# Efficient Attention Methods: Hardware efficient, Sparse, Compact, and Linear Attention

Jintao Zhang, Rundong Su, Chunyu Liu, Jia Wei, Ziteng Wang, Haoxu Wang, Pengle Zhang, Huiqiang Jiang, Haofeng Huang, Chendong Xiang, Haocheng Xi, Shuo Yang, Xingyang Li, Yilong Zhao, Yuezhou Hu, Tianyu Fu, Tianchen Zhao, Yicheng Zhang, Boqun Cao, Youhe Jiang, Kai Jiang, Huayu Chen, Min Zhao, Xiaoming Xu, Yi Wu, Fan Bao, Ion Stoica, Joseph E. Gonzalez, Jianfei Chen, and Jun Zhu, *Fellow*, IEEE

***Abstract*—In modern transformer architectures, 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 $N$ is the sequence length. Recently, a plethora of methods have been proposed to improve the computational efficiency of the attention operation, which are very critical for developing large-scale models. In this paper, we present a unified taxonomy and a comprehensive survey of efficient attention methods. We categorize existing approaches into four classes: (1) Hardware-efficient attention: Optimizes attention computation efficiency by leveraging hardware characteristics; (2) Sparse attention: Selectively performs a subset of computations in attention while omitting others; (3) Compact attention: Compresses the KV cache (e.g., via weight sharing or low-rank decomposition) without changing the computation cost of using a full-sized KV cache; and (4) Linear attention: Reformulate the attention computation to achieve $\mathcal{O}(N)$ time complexity. For each category, we also develop a unified analysis framework and offer perspectives on open challenges and promising directions for future research on efficient attention. To the best of our knowledge, this survey is the most comprehensive survey on efficient attention. Our taxonomy of hardware-efficient, compact, sparse, and linear attention covers nearly all existing efficient attention methods, which prior surveys do not.**

***Index Terms*—Attention mechanisms, Transformers.**

## I. Introduction

Attention is the core of transformers [1], which powers the development and applications of large models across various fields, ranging from language modeling to vision understanding and multimodal generation. Despite their success, a fundamental computational bottleneck exists: the time complexity of attention grows quadratically with sequence length $N$. As models scale to handle increasingly long contexts, this quadratic cost becomes prohibitive, limiting deployment in latency-sensitive or resource-limited settings. This challenge has sparked extensive research on efficient attention. In general, efficient attention methods aim to reduce time or memory costs while preserving the effectiveness of the standard attention.

**Taxonomy.** To better understand the current and predict the future, this survey proposes a unified and comprehensive taxonomy of efficient attention methods. Specifically, we systematically organize the existing approaches into four major categories: *hardware-efficient attention*, *compact attention*, *sparse attention*, and *linear attention*. The high-level ideas of each category are as follows: (1) Hardware-efficient attention methods optimize the implementation efficiency of the original attention without changing its computation logic. These methods utilize modern GPU features through techniques such as matrix tiling, kernel fusion [2], [3], and quantization for low-bit Tensor Core [4]–[8], to accelerate attention. (2) Compact attention methods compress the KV cache [9] to reduce memory overhead during inference. This is typically achieved through weight sharing [10] and low-rank decomposition [11], enabling memory-efficient caching while keeping computational cost unchanged compared with a full-sized KV cache. (3) Sparse attention methods reduce computational cost by skipping calculations for non-critical parts of the softmax attention map. This approach is viable because the attention map 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 sparse mask [12] or a dynamic sparse mask [13] to the matrix, which directs the model to perform attention operations only on the unmasked positions, i.e., the crucial positions. (4) Linear attention methods remove 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 $K^\top V$ [14] and then multiplies the result with the queries $Q$, reducing the computational complexity to linear $\mathcal{O}(N)$.

**Relation to prior surveys.** Although several surveys [15], [16], [16] also summarize work on attention, they each focus on attention mechanisms within a specific domain, e.g., language modeling or computer vision. In contrast, to the best of our knowledge, our survey is the most comprehensive and most recent survey on efficient attention. Our taxonomy (hardware-efficient attention, compact attention, sparse attention, and linear attention) covers almost all existing efficient attention methods; in particular, the categories hardware-efficient attention and compact attention are newly proposed in this work.

**Analysis framework.** We first present the motivation, fundamental idea of the four categories of efficient attention methods in Section III. Furthermore, we develop a unified analysis framework for each category in Sections IV-A, V-A, VI-A, and VII-A. These unified frameworks enable a systematic analysis and comparison of different methods, providing clear insights into their design and practical strengths.

**Trade-off.** Efficient attention methods often involve a trade-off between efficiency and quality. When we remove more attention computations (for example, by using very sparse

Corresponding author: Jun Zhu.

TABLE I: Notations.

| Notation | Shape | Meaning |
|---|---|---|
| $N, h, d$ | $1 \times 1$ | sequence length, number of attention heads, head dimension of attention |
| $D_m, D$ | $1 \times 1$ | 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 |
| $S, P$ | $N \times N$ | $S = QK^\top, P = \mathrm{Softmax}(S)$ |
| $M$ | $N \times N$ | the mask needed to be added to $P$ |
| $b_q, b_{kv}$ | $1 \times 1$ | 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 |
| $\beta$ | $1 \times 1$ | select gate in linear attention |
| $\phi$ | - | kernel functions in linear attention |

patterns or strong low-rank approximations), the risk of performance degradation usually increases. In contrast, methods that only speed up the computation of standard attention, without changing the attention formula itself (such as some hardware-efficient attention methods), often maintain model quality more consistently. However, this trade-off is not strict: with careful design, low-bit quantization can be lossless, sparse attention can sometimes improve generalization, and linear attention can match or even surpass vanilla softmax attention on some tasks.

**Content.** First, in Section II, we introduce the important preliminaries of efficient attention. Next, in Section III, we present the motivation, fundamental idea and common practices of the four categories, i.e., hardware-efficient attention, compact attention, sparse attention, and linear attention. Subsequently, in Sections IV, V, VI, and VII, we discuss each category in detail, outlining their unified frameworks, formulations, key methods and features, and implementation details. Lastly, we analyze the limitations and opportunities of each category of efficient attention methods in Section VIII.

## II. Preliminary

### A. Standard Attention Computation

The core idea of attention [17] is to dynamically retrieve information by calculating a weighted sum of values, where each weight is given by how relevant the corresponding key is to a given query. A standard implementation is the Scaled Dot-Product Attention [1]. 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 enable the model to attend jointly to information from multiple representation subspaces, Multi-Head Attention [1] is operated as follows: $Q$, $K$, and $V$ 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. These $h$ distinct outputs are stacked, and a final linear projection acts to combine them into a single final output. For each individual attention head, the computation is formulated as:

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

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 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 is implemented by subtracting the row-wise maximum $m_i = \max(S_i)$ before exponentiation to prevent overflow: $P_{i,j} = \frac{\exp(S_{i,j} - m_i)}{\sum_k \exp(S_{i,k} - m_i)}$.

### B. 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 [18]. 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 [19], denoted as $\underline{B_w}$ (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 [8], [20], 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 $T_{\text{compute}}$ dominates, it is compute-bound.

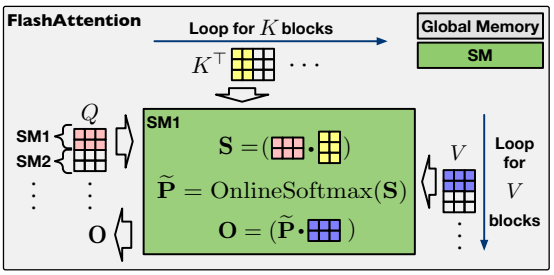


Fig. 1: Illustration of FlashAttention. Different $Q$ blocks are processed on different SMs, where each SM iteratively loads $K$ and $V$ blocks and finally outputs an $O$ block.

### C. FlashAttention

The $Q$, $K$, and $V$ matrices have dimensions $N \times d$, while the score and probability matrices $S$ and $P$ have dimensions $N \times N$. Although $d$ is relatively small (e.g., 64 or 128), $N$ can reach thousands or even millions. Consequently, 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 [2], [3] 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 [21] 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 $\mathbf{O}_i = \text{diag}(l_{ij})^{-1} \mathbf{O}_{ij}$.

### D. Attention in Different Tasks

*1) Attention in Non-Autoregressive Models:* In non-autoregressive models such as BERT [22], Vision Transformer (VIT) [23], and U-VIT [24] (the first backbone of Diffusion Transformer), attention is computed in parallel over a sequence of $N$ input tokens, with each token attending to every other:

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

For $N$ input tokens, the $Q$, $K$, and $V$ matrices have dimensions $N \times d$. The computational cost comes from the matrix multiplications $QK^\top$ and $PV$, which have a time complexity of $\mathcal{O}(N^2 d)$. Most models today use FlashAttention [3] 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) Attention in LLM Training:* To preserve the autoregressive property of LLMs [9], a lower-triangular causal mask is applied during training, ensuring that each token can only attend to itself and preceding tokens:

$$O = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}} + M\right) V, \quad M_{i,j} = \begin{cases} 0 & \text{if } j \le i. \\ -\infty & \text{if } j > i. \end{cases} \tag{6}$$

The $N \times N$ causal mask $M$ does not change the dominant computational complexity of $\mathcal{O}(N^2 d)$. Hence, attention in LLM training remains **compute-bound**.

*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 II-D2), which is typically **compute-bound**. The resulting $K$ and $V$ matrices are stored in the GPU's global memory as the KV cache [9]. In the decoding phase, the model generates tokens sequentially. For each new token, it computes the corresponding $q, k, v$ vectors, and the query $q$ attends to all preceding keys and values. The attention output is computed as:

$$o = \text{softmax}\left(q(K\|k)^\top / \sqrt{d}\right)(V\|v). \tag{7}$$

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 has a computational complexity of $\mathcal{O}(Nd)$ for matrix-vector multiplication and a memory complexity of $\mathcal{O}(Nd)$ for loading the KV cache. As observed in FlashAttention [2], on modern GPUs, I/O operations are often significantly slower than computation, even when both have equivalent complexity. Thus, the decoding phase is **memory-bound**.

## III. Overview

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

### A. Hardware-Efficient Attention

To design efficient operations on modern GPUs, it is crucial to understand their memory hierarchy, as mentioned in Section II-B. 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

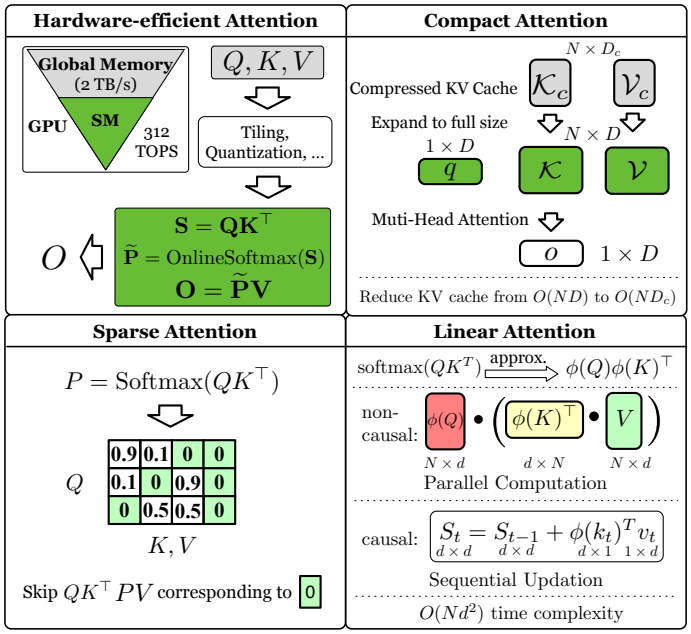


Fig. 2: Overview of efficient attention methods.

access latency, computation and memory I/O can be fully overlapped and executed in parallel through pipelining [8], [20], [25]. 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 II-A, naive attention is memory-bound because of the materialization of two large intermediate matrices, the attention scores $S = QK^\top$ and the probability matrix $P = \mathrm{softmax}(S)$, 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, which has a global memory bandwidth of up to 2.0 TB/s and a peak throughput of 312 `TFLOPS` for `FP16` operations. 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 $\approx 19$`GB` (1 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 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 [21], 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 improving efficiency.

