification, detection, and segmentation tasks, outperforming Swin (Liu et al., 2021) and ConvNeXt (Liu et al., 2022) under comparable parameter and computational budgets.

▶ **PAROAttn.** PAROAttention (Zhao et al., 2025) proposes a simple yet effective pattern-aware token reorder technique to transform the diverse and scattered attention values into unified hardware-friendly block-wise patterns. It observes that seemingly diverse visual attention maps consist of multiple "diagonal lines" all represent "local aggregation" along a particular dimension in 3D space. For example, for a video tensor of shape $[N_{\text{frame}}, W, H]$, locality along the W axis produces tokens in the attention map spaced equally by H, and conducting local aggregation between them produces multiple "diagonal lines" in the attention map. Therefore, permuting the token order from $[N_{\text{frame}}, W, H]$ to $[N_{\text{frame}}, H, W]$ converts these multi-diagonal lines into a regular block-wise structure. This, in turn, enables a simple threshold-based block-sum scheme to derive the attention pattern. Enlightened by the empirical evidence that visual attention patterns generalize across different conditions, it adopts a static sparsity scheme, where the attention patterns are determined offline. PAROAttention follows the concept of hardware-software co-optimization by aligning the locality of visual feature extraction (numerical locality) with the locality of hardware computation (memory and computation locality). It designs a comprehensive suite of efficient CUDA implementations to minimize overhead and maximize efficiency.

6.4 Dynamic Sparse Attention

▶ **SLA.** SLA (Zhang et al., 2025a) (Sparse-Linear Attention) is a trainable attention method for accelerating Diffusion Transformer (DiT) models, especially for video generation. SLA decomposes attention weights into critical, marginal, and negligible parts, applying full attention to critical weights, linear attention to marginal ones, and skipping negligible ones within one GPU kernel. It implements efficient forward and backward GPU kernels. By replacing the standard attention with SLA and performing a few fine-tuning steps before inference, DiT models achieve a 20× reduction in attention computation and a 13.7× attention speedup without compromising generation quality.

▶ **SpargeAttn.** SpargeAttn (Zhang et al., 2025c) proposes a training-free, all model-applicabile sparse attention. The method has two stages to perform sparse attention. In Stage-1, SpargeAttn selectively compresses $\mathbf{Q}_i$ and $\mathbf{K}_j$ whose tokens have high similarity to one token by mean pooling. Then, SpargeAttn computes a compressed attention map $\hat{P}$ using the compressed Q and K. Finally, SpargeAttn selectively compute $\{\mathbf{Q}_i\mathbf{K}_j^\top, \widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j\}$ for those pairs (i, j) where $\{\hat{P}[i, j]\}$ accumulates a high score in the compressed attention map. For those non-self-similar blocks, as a good presentation token for the whole block is hard to find, SpargeAttn chooses to always compute attention related to the non-self-similar blocks. In Stage 2, SpargeAttn further identifies the small enough values in the attention map during the online softmax process. If all values in $\widetilde{\mathbf{P}}_{i,j}$ are close enough to zero, the $\widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j$ will be negligible and can be omitted. If $\max(\text{rowmax}(\mathbf{S}_{i,j}) - m_{i,j})$ is small enough, then $\widetilde{\mathbf{P}}_{i,j} = \exp(\mathbf{S}_{i,j} - m_{i,j})$ are close to 0, allowing skipping the $\widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j$. Lastly, since quantization operations and sparse operations are orthogonal, sparse computation can be directly applied to SageAttention. SpargeAttn could accelerate diverse models, including language, image, and video generation models.

▶ **H2O.** H2O (Zhang et al., 2023) proposes a training-free method that maintains a constant KV cache size during decoding through dynamic token eviction. Let S denote the set of cached token indices (e.g., $S = \{1, 2, 4, 5, \ldots\}$). For each token $j \in S$, they define its importance score as the sum of attention probabilities from all subsequent tokens: $F_j = \sum_{l=j}^{n} P_{lj}$, where n is the current

sequence length. Tokens with high scores are termed "Heavy Hitters" (H2). Once the cache reaches capacity k, each new decoding step evicts the least important token $t = \text{argmin}_{j \in S} F_j$ while adding the new token $S \leftarrow S \cup \{n\} \setminus \{t\}$, maintaining exactly k cached tokens. H2O balances retaining both heavy hitters and recent tokens to preserve generation quality. On OPT (Zhang et al., 2022) models, H2O achieves accuracy comparable to full attention while delivering 3× latency reduction compared to FlexGen (Sheng et al., 2023).

► **InfLLM.** While StreamingLLM (Xiao et al., 2024b) enables infinite-length inference, its sliding window leads to global information loss. Xiao et al. (2024a) propose InfLLM, which augments StreamingLLM's attention sink I and local window L with retrieval from a global KV cache. The global cache is partitioned into blocks $\{B_1, \ldots, B_{\lfloor N/b \rfloor}\}$ of size b. For each block B_i, the top-t_1 tokens T_i selected by cumulative local window attention scores $F(j) = \sum_{l=j}^{j+n_l-1} P_{lj}$, are averaged to create a block representative $\bar{k}_i = \frac{1}{t_1} \sum_{j \in T_i} k_j$. During decoding, each query computes scores $S'_i = q\bar{k}_i^\top$ with all block representatives and retrieves the top-t_2 blocks to form context $C = \{I, L, B_{i_1}, \cdots, B_{i_{t_2}}\}$ for final attention computation. InfLLM extends context to 128K on Mistral and Llama-3 with 100% needle-in-a-haystack accuracy, achieving 1.5× speedup over full attention by computing on fixed-size $|I| + |L| + t_2 b$ context length. Building upon this, InfLLM v2 (Team et al., 2025) introduces a trainable sparse attention that accelerates both prefilling and decoding. It improves the block selection process by replacing the selection of representative tokens with fine-grained "semantic kernels". These kernels are constructed via mean pooling, a design that eliminates the token-level memory access bottleneck of the original method.

► **MInference.** MInference (Jiang et al., 2024) introduces an offline search-based dynamic sparse attention to accelerate long-context LLM inference in the prefilling stage. It comprises three stages: (1) offline pattern search, (2) dynamic sparse mask prediction, and (3) dynamic sparse computation. First, each attention head searches for its optimal pattern from *A-shape* (Xiao et al., 2024b), *vertical-slash* (Jiang et al., 2024) , or *block-sparse*. Vertical-slash denotes an attention pattern concentrated on specific tokens (vertical lines) and tokens at fixed intervals (slash lines). Second, MInference generates the dynamic sparse mask online. For example, in the vertical-slash pattern, MInference computes the attention scores between the queries and all keys/values to identify the top-K vertical and slash lines: $\mathbf{M} = \text{RowTopK}(\text{softmax}\left(Q[-\text{w}:]K^\top/\sqrt{d}\right))$, where `RowTopK` selects the top-K entries in each row. For block-sparse attention, the mask is obtained by computing the attention weights between average-pooled queries and keys: $\hat{Q} = \text{pool}(Q)$, $\hat{K} = \text{pool}(K)$, $\mathbf{M} = \text{RowTopK}(\text{softmax}\left(\hat{Q}\hat{K}^\top/\sqrt{d}\right))$, where `pool` refers to average pooling over tokens. Finally, dynamic sparse attention is computed over the resulting mask at the stripe level (Zheng et al., 2023) (for vertical lines) and block level (for slash lines and block-sparse).

► **FlexPrefill.** FlexPrefill (Lai et al., 2025) proposes a context-aware dynamic sparse attention that adapts both the sparse pattern and sparsity ratio at runtime. It consists of two steps: (1) Pattern selection by different context, which chooses between block-sparse and vertical-slash (Jiang et al., 2024) attention by measuring the Jensen–Shannon divergence (Menéndez et al., 1997) between dense and block-sparse attention scores: $\bar{p} = \text{softmax}\left(\text{pool}(Q[-\text{w}:])\text{pool}(K)^\top/\sqrt{d}\right)$, $\bar{a}_{\text{block}} = \text{sumpool}\left(\text{softmax}\left(Q[-\text{w}:]K^\top/\sqrt{d}\right)\right)$, and $D_{JS}(\bar{a}, \bar{a}_{\text{block}}) = \sqrt{JSD(\bar{a}||\bar{a}_{\text{block}})}$, where `pool`/`sumpool` indicate average/sum pooling over tokens; and (2) Dynamic sparsity ratio determination, which generates the attention mask by applying a row-wise top-P threshold until a target

recall is met. For example, in vertical-slash pattern, $\mathbf{M} = \text{RowTopP}(\text{Softmax}\left(Q_{[-\text{w}:]}K^\top/\sqrt{d}\right))$, where `RowTopP` selects the biggest entries which accumulate to a threshold in each row.

► **SparQAttn.** SparQAttention (Ribar et al., 2024) is designed to alleviate the memory bandwidth bottleneck during the decoding phase of large language model inference. Given a query vector q_t in a decoding step, SparQ first selects the r largest-magnitude components of q_t along the hidden dimension and slices both q_t and the Key cache K along these dimensions. Note that r is a pre-determined hyperparameter and is usually set to 64. This produces a low-dimensional approximation of the attention scores $\hat{S}$ without retrieving the full key vectors. Based on this approximated attention score, SparQ identifies the top-k tokens that are most important to attention computation. SparQ finally extracts the Keys and Values at these k tokens to calculate the approximated attention output. The final output is a weighted combination of the partial attention result and a precomputed mean value vector $\bar{v}$, allowing SparQ to do mean value reallocation. This hybrid attention design enables significant reductions in bandwidth cost with minimal quality degradation.

► **LokiAttn.** LokiAttention (Singhania et al., 2024) is a training-free sparse attention method for LLM decoding. LokiAttn consists of two main stages: (1) Offline Principal Component Analysis (PCA) (Abdi & Williams, 2010) calibration: Loki first generates a set of key vectors from calibration prompts for a target LLM. It then applies PCA to the *headdim* d of key tokens, reducing the d to d_r. This PCA projection matrix is computed offline and stored. The analysis shows that a small number of d (e.g., around 80, despite a full dimension of 128) can explain 90% of the information for key tokens. (2) Low-Rank inference with top-k selection: During inference, instead of computing attention scores using the key tokens in $N \times d$, Loki performs the following steps: It projects the query and key tokens into $N \times d_r$ using the pre-computed PCA matrix. An approximate attention score matrix is calculated in d_r. This step is computationally efficient due to the lower dimensionality. Based on these approximate scores, LokiAttention identifies and selects the top-k most relevant tokens (e.g., 12.5-25% of the total tokens) from the KV-cache. The real P matrix is then computed using the selected subset of top-k tokens in $N \times d$. This approach allows Loki to achieve significant speedups (up to 45% over standard implementations) with minimal degradation in model accuracy.

► **HashAttention.** HashAttention (Desai et al., 2025) is a sparse attention algorithm that leverages learned mappings to accelerate long-context LLM inference. The motivation for HashAttention is framing the task of identifying important tokens as a Maximum Inner Product Search (MIPS) problem. To approximate this MIPS problem efficiently, HashAttention uses a trained linear projection to encode both queries and keys into compact, bit-level signatures. The distance between these binary signatures can be computed with an XOR operation. In this compressed space, a smaller distance between a query's signature and a key's signature indicates that the keys are considered important tokens. During inference, it finds the most relevant tokens for a given query by calculating the Hamming distance between their respective signatures. Attention is computed only on this small subset of selected tokens. By using learned, independent mappings, HashAttention achieves high recall of important tokens with very low auxiliary memory overhead (e.g., 32 bits per token).

► **SeerAttention.** SeerAttention (Gao et al., 2024) introduces a learnable block-sparse attention by inserting a gated linear projection g before the RoPE (Su et al., 2024a) module to predict block-wise sparse mask $\mathbf{M}$: $\hat{Q} = \text{RoPE}(g(\text{pool}(Q_{\text{pre-rope}})))$, $\hat{K} = \text{RoPE}(g(\text{pool}(K_{\text{pre-rope}})))$, $\mathbf{M} = \text{RowTopK}(\text{Softmax}\left(\hat{Q}\hat{K}^\top/\sqrt{d}\right))$, where $K_{\text{pre-rope}}$ denotes the attention keys before applying RoPE, `pool` refers to average pooling over tokens, and `RowTopK` selects the top-K entries in each row. The

parameters of the gated linear projection g are trained to minimize the KL divergence (Joyce, 2011) between SeerAttention and full attention outputs on long-context post-training data, allowing the model to retain sparsity while closely approximating full attention behavior.

