All You Need To Know About The Self-Attention Layer

Damien Benveniste
The AiEdge
damienb@theaiedge.io

Abstract

All You Need to Know About the Self-Attention Layer offers a single, tutorial-style roadmap from the Self-attention mechanism's origins to today's efficiency breakthroughs. After deriving vanilla multi-head self-attention, we survey five families of optimizations:

- **Sparse patterns** – prune the N^2 attention graph to achieve sub-quadratic complexity.
- **Linear kernels** – factorize or approximate the softmax to reach $O(N)$, or even $O(1)$ memory.
- **Memory-centric designs** – trade arithmetic for bandwidth, unlocking 16K-token contexts on current GPUs.
- **Decoding-time tweaks** – shrink the KV-cache and raise generation throughput.
- **Long-sequence architectures** – extend context to the million-token regime by mixing recurrence or external memory.

Each technique is dissected mathematically and paired with hardware considerations. This introduction to the self-attention mechanism offers practitioners clear guidance for assembling efficient attention stacks.

Contents

1 Introduction

The *"Attention Is All You Need"* paper (Vaswani et al., 2017 [32]) is one of the most influential works in modern AI. By replacing recurrence with self-attention mechanisms, the authors introduced the Transformer architecture, a design that enabled parallelized training, captured long-range dependencies in data, and scaled effortlessly to unprecedented model sizes. This innovation not only rendered RNNs obsolete but also laid the groundwork for BERT[12], GPT[25], and the modern LLM revolution, powering breakthroughs from conversational AI to protein folding.

One of the earliest glimmers of what would become self-attention appeared more than two decades before the Transformer. In 1992, Jürgen Schmidhuber's seminal paper *"Learning to Control Fast-Weight Memories"* [26] showed that a neural network could write a short-lived matrix of associations, fast weights, at every time-step. Schmidhuber's fast-weight mechanism already promised the features that would later make attention famous: constant-time access to distant context, flexible variable binding, and the ability to capture long-range dependencies without laborious back-propagation through many recurrent steps.

The first papers to name and systematically exploit attention within a single sequence appeared in 2016. Cheng, Dong, and Lapata's *"Long Short-Term Memory-Networks for Machine Reading"* [5] replaced the fixed LSTM cell with a small memory network and let each token form attention weights over earlier tokens, calling the mechanism intra-attention. In parallel, Parikh et al.'s

"Decomposable Attention Model" [22] for natural language inference showed that sentence pairs could be compared token-wise without any recurrent structure, relying entirely on pairwise alignment scores that are computed in parallel. These works demonstrated that letting tokens look at one another could rival or surpass recurrent encoders on language understanding tasks, paving the way for recurrence-free architectures.

Just months before Vaswani et al., Lin et al.'s *"A Structured Self-Attentive Sentence Embedding"* [20] introduced what is effectively a multi-head self-attention layer: several independent attention maps are learned in parallel, each capturing a different aspect of the sentence. The model discards recurrence entirely at the representation stage and delivers state-of-the-art results on sentiment and entailment, offering a convincing demonstration that stacked attention alone can learn rich linguistic features.

When the Transformer finally appeared, it fused Schmidhuber's associative fast-weight reading, the external-memory alignment of Memory Networks, the intra-sentence focus of 2016 *"self-attention,"* and the multi-head formulation of Lin et al. into a single, fully parallelisable module. By interpreting the entire previous sequence as a writable/readable cache, rather than as a chain of recurrent states, the Transformer inherited the constant-time retrieval, long-range expressiveness, and data-parallel efficiency first hinted at by its precursors, and scaled them to unprecedented model sizes.

The original Transformer architecture opened a whole new avenue of research, but its self-attention mechanism introduces significant computational bottlenecks as sequence lengths grow. This quadratic complexity $\mathcal{O}(N^2)$ in both computation and memory usage makes processing long contexts with thousands of tokens increasingly prohibitive, limiting the practical utility of these powerful models. In this chapter, we explore a diverse ecosystem of innovative approaches that tackle these fundamental constraints from multiple angles:

- Sparse attention patterns that strategically prune connections between tokens, maintaining information pathways while reducing computational demands
- Linear attention mechanisms that mathematically reformulate the attention operation to achieve $\mathcal{O}(N)$ scaling through kernel methods and low-rank approximations
- Memory-efficient implementations that optimize hardware utilization at the CUDA kernel level, greatly reducing memory bandwidth bottlenecks
- Specialized decoding attention variants that accelerate inference through key-value sharing and latent representations
- Long-sequence architectures that enable virtually unlimited context through recurrence, retrieval, and constant-memory representations

These enhancements address the core limitations of vanilla attention, making Transformers more scalable, expressive, and practical for today's demanding language modeling tasks. Modern Transformer architectures rarely use the original blueprint without incorporating one or more of these optimization strategies.

2 The Vanilla Self-Attention

2.1 The Architecture

In the Transformer, the self-attention captures the token interactions within the sequences. It is composed of three linear layers: W^K, W^Q, and W^V. The input vectors to the attention layer are the internal hidden states $\mathbf{h}_i$ resulting from the model inputs. There are as many hidden states as tokens in the input sequence, and $\mathbf{h}_i$ corresponds to the i^{th} token. W^K, W^Q and W^V project the incoming hidden states into the so-called keys $\mathbf{k}_i$, queries $\mathbf{q}_i$ and values $\mathbf{v}_i$:

$$\begin{aligned} \mathbf{k}_i &= W^K \mathbf{h}_i, \quad \text{keys} \\ \mathbf{q}_i &= W^Q \mathbf{h}_i, \quad \text{queries} \\ \mathbf{v}_i &= W^V \mathbf{h}_i, \quad \text{values} \end{aligned} \tag{1}$$

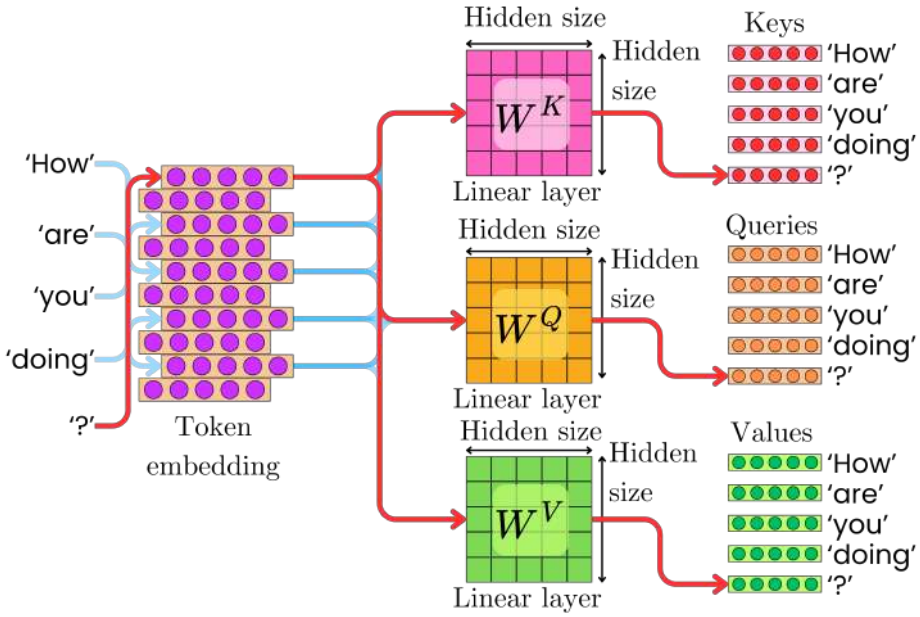

Fig. 1 *W^K, W^Q, and W^V are used to project the hidden states into keys, queries, and values.*

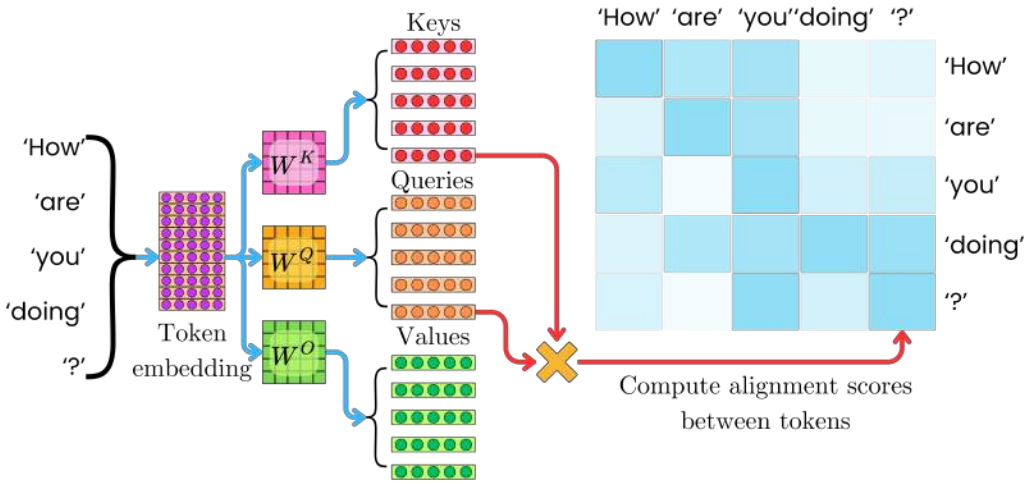

Fig. 2 *The keys and queries are used to compute the alignment scores.*

The keys and queries are used to compute the alignment scores:

$$e_{ij} = \frac{\mathbf{k}_i^\top \mathbf{q}_j}{\sqrt{d_{\text{model}}}} \tag{2}$$

e_{ij} is the alignment score between the i^{th} word and the j^{th} word in the input sequence. d_{model} is the common naming convention for the hidden size:

$$|\mathbf{h}_i| = |\mathbf{k}_i| = |\mathbf{q}_i| = |\mathbf{v}_i| = d_{\text{model}} = \text{Hidden size} \tag{3}$$