As discussed in Section II-D3, after using FlashAttention to optimize I/O, there are two settings: the compute-bound setting for non-autoregressive models and LLM prefilling, and the memory-bound setting for LLM decoding. Hardware-efficient attention methods are designed to address the bottlenecks of each setting. For the compute-bound setting, hardware-efficient methods focus on maximizing parallelism and computational throughput. For the memory-bound setting, hardware-efficient methods focus on accelerating I/O for the KV cache.

### *B. Compact Attention*

**Exploding KV cache memory in LLM inference.** During inference of LLMs, vanilla Multi-Head Attention (MHA) [1] introduces $\mathcal{O}(Nhd)$ memory complexity for KV cache. This consumes significant GPU memory and can 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 Transformer with MHA, the KV cache accounts for 96`GB` memory when using `bfloat16` format. This exceeds the VRAM needed for storing the parameters of the entire model, as well as the capacity of many popular GPUs, including A100 and H100. This problem is exacerbated 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 $\mathcal{K} = [K^{(1)}, \ldots, K^{(h)}]$ denote 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.

$$\mathcal{K}_c, \mathcal{V}_c \in \mathbb{R}^{N\times D_c}, \tag{8}$$

$$\mathcal{K}, \mathcal{V} = \mathrm{Expand}_{\mathcal{K}}(\mathcal{K}_c), \mathrm{Expand}_{\mathcal{V}}(\mathcal{V}_c) \in \mathbb{R}^{N\times D}, \tag{9}$$

$$o = \mathrm{MHA}(q, \mathcal{K}, \mathcal{V}). \tag{10}$$

where $\mathrm{MHA}(\cdot)$ denotes the multi-head attention operation as stated in Section II-A. 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 $\mathrm{Expand}(\cdot)$ function.

### C. Sparse Attention

**Sparsity for matrix multiplication efficiency.** A matrix is considered sparse when it contains a large number of zeros. Such sparsity can be used 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 $A[i, j]B[j, :]$ contributes nothing to $C[i, :]$ and can be omitted; similarly, if $B[i, j] = 0$, the computation of $A[:, i]B[i, j]$ can be skipped.

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 $P = \mathrm{Softmax}(QK^\top/\sqrt{d})$ exhibits inherent sparsity [13], [26], 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 $M$:

$$P = \mathrm{Softmax}(M + QK^\top/\sqrt{d}), \qquad O = PV. \tag{11}$$

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. 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.

### D. Linear Attention

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 $\phi$, e.g., a softmax function, which allows the standard attention computation to be reordered as:

$$O = \phi(Q)(\phi(K)^\top V). \tag{12}$$

By first calculating the term $(\phi(K)^\top V)$ with $\mathcal{O}(Nd^2)$ complexity, 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.

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 are introduced. Then the $H_t$ update can be formulated as:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot \phi(k_t)^\top v_t. \tag{13}$$

Here, inspired by gates of RNNs [27], $G_f^{(t)}$ 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) [28] views the hidden state $H_t$ as a set of learnable parameters, also called 'fast weights' [29]. TTT will continuously update the hidden state via gradient descent [30] in both training and inference.

## IV. Hardware-efficient Attention

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

### A. Unified Framwork

Corresponding to the two stages in LLMs inference as introduced in Section. II-D3, *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:

$$\hat{\mathbf{Q}}, \hat{\mathbf{K}}, \hat{\mathbf{V}} = \Psi(\mathbf{Q}), \Psi(\mathbf{K}), \Theta(\mathbf{V}), \tag{14}$$

$$\mathbf{S} = \hat{\mathbf{Q}}\hat{\mathbf{K}}^\top, \quad \hat{\mathbf{P}} = \Theta(\mathrm{softmax}(\mathbf{S})), \quad \mathbf{O} = \hat{\mathbf{P}}\hat{\mathbf{V}}. \tag{15}$$

TABLE II: Summary of hardware-efficient attention methods. − means no additional preprocessing.

| Method | Category | $\Psi(\cdot)$ Type | $\Theta(\cdot)$ Type |
|---|---|---|---|
| FlashAttention [2] | Prefilling | − | − |
| FlashAttention2 [3] | Prefilling | − | − |
| SageAttention [4] | Prefilling | INT8 quantizer | − |
| SageAttention2 [5] | Prefilling | INT4 quantizer | FP8 quantizer |
| SageAttention2++ [7] | Prefilling | INT4 quantizer | FP8 quantizer |
| SageAttention3 [6] | Prefilling | FP4 quantizer | FP4 quantizer |
| FlashAttention3 [8] | Prefilling | − or FP8 quantizer | − or FP8 quantizer |
| FlashDecoding [31] | Decoding | split KV cache | split KV cache |
| KVQuant [32] | Decoding | INT2/3/4 quantizer | INT2/3/4 quantizer |
| KiVi [33] | Decoding | INT2 quantizer | INT2 quantizer |
| PagedAttention [34] | Decoding | reallocate KV cache | reallocate KV cache |
| FlashInfer [35] | Decoding | reallocate KV cache | reallocate KV cache |

where $\Psi(\cdot), \Theta(\cdot)$ are preprocess functions to accelerate computation, e.g., low-bit 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:

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

$$\mathbf{s} = \mathbf{q}\hat{\mathbf{K}}^\top, \quad \mathbf{p} = \mathrm{softmax}(\mathbf{s}), \quad \mathbf{o} = \mathbf{p}\hat{\mathbf{V}}. \tag{17}$$

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

### *B. Methods*

*1) Non-autoregressive Models and Prefilling in LLM:* 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 [21] technique, FlashAttention [2] 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 II-C). FlashAttention [2] 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 [3] 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.

► **SageAttention.** SageAttention [4] builds on the block-tiling strategy of FlashAttention [2] 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++ [5], [7] 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 and preserve end-to-end accuracy across all models and tasks.

► **FlashAttention3.** While recent GPUs like H100 have massively increased TensorCore throughput, computation is bottlenecked by softmax operations on CUDA cores. FlashAttention 3 [8] addresses this by leveraging Hopper architecture [36] 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 $2\times$ theoretical speedup over `FP16`/`BF16`. These optimizations achieve up to 75% TensorCore utilization, yielding 1.5-2$\times$ speedup over FlashAttention2 and reaching 1.2 `PFLOPS` throughput with `FP8` precision.

► **SageAttention3.** SageAttention3 [6] 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 $1 \times 16$ 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 $[0, 448\times6]$ 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 $5\times$ speedup over FlashAttention) for inference. `SageBwd` accelerate fine-tuning tasks with lossless accuracy.

*2) Decoding in LLM:* As discussed in Section II-D1, the decoding phase is memory-bound due to the large I/O overhead for the KV cache, so hardware-efficient attention methods for decoding primarily aim to accelerate I/O for the KV cache.

► **FlashDecoding.** Though FlashAttention2 [3] 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 [31] 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 $8\times$ speedup over FlashAttention2 on long sequences with CodeLlama-34B.

► **PagedAttention.** PagedAttention [34] 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 [33] 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 [4], 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 $2.6\times$ peak memory reduction while maintaining model quality, enabling $4\times$ larger batch sizes.

► **KVQuant.** KVQuant [32] 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 [37] 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 [35] 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 [34] and radix-tree layouts [38]. 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.

## V. Compact Attention

In this section, we first present a unified formulation for compact attention, then review representative methods, summarizing their key characteristics, and finally elaborate on individual solutions.

### A. Unified Framwork

Compact attention methods are designed to reduce the memory consumption of the KV cache during LLM inference. Instead of storing the full-resolution KV matrices in MHA, compact attention methods decouple storage KV from computation KV, storing compressed KV states and expanding them for computation. This sharply slows KV memory growth, unlocking longer contexts, while keeping the compute-time

TABLE III: Summary of compact attention methods.

| Method | KV Cache | Parameters | Expand Method |
|---|---|---|---|
| MHA [1] | $2hd$ | $4D_m hd$ | None |
| MQA [39] | $2d$ | $2D_m hd + 2D_m d$ | Repeat |
| GQA [10] | $2h_{kv}d$ | $2D_m hd + 2D_m h_{kv} d$ | Repeat |
| MLA [11] | $r_{kv} + d_R$ | $D_m(r_q + r_{kv} + d_R + hd)$ $+r_q h(d_N + d_R) + r_{kv} h(d_N + d)$ | Low Rank Projection + Repeat |
| MFA [40] | $2d$ | $D_m(3d + hd) + hd^2$ | Repeat |
| TPA [41] | $(r_k + r_v)(h + d)$ | $D_m(r_q + r_k + r_v)(h + d) + D_m hd$ | Tensor Product |

tensors unchanged, so quality loss is minimal. The general formulation can be expressed as follows:

$$q, \mathcal{K}_c, \mathcal{V}_c = \mathrm{Proj}_{\mathcal{Q}}(x), \mathrm{Proj}_{\mathcal{K}_c}(X), \mathrm{Proj}_{\mathcal{V}_c}(X), \tag{18}$$
$$\mathcal{K}, \mathcal{V} = \mathrm{Expand}_{\mathcal{K}}(\mathcal{K}_c), \mathrm{Expand}_{\mathcal{V}}(\mathcal{V}_c), o = \mathrm{MHA}(q, \mathcal{K}, \mathcal{V}).$$

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 $D = hd$, analogously for $q$ 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 context states, $\mathrm{Proj}(\cdot)$ and $\mathrm{Expand}(\cdot)$ denote the projection/expansion functions, and $\mathrm{MHA}(\cdot)$ is the standard multi-head attention. Vanilla MHA [1] can be recovered by setting $\mathrm{Proj}_{\mathcal{K}_c}(X) = xW_{\mathcal{K}_c} \in \mathbb{R}^{N \times D}$ where $W_{\mathcal{K}_c} \in \mathbb{R}^{D_m \times D}$ (similarly for $\mathrm{Proj}_{\mathcal{Q}}$ and $\mathrm{Proj}_{\mathcal{V}_c}$), and $\mathrm{Expand}_{\mathcal{K}}(\cdot) = \mathrm{Expand}_{\mathcal{V}}(\cdot) = \mathrm{id}(\cdot)$, i.e., the identity projection. Compact attention reduces the size of KV cache $\mathcal{K}_c$ and $\mathcal{V}_c$ by altering $\mathrm{Proj}(\cdot)$ and using a non-trivial $\mathrm{Expand}(\cdot)$. These methods do not alter the procedure or computational cost of scaled dot-product attention, but can substantially lower the memory consumption of the KV cache, enabling longer contexts and memory-efficient scaling.

Table III summarizes the per-token KV cache size, attention parameter count, and expansion operator for each compact attention method, with the notations detailed in the next section.