▶ **RetrievalAttn.** RetrievalAttention (Liu et al., 2024b) is a training-free, CPU–GPU co-execution dynamic sparse attention for accelerating LLM decoding. It leverages the sparsity of attention by introducing a vector index (Malkov & Yashunin, 2018) to efficiently retrieve top-K KV cache. However, due to the distribution mismatch between queries (Q) and keys (K), directly building the index on K and retrieving with Q requires scanning 20–50% of K to maintain high recall. To address this, RetrievalAttention constructs an out-of-distribution (OOD)-aware vector index (Chen et al., 2024a) on the CPU by offloading the KV cache during prefilling. During decoding, q_t is transferred from GPU to CPU to retrieve the top-K keys, which are then used to compute attention scores on the CPU and merged with a small portion of GPU-computed local attention. This hybrid CPU–GPU design reduces decoding complexity to sub-$\mathcal{O}(n)$ and achieves up to 5 tokens/s for 8B LLMs with a 128K context on a single RTX 4090.

▶ **FPSAttention.** FPSAttention (Liu et al., 2025) is a training-aware FP8 quantization and structured sparsity co-design built on Sliding Tile Attention (STA) (Zhang et al., 2025e) for accelerating video diffusion models. It applies FP8 quantization and sparsity over the same 3-dimensional tiles, aligning with GPU- and FlashAttention-friendly block patterns for optimal hardware efficiency. For fine-tuning, FPSAttention integrates with SageAttention2 (Zhang et al., 2024a), using SageAttention2 in the forward pass and FlashAttention in the backward pass, while adopting a denoising-step-aware adaptive strategy to dynamically adjust sparsity according to the timestep. This design achieves up to a 7.09× speedup in the attention kernel and a 4.96× end-to-end generation speedup at 720p resolution, without compromising generation quality.

▶ **NSA.** Native Sparse Attention (NSA) (Yuan et al., 2025) is a trainable sparse attention method that accelerates attention with both algorithmic design and hardware optimizations. Algorithmically, NSA splits attention into three branches: (1) compression: K and V are blocked and compressed via a learnable MLP (Haykin, 1994), forming $\tilde{K}$ and $\tilde{V}$. Attention between q_t and $\tilde{K}, \tilde{V}$ yields O^{cmp}; (2) selection: top-k scores in the attention map from the compression branch select blocks for a sparse mask M_t. Sparse attention is then computed, producing O^{slc}; (3) sliding window: attention between q_t and $K_{t-w:t}$, $V_{t-w:t}$ within a sliding window of length w gives O^{win}. These outputs are gated and summed for the final result O. For hardware, NSA uses GQA (Ainslie et al., 2023) and aggregates importance scores across grouped query heads, therefore sharing KV blocks within the same group and reducing cache fetches, which boosts GPU efficiency. NSA also implements kernels using the Sparse FlashAttention approach for optimized contiguous memory access. These strategies give NSA a 6 to 11.6x speedup over full attention, while achieving even better results on general, long-context, and reasoning tasks.

▶ **MoBA.** MoBA (Lu et al., 2025), inspired by the Mixture of Experts (MoE) (Shazeer et al., 2017) mechanism, proposes a dynamic Mix of Block Attention. It evenly splits input K and V into n blocks and mean-pools each block into one token, forming compressed sequences $\tilde{K}$ and $\tilde{V}$ of length n. For each query q_t, attention scores are computed with $\tilde{K}$ and $\tilde{V}$, and the top-k blocks are selected to form the sparse mask M_t. Sparse attention is then computed. For causal attention, the block containing the t-th token (i.e., the current block of q_t) is always included and causal masking

is applied. On a 1M-token prefilling task, MoBA achieves up to 6.5x speedup over full attention with similar benchmark performance.

► **VMoBA.** VMoBA (Wu et al., 2025) extends MoBA to Video Diffusion Models (Ho et al., 2022). While MoBA targets only temporal locality, VMoBA also leverages spatial and spatio-temporal locality by dividing input K and V along temporal (1D), spatial (2D), or spatio-temporal (3D) axes in cycles across attention layers. Sparse attention is then computed as in MoBA. To further improve performance: (1) VMoBA selects top-k values across the entire sparse attention matrix $Q\tilde{K}^\top$ to better capture important queries; (2) it introduces a threshold τ per head, selecting the top elements that cumulatively reach τ instead of using a fixed top-k. These optimizations yield a 2.92x FLOPS and 1.48x latency speedup in training, and a 2.40x FLOPS and 1.35x latency speedup in inference, while maintaining similar video generation quality to full attention.

► **CHAI.** CHAI (Agarwal et al., 2024) is a training-free and efficient algorithm that groups the attention heads with high correlation to save both attention computations and memory usage during LLM inference. The method's key insight is that the output of different attention heads can deliver nearly identical attention outputs over the full sequence, which yields great redundancy. Moreover, such redundancy lies mainly in later layers. To turn this observation into a real speedup, deciding the number of clusters and the membership of each attention head has become crucial. To address this, CHAI first runs a quick offline clustering on tiny samples to decide the cluster number for each layer. During runtime, it dynamically conducts the clustering by assigning every head to one of the representative clusters based on the attention outputs of the first five tokens. When computing attention, only the representative head in each cluster keeps its Q and K, while the rest reuse its results. CHAI reduces both computations and K, V storage, achieving up to 1.73× inference speedup and 21.4% saving in KV cache memory size with minimal sacrifice in model accuracy.

► **Quest.** Quest (Tang et al., 2024) is a KV cache selection algorithm free of training that focuses on fast and memory-efficient long-context LLM inference. Quest's motivation is that only a tiny subset of the KV cache dominates attention scores in long-context LLMs, and the criticality of such important tokens depends on the current query token. Quest splits the KV cache into fixed-size pages and keeps track of the upper and lower bounds of the attention weights, leveraging them to approximate the highest possible attention in the page. At runtime, only the Top-k pages with the highest estimated attention scores are loaded to perform the sparse attention operation, thus significantly reducing memory consumption while accelerating the attention. Because the page scores are query-aware, tokens that were previously unimportant can be recalled if the new query values them, which allows Quest to surpass previous query-agnostic algorithms in terms of attention recall rate. Across long-context tasks, Quest delivers up to 7.03× faster self-attention and 2.23× end-to-end speedup with negligible reduction in accuracy.

► **MagicPig.** Standard Top-K sparse attention fails in aggregation tasks where attention is not highly concentrated, leading to significant quality degradation. To address this, MagicPig (Chen et al., 2024b) proposes that sampling from the attention distribution provides a more accurate, unbiased output. MagicPig uses Locality Sensitive Hashing (LSH) (Backurs et al., 2019) to approximate the ideal sampling distribution. The final output is computed with a modified Softmax that incorporates the LSH collision probabilities u_S: $\bar{o} = \text{Softmax}\left(\frac{qK_S^\top}{\sqrt{d}} - \log u_S\right) V_S$, where S is the set of indices sampled via LSH. MagicPig also features an efficient system co-design that partitions the workload: GPU executes compute-bound, low-memory tasks like linear projections and the LSH random projections for the query. CPU leverages its large DRAM to store the entire

KV cache and pre-built LSH hash tables, performing the final sparse attention computation on the sampled subset. This training-free algorithmic and system-level design overcomes the GPU memory bottleneck, while maintaining high accuracy and efficiency on extremely long contexts.

► **DraftAttn** . DraftAttn (Shen et al., 2025) is a training-free block sparse attention method for video diffusion transformers(DiT). The method consists of three main steps: (1) low-resolution draft attention computation, (2) permutation for structured sparsity, and (3) full-resolution sparse attention with original order restoration. In the low-resolution draft attention computation step, the method first applies down-sampling via average pooling to each spatial patch in the latent space, generating a low-resolution draft query ($\widetilde{Q}$) and draft key($\widetilde{K}$): $Q, K \in \mathbb{R}^{(T\cdot H\cdot W)\times d} \rightarrow \widetilde{Q}, \widetilde{K} \in \mathbb{R}^{(T\cdot H/h\cdot W/w)\times d}$, where h, w is height and width patch size in the latent space. The draft attention map is then computed using these down-sampled representations: $P_{\text{draft}} = \text{Softmax}\left(\frac{\widetilde{Q}\widetilde{K}^T}{\sqrt{d}}\right)$, exposing spatial and temporal redundancy. During permutation for structured sparsity, Q, K, and V are permuted based on P_{draft}. This permutation groups tokens in each patch contiguously in memory, aligning patch region-level sparsity with token-level computation and enabling hardware-friendly execution. In the full-resolution sparse attention step, the block mask($\mathbf{M}$) is constructed from the P_{draft} by selecting the highest score with a predefined ratio r. Block sparse attention computation at full resolution was performed with $\mathbf{M}$. After the attention computation, the original order of tokens is restored using an inverse permutation. To ensure efficiency, DraftAttention uses aggressive downsampling (e.g., 8×16 pooling kernel) to minimize the computational overhead of draft attention.

► **XAttention.** (Xu et al., 2025) proposes using the sum of antidiagonal values (i.e., from the lower-left to upper-right) in the attention matrix as an importance metric for attention block selection. To efficiently compute this antidiagonal sum, XAttention rearranges the Q and K matrices by grouping segments of the sequence dimension together while expanding the feature dimension accordingly. By reversing the order of elements within segments of one of the matrices before grouping, the product of the rearranged Q and K effectively captures the sum of the antidiagonal elements. XAttention achieves up to 13.5× speedup in attention computation while maintaining accuracy comparable to full attention.

► **VSA.** Video Sparse Attention (Zhang et al., 2025f) is a trainable, data-dependent sparse attention mechanism designed for video diffusion models, and is compatible with diffusion model distillation. In contrast to post-hoc methods that apply sparsity only at inference, Video Sparse Attention introduces a hierarchical two-stage attention structure that operates natively during both training and inference. In the first stage, the video latent is partitioned into $(4, 4, 4)$ spatio-temporal cubes, and coarse cube-to-cube attention is applied to identify Top-K critical regions. These regions are then used to construct a block-sparse attention mask. In the second stage, fine-grained token-level attention is computed only within the selected blocks, and the outputs from both stages are fused via learnable gating. This design preserves compatibility with block-sparse GPU kernels and achieves 85% of FlashAttention3's MFU (Shah et al., 2024). Under a comparable compute budget, Video Sparse Attention consistently outperforms fixed-pattern sparse baselines. On Wan2.1-1.3B, it reduces training FLOPs by 2.53× and accelerates inference by 1.7×, without compromising generation quality.

► **Sparse VideoGen2.** Sparse VideoGen2 (Yang et al., 2025b) is a training-free sparse attention designed to accelerate video generation. The method comprises three main steps: (1) semantic-aware permutation, (2) centroid-based top-p selection, and (3) dynamic block sparse attention. In the semantic-aware permutation step, the method applies k-means clustering to the query (Q) and key

(K) tokens to group semantically similar tokens. Based on the clustering results, it permutes Q, K, and V to ensure a continuous memory layout. During centroid-based top-p selection, the method estimates the attention scores between clusters by using their centroids, weighting these scores by cluster size. For each query cluster, it selects key clusters in descending order of their weighted attention score until the cumulative score surpasses a predefined threshold p. In the dynamic block sparse attention step, the model computes the attention only for the selected cluster pairs (blocks), while completely skipping the computation for all other blocks. To further improve performance, Sparse VideoGen2 utilizes a centroid cache, which leverages centroids from the previous diffusion step to accelerate K-means convergence. It also employs a customized sparse attention kernel that supports variable block sizes to maximize Tensor Core utilization. Sparse VideoGen2 shows significant acceleration in video generation tasks.

► **TidalDecode.** TidalDecode (Yang et al., 2024a) accelerates LLM decoding with position persistent sparse attention, based on the observation that the most attended tokens show strong overlap across layers within a decoding step. Instead of running full attention in every layer, it begins with a few full attention layers, then periodically selects the top-k important tokens, and lets the remaining layers reuse these token indices to reduce computation and memory cost. To avoid errors from repeated sparse updates, it introduces a cache-correction step that occasionally refreshes the cache with full attention. By exploiting this spatial coherence of token importance, TidalDecode achieves up to 2.1× faster decoding while maintaining performance close to full attention.

► **LessIsMore.** LessIsMore (Yang et al., 2025a) introduces a training-free sparse attention for large reasoning models. The authors find that important tokens show strong overlap across attention heads and that recently generated tokens consistently remain critical . Leveraging these observations, LessIsMore adopts a unified selection strategy that globally ranks tokens across all heads and reserves a fixed proportion of its budget for recent tokens. This design preserves essential context, improves efficiency, and avoids common errors in sparse methods. As a result, LessIsMore reduces the number of attended tokens by up to half without loss, speeds up decoding, and maintains generation length and accuracy comparable to full attention.

► **Twilight.** Twilight (Lin et al., 2025) uses top-p pruning for adaptive sparsity. It adopts a Select-then-Prune framework: a base method first selects candidate tokens, then the Twilight Pruner keeps only those whose cumulative attention exceeds a threshold. This adaptive pruning removes up to 98% of tokens with minimal accuracy loss and yields up to 1.4× speedup over SOTA sparse attention.