The scaling factor $\sqrt{d_{\text{model}}}$ in the scaled dot-product is used to counteract the effect of the dot product's magnitude growing with the dimensionality d_{model}, which stabilizes gradients and ensures numerical stability during training. It is common to represent those operations as matrix multiplications. With the matrix $K = [\mathbf{k}_1, \ldots, \mathbf{k}_N]$ and $Q = [\mathbf{q}_1, \ldots, \mathbf{q}_N]$, we have:

$$E = \frac{Q^\top K}{\sqrt{d_{\text{model}}}} \tag{4}$$

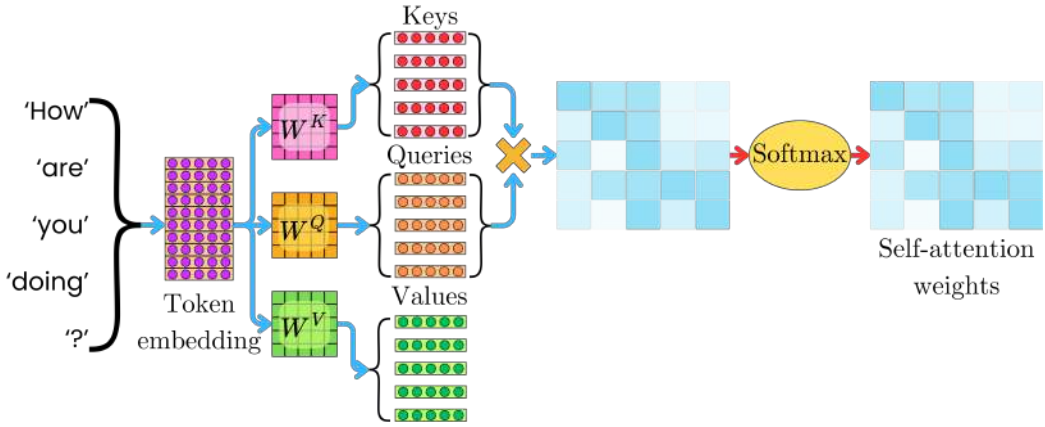

Fig. 3 *The attention weights are the result of normalizing the alignment scores by using the softmax transformation.*

or:

$$E = \frac{1}{\sqrt{d_{\text{model}}}} \begin{bmatrix} \mathbf{q}_1^\top \mathbf{k}_1 & \mathbf{q}_1^\top \mathbf{k}_2 & \cdots & \mathbf{q}_1^\top \mathbf{k}_N \\ \mathbf{q}_2^\top \mathbf{k}_1 & \mathbf{q}_2^\top \mathbf{k}_2 & \cdots & \mathbf{q}_2^\top \mathbf{k}_N \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{q}_N^\top \mathbf{k}_1 & \mathbf{q}_N^\top \mathbf{k}_2 & \cdots & \mathbf{q}_N^\top \mathbf{k}_N \end{bmatrix} \tag{5}$$

with N being the number of tokens in the sequence.

As for the other attentions, the alignment scores are normalized to 1 through a Softmax transformation:

$$a_{ij} = \text{Softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{j=1}^{N} \exp(e_{ij})} \tag{6}$$

where a_{ij} is the attention weight between the tokens i and j, quantifying how strongly the model should attend to token j when processing token i. Because we have $\sum_{j=1}^{N} a_{ij} = 1$, a_{ij} can be interpreted as the probability that token j is relevant to token i.

The attention weights are used to compute a weighted average of the values vectors:

$$\mathbf{c}_i = \sum_{j=1}^{N} a_{ij} \mathbf{v}_j \tag{7}$$

In the jargon used in the previous chapter, $\mathbf{c}_i$ are the context vectors coming out of the attention layer, but we can think of them as another intermediary set of hidden states within the network. Using the more common matrix notation, we have:

$$C = AV^\top \tag{8}$$

where $V = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$, $C = [\mathbf{c}_1, \ldots, \mathbf{c}_N]$ and $A = \text{Softmax}(E)$ is the matrix of attention weights.

The whole set of computations happening in the attention layer can be summarized as the following equation:

$$C = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_{\text{model}}}}\right) V \tag{9}$$

The names "queries," "keys," and "values" are inspired by information retrieval systems (such as databases or search engines). Each token generates a query, key, and value to "retrieve" relevant context from other tokens. The model learns to search for relationships between tokens dynamically.

The queries represent what the current token is "asking for." For example, the word *"it"* in *"The cat sat because it was tired,"* the query seeks antecedents (e.g., *"cat"*). The keys represent what other

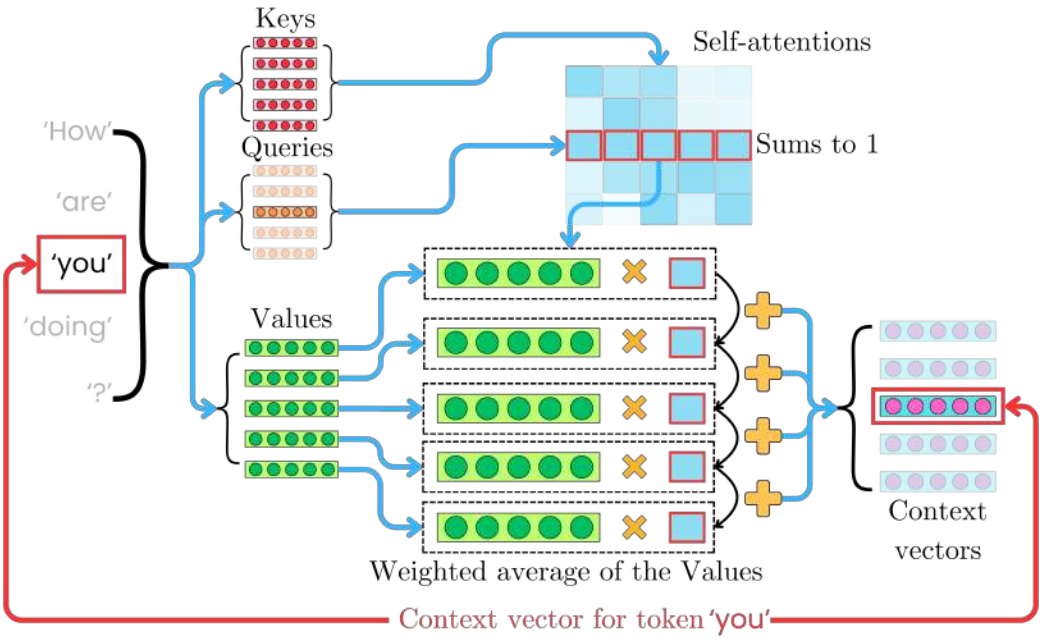

Fig. 4 *Each context vector is the result of a weighted average of the value vectors by using the attention weights.*

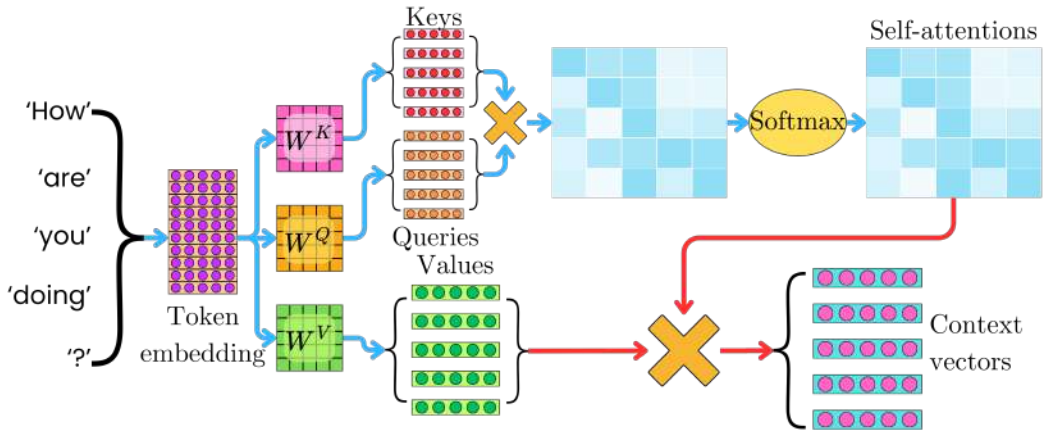

Fig. 5 *The entire attention layer process.*

tokens "offer" as context. In our example, the key for *"cat"* signals it is a candidate antecedent for *"it."* The values are the actual content to aggregate based on attention weights. The value for *"cat"* encodes its contextual meaning (e.g., entity type, role in the sentence, ...). For each query (current token), the model "retrieves" values (context) by comparing the query to all keys (other tokens). For example, let us consider the sentence:

> ***"The bank is steep, so it's dangerous to stand near it."***

- Query (*"it"*): "What does *'it'* refer to?"
- Keys (*"bank," "steep," "dangerous"*): Highlight candidates for reference.
- Values: Encode the meaning of each candidate.

The model computes high attention weights between the query (*"it"*) and keys (*"bank," "steep"*), then aggregates their values to infer *"it"* refers to the riverbank.

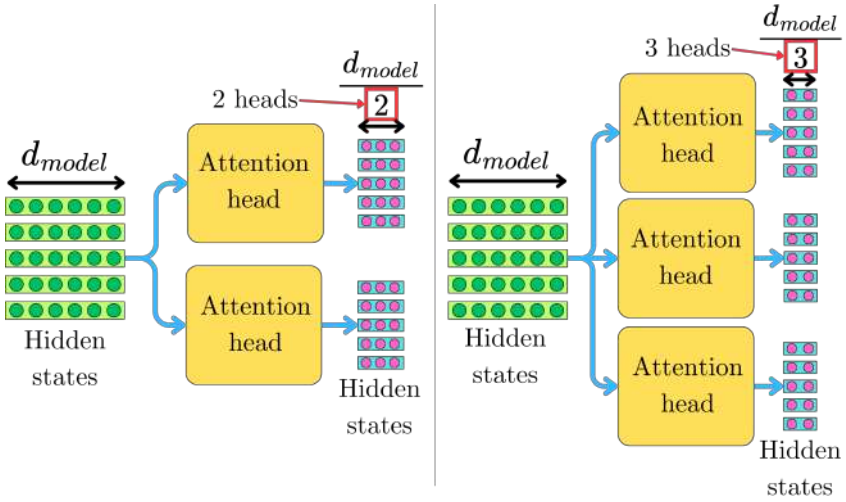

Fig. 6 *Each attention generates vectors of size $d_{head} = \frac{d_{model}}{n_{head}}$, depending of the number of heads.*

2.2 The Multi-head Attention Layer

We have talked about self-attention so far, but we use the so-called multi-head attention layer in the transformer architecture. The multi-head attention layer works as multiple parallel attention mechanisms. By having multiple attention layers in parallel, they will be able to learn various interaction patterns between the different tokens in the input sequence. Combining those will lead to more heterogeneous learning, and we will be able to learn richer information from the input sequence. Think about the multi-head attention layer as an ensemble of self-attentions, a bit like the random forest is an ensemble of decision tree models.