### *B. Methods*

► **MQA.** Multi-Query Attention (MQA) [39] introduces an asymmetric head architecture: it retains $h$ distinct query heads but shares a single key–value head. Concretely, the input sequence $X$ is projected into a single key–value pair using $W_{\mathcal{K}_c}, W_{\mathcal{V}_c} \in \mathbb{R}^{D \times d}$: $\mathcal{K}_c, \mathcal{V}_c = XW_{\mathcal{K}_c}, XW_{\mathcal{V}_c}$ The shared KV pair is broadcast to all $h$ query heads, yielding the final $\mathcal{K}$ and $\mathcal{V}$ for computation. This replication can be written via the Kronecker product $\otimes$ [42]: $\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h}$. where $\mathbf{1}_{1\times h}$ is a length-$h$ row vector of ones that broadcasts the shared projections to all query heads. Equivalently, MQA is weight sharing: $\mathcal{K} = (XW_{\mathcal{K}_c}) \otimes \mathbf{1}_{1\times h} = X(W_{\mathcal{K}_c} \otimes \mathbf{1}_{1\times h})$, so a single projection $W_{\mathcal{K}_c}$ is reused across all $h$ heads. This reduces KV-cache by a factor of $h$ versus MHA. But the lone KV head limits capacity and typically underperforms MHA.

$$q, \mathcal{K}_c, \mathcal{V}_c = x \overset{D_m \times hd}{W_Q}, X \overset{D_m \times d}{W_{\mathcal{K}_c}}, X \overset{D_m \times d}{W_{\mathcal{V}_c}}, \tag{19}$$
$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h}. \tag{20}$$

► **GQA.** Grouped-Query Attention (GQA) [10] interpolates between MHA and MQA by partitioning the $h$ query heads into $h_{kv}$ groups. 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 is obtained: $\mathcal{K}_c = XW_{\mathcal{K}_c}, \mathcal{V}_c = XW_{\mathcal{V}_c}$, where $W_{\mathcal{K}_c}, W_{\mathcal{V}_c} \in \mathbb{R}^{D \times h_{kv} d}$. To serve all $h$ query heads, each KV head is replicated within its group: $\mathcal{K} = \mathcal{K}_c \otimes \mathbf{1}_{1\times(h/h_{kv})}, \mathcal{V} = \mathcal{V}_c \otimes \mathbf{1}_{1\times(h/h_{kv})}$. GQA reduces the KV cache size by $h/h_{kv}$ compared to MHA, achieves a strong quality–efficiency balance, and has been widely adopted in large LMs (e.g., LLaMA [43]).

$$q, \mathcal{K}_c, \mathcal{V}_c = x \overset{D_m \times hd}{W_Q}, X \overset{D_m \times h_{kv} d}{W_{\mathcal{K}_c}}, X \overset{D_m \times h_{kv} d}{W_{\mathcal{V}_c}}, \tag{21}$$
$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h/h_{kv}}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h/h_{kv}}. \tag{22}$$

► **MLA.** Multi-Head Latent Attention (MLA) [11] shrinks KV cache size through low-rank projection. It caches a shared, down-projected latent: $\mathcal{K}_c^N = \mathcal{V}_c = XW_{\mathcal{KV}}^{\mathrm{down}} \in \mathbb{R}^{N \times r_{kv}}$, then up-projects for compute: $\mathcal{K}^N = \mathcal{K}_c^N W_{\mathcal{K}}^{\mathrm{up}} \in \mathbb{R}^{N \times d_N}, \mathcal{V} = \mathcal{V}_c^N W_{\mathcal{V}}^{\mathrm{up}} \in \mathbb{R}^{N \times d}$. The Rotary Position Encoding (RoPE) [44] cannot be directly combined with low-rank KV compression: its rotation acts in the full Q/K feature space, forcing caches to store the up-projected states. Consequently, the aforementioned branch is kept as NoPE (no positional encoding) [45]. To inject positional information, MLA adds another RoPE branch with a single-head key: $\mathcal{K}_c^R = \mathrm{RoPE}(XW_{\mathcal{K}_c^R}) \in \mathbb{R}^{N \times d_R}$, then tiles it across all heads: $\mathcal{K}^R = \mathcal{K}_c^R \otimes \mathbf{1}_{1\times h}$, and concatenates it with the per-head NoPE key 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 likewise low-rank projected to match key dimensions and reduce parameters. With these two branches, the per-token KV cache is reduced from $2hd$ to $r_{kv} + d_R$.

$$q = [\overset{D_m \times r_q}{xW_{\mathcal{Q}}^{\mathrm{down}}} \overset{r_q \times hd_N}{W_{\mathcal{Q}}^{\mathrm{up},N}}, \ \mathrm{RoPE}(\overset{D_m \times r_q}{xW_{\mathcal{Q}}^{\mathrm{down}}} \overset{r_q \times hd_R}{W_{\mathcal{Q}}^{\mathrm{up},R}})], \tag{23}$$

$$\mathcal{K}_c^N = \mathcal{V}_c = \overset{D_m \times r_{kv}}{XW_{\mathcal{KV}}^{\mathrm{down}}}, \quad \mathcal{K}_c^R = \mathrm{RoPE}(\overset{D_m \times d_R}{XW_{\mathcal{K}_c^R}}), \tag{24}$$

$$\mathcal{K} = [\mathcal{K}_c^N \overset{r_{kv} \times hd_N}{W_{\mathcal{K}}^{\mathrm{up}}}, \mathcal{K}_c^R \otimes 1_{1\times h}], \quad \mathcal{V} = \mathcal{V}_c \overset{r_{kv} \times hd}{W_{\mathcal{V}}^{\mathrm{up}}}. \tag{25}$$

► **MFA.** Multi-matrix Factorization Attention (MFA) [40] builds on MQA by sharing a single KV head, and narrows the MQA–MHA gap with two changes: (i) increase the per-head width $d$ to boost each head's capacity; and (ii) factorize the query projection with a shared down-projection $W_{\mathcal{Q}}^{\mathrm{down}} \in \mathbb{R}^{D_m \times d}$ followed by an up-projection $W_{\mathcal{Q}}^{\mathrm{up}} \in \mathbb{R}^{d \times hd}$

to generate all heads. This reduces the parameter cost of scaling $h$ from $\sim 2D_m d$ (query+output) to $\sim D_m d$ (output alone).

$$q = x\overset{D_m\times d}{W_{\mathcal{Q}}^{\text{down}}}\overset{d\times hd}{W_{\mathcal{Q}}^{\text{up}}}, \quad \mathcal{K}_c, \mathcal{V}_c = X\overset{D_m\times d}{W_{\mathcal{K}_c}^{\text{down}}}, X\overset{D_m\times d}{W_{\mathcal{V}_c}^{\text{down}}}, \tag{26}$$

$$\mathcal{K}, \mathcal{V} = \mathcal{K}_c \otimes \mathbf{1}_{1\times h}, \mathcal{V}_c \otimes \mathbf{1}_{1\times h}. \tag{27}$$

► **TPA.** Tensor-Product Attention (TPA) [41], rooted in low-rank decompositions, replaces the fixed head-wise repetition of MQA/GQA with a learnable tensor product. Whereas MQA/GQA tile a shared KV via $\mathcal{K} = \mathcal{K}_c \otimes \mathbf{1}$, TPA learns: $\mathcal{K} = \mathcal{K}_c^A \otimes \mathcal{K}_c^B$, with $\mathcal{K}_c^A \in \mathbb{R}^{N\times h\times 1}$ capturing head-specific scaling factors and $\mathcal{K}_c^B \in \mathbb{R}^{N\times 1\times d}$ capturing shared features; $\otimes$ denotes the Kronecker product along the last two axes. In this basic form, $\mathcal{K}$ is a rank-1 approximation of the full-size key state. To increase capacity, TPA averages $r$ such factorized components: $\mathcal{K} = \frac{1}{r}\sum_{i=1}^{r}\mathcal{K}_i$, yielding a rank-$r$ storage state. The same construction applies to $\mathcal{V}$, reducing the per-token KV cache size to $(r_k + r_v)(h + d)$. The query projection is factorized analogously, preserving structural symmetry.

$$q^A, q^B = x\overset{D_m\times r_q h}{W_{\mathcal{Q}}^A}, x\overset{D_m\times r_q d}{W_{\mathcal{Q}}^B}, \tag{28}$$

$$= [q^{A,1}, \ldots, q^{A,r_q}], [q^{B,1}, \ldots, q^{B,r_q}], \tag{29}$$

$$q = \frac{1}{r_q}\sum_{i=1}^{r_q}\overset{h\times 1}{q^{A,i}} \otimes \overset{1\times d}{q^{B,i}}, \tag{30}$$

$$\mathcal{K}_c^A, \mathcal{K}_c^B = X\overset{D_m\times r_k h}{W_{\mathcal{K}_c}^A}, X\overset{D_m\times r_k d}{W_{\mathcal{K}_c}^B}, \tag{31}$$

$$\mathcal{V}_c^A, \mathcal{V}_c^B = X\overset{D_m\times r_v h}{W_{\mathcal{V}_c}^A}, X\overset{D_m\times r_v d}{W_{\mathcal{V}_c}^A}, \tag{32}$$

$$\mathcal{K}, \mathcal{V} = \frac{1}{r_k}\sum_{i=1}^{r_k}\mathcal{K}_c^{A,i} \otimes \mathcal{K}_c^{B,i}, \frac{1}{r_v}\sum_{i=1}^{r_v}\mathcal{V}_c^{A,i} \otimes \mathcal{V}_c^{B,i}. \tag{33}$$

## VI. Sparse Attention

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

### A. Unified Framwork

The attention map $P = \text{Softmax}(QK^\top/\sqrt{d})$ exhibits inherent sparsity, as the softmax operation often creates many values approaching zero [77]. *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* $M$.

$$P = \text{Softmax}(M + QK^\top/\sqrt{d}), \quad O = PV. \tag{34}$$

Here, $M$ is a $N \times N$ 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 $\tau$, 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 of sparse FlashAttention based on the masks is defined as follows:*

$$\mathbf{M}_{i,j} = -\infty \text{ if } M[ib_q : (i+1)b_q, jb_{kv} : (j+1)b_{kv}] = -\infty,$$
$$\mathbf{Q}_i\mathbf{K}_j^\top, \widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j \text{ are skipped if } \mathbf{M}_{i,j} = -\infty. \tag{35}$$

### B. Preliminaries of Sparse Attention

**LLM prefilling and decoding.** As discussed in Section II-D3, 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 $\mathbf{V}$.

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

**DiT.** Diffusion transformer models [78], commonly used for image and video generation, often adopt vision transformers as the backbone. As discussed in Section II-D1, attention computation speed is the primary bottleneck.

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

Table IV 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.

### C. Pattern-based Sparse Attention

► **StreamingLLM.** Simple window attention [79] collapses once initial tokens are evicted from the KV cache. StreamingLLM [12] identifies a key phenomenon called the attention sink, where a few initial tokens consistently

TABLE IV: Summary of sparse attention methods.

| Category | Method | LLM | Reduce KV Storage | DIT | Training Free |
|---|---|---|---|---|---|
| **Pattern Based** | StreamingLLM [12] | ✓ | ✓ | ✗ | ✓ |
| | DiTFastAttn [46] | ✗ | ✗ | ✓ | ✓ |
| | SampleAttn [47] | ✓ | ✗ | ✗ | ✓ |
| | MoA [48] | ✓ | ✓ | ✗ | ✓ |
| | DuoAttention [49] | ✓ | ✓ | ✗ | ○ |
| | SparseVideoGen [50] | ✗ | ✗ | ✓ | ✓ |
| | Radial Attention [51] | ✗ | ✗ | ✓ | ✓ |
| | STA [52] | ✗ | ✗ | ✓ | ✓ |
| | NeighborAttn [53] | ✗ | ✗ | ✓ | ✓ |
| | PAROAttn [54] | ✗ | ✗ | ✓ | ✓ |
| **Dynamic Sparse** | SLA (sparse-linear attn) [55] | ✗ | ✗ | ✓ | ✗ |
| | SpargeAttn [13] | ✓ | ✗ | ✓ | ✓ |
| | H2O [56] | ✓ | ✓ | ✗ | ✓ |
| | InfLLM [57] | ✓ | ✗ | ✗ | ✓ |
| | MInference [58] | ✓ | ✗ | ✗ | ✓ |
| | FlexPrefill [59] | ✓ | ✗ | ✗ | ✓ |
| | SparQAttn [60] | ✓ | ✓ | ✗ | ✓ |
| | LokiAttn [61] | ✓ | ✗ | ✗ | ✓ |
| | SeerAttention [62] | ✓ | ✗ | ✗ | ○ |
| | RetrievalAttn [63] | ✓ | ✓ | ✗ | ✓ |
| | FPSAttention [64] | ✗ | ✗ | ✓ | ✗ |
| | NSA [65] | ✓ | ✓ | ✗ | ✗ |
| | DSA [66] | ✓ | ✓ | ✗ | ✗ |
| | MoBA [67] | ✓ | ✗ | ✗ | ✗ |
| | VMoBA [68] | ✓ | ✗ | ✓ | ✗ |
| | CHAI [69] | ✓ | ✓ | ✗ | ✓ |
| | Quest [70] | ✓ | ✗ | ✗ | ✓ |
| | MagicPig [71] | ✓ | ✗ | ✗ | ✓ |
| | HashAttn [72] | ✓ | ✗ | ✗ | ○ |
| | XAttention [73] | ✓ | ✗ | ✓ | ✓ |
| | VSA [74] | ✗ | ✗ | ✓ | ✗ |
| | SparseVideoGen2 [75] | ✗ | ✗ | ✓ | ✓ |
| | Twilight [76] | ✓ | ✗ | ✗ | ✓ |

$^{\circ}$ indicates that it requires training a subset of parameters rather than the entire model.