7 Linear Attention

In this section, we provide unified formulations for linear attention, offer a detailed summary of representative methods and their unique attributes, and examine the implementation aspects of individual algorithms.

7.1 Overall Formulation

Linear attention decreases the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ by decomposing the softmax function and using the combination property of matrix multiplication. It can be formulated as:

$$O = \phi(Q)(\phi(K)^\top V), \quad O, Q, K, V \in \mathbb{R}^{N \times d}. \tag{45}$$

where ϕ is a kernel function applied row-wise to queries and keys. When applied to non-autoregressive tasks, it can be computed directly using Equation 45. However, when applied to autoregressive tasks, due to the causal relationships between tokens, the attention computation is formulated as:

$$o_t = \phi(q_t) \sum_{i=1}^{t} (\phi(k_i)^\top v_i). \tag{46}$$

Here, the subscript t(or i) represents the time step t(or i). To avoid the costly computation of historical $\sum_{i=1}^{t-1} k_i^\top v_i$ during inference, a hidden state H_t is maintained to store the historical $\sum_{i=1}^{t-1} k_i^\top v_i$, and recurrently updated. The hidden state and the output is computed as follows:

$$\begin{aligned} H_t &= H_{t-1} + \phi(k_t)^\top v_t. \\ o_t &= \phi(q_t) H_t. \end{aligned} \tag{47}$$

However, compressing all historical information into a fixed-size hidden state inevitably leads to information loss. Forget gate G_f and select gate G_s are introduced to mitigate this problem by forgetting historical information in H_{t-1} and selecting current information in $k_t^\top v_t$. The hidden states update with gates can be formulated as:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot k_t^\top v_t. \tag{48}$$

Here we omit the kernel function ϕ for simplicity; $G^{(t)}$ represents the gates at time t. We call G_f and G_s input-dependent if they rely on the attention computation input, and input-independent otherwise.

Linear attention methods can be classified by their hidden state update method. The first three categories rely on direct computation of H_t: **(1)** Naive Linear Attention: Linear attention without gates, i.e., both $G_f^{(t)}$ and $G_s^{(t)}$ are fixed as $\mathbf{1}^\top \mathbf{1}$, **(2)** Linear Attention with a Forget Gate: Only $G_s^{(t)}$ is fixed as $\mathbf{1}^\top \mathbf{1}$, while the forget gate $G_f^{(t)}$ is predefined or input-dependent, and **(3)** Linear Attention with both Forget and Select Gates: both $G_f^{(t)}$ and $G_s^{(t)}$ are predefined or input-dependent rather than fixed as $\mathbf{1}^\top \mathbf{1}$. In these models, the hidden state H_t at each step is calculated directly from the previous state and the current input. In contrast, the fourth category, **(4)** Test-Time Training (TTT) (Sun et al., 2024), adopts an optimization-based approach. TTT re-conceptualizes the hidden state H_t, not as a computed value, but as a set of learnable parameters known as fast weights (Schlag et al., 2021b). The key distinction is that these fast weights are updated via gradient descent (Robbins & Monro, 1951) during both training and inference. This continuous learning process sets TTT apart from conventional architectures, where model parameters are frozen during inference. We will discuss each of these categories in detail in the following subsections: Naive linear attention in Section 7.4, linear attention with a forget gate in Section 7.5, linear attention with both forget and select gates in Section 7.6, and TTT in Section 7.7.

7.2 Preliminaries of Linear Attention without Gates

As mentioned in Section 3.4, there are three basic forms of linear attention: the parallel form, the recurrent form, and the chunkwise form. The parallel form is the most fundamental implementation and has two variants: a linear parallel form and a quadratic parallel form. The chunkwise form is a hybrid that combines aspects of the recurrent and quadratic parallel forms. In modern models, the application of these forms is task-dependent. For non-autoregressive tasks, the linear parallel form is typically applied during both training and inference. In contrast, for autoregressive tasks, the chunkwise form is employed for training, while the recurrent form is used for efficient inference.

7.2.1 Linear Parallel Form for Non-Autoregressive Tasks

The linear parallel form is optimal for Non-Autoregressive (NAR) settings where causality is not a constraint. It is defined as:

$$O = \phi(Q)\left(\phi(K)^\top V\right). \tag{49}$$

The computation first calculates a global key-value state, $\phi(K)^\top V$, which is then queried by all positions in parallel. This allows for maximum throughput during NAR training and inference. For example, the output for a query Q_n can be computed as:

$$O_n = \frac{\phi(Q_n)\sum_{j=1}^{N}\phi(K_j)^\top V_j}{\phi(Q_n)\sum_{j=1}^{N}\phi(K_j)^\top}. \tag{50}$$

7.2.2 Recurrent Form for Autoregressive Inference

For autoregressive inference, linear attention can be reformulated into a highly efficient recurrent form. At each timestep t, the output o_t depends only on the current query q_t and an accumulated state H_t:

$$H_t = H_{t-1} + \phi(k_t)^\top v_t \quad , \quad o_t = \phi(q_t)H_t. \tag{51}$$

where H_0 is initialized as a zero matrix. This approach offers significant advantages over standard attention's Key-Value (KV) cache: (1) The state H_t has a fixed size, whereas the KV cache grows linearly with the sequence length. (2) Each generation step is an $\mathcal{O}(1)$ state update, independent of the sequence length t.

7.2.3 Chunkwise Form for Parallel Autoregressive Training

During autoregressive training, there is a fundamental tension between computational complexity and hardware parallelism. Two naive approaches present a dilemma: (1) Quadratic parallel form: Applying a causal mask M to the parallel form, as in $O = (\phi(Q)\phi(K)^\top \odot M)V$, is parallelizable but forces the computation of the $N \times N$ matrix, reverting to an inefficient $\mathcal{O}(N^2)$ complexity. (2) Pure recurrent form: The complexity is theoretically $\mathcal{O}(N)$, but its sequential nature makes training on GPUs prohibitively slow as shown in (Schlag et al., 2021a).

The chunkwise form provides a practical hybrid solution for this problem. It divides the sequence into fixed-size chunks, processing each chunk in N^2 parallel form while using a recurrent mechanism to carry information between them. For the i-th chunk, the computation combines an intra-chunk parallel calculation with an inter-chunk recurrent update. Given a causal mask D within a chunk:

$$O_{\text{intra},i} = (Q_{[i]}K_{[i]}^T \odot D)V_{[i]} \qquad \text{(Intra-chunk Parallel)}. \tag{52}$$

$$O_{\text{inter},i} = Q_{[i]}H_{i-1} \qquad \text{(Inter-chunk Recurrent)}. \tag{53}$$

$$O_i = O_{\text{intra},i} + O_{\text{inter},i} \qquad \text{(Final Chunk Output)}. \tag{54}$$

$$H_i = K_{[i]}^T V_{[i]} + H_{i-1} \qquad \text{(Next State Update)}. \tag{55}$$

This hybrid approach effectively balances parallelism and causality, enabling efficient autoregressive training on modern hardware with a complexity of $\mathcal{O}(NCd + Nd^2)$ for a chunk size C.

7.3 Preliminaries of Linear Attention with Gates

We now elaborate on the computational forms from Section 7.2.2, detailing their implementation with gates.

7.3.1 Recurrent Form

The formulation of recurrent form with gates is defined as follows:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot k_t^\top v_t. \tag{56}$$

$$o_t = q_t H_t. \tag{57}$$

The forget gate $G_f^{(t)}$ controls how much of the previous state H_{t-1} is retained. Concurrently, the select gate $G_s^{(t)}$ determines how the new information $k_t^\top v_t$ is incorporated into the new state H_t. Finally, the output o_t is computed by applying the current query q_t to this updated state.

7.3.2 Quadratic Parallel Form

When considering the gates, it is necessary to consider the cumulative effect of the gates along the sequence. The overall formulation is given by Equation 58 (see the Appendix A.1 for the derivation).

$$\begin{aligned} O &= (\tilde{Q}\tilde{K}^\top \odot M)\tilde{V} \odot \Lambda, \\ \text{where} \quad \tilde{Q} = Q \odot \mathcal{A} \quad \tilde{K} &= K/\mathcal{A} \odot \hat{A} \quad \tilde{V} = V/\Lambda \odot \hat{B}. \end{aligned} \tag{58}$$

Here, $\mathcal{A}$ and Λ represent the cumulative forget effects along the sequence, and $\hat{A}, \hat{B}$ represent the cumulative select effects along the sequence. In this formulation, the queries, keys, and values are pre-scaled by the cumulative gate values. This allows the attention-like matrix $\tilde{Q}\tilde{K}^\top$ to correctly model the sequence's causal and gated dependencies in a parallel manner. However, its quadratic time complexity results in inefficient training. Consequently, many modern architectures instead favor the chunkwise form, which sacrifices some parallelism to achieve lower computational complexity.

7.3.3 Chunk-wise Formulation for Gated Linear Attention

Incorporating the gates results in a more detailed chunk-wise formulation defined by Equation 59a through Equation 59d(see Appendix A.2 for the derivation).

$$H_{[i]} = (\zeta_i \odot H_{[i-1]}) + \widehat{K}_{[i]}^\top \widehat{V}_{[i]}. \tag{59a}$$

$$O_{[i+1]} = O_{[i+1]}^{\text{Intra}} + O_{[i+1]}^{\text{Inter}}. \tag{59b}$$

$$O_{[i+1]}^{\text{Intra}} = (\tilde{Q}_{[i+1]}\tilde{K}_{[i+1]}^\top \odot M)\tilde{V}_{[i+1]} \odot \Lambda_{[i+1]}. \tag{59c}$$

$$O_{[i+1]}^{\text{Inter}} = (Q_{[i+1]} \odot A_{[i+1]}^\dagger)H_{[i]} \odot B_{[i]}^\dagger. \tag{59d}$$

The chunk-level hidden state $H_{[i]}$ is updated using Equation 59a, where ζ_i represents the cumulative forgetfulness of the previous chunk. The terms $\widehat{K}_{[i]}$ and $\widehat{V}_{[i]}$ are aggregated from keys and values, where each key value pair are first modulated by select gate $(\hat{a}_t, \hat{b}_t)$ and then decayed by the remaining forget gates within the chunk.

The output for the $(i+1)$-th chunk, $O_{[i+1]}$, is decomposed into two components: an intra-chunk output $O_{[i+1]}^{\text{Intra}}$ and an inter-chunk output $O_{[i+1]}^{\text{Inter}}$. $O_{[i+1]}^{\text{Intra}}$ is computed within the current chunk using the quadratic parallel form. $O_{[i+1]}^{\text{Inter}}$ incorporates historical information from the previous chunk via the recurrent state $H_{[i]}$. The decay matrices $A_{[i+1]}^\dagger$ and $B_{[i]}^\dagger$ apply the appropriate decay based on the relative positions of queries.

Table 5: Summary of naive linear attention methods.

Method	Forget Gate	Select Gate	Time Complexity	Space Complexity	Form
Linear transformer (Katharopoulos et al., 2020)	$\mathbf{1}^\top\mathbf{1}$	$\mathbf{1}$	$\mathcal{O}(N^2d)$	$\mathcal{O}(N^2)$	quadratic
CHELA (Liu et al., 2024c)	$\mathbf{1}^\top\mathbf{1}$	$\mathbf{1}$	$\mathcal{O}(N^2d)$	$\mathcal{O}(N^2)$	quadratic
Lightning Atten (Qin et al., 2024)	$\mathbf{1}^\top\mathbf{1}$	$\mathbf{1}$	$\mathcal{O}(NCd+Nd^2)$	$\mathcal{O}(Nd)$	chunkwise
SLA (Guo et al., 2024)	$\mathbf{1}^\top\mathbf{1}$	$\mathbf{1}$	$\mathcal{O}(N^2d)$	$\mathcal{O}(N^2)$	quadratic
			$\mathcal{O}(NCd+Nd^2)$	$\mathcal{O}(Nd)$	chunkwise

[1] $\mathbf{1}$ in the table represents a $1 \times d$ vector of all ones.

7.4 Naive Linear Attention

When both forget gate G_f and select gate G_s are $\mathbf{1}^\top\mathbf{1}$ (i.e., they apply no transformation), the method is categorized as Naive Linear Attention. Table 5 shows some typical naive linear attention methods. All the complexities shown in the table are for the training phase. In this table, all methods use a recurrent form for inference, with time complexity of $\mathcal{O}(Nd^2)$ and space complexity of $\mathcal{O}(Nd)$.