We call "heads" the parallel attention mechanisms. To ensure that the time complexity of the computations remains independent of the number of attention heads, we need to reduce the size of the internal vectors within the layers. The hidden size dimensionality per head is divided by the number of heads:

$$d_{\text{head}} = \frac{d_{\text{model}}}{n_{\text{head}}} \tag{10}$$

where n_{head} is the number of heads. This implies that the hidden size has to be chosen so that it is divisible by the number of heads.

Let us call $H = [\mathbf{h}_1, \ldots, \mathbf{h}_N]$ the incoming hidden states. Each head h generates resulting hidden states H'_h of size $\frac{d_{\text{model}}}{n_{\text{head}}}$:

$$H'_h = \text{Attention}_h(H) \tag{11}$$

To combine those heads' hidden states, we concatenate them, and we pass them through a final linear layer W^O to mix the signals coming from the different heads:

$$H' = \text{Concat}(H'_1, \ldots, H'_{n_{\text{head}}})W^O \tag{12}$$

To generate smaller hidden states, we need to reduce the dimensionality of the internal matrices. In each head, the projection matrices W^K, W^Q, and W^V take vectors of size d_{model} and generate vectors of size $\frac{d_{\text{model}}}{n_{\text{head}}}$.

2.3 The Tensor Representation

Although the information we have described so far about the multi-head attention layer is accurate, there is a critical subtlety to understand when it comes to its implementation. To illustrate the mathematical properties of the attention heads, we pictured separate "boxes" where each attention mechanism evolved in parallel, but in reality, they are slightly more connected. To fully utilize the

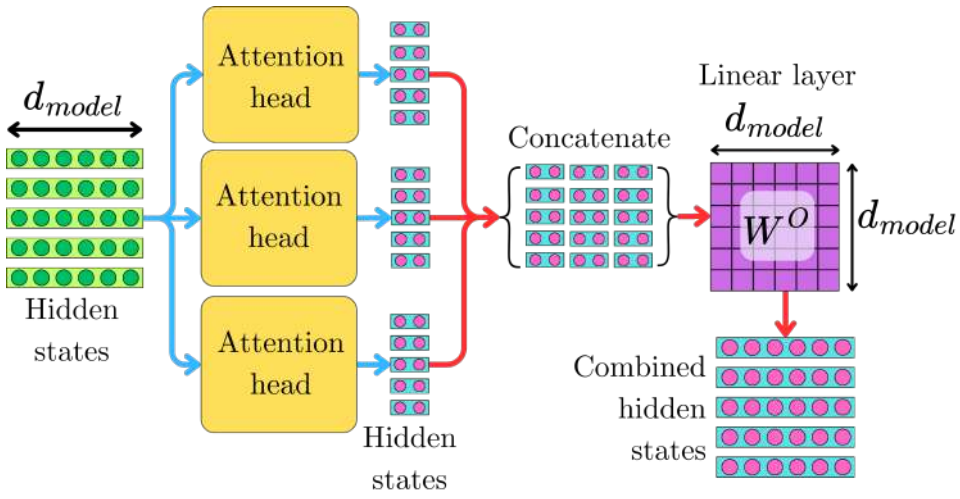

Fig. 7 *The result of each head is concatenated, and the signals are further mixed by a final linear layer W_O.*

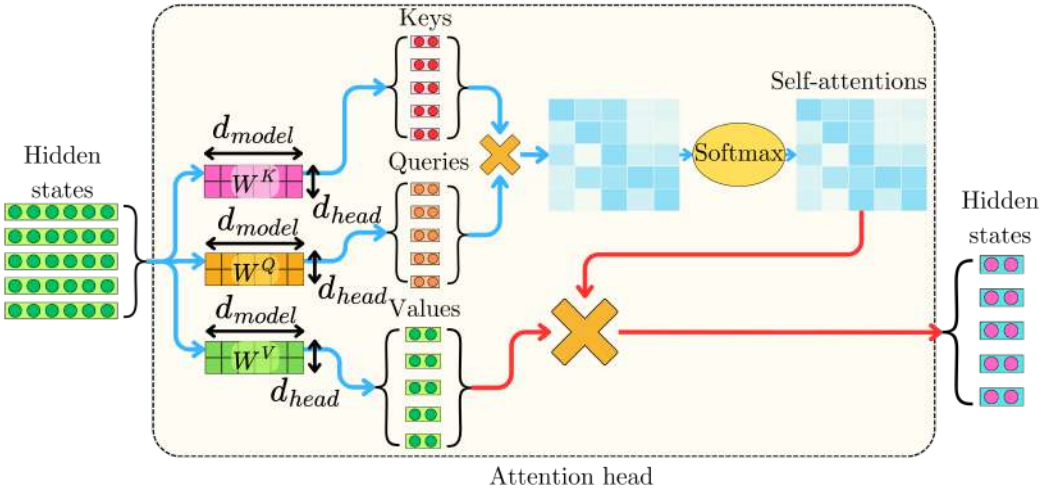

Fig. 8 *To generate smaller context vectors, we underlying projection matrices W^K, W^Q, and W^V need to be of size $d_{model} \times d_{head}$ for each head.*

efficient parallelization capability of the GPU hardware, it is critical to rethink every operation as a tensor operation. We described W^K, W^Q, and W^V of each head as separate matrices, but in practice, it is just three matrices that we conceptually break down by the number of heads needed.

Similarly, there is only one set of keys, queries, and values, and each head processes the entire sequence of tokens but operates on a distinct subset of features. The keys, queries, and values have dimension $d_{\text{model}} \times N$ where N is the number of tokens in the input sequence. To specify each head's sub-segment explicitly, we reshape the matrices into 3-dimensional tensors with dimension $n_{\text{head}} \times d_{\text{head}} \times N$. Let us consider the incoming set of the hidden states. It is first projected into keys, queries, and values:

$$K = W^K H, \quad \text{shape: } d_{\text{model}} \times N \tag{13}$$

$$Q = W^Q H, \quad \text{shape: } d_{\text{model}} \times N \tag{14}$$

$$V = W^V H, \quad \text{shape: } d_{\text{model}} \times N \tag{15}$$

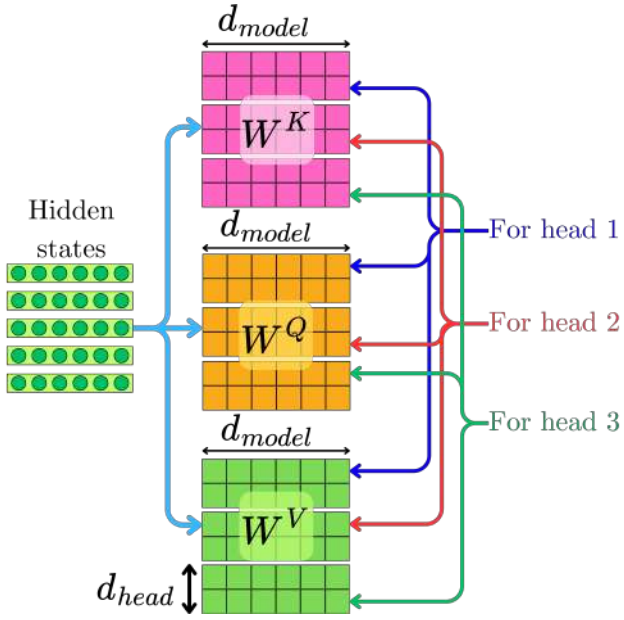

Fig. 9 *In reality, the projection matrices are not spread across multiple heads, but different sections of the matrices handle the projections for the different heads.*

We then reshape the resulting matrices into 3-dimensional tensors:

$$K' = \text{Reshape}(K), \quad \text{shape: } n_{\text{head}} \times d_{\text{head}} \times N \tag{16}$$

$$Q' = \text{Reshape}(Q), \quad \text{shape: } n_{\text{head}} \times d_{\text{head}} \times N \tag{17}$$

$$V' = \text{Reshape}(V), \quad \text{shape: } n_{\text{head}} \times d_{\text{head}} \times N \tag{18}$$

Reshaping is computationally efficient as it only reorganizes the tensor dimensions. When we compute the alignment scores E' from new tensors, this leads to $N \times N$ score for each head:

$$E' = \frac{Q' K'^{\top}}{\sqrt{d_{\text{head}}}}, \quad \text{shape: } n_{\text{head}} \times N \times N \tag{19}$$