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 approach reduces the per-token computational complexity to $O(L)$ for a KV cache of fixed size $L$.

► **DitFastAttn.** DiTFastAttn [46] is a post-training method that accelerates DiT [78] inference by targeting three key redundancies. For spatial redundancy, it uses window attention [79] and adds a cached "residual" (the difference between full and window attention) from a prior step to maintain quality. For temporal redundancy, it reuses attention results from a previous timestep if they are highly similar. To reduce CFG [80] redundancy, it reuses the attention output from the conditional pass for the unconditional pass, skipping the redundant computation.

► **DuoAttention.** DuoAttention [49] 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 attn $= \alpha \cdot$ full_attn $+ (1 - \alpha) \cdot$ 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 $\alpha$, $(\sum |\alpha|)$, to promote sparsity. DuoAttention achieves up to $2.55\times$ prefilling and $2.18\times$ decoding speedups, and can reduce inference memory.

► **Sparse VideoGen.** Sparse VideoGen [50] 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 $M_{\text{spatial}}$ and $M_{\text{temporal}}$, where $M_{\text{spatial}}$ consists of a diagonal sliding window and a first-frame sink, and $M_{\text{temporal}}$ 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 $M_h^{\text{spatial}}$ or $M_h^{\text{temporal}}$ accordingly by comparing the MSE between the ground truth output and the output after applying the sparse attention mask. Sparse VideoGen achieves up to 2.3× speedup on video generation tasks.

► **STA.** Sliding Tile Attention [52] accelerates video diffusion transformers by overcoming the computational inefficiency 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. STA supports both training-free and fine-tuned configurations. In the training-free setting, it calibrates per-layer, per-head window sizes using a small prompt set, attaining up to 58% attention sparsity. In the fine-tuned setting, fixed sparse masks can be optimized to further improve throughput.

► **NeighborAttn.** Neighborhood Attention [53] introduces a pixel-wise sliding window attention that localizes each query's attention span to its immediate spatial neighbors. In contrast to Swin transformer [81], 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 [82] with custom CUDA kernels, achieving up to 40% speedup and 25% memory reduction compared to Swin attention.

### *D. Dynamic Sparse Attention*

► **SLA.** SLA [55] (Sparse-Linear Attention) is a trainable attention method for accelerating Diffusion Transformer models (e.g., U-VIT [24]), offering significant benefits for computationally intensive video generation task. 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 standard attention with SLA and conducting a brief fine-tuning phase before inference, diffusion transformer models can achieve a 20× reduction (95% sparsity) in attention computation and a 13.7× attention speedup without loss of generation quality.

► **SpargeAttn.** SpargeAttn [13] 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. Using the compressed $Q$ and $K$, it builds a reduced attention map $\hat{P}$. SpargeAttn then computes $\mathbf{Q}_i\mathbf{K}_j^\top$ and $\widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j$ only for those (i,j) pairs whose entries in $\hat{P}$ receive high accumulated scores. 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 $\widetilde{\mathbf{P}}_{i,j}\mathbf{V}_j$. Additionally, SpargeAttn is directly applied to SageAttention for further acceleration. SpargeAttn could accelerate language, image, and video generation models without harming end-to-end quality.

► **H2O.** H2O [56] 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 [83] models, H2O achieves accuracy comparable to full attention while delivering 3× latency reduction compared to FlexGen [84].

► **InfLLM.** InfLLM [57] is a training-free, memory-based attention method for large language models. First, each key $k_m$ is assigned a representative score $r_m = \frac{1}{l_L}\sum_{j=1}^{l_L} q_{m+j}k_m$, where $l_L$ is the length of the local window. The global KV cache is then partitioned into blocks $\{B_i\}$ of size $b$. For each block, the top-$r_k$ tokens by $r_m$ are averaged to form a block representative. During decoding, attention between the current query and all block representatives is computed, and the top-$k_m$ blocks are selected to form the sparse attention mask. InfLLM also integrates sliding window attention and attention sink. On Mistral and Llama-3, it extends context length to 128K tokens, and achieves a 1.5× speedup over full attention.

► **MInference.** MInference [58] introduces an offline search-based dynamic sparse attention to accelerate long-context LLM inference during the prefilling stage. It defines three candidate attention patterns: A-shaped, vertical-slash, and block-sparse. Each attention head first performs an offline search to

determine its optimal pattern, and then dynamically predicts the corresponding sparse attention mask online. For A-shaped heads, the mask is static and includes only the initial tokens (attention sink) and those within a sliding window. For vertical-slash, the mask is constructed by selecting the top-$k$ attention scores in each row. For block-sparse, queries and keys are first mean-pooled into blocks, and the top-$k$ scores in the block-level attention form the mask. By restricting computation to pattern-specific regions, MInference effectively reduces computation. It achieves up to $10\times$ speedup in the prefilling stage on A100 GPUs, without degrading model accuracy.

► **SparQAttn.** SparQAttention [60] mitigates the memory bandwidth bottleneck during LLM decoding. At each decoding step, it selects the top-$r$ largest components of the query token $q_t$ and slices both $q_t$ and the key cache $K$ along these dimensions ($r$ is a fixed hyperparameter, typically 64). This yields a low-dimensional approximation of the attention scores $\hat{S}$ without accessing full key tokens. Based on $\hat{S}$, SparQ identifies the top-$k$ most relevant tokens and computes attention only over their corresponding keys and values, significantly reducing memory access and computation.

► **LokiAttn.** LokiAttention [61] is a training-free sparse attention method designed to accelerate LLM decoding. It first generates key vectors from calibration prompts for the target model and applies Principal Component Analysis (PCA) [85] to reduce the head dimension $d$ to a lower rank $d_r$. The resulting PCA projection matrix is computed offline and stored. During inference, queries and keys are projected into the reduced $N \times d_r$ space using this matrix, where attention scores are computed efficiently. The top-$k$ scores from the projected attention are then selected to form the sparse mask, over which the final attention is computed in full dimension. LokiAttn achieves up to 45% speedup over standard implementations.

► **SeerAttention.** SeerAttention [62] introduces a learnable block-sparse attention by inserting a gated linear projection $g$ before the RoPE [37] module to predict block-wise sparse mask $\mathbf{M}$: $\hat{Q} = \mathrm{RoPE}(g(\mathrm{pool}(Q_{\mathrm{pre-rope}})))$, $\hat{K} = \mathrm{RoPE}(g(\mathrm{pool}(K_{\mathrm{pre-rope}})))$, $\mathbf{M} = \mathrm{RowTopK}(\mathrm{Softmax}\left(\hat{Q}\hat{K}^\top/\sqrt{d}\right))$, 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 [86] between SeerAttention and full attention outputs on long-context post-training data, allowing the model to retain sparsity while closely approximating full attention behavior.

► **FPSAttention.** FPSAttention [64] is a training-aware FP8 quantization and structured sparsity co-design built on Sliding Tile Attention (STA) [52] 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 [5], 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. It achieves a high speedup in video generation.

► **NSA.** Native Sparse Attention (NSA) [65] is a trainable sparse attention method that accelerates attention with both algorithmic design and hardware optimizations. For each query $q_t$, NSA combines three branches: (1) Compression: $K$ and $V$ are blocked and compressed by a learnable MLP [87], yielding $\tilde{K}$ and $\tilde{V}$. Attention between $q_t$ and $\tilde{K}, \tilde{V}$ produces $O_t^{\text{cmp}}$. (2) Selection: top-$k$ scores from the compression branch select blocks to form a sparse mask $M_t$, producing sparse attention output $O_t^{\text{slc}}$. (3) Sliding window: local attention within $K_{t-w:t}, V_{t-w:t}$ of window length $w$ gives $O_t^{\text{win}}$. The three outputs are gated and aggregated into the final result $O_t$.

► **DSA.** DeepSeek Sparse Attention (DSA) [66] employs a lightning indexer to efficiently generate sparse masks. Under the MLA architecture [11], the hidden layer $H$ is projected into indexer query $Q^I$, key $K^I$ and weight $w^I$. Here, $Q^I$ has $h^I$ heads while $K^I$ is single headed, and both have head dimension $d^I < d$. The index score between two hidden tokens $h_t$ and $h_s$ is $I_{t,s} = \sum_{j=1}^{h^I} w_{t,j}^I \cdot \mathrm{ReLU}(q_{t,j}^I \cdot k_s^I)$. For each $h_t$, the top-k scores form a sparse mask $M_t$, which is shared across all heads of $Q$ and $K$. The indexer is efficient as it operates with fewer heads and smaller dimensions. It is trained independently before pre-training of the full model.

► **MoBA.** MoBA [67], inspired by the Mixture-of-Experts (MoE) mechanism [88], introduces a dynamic Mixture of Block Attention. It divides $K$ and $V$ into $B$ blocks and mean-pools each into a single token, forming compressed sequences $\tilde{K}$ and $\tilde{V}$ of length $B$. For each query $q_t$, attention scores over $\tilde{K}$ and $\tilde{V}$ select the top-$k$ blocks to construct a sparse mask $M_t$, on which sparse attention is computed. In the causal setting, the block containing the $t$-th token (i.e., the current block of $q_t$) is always included, and causal mask is applied. MoBA achieves up to 6.5× speedup over full attention on an 1M-token prefilling task, with comparable accuracy across benchmarks.

► **VSA.** Video Sparse Attention [74] is a trainable attention mechanism for video diffusion models. The video latent is first divided into $(4, 4, 4)$ spatio-temporal cubes, where coarse cube-to-cube attention identifies the Top-K cubes, forming a block-sparse attention mask. Next, token-level attention is computed within the selected blocks, and the outputs from both the cube-level and the token-level attention are combined via learnable gating to produce the final output. VSA is implemented with a block-sparse attention kernel, achieving 85% of FlashAttention3's MFU [8]. It accelerates attention computation by up to 6× on Wan2.1-1.3B.

### *E. Other Sparse Attention Methods.*

CHAI [69] clusters highly correlated attention heads (more so in later layers) and computes only representative heads while sharing results within clusters, cutting compute and KV memory with minimal loss. SampleAttn [47] combines a local window with a sampled global stripe pattern to form adaptive structured sparsity for prefilling, reducing latency than FlashAttention while retaining most accuracy. Twilight [76] adopts a select–then–prune scheme with row-wise top-$p$ thresholds to realize adaptive sparsity. FlexPrefill [59] performs context-aware dynamic sparse attention by selecting a pattern through

a Jensen–Shannon–divergence test and determining the sparsity ratio via row-wise top-$P$ to a target recall. MagicPig [71] proposes sampling-based sparse attention with an LSH-based approximation and a bias-corrected softmax, together with a CPU–GPU co-design for long contexts. VMoBA [68] extends MoBA to video by exploiting the 1D, 2D, and 3D locality of $K$ and $V$, and further introduces global top-$k$ and per-head cumulative thresholding. Quest [70] partitions the KV cache into fixed-size pages and maintains per-page attention bounds to estimate maximal attention. At runtime, it loads only the top-$k$ pages conditioned on the current query to perform sparse attention. RetrievalAttn [63] constructs an OOD-aware vector index [89] over the KV cache during prefilling and retrieves top-$K$ on the CPU during decoding, which is merged with local GPU attention for sub-$\mathcal{O}(n)$ complexity. Radial Attention [51] uses a fixed sparse mask in video diffusion transformers. It splits the attention map into time bands that grow wider away from the diagonal and keeps a small diagonal window in each frame pair. PAROAttention [54] reorders tokens to turn scattered visual attention into hardware-friendly block patterns, enabling efficient masking via a simple thresholded block-sum. It exploits pattern generality and uses offline static sparsity. MoA [48] extends the uniform sliding-window scheme of StreamingLLM [12] by assigning heterogeneous, input-length–dependent window sizes to different attention heads. Window lengths are tuned offline. XAttention [73] measures block importance via antidiagonal sums of the attention matrix and rearranges $Q/K$ to compute these efficiently in one product, achieving up to $13.5\times$ speedup with accuracy comparable to full attention. HashAttention [72] recasts token selection as MIPS: it learns linear projections to binary signatures and retrieves keys by Hamming distance, computing attention only on the retrieved subset with about tens of bits per token overhead. SVG2 [75] accelerates video generation via training-free sparse attention: cluster-based semantic permutation, centroid-driven top-$p$ block selection, and dynamic block-sparse kernels yield significant speedups.