► **Linear transformer.** Linear Transformer (Katharopoulos et al., 2020) decomposes the $\text{softmax}(QK^\top)$ into a kernelized dot product $\phi(Q)\phi(K)^\top$ and utilizes the matrix product associativity to reduce the time complexity to linear. The attention output for a single query q_i is formulated as:

$$o_i = \frac{\phi(q_i)^\top \left(\sum_{j=1}^N \phi(k_j)v_j^\top\right)}{\phi(q_i)^\top \left(\sum_{j=1}^N \phi(k_j)\right)} \tag{60}$$

By first computing $\sum_{j=1}^N \phi(k_j)v_j^\top = \phi(K)^\top V$, the overall complexity becomes $\mathcal{O}(Nd^2)$. For autoregressive tasks, this method maintains two state matrices, H_t and Z_t, to compute attention output, and can be formulated as

$$\begin{aligned} H_t &= H_{t-1} + \phi(k_t)v_t^\top \\ Z_t &= Z_{t-1} + \phi(k_t) \\ o_t &= \frac{\phi(q_t)^\top H_t}{\phi(q_t)^\top Z_t} \end{aligned} \tag{61}$$

This state-based update reduces the memory complexity during inference from $\mathcal{O}(Nd)$ to a constant $\mathcal{O}(d^2)$, eliminating the need for a growing KV cache.

► **SANA.** SANA text-to-image framework (Xie et al., 2025) is an application of naive linear attention in a non-autoregressive task. SANA replaces all standard quadratic self-attention of the DiT architecture (Peebles & Xie, 2023) with linear attention. SANA is well-suited for this optimization because it processes the entire image representation at once and does not require a causal mask. Specifically, SANA employs ReLU Linear Attention (Katharopoulos et al., 2020).

This method reformulates the attention calculation by first computing two shared terms: $H_N = \sum_{j=1}^{N} \mathrm{ReLU}(K_j)^\top V_j, Z_N = \sum_{j=1}^{N} \mathrm{ReLU}(K_j)^\top$. The final output is computed as $O_n = \frac{\mathrm{ReLU}(Q_n)H_N}{\mathrm{ReLU}(Q_n)Z_N}$.

► **CHELA.** CHELA (Liu et al., 2024c) leverages a tiling strategy inspired by FlashAttention (Yang et al., 2024b) to achieve a hardware-efficient implementation of linear attention. To enhance training stability, it incorporates short-long convolution operation to before compute Q, K matrices. This operation is defined as $Z = \overline{K}_l \left(\phi_{\text{silu}} \left(\overline{K}_s(X)\right)\right)$, $\overline{K}_l$ and $\overline{K}_s$ represent long and short convolution kernels, respectively. ϕ_{silu} is the SiLU activation function (Ramachandran et al., 2017), and X is the current token representations. Then, Q, K, V, O are computed as follows:

$$\begin{aligned} Q &= \alpha_q \odot Z + \beta_q \\ K &= \alpha_k \odot Z + \beta_k \\ V &= \phi_{\text{silu}}(XW_v + b_v) \\ O &= \mathrm{Norm}\left(Q(K^\top V)\right) \odot G_a \end{aligned} \tag{62}$$

Here, α_*, β_*, b_* are all learnable parameters, and $G_a = \phi_{\text{silu}}(ZW_v + b_g)$. functions as an input-dependent filter on the final attention score, leveraging the global information from the convolutions to refine the final output. For inference, CHELA employs: $H_t = H_{t-1} + K_t^\top V_t$ and $o_t = \mathrm{Norm}(q_t H_t) \odot G_a$. For training, it uses chunkwise form as introduced in Section.7.2.3.

► **LightningAttention.** Lightning Attention (Qin et al., 2024) adopts the chunkwise form for hardware-efficient implementation, formulated as follows:

$$H_t = H_{t-1} + K_t^\top V_t. \tag{63}$$

$$O_t^{\text{intra}} = \left((Q_t K_t^\top) \odot M\right) V_t. \tag{64}$$

$$O_t^{\text{inter}} = Q_t H_{t-1}. \tag{65}$$

$$O_t = O_{\text{intra}} + O_{\text{inter}}. \tag{66}$$

The attention computation is split into intra-chunk computation and inter-chunk computation. O_t^{intra} is computed using the quadratic parallel form. O_t^{inter} incorporates information from all previous blocks. The hidden state H_t is updated using the recurrent form. To minimize I/O cost, during each iteration, only the current blocks (Q_t, K_t, V_t) are loaded from global memory to shared memory. H_t is also maintained and updated entirely within shared memory. As a result, LightningAttention achieves a constant training speed and memory footprint, regardless of the input sequence length, significantly improving the efficiency of linear attention for language modeling.

► **SLA.** Unlike prior linear attention methods that employ complex kernel functions, Simplified Linear Attention(SLA) (Guo et al., 2024) uses simpler ReLU as its kernel function. And the computation of SLA can be formulated as Equation 67.

$$\begin{aligned} H_t &= H_{t-1} + \phi(K_t)^\top V_t \\ Z_t &= Z_{t-1} + \phi(K_t)^\top \\ o_t &= \frac{\phi(Q_t)H_t}{\phi(Q_t)Z_t} + \mathrm{DWC}(V) \end{aligned} \tag{67}$$

where $\phi(x) = \mathrm{ReLU}(x)$ (Nair & Hinton, 2010), and DWC represents depth-wise convolution (Chollet, 2017) used to ensure model captures local contextual patterns. This design effectively balances

performance with computational efficiency, achieving significant latency reductions by leveraging hardware-friendly operations (ReLU and DWC).

► **QT-ViT.** QT-ViT (Xu et al., 2024) approximates the softmax function with a second-order Taylor expansion (Dass et al., 2023), achieving better performance while maintaining efficiency. The similarity function is computed as:

$$Sim(q,k) \approx 1 + \frac{\langle q,k\rangle}{\sqrt{d}} + \frac{\langle q,k\rangle^2}{2d}. \tag{68}$$

However, the quadratic term $\langle q,k\rangle^2$ prevents the decomposition into two separate kernel functions. To overcome this, QT-ViT leverages the Kronecker product $\otimes$ (Broxson, 2006), using the property that $\langle a,b\rangle^2 = \langle K_r(a), K_r(b)\rangle$, where $K_r(x) = \mathrm{vec}(x \otimes x)$ vectorizes the outer product of the vector with itself. However, this decomposition increases the feature dimension from d to d^2, resulting in a time complexity of $\mathcal{O}(Nd^3)$. Thus, a fast and compact approximation of the Kronecker product is introduced. Instead of computing all d^2 cross-product terms, it was empirically found that using only the d self-multiplication terms $\{x_i^2\}_{i=1}^d$ is sufficient to effectively represent the quadratic information. This leads to a compact kernel function, $\tilde{K}_r(\phi(x))$, which is a concatenation of three components: the approximated quadratic terms, the linear terms, and a constant, each scaled by a learnable parameter (α,β,γ). The function is defined as:

$$\tilde{K}_r(\phi(x)) = \mathrm{concat}\left(\alpha \cdot \{x_i^2\}_{i=1}^d, \beta \cdot \sqrt[4]{\frac{4}{d}}\{x_i\}_{i=1}^d, \gamma\right). \tag{69}$$

By using $\tilde{K}_r(\phi(x))$, the output vector has a manageable dimension of $2d+1$. This allows QT-ViT to achieve $\mathcal{O}(Nd^2)$ complexity while effectively capitalizing on the higher-order information.

7.5 Linear Attention with a Forget Gate

A method is categorized as linear attention with a forget gate if its hidden state update uses $G_f^{(t)} \neq \mathbf{1}^\top\mathbf{1}$ while $G_s^{(t)} = \mathbf{1}^\top\mathbf{1}$. Table 6 shows some typical linear attention methods with a forget gate. All the complexities shown in the table are the complexities of the training phase. In the table, all methods use the recurrent form for inference.

► **RetNet.** RetNet (Sun et al., 2023) introduces a forget gate $\gamma\mathbf{1}^\top\mathbf{1}$, along with a chunkwise representation to enable efficient training on long sequences. For inference, RetNet use the recurrent form: $H_t = \gamma H_{t-1} + k_t^\top v_t, o_t = q_t H_t$. The decay factor γ controls how much of the past information is retained at each step. The chunkwise form balances parallelism and recurrence. The intra-chunk output is calculated in quadratic parallel form, $O_{[i]}^{\text{intra}} = (QK^\top \odot D)V$, $D_{nm} = \begin{cases}\gamma^{n-m}, & n \geq m\\ 0, & n < m\end{cases}$. The matrix D combines causal masking with an exponential decay based on the relative distance between tokens. Finally, the chunkwise form can be formulated as:

$$H_{[i]} = K_{[i]}^\top(V_{[i]} \odot \zeta) + \gamma^C H_{i-1}, \text{where } \zeta_{i,j} = \gamma^{C-i-1}. \tag{70}$$

$$O_{[i+1]}^{\text{inter}} = (Q_{[i]}H_{i-1}) \odot \xi, \text{where } \xi_{i,j} = \gamma^{i+1}. \tag{71}$$

$$O_{[i+1]} = O_{[i+1]}^{\text{intra}} + O_{[i+1]}^{\text{inter}}. \tag{72}$$

Table 6: Summary of linear attention with forget gate only.

Method	Forget Gate	Select Gate	Time Complexity	Space Complexity	Form
Retnet (Sun et al., 2023)	$\gamma \mathbf{1}^\top \mathbf{1}$	$\mathbf{1}$	$O(N^2 d)$	$O(N^2)$	quadratic
			$O(NCd + Nd^2)$	$O(Nd^2/C + Nd)$	chunkwise
GLA (Yang et al., 2023)	$\alpha_t^\top \mathbf{1}$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C + Nd)$	chunkwise
RWKV4 (Peng et al., 2023)	e^{-w}	$\mathbf{1}$	$O(Nd)$	$O(d)$	parallel
RWKV5 (Peng et al., 2024)	$w^\top \mathbf{1}$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C)$	chunkwise
RWKV6 (Peng et al., 2024)	$w_t^\top \mathbf{1}$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C)$	chunkwise
RWKV7 (Peng et al., 2025)	$(\mathrm{diag}(w_t) - \kappa_t(a_t \odot \kappa_t))^*$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C)$	chunkwise
GSA (Zhang et al., 2024b)	$\alpha_t^\top \mathbf{1}$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C + Nd)$	chunkwise
MetaLA (Chou et al., 2024)	$\alpha_t^\top \mathbf{1}$	$\mathbf{1}$	$O(NCd + Nd^2)$	$O(Nd^2/C + Nd)$	chunkwise

[1] $\mathbf{1}$ in the table represents a $1 \times d$ vector of all ones.
[2] The blue gates are input-dependent and the black gates are input-independent.

This hybrid form allows for parallel processing within local blocks for speed while efficiently summarizing global information recurrently to save memory.

► **GLA.** Gated Linear Attention(GLA) (Yang et al., 2023) employs an input-dependent and vector-valued forget gate and introduced a hardware-efficient implementation with two variants. The gate, $\alpha_t \in (0,1)^{d_k}$, is computed dynamically from each input token x_t, allowing different feature dimensions to decay at different rates. This fine-grained control contrasts with models like RetNet, which use a fixed, input-independent scalar decay. For inference, GLA use the recurrent form: $H_t = (\alpha_t^\top \mathbf{1}) \odot H_{t-1} + k_t^\top v_t, o_t = q_t H_t$. The gate vector α_t is generated via a low-rank linear layer, followed by a sigmoid activation and a scaling term τ: $\alpha_t = \sigma\left(x_t W_\alpha^1 W_\alpha^2 + b_\alpha\right)^{\frac{1}{\tau}}$. By defining a cumulative gate product $b_t = \prod_{j=1}^{t} \alpha_j$, the output can be written in quadratic parallel form, which is used to compute the intra-chunk output of chunkwise form: $O_{[i+1]}^{\text{intra}} = \left(\left((Q_{[i+1]} \odot B_{[i+1]})\left(\frac{K_{[i+1]}}{B_{[i+1]}}\right)^\top\right) \odot M\right) V_{[i+1]}$, where M is the causal mask, and B is the matrix of stacked b_t vectors. Due to potential numerical instability from the cumulative product, this form is often computed in log space. Finally, the chunkwise form can be formulated as follows, where matrices Γ and Λ are derived from the per-token α_t values and manage the decay across chunks:

$$H_{[i+1]} = (\gamma_{i+1}^\top 1) \odot H_{[i]} + (K_{[i+1]} \odot \Gamma_{[i+1]})^\top V_{[i+1]}. \tag{73}$$

$$O_{[i+1]}^{\text{inter}} = (Q_{[i+1]} \odot \Lambda_{[i+1]}) H_{[i]}. \tag{74}$$

$$O_{[i+1]} = O_{[i+1]}^{\text{intra}} + O_{[i+1]}^{\text{inter}}. \tag{75}$$

GLA also provides an I/O-aware implementation with two variants: (1) The non-materialization version computes chunks sequentially, using fast shared memory to carry the hidden state $H_{[n]}$

between chunks without writing to slower global memory. It is memory-efficient with $O(Nd)$ training space complexity but lacks sequence-level parallelism. (2) The materialization version first runs the inter-chunk recurrence and stores all intermediate states $H_{[n]}$ in global memory. Subsequently, the outputs $O_{[n]}$ for all chunks are computed in parallel. This boosts parallelism at the cost of increased space complexity $O(Nd^2/C + Nd)$.