Here, we use the shorthand notation $K'^{\top}$ to streamline the notation and imply permutation on the last two indices of the tensor, similar to the transpose operation for matrices:

$$k'_{ikj} = k'^{\top}_{ijk}, \quad \text{shape: } n_{\text{head}} \times N \times d_{\text{head}} \tag{20}$$

where k'_{ijk} is an element of K'. Notice that the way the operations are performed ensures the computation of $N \times N$ attention weights per head while keeping the number of arithmetic operations constant compared to the vanilla attention layer. The attention weights A' are obtained by normalizing on the last dimension:

$$a'_{ijk} = \text{Softmax}(e'_{ijk}) = \frac{\exp(e'_{ijk})}{\sum_{m=1}^{N} \exp(e'_{ijm})}, \quad \text{shape: } n_{\text{head}} \times N \times N \tag{21}$$

again, e'_{ijk} is an element of the tensor E' and a'_{ijk} of the tensor A'. The context vectors are computed as the weighted average of the values with the attention weights:

$$c'_{ijl} = \sum_{k=1}^{N} a'_{ijk} v'_{ilk}, \quad \text{shape: } n_{\text{head}} \times d_{\text{head}} \times N \tag{22}$$

or in tensor notation:

$$C' = A' V'^{\top}, \quad \text{shape: } n_{\text{head}} \times d_{\text{head}} \times N \tag{23}$$

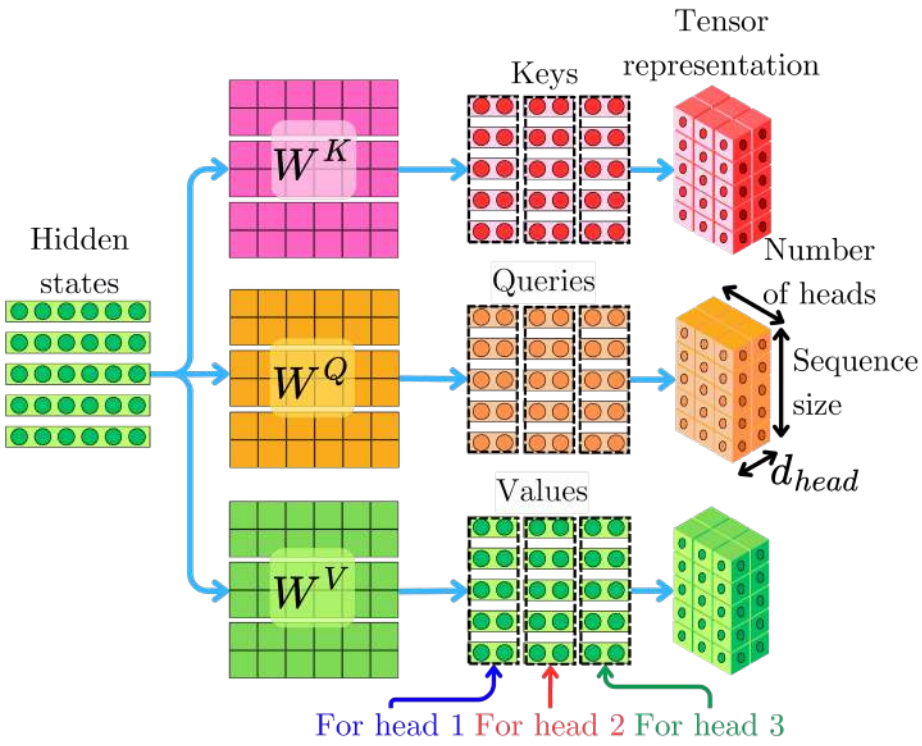

Fig. 10 *The keys, queries, and values are reshaped into 3-dimensional tensors with dimension $n_{head} \times d_{head} \times N$, where each slice of the tensors corresponds to one head.*

At this point, we have N context vectors of size d_{head} per head. We can reshape this tensor such that we have N context vectors of size $d_{\text{model}} = n_{\text{head}} d_{\text{head}}$:

$$C = \text{Reshape}(C'), \quad \text{shape: } d_{\text{model}} \times N \tag{24}$$

We described this earlier as the concatenation of the different heads' context vectors. As a way to combine further the signal coming from the different heads, we pass the resulting context vectors through a final linear layer:

$$C_{\text{final}} = W^O C, \quad \text{shape: } d_{\text{model}} \times N \tag{25}$$

This approach lets the model process information more efficiently than sequential methods, making it better at understanding both nearby and far-apart relationships in the data.

3 Sparse Attention Mechanisms

In the original Transformer, each query token attends to all tokens in the sequence (including itself), resulting in $\mathcal{O}(N^2)$ time and memory complexity for a sequence of length N. As context windows grow into the thousands or tens of thousands of tokens, this quadratic scaling becomes impractical, consuming excessive memory and computational resources.

Sparse attention addresses this bottleneck by restricting, or "sparsifying", which tokens can attend to which. Instead of forming attention connections from every token to every other token, sparse mechanisms allow each token to attend to a subset of the sequence according to a specific pattern. By reducing the total number of key/value pairs, sparse attention can often achieve $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$ complexity.

3.1 The First Sparse Attention: Sparse Transformers

One of the first attempts at sparse attention was proposed by OpenAI in 2019 with the *Sparse Transformers* [6], and this was one of the strategies chosen to scale GPT-3[4]. The idea is to limit the

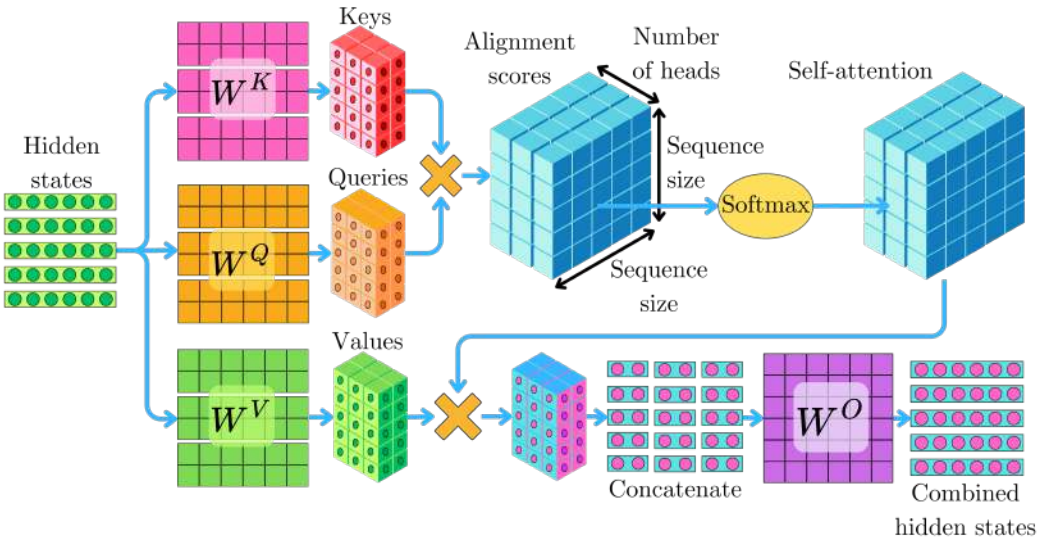

Fig. 11 *The computation of the attention and context vectors across multiple heads happens in parallel by making use of efficient tensor operations for GPU computing.*

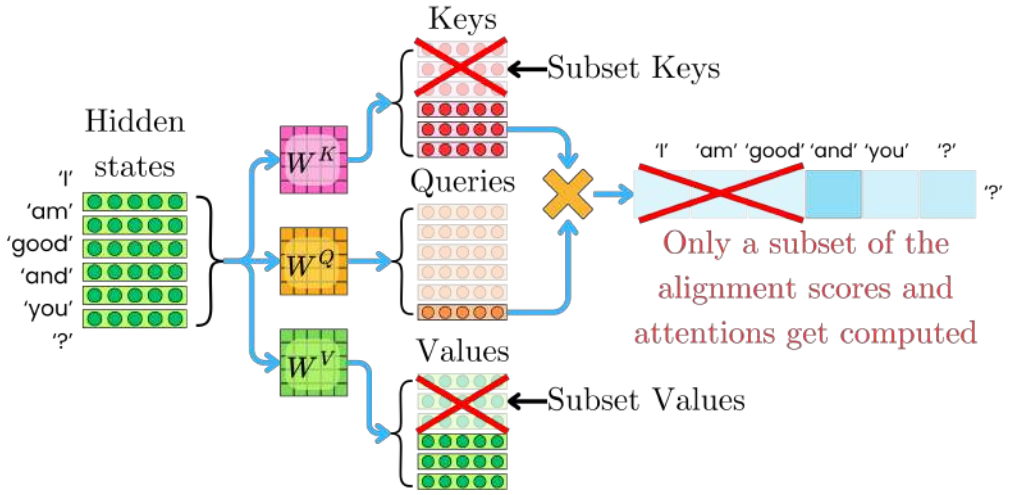

Fig. 12 *A Sparse attention consists of a vanilla self-attention where the key-value pairs are filtered to focus on those capturing the most information in the sequence.*

number of keys each query can attend when computing the alignment scores (see fig. 12). This will reduce the number of alignment scores and attention weights computed. Because of this, we also need to subset the values to ensure we can compute the context vectors $C = AV^\top$.

OpenAI suggested two different sparse patterns. The first pattern is the *strided* pattern (see fig. 13). One head focuses on a local window of nearby tokens by having the i-th query only attending the keys in $[i - w, i]$, where w is the window size. For example, if $w = 64$, it means we only select the keys $[i-64, i-63, \ldots, i-1, i]$. The other heads focus on more global token interactions by having the i-th query attending every c key. c is the stride and can be different for each head. For example, if $c = 8$, then we would only select the keys $[0, \ldots, i - 24, i - 16, i - 8, i]$.

They observed that the strided pattern (attending every k-th position) did not work well for text data, which lacks a naturally periodic structure. As a result, they introduced a *fixed* attention pattern. In one attention head, the sequence is divided into fixed-size blocks (e.g., 128 tokens). Each token within a block attends only to other tokens in that same block, capturing local dependencies in a more straightforward manner (see fig. 14).