## VII. Linear Attention

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

### A. Unified Framwork

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:

$$O = \phi(Q)(\phi(K)^\top V), \quad O, Q, K, V \in \mathbb{R}^{N\times d}. \tag{36}$$

where $\phi$ is a kernel function applied row-wise. In non-autoregressive tasks, computation follows Eq. 36. In autoregressive tasks, the causal mask reformulates the computation:

$$o_t = \phi(q_t)\sum_{i=1}^{t}(\phi(k_i)^\top v_i). \tag{37}$$

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 output is computed as follows:

$$H_t = H_{t-1} + \phi(k_t)^\top v_t, \quad o_t = \phi(q_t)H_t. \tag{38}$$

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 updated with gates can be formulated as:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot k_t^\top v_t. \tag{39}$$

Here we omit the kernel function $\phi$ 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 input, and input-independent otherwise.

Linear attention methods can be classified by their hidden state updating method. The first three categories rely on direct computation of $H_t$: **(1)** Naive Linear Attention: 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 $G_f^{(t)}$ 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. 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) [28], adopts an optimization-based approach. It treats the hidden state $H_t$, not as a computed value, but as learnable parameters known as fast weights [29] updated via gradient descent [30]. We will discuss naive linear attention in Section VII-D, linear attention with a forget gate in Section VII-E, linear attention with both forget and select gates in Section VII-F, and TTT in Section VII-G.

### B. Preliminaries of Linear Attention without Gates

As mentioned in Section III-D, there are three basic forms of linear attention: parallel form, recurrent form, and chunkwise form. In following content, we will discuss these forms in detail.

*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:

$$O = \phi(Q)\left(\phi(K)^\top V\right). \tag{40}$$

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. The output for a query $Q_n$ can be computed as:

$$O_n = \frac{\phi(Q_n)\sum_{j=1}^{N}\phi(K_j)^\top V_j}{\phi(Q_n)\sum_{j=1}^{N}\phi(K_j)^\top}. \tag{41}$$

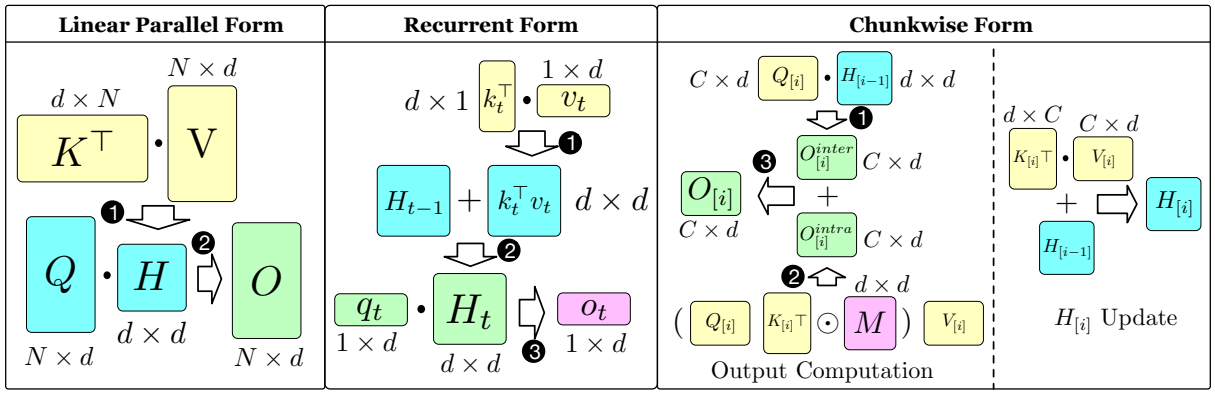


Fig. 3: Illustration of different computation forms for linear attention.

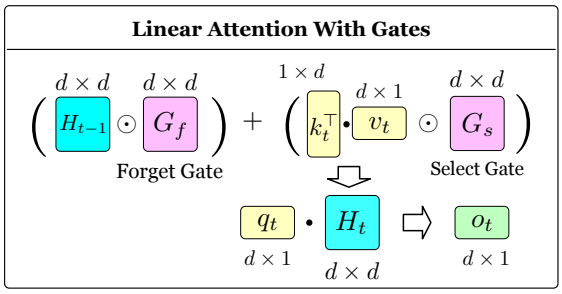


Fig. 4: Linear attention with forget and select gates.

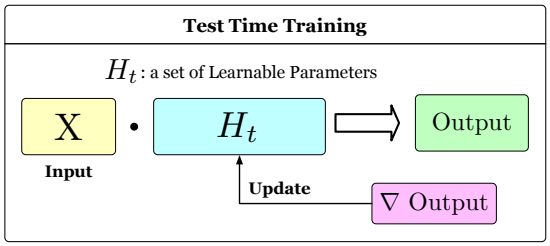


Fig. 5: Overview of test time training.

*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 a hidden state $H_t$:

$$H_t = H_{t-1} + \phi(k_t)^\top v_t, \quad o_t = \phi(q_t) H_t. \tag{42}$$

where $H_0$ is initialized as zeros. 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 $t$.

*3) Chunkwise Form for Parallel Autoregressive Training:* Autoregressive training faces a trade-off between efficiency and parallelism. The quadratic parallel form allows full parallelization but incurs $\mathcal{O}(N^2)$ cost. The purely recurrent form achieves $\mathcal{O}(N)$ complexity but is sequential and computationally inefficient on GPUs [90].

The chunkwise form provides a practical approach by dividing the sequence into fixed-size chunks. Each chunk is processed in parallel with an inter-chunk recurrent update. Formally, for the $i$-th chunk with a causal mask $D$:

$$O_{\text{intra},i} = (Q_{[i]} K_{[i]}^T \odot D) V_{[i]} \quad \text{(Intra-chunk Parallel)}, \tag{43}$$
$$O_{\text{inter},i} = Q_{[i]} H_{i-1} \quad \text{(Inter-chunk Recurrent)}, \tag{44}$$
$$O_i = O_{\text{intra},i} + O_{\text{inter},i} \quad \text{(Final Chunk Output)}, \tag{45}$$
$$H_i = K_{[i]}^T V_{[i]} + H_{i-1} \quad \text{(Next State Update)}. \tag{46}$$

This hybrid approach balances parallelism and causality, with a complexity of $\mathcal{O}(NCd + Nd^2)$ for chunk size $C$.

### C. Preliminaries of Linear Attention with Gates

We now elaborate on the three computational forms from Section VII-B2, explaining their implementation with gates.

*1) Recurrent Form:* The gated recurrent form is defined as:

$$H_t = G_f^{(t)} \odot H_{t-1} + G_s^{(t)} \odot k_t^\top v_t, \quad o_t = q_t H_t. \tag{47}$$

The forget gate $G_f^{(t)}$ and select gate $G_s^{(t)}$ control the retention of the previous state and incorporation of new information, respectively. And the output $o_t$ is computed by applying the current query $q_t$ to updated state $H_t$.

*2) Quadratic Parallel Form:* When using gates, considering the cumulative effect of the gates along the sequence is needed. The formulation is given by Eq. 48.

$$O = (\tilde{Q}\tilde{K}^\top \odot M)\tilde{V} \odot \Lambda, \tag{48}$$
$$\text{where} \quad \tilde{Q} = Q \odot \mathcal{A} \quad \tilde{K} = K / \mathcal{A} \odot \hat{A} \quad \tilde{V} = V / \Lambda \odot \hat{B}.$$

Here, the queries, keys, and values are pre-scaled by cumulative forget effects($\mathcal{A}$, $\Lambda$) and cumulative select effects($\hat{A}, \hat{B}$) to model causal and gated dependencies in parallel. However, its quadratic complexity makes training inefficient. Thus, modern architectures adopt the chunkwise form for better efficiency.

*3) Chunk-wise Formulation for Gated Linear Attention:* Incorporating the gates results in a more detailed chunk-wise formulation defined by Eq. 49a through Eq. 49d.

$$H_{[i]} = (\zeta_i \odot H_{[i-1]}) + \widehat{K}_{[i]}^\top \widehat{V}_{[i]}, \tag{49a}$$
$$O_{[i+1]} = O_{[i+1]}^{\text{Intra}} + O_{[i+1]}^{\text{Inter}}, \tag{49b}$$
$$O_{[i+1]}^{\text{Intra}} = (\tilde{Q}_{[i+1]} \tilde{K}_{[i+1]}^\top \odot M) \tilde{V}_{[i+1]} \odot \Lambda_{[i+1]}, \tag{49c}$$
$$O_{[i+1]}^{\text{Inter}} = (Q_{[i+1]} \odot A_{[i+1]}^\dagger) H_{[i]} \odot B_{[i]}^\dagger. \tag{49d}$$

The chunk-level hidden state $H_{[i]}$ is updated by Eq. 49a, where $\zeta_i$ represents the cumulative forgetfulness of the previous chunk.

$\widehat{K}_{[i]}$ and $\widehat{V}_{[i]}$ are aggregated from keys and values, where each key value pair is first modulated by selecting gate $(\hat{a}_t, \hat{b}_t)$ and then decayed by remaining forget gates within the chunk.

The $(i+1)$-th chunk output, $O_{[i+1]}$, is composed of two parts: an intra-chunk output $O^{\text{Intra}}_{[i+1]}$, computed within current chunk using quadratic parallel form, and an inter-chunk output $O^{\text{Inter}}_{[i+1]}$, which incorporates historical information from the previous chunk via the recurrent state $H_{[i]}$. $A^{\dagger}_{[i+1]}$ and $B^{\dagger}_{[i]}$ apply the right decay based on the relative positions of queries.

### *D. Naive Linear Attention*

When both forget gate $G_f$ and select gate $G_s$ are $\mathbf{1}^\top\mathbf{1}$, the method is viewed as Naive Linear Attention. Table V shows some typical methods. The listed complexities are for training. In this table, all methods use a recurrent form for inference, with $\mathcal{O}(Nd^2)$ time complexity and $\mathcal{O}(Nd)$ space complexity.

► **Linear Transformer.** Linear Transformer [14] decomposes the softmax$(QK^\top)$ into a kernelized dot product $\phi(Q)\phi(K)^\top$ and utilizes matrix product associativity to achieve a linear complexity. The attention output for a single query $q_i$ is:

$$o_i = \frac{\phi(q_i)^\top \left(\sum_{j=1}^{N} \phi(k_j) v_j^\top\right)}{\phi(q_i)^\top \left(\sum_{j=1}^{N} \phi(k_j)\right)}. \tag{50}$$

By first computing $\phi(K)^\top V$(i.e.,$\sum_{j=1}^{N} \phi(k_j) v_j^\top$), the complexity becomes $\mathcal{O}(Nd^2)$. For autoregressive tasks, this method keeps two state matrices, $H_t$ and $Z_t$, and is formulated as:

$$H_t = H_{t-1} + \phi(k_t)v_t^\top, Z_t = Z_{t-1} + \phi(k_t),\ o_t = \frac{\phi(q_t)^\top H_t}{\phi(q_t)^\top Z_t}.$$

This update reduces the memory complexity from $\mathcal{O}(Nd)$ to $\mathcal{O}(d^2)$ during inference, avoiding growing KV cache.