► **GSA.** GSA (Zhang et al., b) uses a input-dependent forget gate α_t to selectively manage its hidden states($\widetilde{K}_t$ and $\widetilde{V}_t$). It can be expressed as:

$$\begin{aligned} \widetilde{K}_t &= \mathrm{Diag}(\alpha_t) \cdot \widetilde{K}_{t-1} + (1-\alpha_t)^\top k_t, \\ \widetilde{V}_t &= \mathrm{Diag}(\alpha_t) \cdot \widetilde{V}_{t-1} + (1-\alpha_t)^\top v_t, \\ o_t &= \widetilde{V}_t^\top \mathrm{softmax}(\widetilde{K}_t^\top q_t). \end{aligned} \tag{76}$$

where α_t is the forget gate obtained via a linear transformation followed by a sigmoid activation with a damping factor (Qin et al., 2023). For parallel and hardware-efficient training, this recurrence is reformulated as a two-pass GLA, where the output of the first pass becomes the query for the second pass after a softmax non-linearity:

$$\begin{aligned} H_t^1 &= (\alpha_t^\top \mathbf{1}) \odot H_{t-1}^1 + k_t^\top (1-\alpha_t), \\ o_t' &= q_t H_t^1, \\ H_t^2 &= (\mathbf{1}^\top \alpha_t) \odot H_{t-1}^2 + (1-\alpha_t)^\top v_t, \\ o_t &= \mathrm{Softmax}(o_t') H_t^2. \end{aligned} \tag{77}$$

This two-pass structure allows GSA to leverage existing optimized chunkwise training algorithms, ensuring both high performance in recall-intensive tasks and efficient computation.

► **MetaLA.** MetaLA (Chou et al., 2024) proposes three critical conditions for an optimal linear attention: (1)dynamic memory, (2)accurate approximation of softmax attention, and (3) parameter efficiency. Based on these principles, they propose MetaLA, which can be formulated as:

$$\begin{aligned} H_t &= \alpha_t^\top 1 \odot H_{t-1} + (1-\alpha_t)^\top v_t, \\ o_t &= q_t H_t + \sigma_{\text{aug}}(q_t(w_{\text{aug}} \odot (1-\alpha_t))^\top v_t). \end{aligned} \tag{78}$$

Here, α_t^h is derived from the current input x_t, which allows the model to selectively forget or remember information based on the input context. Notably, MetaLA replaces the traditional k_t with $1-\alpha_t$, enhancing parameter efficiency. $\sigma_{\text{aug}}(q_t^h(w_{\text{aug}}^h \odot (1-\alpha_t^h))^T v_t)$ enhances the current token's attention to itself. A learnable parameter w_{aug}^h controls the extent of this self-augmentation. MetaLA employs a chunk-wise parallel form during training for hardware efficiency and a recurrent form during inference. A short convolution layer can be introduced before attention computation to further enhance local interactions.

► **LogLinearAttention.** Log-linear Attention (LLA) (Guo et al., 2025) employs a hidden state whose size grows logarithmically with the sequence length. It hierarchically partitions the past context using a Fenwick tree (Fenwick, 1994) structure (binary indexed trees). For any position t in the sequence, this divides the history $[0, t)$ into a logarithmic number. This partitioning scheme creates finer-grained segments for recent history and progressively coarser segments for more distant history, allowing a finer representation of recent context. Due to this implicit down-weighting of

older information, we categorize it as a linear attention variant with a forget mechanism, even though it lacks an explicit gate.

Instead of compressing the entire past into a single hidden state, LLA maintains a hidden state $H_t^{(i)}$ for each bucket i, and the hidden state of each bucket is an add of the past $k^\top v$ pairs in the bucket. The final output is then computed by taking a weighted sum of the attention scores from all these hidden states, which can be formulated as:

$$o_t = \sum_{l=0}^{L-1} \lambda_t^{(l)} q_t^\top H_t^{(l)}. \tag{79}$$

Here, $H_t^{(l)}$ is the hidden state that summarizes the information for the segment at level l. $\lambda_t^{(l)}$ is a learned, input-dependent scalar weight that modulates the contribution of the hidden state from each level l, allowing the model to adaptively focus on the most relevant time scales. This approach allows for a more expressive representation of context while maintaining an efficient $\mathcal{O}(N \log N)$ training time and $\mathcal{O}(\log N)$ memory cost during decoding. LLA is not included in Table 6 as its multi-state architecture does not conform to the single-state update formulation used for categorization.

▶ **RWKV Series** . Receptance Weighted Key Value (RWKV) series models replace the self-attention in the standard transformer with the time-mixing module. This time-mixing module is a variant of gated linear attention, with linear time complexity and constant memory usage in inference. In the following, we introduce representative RWKV models with improvements for the time-mixing module.

RWKV-4. The update rule and output of time-mixing module in RWKV-4 (Peng et al., 2023) can be formulated as:$H_t = e^{-w} \odot H_{t-1} + e^{k_t} \odot v_t, H_t^{'} = e^{-w} \odot H_{t-1}^{'} + e^{k_t}, o_t = \sigma(q_t) \odot (H_t / H_t^{'})$. Here, all operations are performed element-wise. Here H_t and $H_t^{'}$ are vectors in $\mathbb{R}^d$. The time-decay parameter w is a learnable, input-independent vector. The forget gate e^{-w} applies a channel-wise decay rate to each feature of the hidden state. During inference, the time complexity of RWKV-4 is $O(Nd)$ and the space complexity is $O(d)$.

RWKV-5 & 6. RWKV-5 & 6 (Peng et al., 2024) expand the hidden state of the time mixing module to a matrix H_t. The hidden state of H_t of RWKV-5 is updated as:

$$H_t = H_{t-1} diag(w) + v_t^\top k_t \tag{80}$$

where $diag(w)$ is a diagonal input-independent forget gate. RWKV-6 (Peng et al., 2024) changes the forget gate in Equation 80 into a input-dependent one: $H_t = H_{t-1} diag(w_t) + v_t^\top k_t$.

RWKV-7. RWKV-7 (Peng et al., 2025) further enhances the expressive capability of the time mixing module through a more general forget gate. The state update is defined as follows: $H_t = (diag(w_t) - \kappa_t(\alpha_t \odot \kappa_t))H_{t-1} + v_t^\top k_t$. Here, $\kappa_t = k_t \odot \epsilon, \alpha_t = \text{sigmoid}(\text{MLP}(\text{x}_\text{t}))$, and ϵ is a learnable parameter and x_t is the current token representation. The term w_t is an input-dependent, vector-valued decay parameter. This formulation of forget gates is a diagonal matrix plus a rank-one matrix. This special algebraic structure allows a hardware-efficient chunkwise form.

7.6 Linear Attention with Forget and Select Gates

Table 7 shows some typical linear attention methods with forget and select gates. All complexities shown in the table are the complexities of the training phase. For inference of Mamba, the time

Table 7: Summary of linear attention methods with forget gate and select gate.

Method	Forget Gate	Select Gate	Time Complexity	Space Complexity	Form
RODIMUS (He et al., 2024)	$\exp(-g_t \odot \tau_t)^\top \mathbf{1}$	$(g_t^{\tau_t})^\top \hat{\beta}_t$	$\mathcal{O}(NCd + Nd^2)$	$\mathcal{O}(Nd^2/C + Nd)$	chunkwise
Deltanet (Schlag et al., 2021a)	$\mathbf{I} - \beta_t k_t k_t^\top$	β_t	$\mathcal{O}(Nd^2)$	$\mathcal{O}(d^2)$	sequential
Deltanet2 (Yang et al., 2024c)	$\mathbf{I} - \beta_t k_t k_t^\top$	β_t	$\mathcal{O}(NCd + Nd^2)$	$\mathcal{O}(d)$	chunkwise
Mamba (Gu & Dao, 2023)	$\exp(\Delta A)$	$(\Delta A)^{-1}(\exp(\Delta A) - I) \cdot \Delta B$	$\mathcal{O}(Nd)$	$\mathcal{O}(Nd)$	parallel scan
gDeltanet (Yang et al., 2024b)	$\alpha_t(\mathbf{I} - \beta_t k_t k_t^\top)$	β_t	$\mathcal{O}(NCd + Nd^2)$	$\mathcal{O}(d)$	chunkwise
Mamba2 (Dao & Gu, 2024)	A_t	$\mathbf{1}$	$\mathcal{O}(Nd^2)$	$\mathcal{O}(Nd)$	chunkwise

[1] The blue gates are input-dependent and the black gates are input-independent.
[2] $\mathbf{1}$ in the table represents a $1 \times d$ vector of all ones.
[3] I in the table represents a $d \times d$ identity matrix.

complexity is $\mathcal{O}(Nd^2)$, the space complexity is $\mathcal{O}(d)$. All other methods use the recurrent form for inference, with time complexity of $\mathcal{O}(Nd^2)$ and space complexity of $\mathcal{O}(Nd)$.

▶ **RODIMUS.** RODIMUS (He et al., 2024) aims to solve the attention dilution problem, which is common in linear attention methods that compress historical information into a fixed-size state using a constant decay factor. This can lead to the loss of crucial long-range dependencies. The RODIMUS can be formulated as:

$$g_t = \zeta(x_t W_g + b_g) \tag{81}$$
$$\tau_t = \sigma(x_t W_\tau + b_\tau) \tag{82}$$
$$H_t = \left(\exp(-g_t \odot \tau_t)^\top \mathbf{1}_m\right) \odot H_{t-1} + \left((g_t^{\tau_t})^\top \hat{\beta}_t\right) \odot (k_t^\top v_t), \tag{83}$$
$$o_t = q_t H_t + d_t \odot x'_t \tag{84}$$
$$x'_t = \text{short\_conv}(X)[t] \tag{85}$$

To enhance local interactions and non-linearity, input tokens are processed by a short convolution layer and transformed into x'_t. Input-dependent select gate g_t and temperature gate τ_t dynamically regulate information retention and forgetting. d_t is a learnable parameter that allows x'_t to directly influence the output. This special input-dependent gate design makes RODIMUS achieve significant accuracy compared with the previous linear attention models when using the same size hidden states. To improve efficiency, RODIMUS uses the recurrent form for inference and the chunkwise form for training.

▶ **DeltaNet.** The purely additive update rule ($H_t = H_{t-1} + v_t k_t^\top$) of standard linear attention struggles to edit or discard outdated information. This limitation can lead to "key collisions" when the sequence length surpasses the key dimension, degrading the model's associative recall capabilities. To resolve this, DeltaNet (Schlag et al., 2021a) replaces the additive update with the error-correcting delta rule, which updates H_t by minimizing the error between the predicted value ($H_{t-1}k_t$) and

the target value (v_t). This update rule implicitly combines forgetting and selection, and can be formulated in the following two forms:

$$H_t = H_{t-1} - \beta_t(H_{t-1}k_t - v_t)k_t^\top. \tag{86}$$

$$H_t = H_{t-1} - v_t^{\text{old}}k_t^\top + v_t^{\text{new}}k_t^\top. \tag{87}$$

Here, β_t is a learnable, input-dependent "writing strength". The influence of the old value ($v_t^{\text{old}} = H_{t-1}k_t$) is deleted from H_t and the influence of the new value ($v_t^{\text{new}} = v_t$) is written back. However, this training algorithm is inherently sequential, which makes it hardware-inefficient and difficult to scale on modern parallel processors like GPUs.

► **DeltaNet2.** DeltaNet2 (Yang et al., 2024c) was introduced to address the inefficiency caused by the sequential training of DeltaNet. The key insight is a memory-efficient reparameterization of the delta rule recurrence. By viewing the update as a generalized Householder transformation (Bischof & Van Loan, 1987), it becomes possible to use the compact WY representation (Bischof & Van Loan, 1987) to compute the cumulative state updates. This reparameterization enables a chunkwise form. The chunk-level recurrence is given by:

$$H_{[t+1]} = H_{[t]} + (U_{[t]} - W_{[t]}H_{[t]}^\top)^\top K_{[t]}. \tag{88}$$

$$O_{[t]} = Q_{[t]}H_{[t]}^\top + (Q_{[t]}K_{[t]}^\top \odot M_C)(U_{[t]} - W_{[t]}H_{[t]}^\top). \tag{89}$$

where $U_{[t]}$ and $W_{[t]}$ are pseudo-value matrices derived within the chunk. This algorithmic breakthrough makes it efficient to train DeltaNet more parallelizable at a large scale.