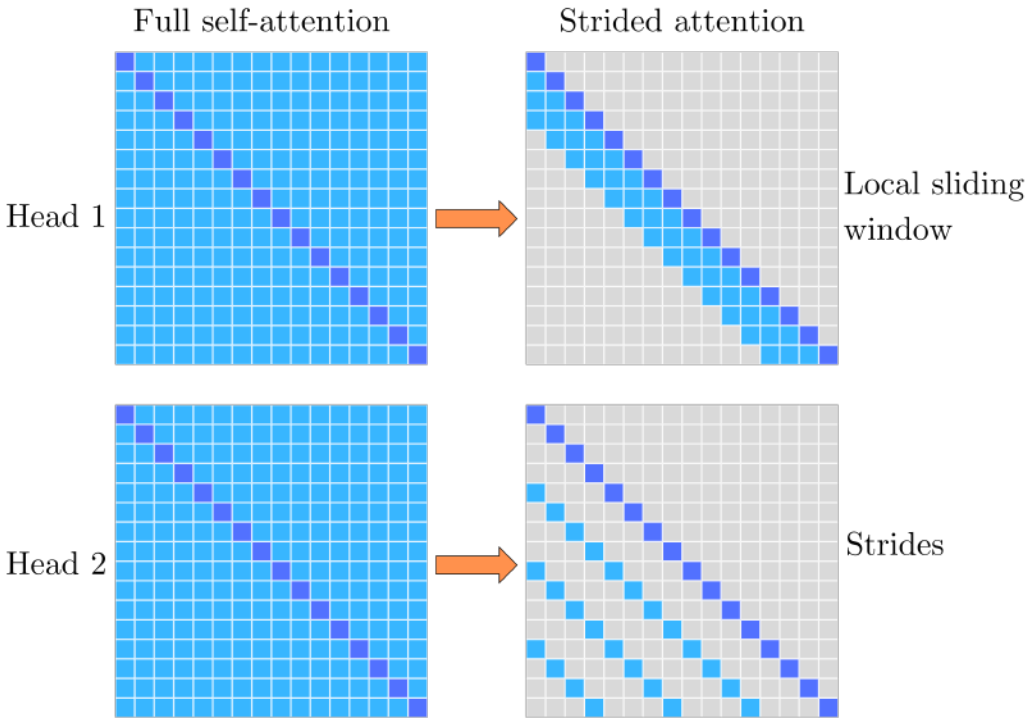

Fig. 13 *On the left, we represent the typical full self-attention where all the queries interact with all the keys. On the right, we represent the strided pattern described in the Sparse Transformers. Different attention heads express different sparse query-key interaction patterns.*

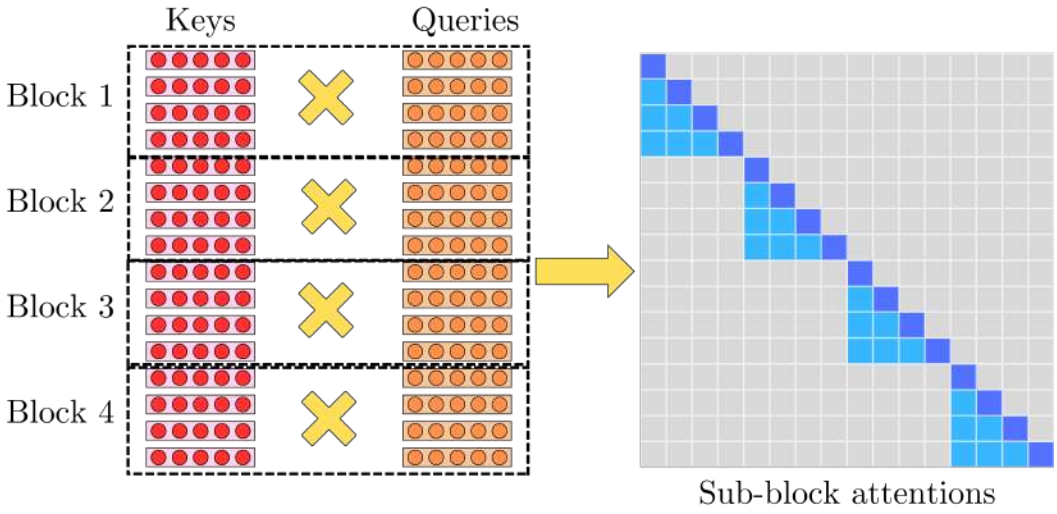

Fig. 14 *In the fixed pattern, some of the heads follow the per-block attention computation. Key-query pairs are grouped together, and the attention is only computed within those blocks.*

However, using purely local attention inside blocks would prevent information from flowing across blocks in deeper layers (see fig. 15). To address this, another head connects the "summary token" at the end of each block to the corresponding summary tokens of all previous blocks (see fig. 16). Because that last token has attended to all tokens in its own block, its hidden state acts as a summary

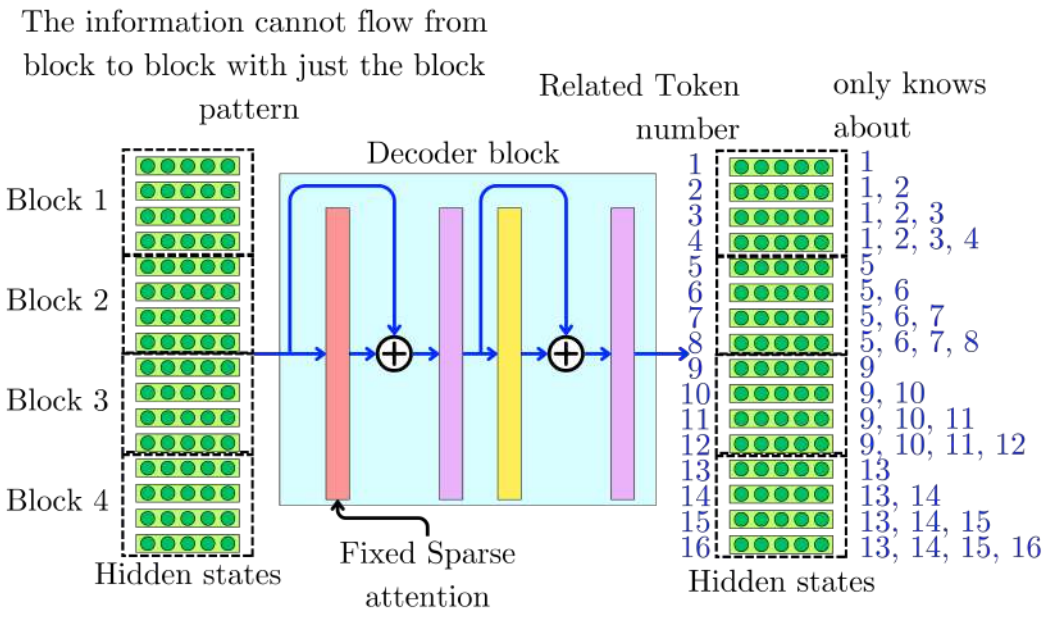

Fig. 15 *In the pure in-block attention pattern, the information cannot flow from block to block.*

of that entire sub-sequence. By letting future blocks attend to these summary tokens, the model propagates information globally across blocks, layer by layer.

Thus, the in-block pattern computes context vectors (weighted averages of values) only from nearby tokens, while the across-block pattern computes context from previous summary tokens. In combination, these patterns ensure both local and long-range context can be aggregated throughout the Transformer stack, even without a dense (i.e., fully quadratic) attention mechanism.

Let's estimate the time complexity of those sparse attentions. In the strided case, we have a local window of size w and a stride c. Therefore, each query attends to roughly $w + \frac{N}{c}$ keys (the local window plus the strided tokens). For N queries, the total cost is $\approx N\left(w + \frac{N}{c}\right)$. By tuning w and c, one can achieve sub-quadratic complexity. For example:

$$N\left(w + \frac{N}{c}\right) \sim \mathcal{O}(N\sqrt{N}), \quad \text{if } c = \sqrt{N}$$
$$N\left(w + \frac{N}{c}\right) \sim \mathcal{O}(N \log N), \quad \text{if } c = \frac{N}{\log N}$$

In the fixed case, the sequence is split into blocks of length l. Within each block, each query attends to at most l keys. Furthermore, each query attends $c = \frac{N}{l}$ summary tokens. Therefore, a query sees $\mathcal{O}(l + c)$ keys, and the total cost for all queries is $\sim \mathcal{O}(N(l + c))$. Typically, we choose l so that it grows sub-linearly with N. For example:

$$N(l + c) \sim \mathcal{O}(N\sqrt{N}), \quad \text{if } l = \sqrt{N} \tag{26}$$

Despite the improved time complexity, those computations must be performed as tensor operations to fully utilize the high parallelism of the GPU hardware. For example, let's assume we want to compute the attentions for the local sliding window described in the strided case. The original keys tensor K is of dimension $n_{\text{head}} \times d_{\text{head}} \times N$. We can construct the windowed keys tensor K^w to compute the sliding window all at once by adding another dimension representing the window size w. Constructing this tensor is a $\mathcal{O}(N)$ operation. The resulting tensor is of size $n_{\text{head}} \times d_{\text{head}} \times N \times w$ and each slice of size $d_{\text{head}} \times N \times w$ contains the necessary keys to compute the windowed attentions for each head (see fig. 17).