► **LightningAttention.** Lightning Attention [92] adopts the chunkwise form for hardware-efficient implementation, formulated as follows:

$$H_t = H_{t-1} + K_t^\top V_t, \quad O_t^{\text{intra}} = \left((Q_t K_t^\top) \odot M\right) V_t. \tag{51}$$

$$O_t^{\text{inter}} = Q_t H_{t-1}, \quad O_t = O_{\text{intra}} + O_{\text{inter}}. \tag{52}$$

The attention is divided into intra-chunk and inter-chunk computations: $O_t^{\text{intra}}$ is computed using the quadratic parallel form, while $O_t^{\text{inter}}$ aggregates information from previous blocks. $H_t$ is recurrently updated in shared memory to incorporate historical data from preceding chunks. By loading only the current blocks $(Q_t, K_t, V_t)$ during each iteration, the system ensures constant training speed and memory footprint.

### *E. Linear Attention with a Forget Gate*

A method is categorized as linear attention with a forget gate if $G_f^{(t)} \neq \mathbf{1}^\top\mathbf{1}$ while $G_s^{(t)} = \mathbf{1}^\top\mathbf{1}$. Table VI shows some representative methods of this type. The listed complexities are for training. In the table, all methods use the recurrent form for inference.

► **RetNet.** RetNet [94] introduces a forget gate $\gamma\mathbf{1}^\top\mathbf{1}$ and uses a chunkwise form for efficient training. For inference, it uses the recurrent form: $H_t = \gamma H_{t-1} + k_t^\top v_t, o_t = q_t H_t$. The decay factor $\gamma$ controls how much of the past information is retained at each step. The intra-chunk output is calculated in quadratic parallel form, $O_{[i]}^{\text{intra}} = (QK^\top \odot D)V$, where $D_{nm} = \gamma^{n-m}$ for $n \geq m$ and $D_{nm} = 0$ otherwise. The matrix $D$ combines causal masking with an exponential decay based on relative distance between tokens. The chunkwise form is:

$$H_{[i]} = K_{[i]}^\top (V_{[i]} \odot \zeta) + \gamma^C H_{i-1}, \text{where } \zeta_{i,j} = \gamma^{C-i-1}, \tag{53}$$

$$O_{[i+1]}^{\text{inter}} = (Q_{[i]} H_{i-1}) \odot \xi, \quad \text{where } \xi_{i,j} = \gamma^{i+1}, \tag{54}$$

$$O_{[i+1]} = O_{[i+1]}^{\text{intra}} + O_{[i+1]}^{\text{inter}}. \tag{55}$$

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) [95] introduces an input-dependent, vector-valued forget gate and a hardware-efficient implementation. The gate, $\alpha_t \in (0,1)^{d_k}$, is computed dynamically from input token $x_t$, allowing different feature dimensions to decay at different rates. 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 $\alpha_t$ is generated via a low-rank linear layer, followed by a sigmoid activation and a scaling term $\tau$: $\alpha_t = \sigma\left(x_t W_\alpha^1 W_\alpha^2 + b_\alpha\right)^{\frac{1}{\tau}}$. 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 $\Gamma$ and $\Lambda$ are derived from the per-token $\alpha_t$ values and manage the decay across chunks:

$$H_{[i+1]} = (\gamma_{i+1}^\top 1) \odot H_{[i]} + (K_{[i+1]} \odot \Gamma_{[i+1]})^\top V_{[i+1]}, \tag{56}$$

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

GLA also provides an I/O-aware implementation with two variants: (1) The non-materialization version processes chunks sequentially, keeping the hidden state $H_{[n]}$ in fast shared memory for high memory efficiency ($O(Nd)$) 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, enabling parallel computation of all chunks' outputs at an $O(Nd^2/C + Nd)$ space complexity.

► **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 [96] can be formulated as:

$$H_t = e^{-w} \odot H_{t-1} + e^{k_t} \odot v_t, \tag{57}$$

$$H_t^{'} = e^{-w} \odot H_{t-1}^{'} + e^{k_t}, \quad o_t = \sigma(q_t) \odot (H_t / H_t^{'}). \tag{58}$$

TABLE V: Summary of naive linear attention methods.

| Method | Forget Gate | Select Gate | Time Complexity | Space Complexity | Form |
|---|---|---|---|---|---|
| Linear transformer [14] | $\mathbf{1}^\top\mathbf{1}$ | $\mathbf{1}$ | $\mathcal{O}(N^2d)$ | $\mathcal{O}(N^2)$ | quadratic |
| CHELA [91] | $\mathbf{1}^\top\mathbf{1}$ | $\mathbf{1}$ | $\mathcal{O}(N^2d)$ | $\mathcal{O}(N^2)$ | quadratic |
| Lightning Attention [92] | $\mathbf{1}^\top\mathbf{1}$ | $\mathbf{1}$ | $\mathcal{O}(NCd+Nd^2)$ | $\mathcal{O}(Nd)$ | chunkwise |
| SLAB [93] | $\mathbf{1}^\top\mathbf{1}$ | $\mathbf{1}$ | $\mathcal{O}(N^2d)$ | $\mathcal{O}(N^2)$ | 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.

TABLE VI: Summary of linear attention with forget gate only.

| Method | Forget Gate | Select Gate | Time Complexity | Space Complexity | Form |
|---|---|---|---|---|---|
| Retnet [94] | $\gamma\mathbf{1}^\top\mathbf{1}$ | $\mathbf{1}$ | $O(N^2d)$ | $O(N^2)$ | quadratic |
| | | | $O(NCd+Nd^2)$ | $O(Nd^2/C+Nd)$ | chunkwise |
| GLA [95] | $\alpha_t^\top\mathbf{1}$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C+Nd)$ | chunkwise |
| RWKV4 [96] | $e^{-w}$ | $\mathbf{1}$ | $O(Nd)$ | $O(d)$ | parallel |
| RWKV5 [97] | $w^\top\mathbf{1}$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C)$ | chunkwise |
| RWKV6 [97] | $w_t^\top\mathbf{1}$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C)$ | chunkwise |
| RWKV7 [98] | $(\mathrm{diag}(w_t)-\kappa_t(a_t\odot\kappa_t))^*$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C)$ | chunkwise |
| GSA [99] | $\alpha_t^\top\mathbf{1}$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C+Nd)$ | chunkwise |
| MetaLA [100] | $\alpha_t^\top\mathbf{1}$ | $\mathbf{1}$ | $O(NCd+Nd^2)$ | $O(Nd^2/C+Nd)$ | chunkwise |

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

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 [97] 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: $H_t = H_{t-1}diag(w) + v_t^\top k_t$, where $diag(w)$ is a diagonal input-independent forget gate. RWKV-6 [97] changes the forget gate into an input-dependent one: $H_t = H_{t-1}diag(w_t) + v_t^\top k_t$.

**RWKV-7.** RWKV-7 [98] further enhances the time-mixing module's expressiveness with a more general forget gate. The state update is defined as: $H_t = (diag(w_t) - \kappa_t(\alpha_t \odot \kappa_t))H_{t-1} + v_t^\top k_t$. Here, $\kappa_t = k_t \odot \epsilon, \alpha_t = \mathrm{sigmoid}(\mathrm{MLP}(\mathrm{x_t}))$, and $\epsilon$ 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.

### *F. Linear Attention with Forget and Select Gates*

Table VII 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 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 $\mathcal{O}(Nd)$.

► **DeltaNet.** DeltaNet [90] improves the standard linear attention's additive update rule ($H_t = H_{t-1} + v_t k_t^\top$), addressing the "key collision" problem that degrades memory performance when the sequence length exceeds the key dimension. It uses an error-correcting delta rule to update $H_t$ by minimizing the error between the predicted value ($H_{t-1}k_t$) and the target value ($v_t$). The update can be expressed in two forms:

$$H_t = H_{t-1} - \beta_t(H_{t-1}k_t - v_t)k_t^\top, \tag{59}$$

$$H_t = H_{t-1} - v_t^{\text{old}}k_t^\top + v_t^{\text{new}}k_t^\top. \tag{60}$$

Here, $\beta_t$ is a learnable, input-dependent "writing strength". $H_t$ is updated by deleting the old value's influence ($v_t^{\text{old}}$) and writing back the new value's ($v_t^{\text{new}}$). This rule combines forgetting and selection. However, the algorithm is sequential, leading to inefficiency on parallel processors like GPUs.

► **Mamba.** Mamba [102] enhances structured state space models (SSMs) [105] with a selection mechanism, making them content-aware. Unlike the Transformer architecture, Mamba directly maps the input $x_t$ to the output $y_t$ without using the Query-Key-Value (QKV) mechanism. The core idea of Mamba is a time-varying linear recurrence that updates the hidden state $h_t$ based on the input $x_t$:

$$h_t = \overline{\mathbf{A}}_t h_{t-1} + \overline{\mathbf{B}}_t x_t, \quad y_t = \mathbf{C}_t h_t, \tag{61}$$

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

TABLE VII: Summary of linear attention methods with forget gate and select gate.

| Method | Forget Gate | Select Gate | Time Complexity | Space Complexity | Form |
|---|---|---|---|---|---|
| Deltanet [90] | $\mathbf{I}-\beta_t k_t k_t^\top$ | $\beta_t$ | $\mathcal{O}(Nd^2)$ | $\mathcal{O}(d^2)$ | sequential |
| Deltanet2 [101] | $\mathbf{I}-\beta_t k_t k_t^\top$ | $\beta_t$ | $\mathcal{O}(NCd+Nd^2)$ | $\mathcal{O}(d)$ | chunkwise |
| Mamba [102] | $\exp(\Delta A)$ | $(\Delta A)^{-1}(\exp(\Delta A)-I)\cdot\Delta B$ | $\mathcal{O}(Nd)$ | $\mathcal{O}(Nd)$ | parallel scan |
| gDeltanet [103] | $\alpha_t(\mathbf{I}-\beta_t k_t k_t^\top)$ | $\beta_t$ | $\mathcal{O}(NCd+Nd^2)$ | $\mathcal{O}(d)$ | chunkwise |
| Mamba2 [104] | $A_t$ | $\mathbf{1}$ | $\mathcal{O}(Nd^2)$ | $\mathcal{O}(Nd)$ | 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.

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 dynamically manage information flow by selectively focusing on or ignoring inputs. For efficient implementation, Mamba uses a special form called parallel scan. This method allows Mamba to dynamically adjust its focus on or ignore parts of the input sequence.

► **Mamba2.** Mamba2 [104] builds upon the original Mamba by introducing the structured State Space Duality (SSD) framework, which reveals a connection between SSMs and attention. Its recurrent formulation is: $h_t=\mathbf{A}_t h_{t-1}+\mathbf{B}_t x_t, y_t=\mathbf{C}_t h_t$. Mamba2 achieves efficient computation through block decomposition: it uses a quadratic, attention-like computation for diagonal blocks and 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.

### G. Test Time Training

LLM can compress a massive training set into weights through self-supervised learning [9], 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) [28] 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: $g=\sum_{i=1}^{C}\nabla\mathcal{L}(f_{H_{t-1}}(k_t),v_t)$. Then the update rule of $H_t$ is $H_t=H_{t-1}-g$. 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 [28] updates the hidden states using gradient-based rules. TTT further develops two variants of $f_{H_t}$, namely TTT-Linear and TTT-MLP. The $f_{H_t}$ of TTT-MLP is a two-layer MLP with a nonlinear activation GELU [106] 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 [95]. The time and space complexity of TTT-Linear aligns with chunk-wise linear attention methods.

► **Titans** . Titans [107] shares the same motivation as test time training but differs in its update rule for $H_t$, where it further introduces momentum and a forget gate: $H_t=(1-\alpha_t)H_{t-1}+M_t, M_t=\eta_t M_{t-1}-\theta_t g$. Here $\alpha_t\in[0,1]$ is forget gate, $\eta_t$ is the weight decay and $\theta_t$ 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 [108] proposes a hardware-friendly method to make the test-time training method more efficient on GPUs. Compared with TTT [28] and Titans [107], LaCT [108] introduces a much larger chunk size of $4096\sim 1$ million tokens and makes 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 [109] 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: $f_{H_{t-1}}(k_t)=W_2[\mathrm{SiLU}(W_1k_t)\odot W_3k_t]$. 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 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 [110].