► **Mamba.** Mamba (Gu & Dao, 2023) enhances structured State Space Models (SSMs) (Gu et al., 2021) with a selection mechanism, making them content-aware. The Mamba and Transformer architectures are distinct. Consequently, Mamba does not employ the Query-Key-Value (QKV) mechanism. Instead, Mamba directly maps an input x_t to an output y_t. The core idea of Mamba is a time-varying linear recurrence that updates a hidden state h_t based on an input x_t:

$$h_t = \overline{\mathbf{A}}_t h_{t-1} + \overline{\mathbf{B}}_t x_t. \tag{90}$$

$$y_t = \mathbf{C}_t h_t. \tag{91}$$

$$\overline{\mathbf{A}}_t = \exp(\Delta_t \mathbf{A}). \tag{92}$$

$$\overline{\mathbf{B}}_t = (\Delta_t \mathbf{A})^{-1}(\exp(\Delta_t \mathbf{A}) - \mathbf{I}) \cdot \Delta_t \mathbf{B}_t. \tag{93}$$

A key innovation is that the state transition ($\overline{\mathbf{A}}_t$) and input projection ($\overline{\mathbf{B}}_t$) matrices are input-dependent and vary at each timestep, allowing the model to selectively focus on or ignore inputs. These discrete parameters are generated from continuous parameters through a discretization rule, enabling Mamba to dynamically manage information flow. For hardware-efficient implementation, Mamba uses a special form called parallel scan. This selection mechanism allows Mamba to dynamically adjust its focus on or ignore parts of the input sequence.

► **Mamba2.** Mamba2 (Dao & Gu, 2024) builds upon the original Mamba by introducing the structured State Space Duality (SSD) framework, which reveals a connection between SSMs and attention. The recurrent formulation of Mamba2 is a specialized version of the selective SSM: $h_t = \mathbf{A}_t h_{t-1} + \mathbf{B}_t x_t, y_t = \mathbf{C}_t h_t$. Mamba2 computes the SSM through a block decomposition of its matrix form. This method combines a quadratic, attention-like computation for diagonal blocks with a linear, recurrent computation for off-diagonal blocks, significantly improving training speed.

The projections for the SSM parameters $(\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t)$ and the input x_t are all computed in parallel at the beginning of the block. This is analogous to the Q, K, V projections in Transformers and facilitates more efficient tensor parallelism.

► **gDeltaNet.** gDeltaNet (Yang et al., 2024b) addresses the limitations of its predecessors—DeltaNet's (Schlag et al., 2021a) slow memory clearance and Mamba2's(Dao & Gu, 2024) uniform decay. The core idea is the gated delta rule, which introduces an input-dependent gate that results in the following state transition: $H_t = H_{t-1}(\alpha_t(I - \beta_t k_t k_t^\top)) + \beta_t v_t k_t^\top$. where β_t is the writing strength. The gating term $\alpha_t \in (0, 1)$ achieves more flexible memory control. By setting $\alpha_t \to 0$, the model can rapidly erase the prior state, which is crucial during context switches. By setting $\alpha_t \to 1$, the update effectively becomes the pure delta rule, enabling precise modification of a single key-value association while preserving all other information. This combination of adaptive memory clearance and precise, targeted updates enables gDeltaNet to achieve a more robust memory management system.

7.7 Test Time Training

LLM can compress a massive training set into weights through self-supervised learning (Zhao et al., 2023), and what linear attention does can also be viewed as compressing context into a fixed-size hidden state. Inspired by this observation, Test Time Training (TTT) (Sun et al., 2024) views the hidden state H_t as a set of online updated parameters to compress the context. The gradient of the H_t, is computed as follow:

$$g = \sum_{i=1}^{C} \nabla \mathcal{L}(f_{H_{t-1}}(k_t), v_t). \tag{94}$$

Then the update rule of H_t is

$$H_t = H_{t-1} - g. \tag{95}$$

The attention output o_t can be finally computed as: $o_t = f_{H_t}(q_t)$. Here $f_{H_{t-1}}$ and f_{H_t} are networks with H_{t-1} and H_t as their parameters respectively, $\mathcal{L}$ is a loss function between $f_{H_{t-1}}(k_t)$ and value v_t, commonly Mean Square Error, C is the chunk size.

► **TTT** . TTT (Sun et al., 2024) uses the Equation. 94 and 95 to update hiddden states. TTT develops two forms of f_{H_t}, TTT-Linear and TTT-MLP. The f_{H_t} of TTT-MLP is a two-layer MLP with a nonlinear activation GELU (Hendrycks & Gimpel, 2016) similar to Transformer. The nonlinear activation strengthens the expressive ability of TTT, but it makes TTT-MLP cannot be paralleled on a GPU, thus infeasible in reality. The f_{H_t} of TTT-linear is a simple linear transformation: $f_{H_{t-1}}(k_t) = H_{t-1}k_t$ and can be efficiently parallelized through chunk-wise algorithms proposed in GLA (Yang et al., 2023). The time and space complexity of TTT-Linear aligns with chunk-wise linear attention methods.

► **Titans** . Titans (Behrouz et al., 2024) shares the same motivation as test time training. Different from Equation 95, Titans (Behrouz et al., 2024) further introduces momentum into the update rule of H_t: $H_t = (1 - \alpha_t)H_{t-1} + M_t, M_t = \eta_t M_{t-1} - \theta_t g$. Here $\alpha_t \in [0, 1]$ is forget gate, η_t is the weight decay and θ_t is the learning rate. Titans keeps the f_{H_t} a simple linear transformation as TTT-Linear. Titans outperforms both full attention and modern linear recurrent models across

tasks, including language modeling, common-sense reasoning, genomics, time-series forecasting, and needle-in-a-haystack retrieval.

▶ **LaCT** . LaCT (Zhang et al., 2025g) proposes a hardware-friendly method to make the test-time training method more efficient on a GPU. TTT (Sun et al., 2024) and Titans (Behrouz et al., 2024) both take a small chunk to update fast weights, i.e., every 16 to 64 tokens, resulting in low utilization of the GPU. LaCT (Zhang et al., 2025g) introduces a much larger chunk size of 4096 $\sim$ 1 million tokens and make TTT more hardware efficient. Large chunk size enables LaCT to use a more complex design of f_H and update rule. LaCT uses SwiGLU-MLP (Shazeer, 2020) as the sub network f_H, which consists of three weight matrix $W = W_1, W_2, W_3$, and the output is computed as: $f_{H_{t-1}}(k_t) = W_2[\text{SiLU}(W_1 k_t) \odot W_3 k_t]$. The loss function is dot product loss: $\mathcal{L}(f_{H_{t-1}}(k_t), v_T) = -f_{H_{t-1}}(k_t)^T v_t$. Naive update Equation 95 suffers from magnitude explosion because of the accumulated memory of a large chunk. To improve stability and effectiveness, LaCT develops a more robust weight update rule: $H_t = \text{L2-Normalization}(H_{t-1} - \text{Muon}(g))$ where g is the gradient and Muon is a nonlinear optimizer (Jordan et al.). LaCT outperforms other modern linear attention in various language modeling tasks.

8 Conclusion

In this survey, we have provided a systematic and comprehensive review of efficient attention methods, categorizing them into hardware-efficient, sparse, compact, and linear attention methods. Each category targets different aspects of the attention bottleneck, i.e., whether by optimizing low-level I/O and computation, exploiting sparsity, reducing KV cache memory usage, or reformulating the attention to achieve sub-quadratic complexity.

References

Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

Saurabh Agarwal, Bilge Acun, Basil Hosmer, Mostafa Elhoushi, Yejin Lee, Shivaram Venkataraman, Dimitris Papailiopoulos, and Carole-Jean Wu. Chai: clustered head attention for efficient llm inference. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models, 2023. URL `https://arxiv.org/abs/2311.16867`.

Arturs Backurs, Piotr Indyk, and Tal Wagner. Space and time efficient kernel density estimation in high dimensions. *Advances in neural information processing systems*, 32, 2019.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL `https://arxiv.org/abs/1409.0473`.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Christian Bischof and Charles Van Loan. The wy representation for products of householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.

Bobbi Jo Broxson. The kronecker product. 2006.

Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X Sean Wang. Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search. *Proc. VLDB Endow.*, 2024a.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179*, 2024b.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019. URL `https://arxiv.org/abs/1904.10509`.

François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 1251–1258, 2017.

Yuhong Chou, M. Yao, K. Wang, Y. Pan, R.J. Zhu, J. Wu, Y. Zhong, Y. Qiao, B. Xu, and G. Li. Metala: Unified optimal linear approximation to softmax attention map. *Advances in Neural Information Processing Systems*, 37:71034–71067, 2024.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. `https://crfm.stanford.edu/2023/10/12/flashdecoding.html`, 2023. [Online].

Jyotikrishna Dass, Shang Wu, Huihong Shi, Chaojian Li, Zhifan Ye, Zhongfeng Wang, and Yingyan Lin. Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 415–428. IEEE, 2023.

Yichuan Deng, Zhao Song, and Chiwun Yang. Attention is naturally sparse with gaussian distributed input. *arXiv preprint arXiv:2404.02690*, 2024.

Aditya Desai, Shuo Yang, Alejandro Cuadron, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Hashattention: Semantic sparsity for faster inference, 2025. URL `https://arxiv.org/abs/2412.14468`.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994. doi: 10.1002/spe.4380240306.

Tianyu Fu, Haofeng Huang, Xuefei Ning, Genghan Zhang, Boju Chen, Tianqi Wu, Hongyi Wang, Zixiao Huang, Shiyao Li, Shengen Yan, et al. Moa: Mixture of sparse attention for automatic large language model compression. *arXiv preprint arXiv:2406.14909*, 2024.

Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. Seerattention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276*, 2024.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Han Guo, Songlin Yang, Tarushii Goel, Eric P. Xing, Tri Dao, and Yoon Kim. Log-linear attention. *arXiv preprint arXiv:2506.04761*, June 2025. doi: 10.48550/arXiv.2506.04761. URL `https://arxiv.org/abs/2506.04761`. Submitted on 5 Jun 2025 (v1), last revised 25 Jun 2025 (v2).

Jialong Guo, Xinghao Chen, Yehui Tang, and Yunhe Wang. Slab: Efficient transformers with simplified linear attention and progressive re-parameterized batch normalization. *arXiv preprint arXiv:2405.11582*, 2024.

Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence*, 45(1):87–110, 2022.

Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Neighborhood attention transformer. In *Proceedings of CVPR*, pp. 6185–6194, June 2023a.

Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Natten: Neighborhood attention extension. `https://github.com/SHI-Labs/NATTEN`, 2023b.

Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

Zhihao He, Hang Yu, Zi Gong, Shizhan Liu, Jianguo Li, and Weiyao Lin. Rodimus*: Breaking the accuracy-efficiency trade-off with efficient attentions. *arXiv preprint arXiv:2410.06577*, 2024.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. *arXiv preprint arXiv:2207.12598*, 2022.

Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J Fleet. Video diffusion models. *arXiv:2204.03458*, 2022.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. *arXiv e-prints*, 2024.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Jingcheng Hu, Houyi Li, Yinmin Zhang, Zili Wang, Shuigeng Zhou, Xiangyu Zhang, Heung-Yeung Shum, and Daxin Jiang. Multi-matrix factorization attention. *arXiv preprint arXiv:2412.19255*, 2024.

Ziqi Huang, Yinan He, Jiashuo Yu, Fan Zhang, Chenyang Si, Yuming Jiang, Yuanhan Zhang, Tianxing Wu, Qingyang Jin, Nattapol Chanpaisit, et al. Vbench: Comprehensive benchmark suite for video generative models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21807–21818, 2024.

Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *ArXiv*, abs/2310.06825, 2023. URL `https://arxiv.org/abs/2310.06825`.

Huiqiang Jiang, YUCHENG LI, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cecista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. *URL https://kellerjordan. github. io/posts/muon*, 6.

James M Joyce. Kullback-leibler divergence. In *International encyclopedia of statistical science*, pp. 720–722. Springer, 2011.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. *Advances in Neural Information Processing Systems*, 36:24892–24928, 2023.

Mikhail V Koroteev. Bert: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*, 2021.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. *arXiv e-prints*, 2023.

Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=OfjIlbelrT`.

Xingyang Li, Muyang Li, Tianle Cai, Haocheng Xi, Shuo Yang, Yujun Lin, Lvmin Zhang, Songlin Yang, Jinbo Hu, Kelly Peng, Maneesh Agrawala, Ion Stoica, Kurt Keutzer, and Song Han. Radial attention: $\mathcal{O}(n \log n)$ sparse attention with energy decay for long video generation. *arXiv preprint arXiv:2506.19852*, 2025.