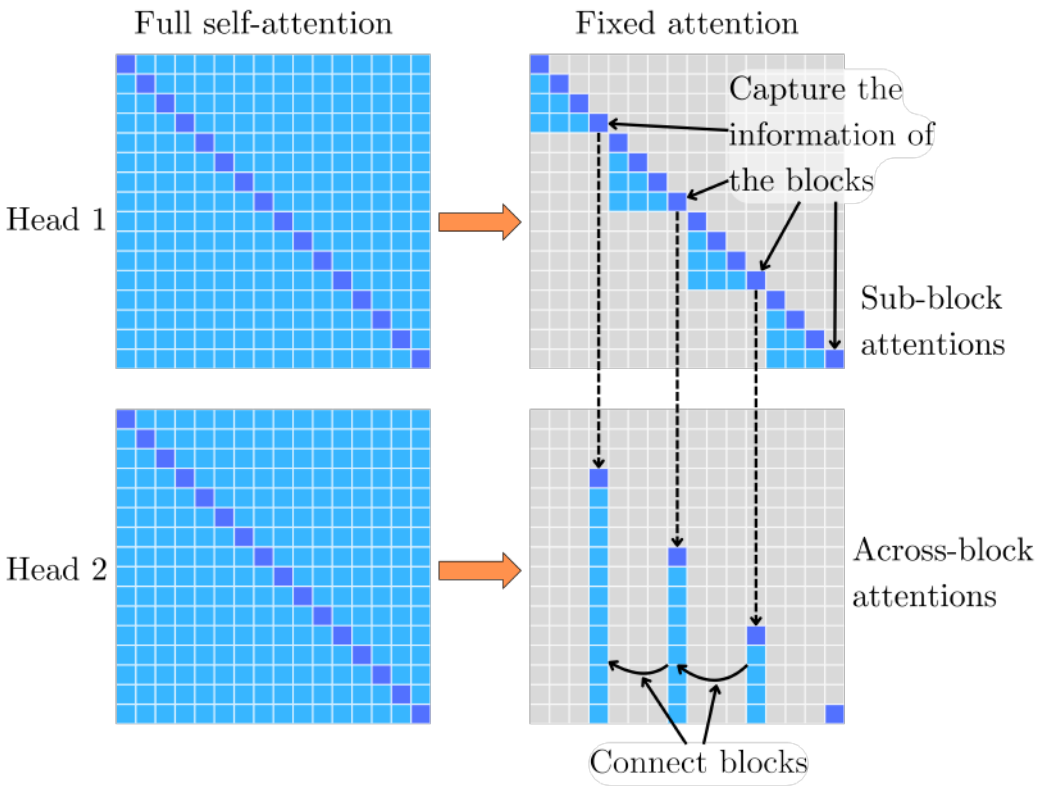

Fig. 16 *In the fixed sparse attention pattern, some heads focus on local attention with the in-block computation, while others focus on propagating the information learned within each block across blocks.*

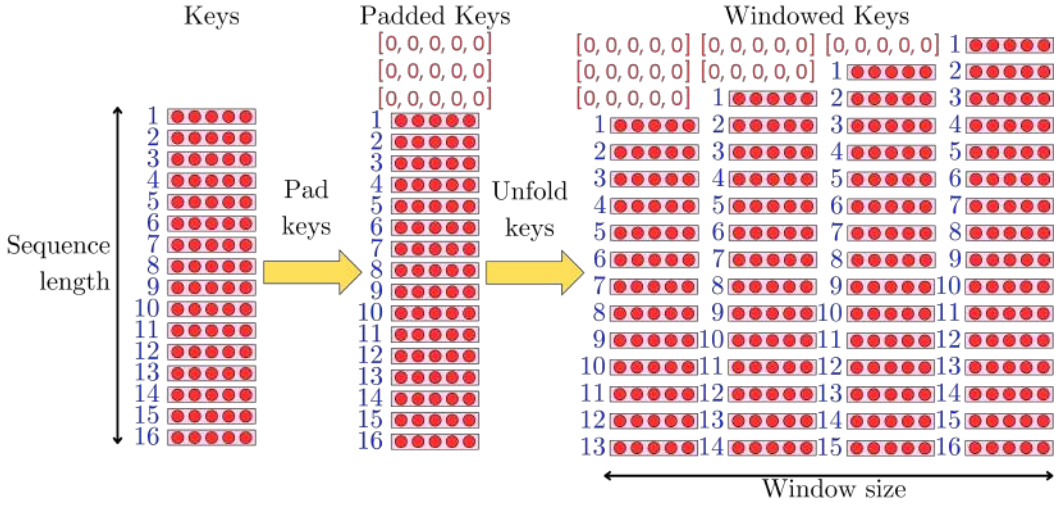

Fig. 17 *To compute the attentions with a local window, we can create the necessary tensor to perform this operation with one tensor multiplication. We first pad the keys to account for the edge case, and then generate the windowed keys. If the window is of size 4, for each related query position, we need four consecutive keys capturing the local query-key interactions.*

Let's now compute the product between the windowed keys K^w and the queries Q:

$$E_{hnw} = \sum_d \frac{Q_{hdn} K^w_{hdnw}}{\sqrt{d_{\text{head}}}}, \quad \text{shape: } n_{\text{head}} \times N \times w \tag{27}$$

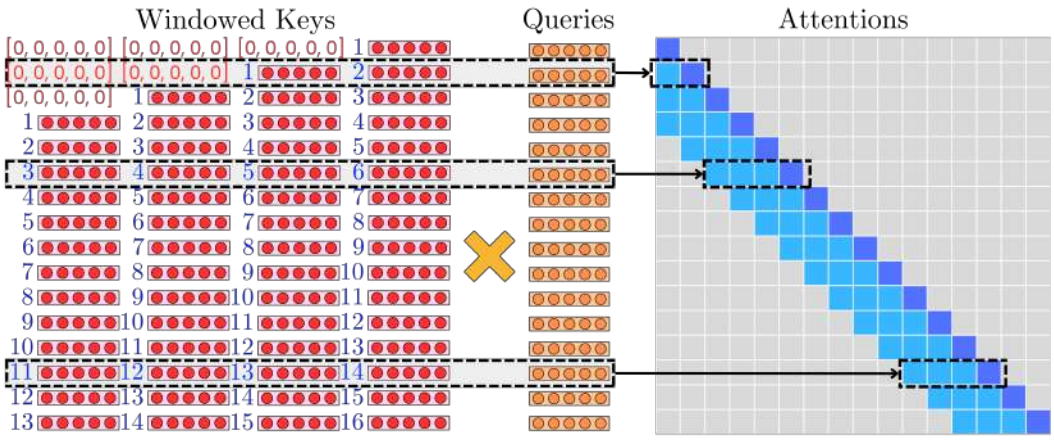

Fig. 18 *With the windowed keys, we compute the windowed attention with one tensor operation* $\sum_d \frac{Q_{hdn} K^w_{hdnw}}{\sqrt{d_{head}}}$*, efficiently using the high parallelism capability of the GPU.*

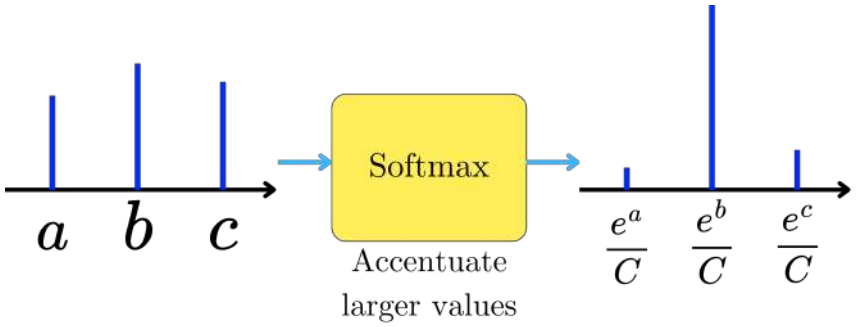

Fig. 19 *The softmax function makes the small values even smaller and accentuates the largest values.*

where h represents the head dimension (n_{head}), n the sequence dimension (N), d the hidden size per head dimension (d_{head}) and w the window dimension (see fig. 18). The resulting alignment scores tensor E is of shape $n_{\text{head}} \times N \times w$. The time complexity of this operation is $\mathcal{O}(d_{\text{model}} N w)$ instead of the vanilla $\mathcal{O}(d_{\text{model}} N^2)$.

3.2 Choosing Sparsity Efficiently: Reformer

The *Reformer Transformer* [18] introduced an efficient way to bucket keys and queries into blocks (as for the Sparse Transformer) that dynamically adapt to the input data. The main idea comes from the fact that, when computing the context vectors, most of the weighted average comes from the high attention weights:

$$\mathbf{c}_i = \sum_{j=1}^{N} a_{ij} \mathbf{v}_j \tag{28}$$

a_{ij} is the result of the softmax transformation of the alignment scores, and is pretty skewed toward the highest values:

$$a_{ij} = \text{Softmax}(e_{ij}) \tag{29}$$

The softmax function (or "soft maximum") exacerbates the largest values while minimizing the others.

As a consequence, most of the weighted average comes from the high alignment scores:

$$e_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}} \tag{30}$$

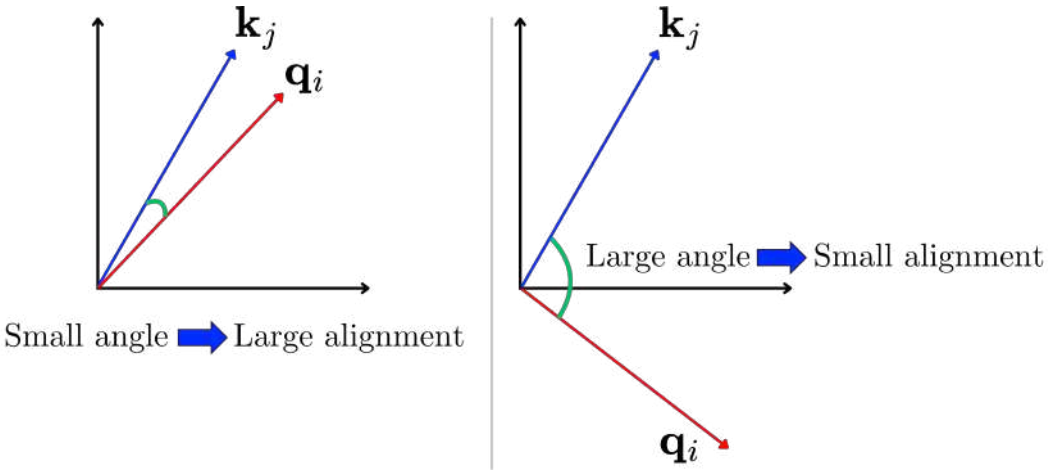

Fig. 20 *large alignment scores and attention result from angular similarity between the keys and queries.*

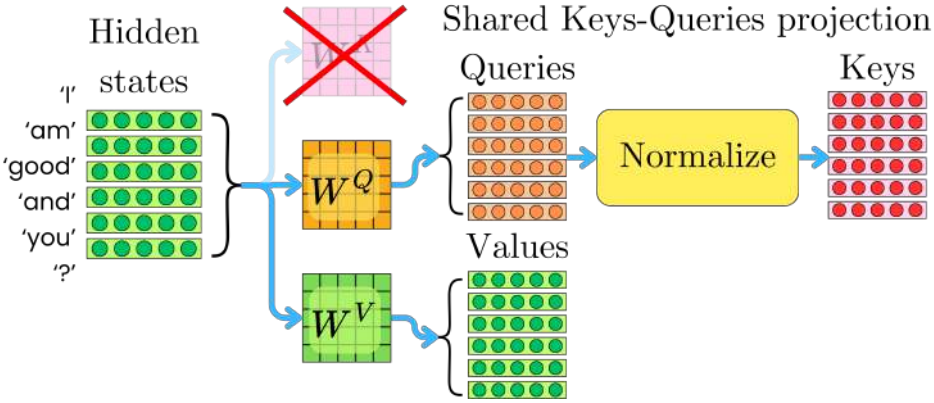

Fig. 21 *In Reformer, the key representations are simplified as the normalized query representations.*

Because the alignment scores result from a dot product, the highest alignment scores come from the keys and queries with a small angle between them (see fig. 20).