### H. Other Linear Attention Methods

CHELA [91] leverages a tiling strategy for a hardware-efficient linear attention implementation and combines a short-long convolutions module before linear attention to stabilize training, enabling effective long-range sequence modeling with linear complexity. SLAB [93] uses a ReLU [111] kernel to compute linear attention, and adds it with the output of a parallel depth-wise convolution [112] branch to capture local contextual information. SANA [113] applies naive linear attention to text-to-image tasks by replacing the quadratic self-attention of DiT [78] with linear attention [14]. GSA [114] employs a two-pass gated linear attention [95] with an input-dependent forget gate, where queries and keys are processed in the first pass, and the softmax-normalized result is used as the query in the second pass to aggregate values. MetaLA [100] omits the theoretically unnecessary Key matrix $(K)$ for parameter efficiency, while incorporating self-augmentation and short convolution to enhance modeling capabilities. DeltaNet2 [101] improves the

efficiency of DeltaNet [90] by introducing a memory-efficient reparameterization of the delta rule recurrence, allowing for chunkwise training. gDeltaNet [103] introduces the gated delta rule, combining Mamba2's [104] gating with DeltaNet's [90] delta rule, achieving more flexible memory management. LogLinear Attention (LLA) [115] keeps several hidden states whose count grows only with the logarithm of the sequence length. It splits the past into $O(\log t)$ buckets (finer for recent, coarser for older) via a Fenwick tree [116] and outputs a weighted sum over these buckets, giving an implicit forget effect with $\mathcal{O}(N \log N)$ training and $\mathcal{O}(\log N)$ decoding.

## VIII. Limitations and Opportunities

We analyze the limitations and opportunities for the four categories of efficient attention methods:

First, hardware-efficient attention methods face several limitations: (1) They are difficult to develop, because they require strong GPU programming skills and detailed knowledge of GPU hardware. (2) Their speedups are fundamentally bounded by the peak compute capability of the device, so there is a clear upper limit on the possible gains. (3) Most existing implementations are heavily tailored to NVIDIA GPUs, while other accelerators, such as NPUs, TPUs, and AMD GPUs, receive much less support. Therefore, future directions include: (1) developing simple toolkits that make it easy to write custom kernels; and (2) porting and tuning hardware-efficient attention methods for a broader range of hardware backends.

Second, compact attention methods introduce a trade-off between KV-cache efficiency and model capacity. By compressing the KV states, they reduce the representational diversity, which can harm multi-step reasoning and long-range recall performance of models. Promising future directions include importance-aware KV allocation at the token, head, and layer levels, as well as training-time distillation from a full-MHA teacher to better preserve model quality under compression.

Third, for sparse attention methods, the main limitation is that it is challenging to push sparsity very high, as large sparsity often hurts end-to-end accuracy. A promising direction is trainable sparse attention, such as NSA [65], VSA [74], and SLA [55], where the model is trained or fine-tuned to adapt to sparse attention while maintaining high accuracy at high attention sparsity.

Fourth, for linear attention methods, while computationally efficient, they exhibit significant performance degradation and underperform standard softmax attention, especially in long-context modeling. The limitation suggests promising research directions, including enhancing the hidden state update rules (e.g., with more powerful forget and selection gates) and developing hybrid methods that combine it with softmax or sparse attention, such as sparse-linear attention [55].

## IX. Conclusion

In this survey, we present a unified and comprehensive view of efficient attention methods. We propose to categorize existing methods into four classes: hardware-efficient attention, sparse attention, compact attention, and linear attention methods. Each class targets a different aspect of the attention bottleneck, such as low-level computation and memory I/O, attention sparsity, KV cache memory usage, or the quadratic time complexity of standard attention. For each class, we provide a unified formulation, a common analysis framework, and a consolidated summary with detailed descriptions of representative methods. Finally, we analyze the limitations and opportunities of each class of efficient attention methods and outline promising directions for future research.

## References


[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16344–16359, 2022.

[3] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *The Twelfth International Conference on Learning Representations*, 2024.

[4] J. Zhang, J. Wei, P. Zhang, J. Zhu, and J. Chen, "Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration," in *International Conference on Learning Representations (ICLR)*, 2025.

[5] J. Zhang, H. Huang, P. Zhang, J. Wei, J. Zhu, and J. Chen, "Sageattention2: Efficient attention with thorough outlier smoothing and per-thread int4 quantization," *arXiv preprint arXiv:2411.10958*, 2024.

[6] J. Zhang, J. Wei, P. Zhang, X. Xu, H. Huang, H. Wang, K. Jiang, J. Zhu, and J. Chen, "Sageattention3: Microscaling fp4 attention for inference and an exploration of 8-bit training," *arXiv preprint arXiv:2505.11594*, 2025.

[7] J. Zhang, X. Xu, J. Wei, H. Huang, P. Zhang, C. Xiang, J. Zhu, and J. Chen, "Sageattention2++: A more efficient implementation of sageattention2," *arXiv preprint arXiv:2505.21136*, 2025.

[8] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[9] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[10] J. Ainslie, J. Lee-Thorp, M. De Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[11] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, *et al.*, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[12] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," in *The Twelfth International Conference on Learning Representations*, 2024.

[13] J. Zhang, C. Xiang, H. Huang, J. Wei, H. Xi, J. Zhu, and J. Chen, "Spargeattn: Accurate sparse attention accelerating any model inference," *arXiv preprint arXiv:2502.18137*, 2025.

[14] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International conference on machine learning*, pp. 5156–5165, PMLR, 2020.

[15] M.-H. Guo, T.-X. Xu, J.-J. Liu, Z.-N. Liu, P.-T. Jiang, T.-J. Mu, S.-H. Zhang, R. R. Martin, M.-M. Cheng, and S.-M. Hu, "Attention mechanisms in computer vision: A survey," *Computational visual media*, vol. 8, no. 3, pp. 331–368, 2022.

[16] S. Chaudhari, V. Mithal, G. Polatkan, and R. Ramanath, "An attentive survey of attention models," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 12, no. 5, pp. 1–32, 2021.

[17] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016.

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

[19] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[20] NVIDIA, "Cutlass: Cuda templates for linear algebra subroutines and solvers," 2025. Accessed: 2025-08-17.

[21] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[22] M. V. Koroteev, "Bert: a review of applications in natural language processing and understanding," *arXiv preprint arXiv:2103.11943*, 2021.

[23] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, *et al.*, "A survey on vision transformer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 1, pp. 87–110, 2022.

[24] F. Bao, S. Nie, K. Xue, Y. Cao, C. Li, H. Su, and J. Zhu, "All are worth words: A vit backbone for diffusion models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 22669–22679, 2023.

[25] S. W. Williams, A. Waterman, and D. A. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," tech. rep., Technical Report UCB/EECS-2008-134, EECS Department, University of . . . , 2008.

[26] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[27] L. R. Medsker, L. Jain, *et al.*, "Recurrent neural networks," *Design and applications*, vol. 5, no. 64-67, p. 2, 2001.

[28] Y. Sun, X. Li, K. Dalal, J. Xu, A. Vikram, G. Zhang, Y. Dubois, X. Chen, X. Wang, S. Koyejo, *et al.*, "Learning to (learn at test time): Rnns with expressive hidden states," *arXiv preprint arXiv:2407.04620*, 2024.

[29] I. Schlag, K. Irie, and J. Schmidhuber, "Linear transformers are secretly fast weight programmers," in *International conference on machine learning*, pp. 9355–9366, PMLR, 2021.

[30] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[31] T. Dao, D. Haziza, F. Massa, and G. Sizov, "Flash-decoding for long-context inference." https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023. [Online].

[32] C. Hooper, S. Kim, H. Mohammadzadeh, M. W. Mahoney, Y. S. Shao, K. Keutzer, and A. Gholami, "KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization," *arXiv e-prints*, 2024.

[33] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu, "Kivi: A tuning-free asymmetric 2bit quantization for kv cache," in *International Conference on Machine Learning*, pp. 32332–32344, PMLR, 2024.

[34] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. Hao Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," *arXiv e-prints*, 2023.

[35] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, *et al.*, "Flashinfer: Efficient and customizable attention engine for llm inference serving," *arXiv preprint arXiv:2501.01005*, 2025.

[36] NVIDIA, "Nvidia h100 tensor core gpu architecture," 2022.

[37] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[38] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, *et al.*, "Sglang: Efficient execution of structured language model programs," *Advances in neural information processing systems*, vol. 37, pp. 62557–62583, 2024.

[39] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019.

[40] J. Hu, H. Li, Y. Zhang, Z. Wang, S. Zhou, X. Zhang, H.-Y. Shum, and D. Jiang, "Multi-matrix factorization attention," *arXiv preprint arXiv:2412.19255*, 2024.

[41] Y. Zhang, Y. Liu, H. Yuan, Z. Qin, Y. Yuan, Q. Gu, and A. C. Yao, "Tensor product attention is all you need," *arXiv preprint arXiv:2501.06425*, 2025.

[42] B. J. Broxson, "The kronecker product," 2006.

[43] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[44] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[45] A. Kazemnejad, I. Padhi, K. Natesan Ramamurthy, P. Das, and S. Reddy, "The impact of positional encoding on length generalization in transformers," *Advances in Neural Information Processing Systems*, vol. 36, pp. 24892–24928, 2023.

[46] Z. Yuan, H. Zhang, L. Pu, X. Ning, L. Zhang, T. Zhao, S. Yan, G. Dai, and Y. Wang, "DiTFastattn: Attention compression for diffusion transformer models," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[47] Q. Zhu, J. Duan, C. Chen, S. Liu, X. Li, G. Feng, X. Lv, H. Cao, X. Chuanfu, X. Zhang, D. Lin, and C. Yang, "Sampleattention: Near-lossless acceleration of long context llm inference with adaptive structured sparse attention," 2024.

[48] T. Fu, H. Huang, X. Ning, G. Zhang, B. Chen, T. Wu, H. Wang, Z. Huang, S. Li, S. Yan, *et al.*, "Moa: Mixture of sparse attention for automatic large language model compression," *arXiv preprint arXiv:2406.14909*, 2024.

[49] G. Xiao, J. Tang, J. Zuo, J. Guo, S. Yang, H. Tang, Y. Fu, and S. Han, "Duoattention: Efficient long-context llm inference with retrieval and streaming heads," in *The International Conference on Learning Representations*, 2025.

[50] H. Xi, S. Yang, Y. Zhao, C. Xu, M. Li, X. Li, Y. Lin, H. Cai, J. Zhang, D. Li, *et al.*, "Sparse videogen: Accelerating video diffusion transformers with spatial-temporal sparsity," *arXiv preprint arXiv:2502.01776*, 2025.

[51] X. Li, M. Li, T. Cai, H. Xi, S. Yang, Y. Lin, L. Zhang, S. Yang, J. Hu, K. Peng, M. Agrawala, I. Stoica, K. Keutzer, and S. Han, "Radial attention: $\mathcal{O}(n \log n)$ sparse attention with energy decay for long video generation," *arXiv preprint arXiv:2506.19852*, 2025.

[52] P. Zhang, Y. Chen, R. Su, H. Ding, I. Stoica, Z. Liu, and H. Zhang, "Fast video generation with sliding tile attention," *arXiv preprint arXiv:2502.04507*, 2025.

[53] A. Hassani, S. Walton, J. Li, S. Li, and H. Shi, "Neighborhood attention transformer," in *Proceedings of CVPR*, pp. 6185–6194, June 2023.

[54] T. Zhao, K. Hong, X. Yang, X. Xiao, H. Li, F. Ling, R. Xie, S. Chen, H. Zhu, Y. Zhang, *et al.*, "Paroattention: Pattern-aware reordering for efficient sparse and quantized attention in visual generation models," *arXiv preprint arXiv:2506.16054*, 2025.