Chaofan Lin, Jiaming Tang, Shuo Yang, Hanshuo Wang, Tian Tang, Boyu Tian, Ion Stoica, Song Han, and Mingyu Gao. Twilight: Adaptive attention sparsity with hierarchical top-p pruning. *arXiv preprint arXiv:2502.02770*, 2025.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024a.

Akide Liu, Zeyu Zhang, Zhexin Li, Xuehai Bai, Yizeng Han, Jiasheng Tang, Yuanjie Xing, Jichao Wu, Mingyang Yang, Weihua Chen, Jiahao He, Yuanyu He, Fan Wang, Gholamreza Haffari, and Bohan Zhuang. Fpsattention: Training-aware fp8 and sparsity co-design for fast video diffusion, 2025. URL `https://arxiv.org/abs/2506.04648`.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024b.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

Zicheng Liu, Siyuan Li, Li Wang, Zedong Wang, Yunfan Liu, and Stan Z Li. Short-long convolutions help hardware-efficient linear attention to focus on long sequences. *arXiv preprint arXiv:2406.08128*, 2024c.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. In *International Conference on Machine Learning*, pp. 32332–32344. PMLR, 2024d.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. Moba: Mixture of block attention for long-context llms. *arXiv preprint arXiv:2502.13189*, 2025.

Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

Larry R Medsker, Lakhmi Jain, et al. Recurrent neural networks. *Design and applications*, 5(64-67): 2, 2001.

María Luisa Menéndez, Julio Angel Pardo, Leandro Pardo, and María del C Pardo. The jensen-shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318, 1997.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, 2010. URL `https://api.semanticscholar.org/CorpusID:15539264`.

NVIDIA. Nvidia ampere architecture whitepaper. Technical report, NVIDIA. URL `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`.

NVIDIA. Nvidia h100 tensor core gpu architecture, 2022.

NVIDIA. Cutlass: Cuda templates for linear algebra subroutines and solvers, 2025. URL `https://github.com/NVIDIA/cutlass`. Accessed: 2025-08-17.

NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Architecture (Ampere Architecture Whitepaper). `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020.

NVIDIA Corporation. CUDA C+/+ Programming Guide (Release 13.0). https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Aug 2025.

William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Xingjian Du, Teddy Ferdinan, Haowen Hou, et al. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence. *arXiv preprint arXiv:2404.05892*, 2024.

Bo Peng, Ruichong Zhang, Daniel Goldstein, Eric Alcaide, Xingjian Du, Haowen Hou, Jiaju Lin, Jiaxing Liu, Janna Lu, William Merrill, et al. Rwkv-7" goose" with expressive dynamic state evolution. *arXiv preprint arXiv:2503.14456*, 2025.

Zhen Qin, Songlin Yang, Weihan Sun, and Yuxuan Zhong. Hierarchically gated recurrent neural network for sequence modeling. *Advances in Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=PITCHk_w_B.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Various lengths, constant speed: Efficient language modeling with lightning attention. *arXiv preprint arXiv:2405.17381*, 2024.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient LLM inference. In *Forty-first International Conference on Machine Learning*, 2024.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, pp. 9355–9366. PMLR, 2021a.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International conference on machine learning*, pp. 9355–9366. PMLR, 2021b.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Xuan Shen, Chenxia Han, Yufa Zhou, Yanyue Xie, Yifan Gong, Quanyi Wang, Yiwei Wang, Yanzhi Wang, Pu Zhao, and Jiuxiang Gu. Draftattention: Fast video diffusion via low-resolution attention guidance, 2025. URL https://arxiv.org/abs/2505.14708.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. Loki: Low-rank keys for efficient sparse attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

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

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

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yue Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: query-aware sparsity for efficient long-context llm inference. ICML'24. JMLR.org, 2024.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

MiniCPM Team, Chaojun Xiao, Yuxuan Li, Xu Han, Yuzhuo Bai, Jie Cai, Haotian Chen, Wentong Chen, Xin Cong, Ganqu Cui, et al. Minicpm4: Ultra-efficient llms on end devices. *arXiv preprint arXiv:2506.07900*, 2025.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jianzong Wu, Liang Hou, Haotian Yang, Xin Tao, Ye Tian, Pengfei Wan, Di Zhang, , and Yunhai Tong. Vmoba: Mixture-of-block attention for video diffusion models. *arXiv preprint arXiv:2506.23858*, 2025.

Haocheng Xi, Shuo Yang, Yilong Zhao, Chenfeng Xu, Muyang Li, Xiuyu Li, Yujun Lin, Han Cai, Jintao Zhang, Dacheng Li, et al. Sparse videogen: Accelerating video diffusion transformers with spatial-temporal sparsity. *arXiv preprint arXiv:2502.01776*, 2025.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 119638–119661. Curran Associates, Inc., 2024a. URL `https://proceedings.neurips.cc/paper_files/paper/2024/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf`.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024b.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. In *The International Conference on Learning Representations*, 2025.

Enze Xie, Junsong Chen, Junyu Chen, Han Cai, Haotian Tang, Yujun Lin, Zhekai Zhang, Muyang Li, Ligeng Zhu, Yao Lu, et al. Sana: Efficient high-resolution text-to-image synthesis with linear diffusion transformers. In *The Thirteenth International Conference on Learning Representations*, 2025.

Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. XAttention: Block sparse attention with antidiagonal scoring. In *Forty-second International Conference on Machine Learning*, 2025.

Yixing Xu, Chao Li, Dong Li, Xiao Sheng, Fan Jiang, Lu Tian, and Emad Barsoum. Qt-vit: Improving linear attention in vit with quadratic taylor expansion. *Advances in Neural Information Processing Systems*, 37:83048–83067, 2024.

Lijie Yang, Zhihao Zhang, Zhuofu Chen, Zikun Li, and Zhihao Jia. Tidaldecode: Fast and accurate llm decoding with position persistent sparse attention. *arXiv preprint arXiv:2410.05076*, 2024a.

Lijie Yang, Zhihao Zhang, Arti Jain, Shijie Cao, Baihong Yuan, Yiwei Chen, Zhihao Jia, and Ravi Netravali. Less is more: Training-free sparse attention with global locality for efficient reasoning. *arXiv preprint arXiv:2508.07101*, 2025a.

Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Chenfeng Xu, Kelly Peng, et al. Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation. *arXiv preprint arXiv:2505.18875*, 2025b.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. *arXiv preprint arXiv:2412.06464*, 2024b.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *arXiv preprint arXiv:2406.06484*, 2024c.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. 2025. URL `https://api.semanticscholar.org/CorpusID:276408911`.

Zhihang Yuan, Hanling Zhang, Lu Pu, Xuefei Ning, Linfeng Zhang, Tianchen Zhao, Shengen Yan, Guohao Dai, and Yu Wang. DiTFastattn: Attention compression for diffusion transformer models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Jintao Zhang, Pengle Zhang, Jun Zhu, Jianfei Chen, et al. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. In *The Thirteenth International Conference on Learning Representations*, a.

Jintao Zhang, Haofeng Huang, Pengle Zhang, Jia Wei, Jun Zhu, and Jianfei Chen. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread int4 quantization. *arXiv preprint arXiv:2411.10958*, 2024a.

Jintao Zhang, Haoxu Wang, Kai Jiang, Shuo Yang, Kaiwen Zheng, Haocheng Xi, Ziteng Wang, Hongzhou Zhu, Min Zhao, Ion Stoica, et al. Sla: Beyond sparsity in diffusion transformers via fine-tunable sparse-linear attention. *arXiv preprint arXiv:2509.24006*, 2025a.

Jintao Zhang, Jia Wei, Pengle Zhang, Xiaoming Xu, Haofeng Huang, Haoxu Wang, Kai Jiang, Jun Zhu, and Jianfei Chen. Sageattention3: Microscaling fp4 attention for inference and an exploration of 8-bit training. *arXiv preprint arXiv:2505.11594*, 2025b.

Jintao Zhang, Chendong Xiang, Haofeng Huang, Jia Wei, Haocheng Xi, Jun Zhu, and Jianfei Chen. Spargeattn: Accurate sparse attention accelerating any model inference. *arXiv preprint arXiv:2502.18137*, 2025c.

Jintao Zhang, Xiaoming Xu, Jia Wei, Haofeng Huang, Pengle Zhang, Chendong Xiang, Jun Zhu, and Jianfei Chen. Sageattention2++: A more efficient implementation of sageattention2. *arXiv preprint arXiv:2505.21136*, 2025d.

Peiyuan Zhang, Yongqi Chen, Runlong Su, Hangliang Ding, Ion Stoica, Zhengzhong Liu, and Hao Zhang. Fast video generation with sliding tile attention. *arXiv preprint arXiv:2502.04507*, 2025e.

Peiyuan Zhang, Haofeng Huang, Yongqi Chen, Will Lin, Zhengzhong Liu, Ion Stoica, Eric Xing, and Hao Zhang. Vsa: Faster video diffusion with trainable sparse attention. *arXiv preprint arXiv:2505.13389*, 2025f.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Tianyuan Zhang, Sai Bi, Yicong Hong, Kai Zhang, Fujun Luan, Songlin Yang, Kalyan Sunkavalli, William T Freeman, and Hao Tan. Test-time training done right. *arXiv preprint arXiv:2505.23884*, 2025g.

Yifan Zhang, Yifeng Liu, Huizhuo Yuan, Zhen Qin, Yang Yuan, Quanquan Gu, and Andrew C Yao. Tensor product attention is all you need. *arXiv preprint arXiv:2501.06425*, 2025h.

Yu Zhang, Songlin Yang, Ruijie Zhu, Yue Zhang, Leyang Cui, Yiqiao Wang, Bolun Wang, Freda Shi, Bailin Wang, Wei Bi, et al. Gated slot attention for efficient linear-time sequence modeling. 2024. *URL https://api. semanticscholar. org/CorpusID*, 272593079, b.

Yu Zhang, Songlin Yang, Rui-Jie Zhu, Yue Zhang, Leyang Cui, Yiqiao Wang, Bolun Wang, Freda Shi, Bailin Wang, Wei Bi, et al. Gated slot attention for efficient linear-time sequence modeling. *Advances in Neural Information Processing Systems*, 37:116870–116898, 2024b.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Tianchen Zhao, Ke Hong, Xinhao Yang, Xuefeng Xiao, Huixia Li, Feng Ling, Ruiqi Xie, Siqi Chen, Hongyu Zhu, Yichong Zhang, et al. Paroattention: Pattern-aware reordering for efficient sparse and quantized attention in visual generation models. *arXiv preprint arXiv:2506.16054*, 2025.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37: 62557–62583, 2024.

Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, et al. Pit: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 331–347, 2023.

Qianchao Zhu, Jiangfei Duan, Chang Chen, Siran Liu, Xiuhong Li, Guanyu Feng, Xin Lv, Huanqi Cao, Xiao Chuanfu, Xingcheng Zhang, Dahua Lin, and Chao Yang. Sampleattention: Near-lossless acceleration of long context llm inference with adaptive structured sparse attention, 2024. URL `https://arxiv.org/abs/2406.15486`.

A Appendix - Linear Attentions with Gates

The recurrent form of linear attention with gates can be expressed as:

$$\begin{aligned} H_t &= G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot k_t^T v_t \\ o_t &= q_t H_t \end{aligned}$$

And usually the gates can be decomposed into the outer product of two vectors, as Equation 96 shows, we also rewrite it here:

$$\begin{aligned} G_f^{(t)} &= a_t^T b_t, \quad a_t, b_t \in \mathbb{R}^{1\times d} \\ G_s^{(t)} &= \hat{a}_t^T \hat{b}_t, \quad \hat{a}_t, \hat{b}_t \in \mathbb{R}^{1\times d} \end{aligned} \tag{96}$$

Then H_t can be expressed as $H_t = G_f^{(t)} \odot H_{t-1} + (\hat{a}_t \odot k_t)^T (\hat{b}_t \odot v_t)$.