The *Reformer* bucketing strategy involves finding the keys most similar to the queries and neglecting the others. The first simplification was to share the query and key projection as they realized that it does not affect the model performance and to normalize queries as the key representations (see fig. 21). Formally, this means that:

$$\begin{aligned} \mathbf{q}_i &= W^Q \mathbf{h}_i \\ \mathbf{k}_i &= \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|} \end{aligned} \tag{31}$$

Therefore, it only uses two projection matrices W^Q and W^V and discards W^K. By normalizing the keys by $\|\mathbf{q}_i\|$, the dot product becomes:

$$\mathbf{q}_i^\top \mathbf{k}_j = \|\mathbf{q}_i\| \cos \theta_{ij} \tag{32}$$

where θ_{ij} is the angle between $\mathbf{q}_i$ and $\mathbf{q}_j$. This allows us to compare keys only based on their angular similarity to the query $\mathbf{q}_i$.

To find the keys nearest neighbors to queries, we use the Locality Sensitive Hashing (LSH) strategy. It aims to group vectors together based on similarity. To partition the vector space into buckets, we project the vectors onto a space of a specific dimensionality and find the axes they are closest to.

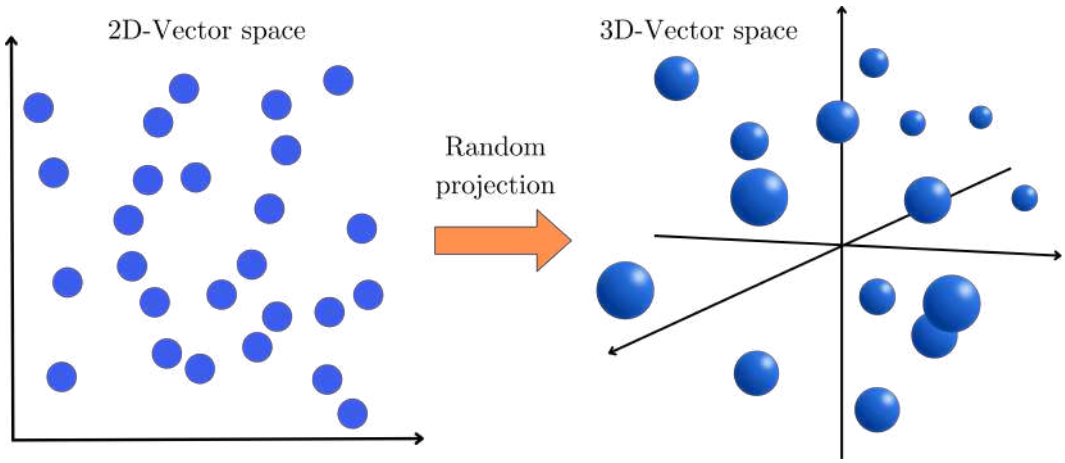

Fig. 22 *In Locally Sensitive Hashing, we first project the keys and queries in a different dimensional space. The specific dimensionality of the projection space allows us to fully control the level of approximation we are willing to accept. A higher-dimensional space will lead to smaller but more buckets, while a lower dimensionality will lead to fewer buckets but more approximated hashing.*

The projection is done using a random matrix R of dimension $d_{\text{model}} \times b$ where b is the dimension of the space we want to project the vectors into, and $2b$ is the number of buckets that will be used to group the queries.

$$\begin{aligned} Q' &= RQ \\ K' &= RK \end{aligned} \tag{33}$$

For example, if $d_{\text{model}} = 2$ and $b = 3$, we would have something like in the figure 22.

Now, we can construct the locality-sensitive hash for each vector:

$$\text{hash}(\mathbf{q}_i) = \arg\max\left[R\mathbf{q}_i; -R\mathbf{q}_i\right] \tag{34}$$

where $[x; y]$ denotes the concatenation of vectors x and y. Vectors with the same hash are bucketed together as approximate nearest neighbors, and we have exactly $2b$ buckets. Let's assume for example $R\mathbf{q}_i = [0.5, -0.3, 0.7]$. Then:

$$[R\mathbf{q}_i; -R\mathbf{q}_i] = [0.5, -0.3, 0.7, -0.5, 0.3, -0.7] \tag{35}$$

The $\arg\max$ is 2 (third element, value = 0.7). For a similar vector $R\mathbf{k}_j = [0.6, -0.2, 0.8]$:

$$[R\mathbf{k}_j; -R\mathbf{k}_j] = [0.6, -0.2, 0.8, -0.6, 0.2, -0.8] \tag{36}$$

The $\arg\max$ is also 2 (third element, value = 0.8), and both $\mathbf{q}_i$ and $\mathbf{k}_j$ are grouped into the same bucket. It works because the $\arg\max$ operation finds, for each vector, the closest axis direction, highlighting the main rough angular similarity (see fig. 23).

So far, we have bucketed the different queries into blocks of similar queries. However, the resulting buckets may vary in size (some very large, others small), making parallel computation inefficient, and we need a second level of equal-size chunking. The queries are then sorted by bucket number and sequence position within each bucket and rebucketed into equal-size chunks (see fig. 24).

Each query only attends to keys within its own chunk and one chunk back. However, the query reordering induces a violation of causality for some keys (some queries can peek into the future), which is a problem for causal language modeling, so we apply a causal mask to prevent this from happening (see fig. 25).

The LSH bucketing is typically done with multiple rounds n_{rounds} to reduce the chance of missing relevant keys. The time complexity of the bucketing is $\mathcal{O}(n_{\text{rounds}} N b)$ where b is the number of buckets. Typically, n_{rounds} and b are constant (e.g. $n_{\text{rounds}} = 4$, $b = 64$) so this becomes $\mathcal{O}(N)$. Sorting the buckets and queries is $\mathcal{O}(N \log N)$, and if we have m final chunks, chunking takes $\mathcal{O}(m)$. Each query attends to at most $2m$ keys, which leads to $\mathcal{O}(Nm)$ for all the queries. Since m is constant, it simplifies to $\mathcal{O}(N)$. Therefore, the total LSH attention complexity:

$$\mathcal{O}(N \log N + 2N) = \mathcal{O}(N \log N) \tag{37}$$

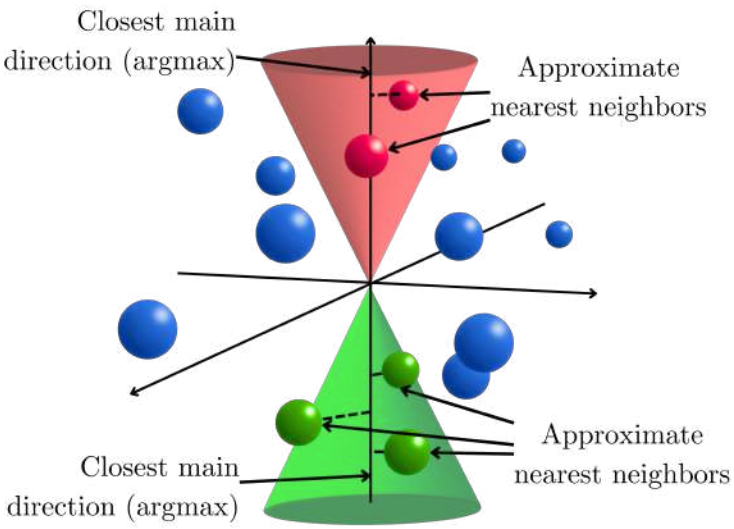

Fig. 23 *The* $\arg\max$ *function will find the closest axis and direction for each element. If two elements have the same closest axis, then they tend to have higher angular similarity, and the bucketing strategy groups them as approximate nearest neighbors.*

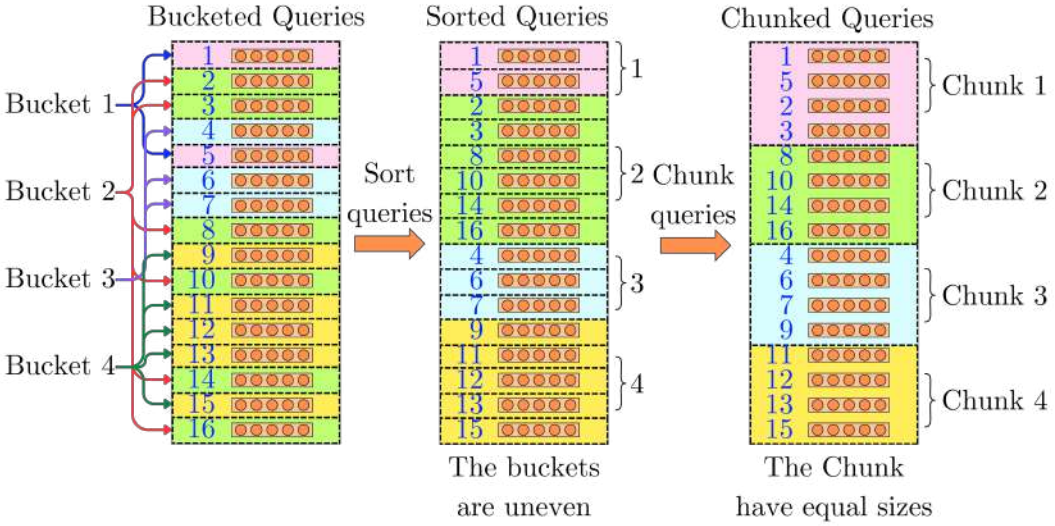

Fig. 24 *Once the queries are bucketed with LSH, they are sorted by bucket index and sequence position within each bucket. Then they are rechunked so the resulting chunks are equal in size, enabling easier tensor operations.*

3.3 Local vs Global Attention: Longformer and BigBird

Longformer [3] and *BigBird* [35] are similar approaches to OpenAI's *Sparse Transformer*, but they explicitly focus on combining local efficiency with global task-specific attention, outperforming both *Sparse Transformers* and *Reformer* in tasks requiring long-range dependencies.