[55] J. Zhang, H. Wang, K. Jiang, S. Yang, K. Zheng, H. Xi, Z. Wang, H. Zhu, M. Zhao, I. Stoica, *et al.*, "Sla: Beyond sparsity in diffusion transformers via fine-tunable sparse-linear attention," *arXiv preprint arXiv:2509.24006*, 2025.

[56] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, *et al.*, "H2o: Heavy-hitter oracle for efficient generative inference of large language models," *Advances in Neural Information Processing Systems*, vol. 36, pp. 34661–34710, 2023.

[57] C. Xiao, P. Zhang, X. Han, G. Xiao, Y. Lin, Z. Zhang, Z. Liu, and M. Sun, "Infllm: Training-free long-context extrapolation for llms with an efficient context memory," in *Advances in Neural Information Processing Systems* (A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, eds.), vol. 37, pp. 119638–119661, Curran Associates, Inc., 2024.

[58] H. Jiang, Y. LI, C. Zhang, Q. Wu, X. Luo, S. Ahn, Z. Han, A. H. Abdi, D. Li, C.-Y. Lin, Y. Yang, and L. 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.

[59] X. Lai, J. Lu, Y. Luo, Y. Ma, and X. Zhou, "Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference," in *The Thirteenth International Conference on Learning Representations*, 2025.

[60] L. Ribar, I. Chelombiev, L. Hudlass-Galley, C. Blake, C. Luschi, and D. Orr, "Sparq attention: Bandwidth-efficient LLM inference," in *Forty-first International Conference on Machine Learning*, 2024.

[61] P. Singhania, S. Singh, S. He, S. Feizi, and A. Bhatele, "Loki: Low-rank keys for efficient sparse attention," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[62] Y. Gao, Z. Zeng, D. Du, S. Cao, H. K.-H. So, T. Cao, F. Yang, and M. Yang, "Seerattention: Learning intrinsic sparse attention in your llms," *arXiv preprint arXiv:2410.13276*, 2024.

[63] D. Liu, M. Chen, B. Lu, H. Jiang, Z. Han, Q. Zhang, Q. Chen, C. Zhang, B. Ding, K. Zhang, *et al.*, "Retrievalattention: Accelerating long-context llm inference via vector retrieval," *arXiv preprint arXiv:2409.10516*, 2024.

[64] A. Liu, Z. Zhang, Z. Li, X. Bai, Y. Han, J. Tang, Y. Xing, J. Wu, M. Yang, W. Chen, J. He, Y. He, F. Wang, G. Haffari, and B. Zhuang, "Fpsattention: Training-aware fp8 and sparsity co-design for fast video diffusion," 2025.

[65] J. Yuan, H. Gao, D. Dai, J. Luo, L. Zhao, Z. Zhang, Z. Xie, Y. X. Wei, L. Wang, Z. Xiao, Y. Wang, C. Ruan, M. Zhang, W. Liang, and W. Zeng, "Native sparse attention: Hardware-aligned and natively trainable sparse attention," 2025.

[66] DeepSeek-AI, "Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention," 2025.

[67] E. Lu, Z. Jiang, J. Liu, Y. Du, T. Jiang, C. Hong, S. Liu, W. He, E. Yuan, Y. Wang, Z. Huang, H. Yuan, S. Xu, X. Xu, G. Lai, Y. Chen, H. Zheng, J. Yan, J. Su, Y. Wu, Y. Zhang, Z. Yang, X. Zhou, M. Zhang, and J. Qiu, "Moba: Mixture of block attention for long-context llms," *arXiv preprint arXiv:2502.13189*, 2025.

[68] J. Wu, L. Hou, H. Yang, X. Tao, Y. Tian, P. Wan, D. Zhang, , and Y. Tong, "Vmoba: Mixture-of-block attention for video diffusion models," *arXiv preprint arXiv:2506.23858*, 2025.

[69] S. Agarwal, B. Acun, B. Hosmer, M. Elhoushi, Y. Lee, S. Venkataraman, D. Papailiopoulos, and C.-J. Wu, "Chai: clustered head attention for efficient llm inference," in *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, JMLR.org, 2024.

[70] J. Tang, Y. Zhao, K. Zhu, G. Xiao, B. Kasikci, and S. Han, "Quest: query-aware sparsity for efficient long-context llm inference," ICML'24, JMLR.org, 2024.

[71] Z. Chen, R. Sadhukhan, Z. Ye, Y. Zhou, J. Zhang, N. Nolte, Y. Tian, M. Douze, L. Bottou, Z. Jia, *et al.*, "Magicpig: Lsh sampling for efficient llm generation," *arXiv preprint arXiv:2410.16179*, 2024.

[72] A. Desai, S. Yang, A. Cuadron, M. Zaharia, J. E. Gonzalez, and I. Stoica, "Hashattention: Semantic sparsity for faster inference," 2025.

[73] R. Xu, G. Xiao, H. Huang, J. Guo, and S. Han, "XAttention: Block sparse attention with antidiagonal scoring," in *Forty-second International Conference on Machine Learning*, 2025.

[74] P. Zhang, H. Huang, Y. Chen, W. Lin, Z. Liu, I. Stoica, E. Xing, and H. Zhang, "Vsa: Faster video diffusion with trainable sparse attention," *arXiv preprint arXiv:2505.13389*, 2025.

[75] S. Yang, H. Xi, Y. Zhao, M. Li, J. Zhang, H. Cai, Y. Lin, X. Li, C. Xu, K. Peng, *et al.*, "Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation," *arXiv preprint arXiv:2505.18875*, 2025.

[76] C. Lin, J. Tang, S. Yang, H. Wang, T. Tang, B. Tian, I. Stoica, S. Han, and M. Gao, "Twilight: Adaptive attention sparsity with hierarchical top-$p$ pruning," *arXiv preprint arXiv:2502.02770*, 2025.

[77] Y. Deng, Z. Song, and C. Yang, "Attention is naturally sparse with gaussian distributed input," *arXiv preprint arXiv:2404.02690*, 2024.

[78] W. Peebles and S. Xie, "Scalable diffusion models with transformers," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

[79] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[80] J. Ho and T. Salimans, "Classifier-free diffusion guidance," *arXiv preprint arXiv:2207.12598*, 2022.

[81] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

[82] A. Hassani, S. Walton, J. Li, S. Li, and H. Shi, "Natten: Neighborhood attention extension." https://github.com/SHI-Labs/NATTEN, 2023.

[83] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[84] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. 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.

[85] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[86] J. M. Joyce, "Kullback-leibler divergence," in *International encyclopedia of statistical science*, pp. 720–722, Springer, 2011.

[87] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[88] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.

[89] M. Chen, K. Zhang, Z. He, Y. Jing, and X. S. Wang, "Roargraph: A projected bipartite graph for efficient cross-modal approximate nearest neighbor search," *Proc. VLDB Endow.*, 2024.

[90] I. Schlag, K. Irie, and J. Schmidhuber, "Linear transformers are secretly fast weight programmers," in *International conference on machine learning*, pp. 9355–9366, PMLR, 2021.

[91] Z. Liu, S. Li, L. Wang, Z. Wang, Y. Liu, and S. Z. Li, "Short-long convolutions help hardware-efficient linear attention to focus on long sequences," *arXiv preprint arXiv:2406.08128*, 2024.

[92] Z. Qin, W. Sun, D. Li, X. Shen, W. Sun, and Y. Zhong, "Various lengths, constant speed: Efficient language modeling with lightning attention," *arXiv preprint arXiv:2405.17381*, 2024.

[93] J. Guo, X. Chen, Y. Tang, and Y. Wang, "Slab: Efficient transformers with simplified linear attention and progressive re-parameterized batch normalization," *arXiv preprint arXiv:2405.11582*, 2024.

[94] Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei, "Retentive network: A successor to transformer for large language models," *arXiv preprint arXiv:2307.08621*, 2023.

[95] S. Yang, B. Wang, Y. Shen, R. Panda, and Y. Kim, "Gated linear attention transformers with hardware-efficient training," *arXiv preprint arXiv:2312.06635*, 2023.

[96] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, S. Biderman, H. Cao, X. Cheng, M. Chung, M. Grella, *et al.*, "Rwkv: Reinventing rnns for the transformer era," *arXiv preprint arXiv:2305.13048*, 2023.

[97] B. Peng, D. Goldstein, Q. Anthony, A. Albalak, E. Alcaide, S. Biderman, E. Cheah, X. Du, T. Ferdinan, H. Hou, *et al.*, "Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence," *arXiv preprint arXiv:2404.05892*, 2024.

[98] B. Peng, R. Zhang, D. Goldstein, E. Alcaide, X. Du, H. Hou, J. Lin, J. Liu, J. Lu, W. Merrill, *et al.*, "Rwkv-7" goose" with expressive dynamic state evolution," *arXiv preprint arXiv:2503.14456*, 2025.

[99] Y. Zhang, S. Yang, R.-J. Zhu, Y. Zhang, L. Cui, Y. Wang, B. Wang, F. Shi, B. Wang, W. Bi, *et al.*, "Gated slot attention for efficient linear-time sequence modeling," *Advances in Neural Information Processing Systems*, vol. 37, pp. 116870–116898, 2024.

[100] Y. Chou, M. Yao, K. Wang, Y. Pan, R. 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*, vol. 37, pp. 71034–71067, 2024.

[101] S. Yang, B. Wang, Y. Zhang, Y. Shen, and Y. Kim, "Parallelizing linear transformers with the delta rule over sequence length," *arXiv preprint arXiv:2406.06484*, 2024.

[102] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023.

[103] S. Yang, J. Kautz, and A. Hatamizadeh, "Gated delta networks: Improving mamba2 with delta rule," *arXiv preprint arXiv:2412.06464*, 2024.

[104] T. Dao and A. Gu, "Transformers are ssms: Generalized models and efficient algorithms through structured state space duality," *arXiv preprint arXiv:2405.21060*, 2024.

[105] A. Gu, K. Goel, and C. Ré, "Efficiently modeling long sequences with structured state spaces," *arXiv preprint arXiv:2111.00396*, 2021.

[106] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[107] A. Behrouz, P. Zhong, and V. Mirrokni, "Titans: Learning to memorize at test time," *arXiv preprint arXiv:2501.00663*, 2024.

[108] T. Zhang, S. Bi, Y. Hong, K. Zhang, F. Luan, S. Yang, K. Sunkavalli, W. T. Freeman, and H. Tan, "Test-time training done right," *arXiv preprint arXiv:2505.23884*, 2025.

[109] N. Shazeer, "Glu variants improve transformer," *arXiv preprint arXiv:2002.05202*, 2020.

[110] K. Jordan, Y. Jin, V. Boza, Y. Jiacheng, F. Cecista, L. Newhouse, and J. Bernstein, "Muon: An optimizer for hidden layers in neural networks, 2024," *URL https://kellerjordan. github. io/posts/muon*, vol. 6.

[111] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *International Conference on Machine Learning*, 2010.

[112] F. 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.

[113] E. Xie, J. Chen, J. Chen, H. Cai, H. Tang, Y. Lin, Z. Zhang, M. Li, L. Zhu, Y. Lu, *et al.*, "Sana: Efficient high-resolution text-to-image synthesis with linear diffusion transformers," in *The Thirteenth International Conference on Learning Representations*, 2025.

[114] Y. Zhang, S. Yang, R. Zhu, Y. Zhang, L. Cui, Y. Wang, B. Wang, F. Shi, B. Wang, W. Bi, *et al.*, "Gated slot attention for efficient linear-time sequence modeling. 2024," *URL https://api. semanticscholar. org/CorpusID*, vol. 272593079.

[115] H. Guo, S. Yang, T. Goel, E. P. Xing, T. Dao, and Y. Kim, "Log-linear attention," *arXiv preprint arXiv:2506.04761*, June 2025. Submitted on 5 Jun 2025 (v1), last revised 25 Jun 2025 (v2).

[116] P. M. Fenwick, "A new data structure for cumulative frequency tables," *Software: Practice and Experience*, vol. 24, no. 3, pp. 327–336, 1994.