A.1 Derivation of Quadratic Parallel Form with Gates

► Unrolling the Recurrence

By repeatedly substituting the definition of H_{t-1}, we can express H_t as a sum over all previous time steps. Let the state decay term be $\gamma_j = a_j^T b_j$ and the state update term be $U_i = (\hat{a}_i \odot k_i)^\top (\hat{b}_i \odot v_i)$. The recurrence $H_t = \gamma_t \odot S_{t-1} + U_t$ expands to:

$$H_t = \sum_{i=1}^{t} \left[\left(\prod_{j=i+1}^{t} \gamma_j \right) \odot U_i \right] \tag{97}$$

Substituting the original terms back gives the explicit form for H_t:

$$\begin{aligned} H_t &= \sum_{i=1}^{t} \left[\left(\prod_{j=i+1}^{t} (a_j^T b_j) \right) \odot \left((\hat{a}_i \odot k_i)^\top (\hat{b}_i \odot v_i) \right) \right] \\ &= \sum_{i=1}^{t} \left[\left((\prod_{j=i+1}^{t} a_j)^\top \odot (\hat{a}_i \odot k_i)^\top \right) \left((\prod_{j=i+1}^{t} b_j) \odot (\hat{b}_i \odot v_i) \right) \right] \\ &= \sum_{i=1}^{t} \left[\left((\prod_{j=i+1}^{t} a_j \odot \hat{a}_i \odot k_i)^\top \right) \left((\prod_{j=i+1}^{t} b_j \odot \hat{b}_i \odot v_i) \right) \right] \end{aligned} \tag{98}$$

► Deriving the Parallel Output Equation

We now substitute the unrolled H_t into the output equation $o_t = q_t^\top H_t$.We can rewrite o_t as:

$$o_t = q_t \sum_{i=1}^{t} \left(\left(\prod_{j=i+1}^{t} a_j \odot k_i \odot \hat{a}_i \right)^\top \left(\prod_{j=i+1}^{t} b_j \odot v_i \odot \hat{b}_i \right) \right) \tag{99}$$

To facilitate parallelization, we define the cumulative products, which can be computed efficiently using a prefix scan:

$$\mathcal{A}_t = \prod_{j=1}^{t} a_j \quad \text{and} \quad \Lambda_t = \prod_{j=1}^{t} b_j \tag{100}$$

This allows us to express the inner products as $\prod_{j=i+1}^{t} a_j = \mathcal{A}_t/\mathcal{A}_i$ and $\prod_{j=i+1}^{t} b_j = \Lambda_t/\Lambda_i$. Substituting these into the equation for o_t and rearranging terms yields:

$$\begin{aligned} o_t &= q_t \sum_{i=1}^{t} \left(\frac{\mathcal{A}_t}{\mathcal{A}_i} \odot \hat{a}_i \odot k_i\right)^T \left(\frac{\mathbf{\Lambda}_t}{\mathbf{\Lambda}_i} \odot \hat{b}_i \odot v_i\right) \\ &= (q_t \odot \mathcal{A}_t) \sum_{i=1}^{t} \left[(\hat{a}_i \odot k_i/\mathcal{A}_i)^\top (\hat{b}_i \odot v_i/\mathbf{\Lambda}_i)\right] \odot \mathbf{\Lambda}_t \end{aligned} \tag{101}$$

This form isolates the dependencies and prepares the expression for matrix representation.

► Final Matrix Formulation

We can now express the entire computation for a sequence of length N using matrix operations. First, we stack the time-series vectors into matrices $Q, K, V, \hat{A}, \hat{B} \in \mathbb{R}^{L\times d}$, and the cumulative products into $\mathcal{A}, \Lambda \in \mathbb{R}^{N\times d}$.

Defining the transformed query, key, and value matrices as:

$$\begin{aligned} \tilde{Q} &= Q \odot \mathcal{A} \\ \tilde{K} &= (K \odot \hat{A}) \oslash \mathcal{A} \\ \tilde{V} &= (V \odot \hat{B}) \oslash \mathbf{\Lambda} \end{aligned} \tag{102}$$

where $\oslash$ denotes element-wise division.

The summation over $i = 1, \ldots, t$ corresponds to a masked matrix multiplication. Let M be a causal mask matrix where $M_{ti} = 1$ if $i \le t$ and 0 otherwise. The full output sequence $O \in \mathbb{R}^{N\times d}$ can be computed in parallel as:

$$O = \left(((\tilde{Q}\tilde{K}^\top) \odot M)\tilde{V}\right) \odot \Lambda \tag{103}$$

This final equation is fully parallelizable, removing the sequential bottleneck of the original recurrent formulation.

A.2 Derivation of Chunkwise Form with Gates

Let $H_{[i]} \in \mathbb{R}^{d\times d}$ denote the hidden state after processing the i-th chunk (i.e., $H_{[i]} = H_{iC}$), and $H_{[0]} = 0$. And we will use $\mathcal{A}, \Lambda, \tilde{Q}, \tilde{K}$ and $\tilde{V}$ defined in Appendix A.1

► Decomposition of the State

To derive the state H_{iC+j} for the j-th token in the $(i+1)$-th chunk, we begin with the state at the previous chunk boundary, $H_{[i]}$. By recursively applying the update rule for the j steps within the current chunk (from token $iC+1$ to $iC+j$), we can expand the expression for H_{iC+j} as follows:

$$H_{iC+j} = \underbrace{\left(\prod_{m=iC+1}^{iC+j} a_m^\top b_m\right) \odot H_{[i]}}_{\text{Inter-chunk component}} + \underbrace{\sum_{m=iC+1}^{iC+j} \left(\left(\prod_{n=m+1}^{iC+j} a_n^\top b_n\right) \odot \left((\hat{a}_m \odot k_m)^\top (\hat{b}_m \odot v_m)\right)\right)}_{\text{Intra-chunk component}} \tag{104}$$

This decomposition separates the output $o_{iC+j} = q_{iC+j} H_{iC+j}$ into an inter-chunk and an intra-chunk component.

► Chunk-level Computation and State Update

Intra-Chunk Computation The intra-chunk part of o_{iC+j} can be expressed as Equation 105.

$$\begin{aligned} o_{iC+j}^{Intra} &= q_{iC+j} \sum_{m=iC+1}^{iC+j} \left[\left(\prod_{n=m+1}^{iC+j} a_n^\top b_n \right) \odot \left((\hat{a}_n \odot k_n)^\top (\hat{b}_n \odot v_n) \right) \right] \\ &= q_{iC+j} \sum_{m=iC+1}^{iC+j} \left[\left(\prod_{n=m+1}^{iC+j} a_n \odot \hat{a}_n \odot k_n \right)^\top \left(\prod_{n=m+1}^{iC+j} b_n \odot \hat{b}_n \odot v_n \right) \right] \end{aligned} \tag{105}$$

Let $O_{[i+1]}$, $Q_{[i+1]}$, etc., be the sub-matrices corresponding to the $(i+1)$-th chunk. According to Appendix A.1, the intra-chunk output is:

$$O_{[i+1]}^{\text{Intra}} = \left(((\tilde{Q}_{[i+1]} \tilde{K}_{[i+1]}^\top) \odot M_C) \tilde{V}_{[i+1]} \right) \odot \Lambda_{[i+1]} \tag{106}$$

where M_C is a causal mask of size $C \times C$, and all $\Lambda, \tilde{Q}, \tilde{K}, \tilde{V}$ are same as defined in Equation 100 and Equation 102.

Inter-Chunk Computation The inter-chunk part of o_{iC+j} can be expressed as follows:

$$\begin{aligned} o_{iC+j}^{\text{Inter}} &= q_{iC+j} \left[\left(\prod_{m=iC+1}^{iC+j} (a_m^\top b_m) \right) \odot H_{[i]} \right] \\ &= \left[(q_{iC+j} \odot \prod_{m=iC+1}^{iC+j} a_m) H_{[i]} \right] \odot (\prod_{m=iC+1}^{iC+j} b_m) \end{aligned} \tag{107}$$

Let $(A_{[i+1]}^\dagger)_j = \prod_{m=1}^{j} a_{iC+m}$ and $(B_{[i+1]}^\dagger)_j = \prod_{m=1}^{j} b_{iC+m}$ be the local cumulative products within the chunk, which can be futher expressed as Equation 108.

$$(A_{[i+1]}^\dagger)_j = \frac{\mathcal{A}_{iC+j}}{\mathcal{A}_{iC}} \in \mathbb{R}^{1\times d}, \quad (B_{[i+1]}^\dagger)_j = \frac{\Lambda_{iC+j}}{\Lambda_{iC}} \in \mathbb{R}^{1\times d}, \quad j = 1, \ldots, C \tag{108}$$

Then the inter-chunk outputs for the entire chunk can be shown as Equation 109:

$$O_{[i+1]}^{\text{Inter}} = \left[\left(Q_{[i+1]} \odot A_{[i+1]}^\dagger \right) H_{[i]} \right] \odot B_{[i+1]}^\dagger \tag{109}$$

Chunk-to-Chunk State Update To process the next chunk, we must compute the new hidden state $H_{[i+1]} = H_{(i+1)C}$. Using the Equation 104 again with $j = C$:

$$\begin{aligned} H_{[i+1]} &= (\prod_{m=iC+1}^{(i+1)C} a_m^\top b_m) \odot H_{[i]} + \sum_{m=iC+1}^{(i+1)C} \left[(\prod_{n=m+1}^{(i+1)C} a_n^\top b_n) \odot \left((\hat{a}_m \odot k_m)^\top (\hat{b}_m \odot v_m) \right) \right] \\ &= (\prod_{m=iC+1}^{(i+1)C} a_m^\top b_m) \odot H_{[i]} + \sum_{m=iC+1}^{(i+1)C} \left[(\prod_{n=m+1}^{(i+1)C} a_n \odot \hat{a}_m \odot k_m)^\top (\prod_{n=m+1}^{(i+1)C} b_n \odot \hat{b}_m \odot v_m) \right] \end{aligned}$$

Using $\mathcal{A}$ and Λ defined in Equation 100, then

$$H_{[i+1]} = \left[(\frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_{iC}})^T (\frac{\Lambda_{(i+1)C}}{\Lambda_{iC}}) \right] \odot H_{[i]} + \sum_{m=iC+1}^{(i+1)C} \left[(\frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_m} \odot \hat{a}_m \odot k_m)^\top (\frac{\Lambda_{(i+1)C}}{\Lambda_m} \odot \hat{b}_m \odot v_m) \right]$$

Let $\hat{A}, \hat{B}$ be the matrices formed by stacking the time-series vectors $\hat{a}_t, \hat{b}_t$ (same as in Appendix A.2) and let:

$$(\mathcal{A}'_{[i]})_j = \frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_{iC+j}} \quad (\mathcal{B}'_{[i]})_j = \frac{\Lambda_{(i+1)C}}{\Lambda_{iC+j}} \in \mathbb{R}^{1\times d}, \quad j = 0, 1, \ldots, C-1 \tag{110}$$

Then:

$$\begin{aligned} H_{[i+1]} &= \left[(\frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_{iC}})^T (\frac{\Lambda_{(i+1)C}}{\Lambda_{iC}}) \right] \odot H_{[i]} + \sum_{m=iC+1}^{(i+1)C} \left[((\mathcal{A}'_{[i]})_m \odot \hat{a}_m \odot k_m)^\top ((\mathcal{B}'_{[i]})_m \odot \hat{b}_m \odot v_m) \right] \\ &= \left[(\frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_{iC}})^T (\frac{\Lambda_{(i+1)C}}{\Lambda_{iC}}) \right] \odot H_{[i]} + (\mathcal{A}'_{[i]} \odot K_{[i]} \odot \hat{A}_{[i]})^\top (\mathcal{B}'_{[i]} \odot V_{[i]} \odot \hat{B}_{[i]}) \end{aligned} \tag{111}$$

Let

$$\begin{aligned} \zeta_i &= \left(\frac{\mathcal{A}_{(i+1)C}}{\mathcal{A}_{iC}} \right)^T \left(\frac{\Lambda_{(i+1)C}}{\Lambda_{iC}} \right) \\ \widehat{K}_{[i]} &= \mathcal{A}'_{[i]} \odot K_{[i]} \odot \hat{A}_{[i]} \\ \widehat{V}_{[i]} &= \mathcal{B}'_{[i]} \odot V_{[i]} \odot \hat{B}_{[i]} \end{aligned} \tag{112}$$

The chuk-level state update can be further simplified to

$$H_{[i+1]} = \zeta_i \odot S_{[i-1]} + \widehat{K}^T_{[i]} \widehat{V}_{[i]} \tag{113}$$

► Final Matrix Formulation

Finally, the complete chunkwise algorithm for the $(i+1)$-th chunk is summarized as follows:

$$\begin{aligned} O_{[i+1]} &= O^{\text{Intra}}_{[i+1]} + O^{\text{Inter}}_{[i+1]} \\ O^{\text{Intra}}_{[i+1]} &= (\tilde{Q}_{[i+1]} \tilde{K}^T_{[i+1]} \odot M) \tilde{V}_{[i+1]} \odot \Lambda_{[i+1]} \\ O^{\text{Inter}}_{[i+1]} &= (Q_{[i+1]} \odot A^\dagger_{[i+1]}) H_{[i]} \odot B^\dagger_{[i]} \\ H_{[i]} &= (\zeta_i \odot H_{[i-1]}) + \widehat{K}^T_{[i]} \widehat{V}_{[i]} \end{aligned}$$