They were mainly used for encoder-only BERT-like [12] models where causality is not imposed. Those models tend to be used to process the input sequence and generate features for specific learning tasks like classification problems. Global attention is addressed in both *Longformer* and *BigBird* by having a small subset of tokens, such as the [CLS] token, that can attend to all the other tokens. In BERT-like models, it is typical to append an additional classification token [CLS] at the start of the input sequence during the pretraining phase. During fine-tuning for classification tasks, the

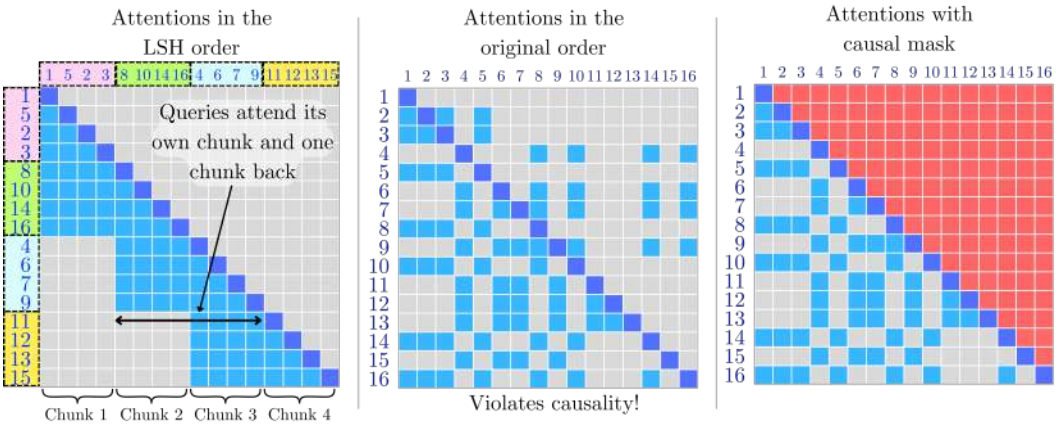

Fig. 25 *On the left, we see a common sparse pattern induced by the reformer bucketing strategy. However, the bucketing involved reordering the queries that do not follow the original sequence ordering. In the middle, we see an example of the same sparse pattern if we look at the original sequence ordering. In the case of causal language modeling, this specific approach leads to causality violations for some of the tokens, and we apply a typical causal mask that needs to be mapped to the new token order caused by the bucketing strategy.*

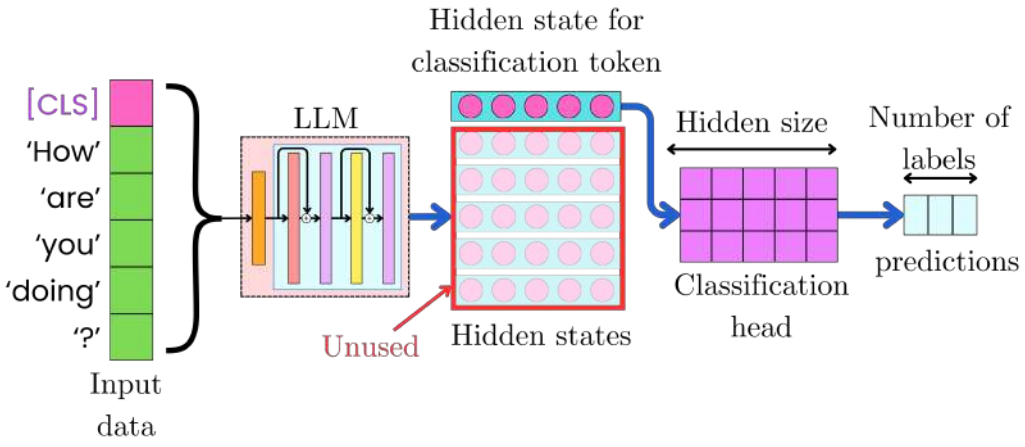

Fig. 26 *In an encoder model like BERT, it is common to prepend the input sequences with the* [CLS] *token so the model generates a hidden state dedicated to encoding the whole input sequence. The vector representation can be used as input to a classifier for sequence classification tasks.*

corresponding hidden state to the [CLS] token is used as input to a classifier. It is so that we have a dedicated hidden state for classification tasks (see fig. 26).

The Global attention is captured by using [CLS] as an information aggregator for the whole sequence. By having the [CLS] token attend to all the other tokens, its resulting hidden state captures the information related to them. By having all the different tokens attending the [CLS] token, they attempt to "source" the information on the whole sequence captured by the [CLS] token in the previous Transformer layers. This ensures that the internal hidden states related to [CLS] carry the signal related to all the input tokens at any layer within the Transformer (see fig. 27).

Furthermore, the local attention is captured by the common windowed attention (typically 512 tokens) in both *Longformer* and *BigBird*. *Longformer* uses separate projection matrices $\{W^K_{\text{global}}, W^Q_{\text{global}}, W^V_{\text{global}}\}$ and $\{W^K_{\text{local}}, W^Q_{\text{local}}, W^V_{\text{local}}\}$ for the global and local patterns while *Big-*

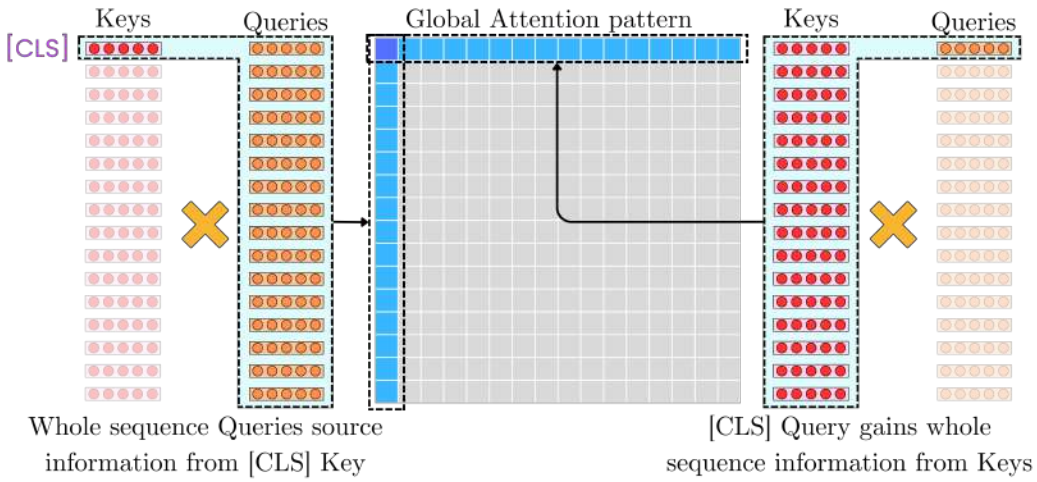

Fig. 27 *Each resulting hidden state can be mapped to a specific query. Having the* [CLS] *key attending to all the queries is a way to spread the information carried by the hidden state corresponding to* [CLS] *to all the other hidden states. By having the* [CLS] *query attend to all the keys, the model can extract the information carried by all the incoming hidden states from the whole sequence to inject it into the resulting hidden state corresponding to* [CLS]. *With this global attention pattern, layer after layer, the model can propagate the information from the whole sequence to all the tokens through the* [CLS] *token.*

Bird cumulates the patterns in the same attentions. BigBird expands on the Global/Local attention by proposing intermediate connections between tokens in the form of random attention (see fig. 28).

If the model only had local window attention, then tokens could not read or update cells far away in just a few layers (the path length would grow with the sequence). If it only had global tokens (like a single [CLS]), then that might help gather information in one place, but you still might not have an efficient way to route data between any two non-global positions. The random edges ensure that, with high probability, every position is connected to every other position in a small number of hops. In other words, the random edges, together with the local and global links, guarantee the entire graph is robustly connected, and that is what preserves the full set of attentions between tokens. For the vanilla attention, the tokens are fully connected at every layer, while for the random attention, the token connections build up layer after layer (see fig. 29). BigBird is proven to theoretically be a universal approximator (can approximate any continuous function) of sequence functions as the Transformer with vanilla attention is.

Each token attends to a fixed-size local window of w tokens for the sliding window, leading to $\mathcal{O}(Nw)$ time complexity. For the global attention, we can choose more global tokens than just [CLS]. With g global tokens, we have a time complexity of $\mathcal{O}(Ng)$. In the random attention case, we choose r keys randomly per query, leading to $\mathcal{O}(Nr)$ time complexity. With fixed w, g, and r, the total complexity simplifies to:

$$\begin{aligned} &\mathcal{O}\left(N(w+g)\right) = \mathcal{O}\left(N\right) \quad \text{for Longformer} \\ &\mathcal{O}\left(N(w+g+r)\right) = \mathcal{O}\left(N\right) \quad \text{for BigBird} \end{aligned} \tag{38}$$

With the right choice for the different parameters, this leads to a linear time complexity.

3.4 To Summarize

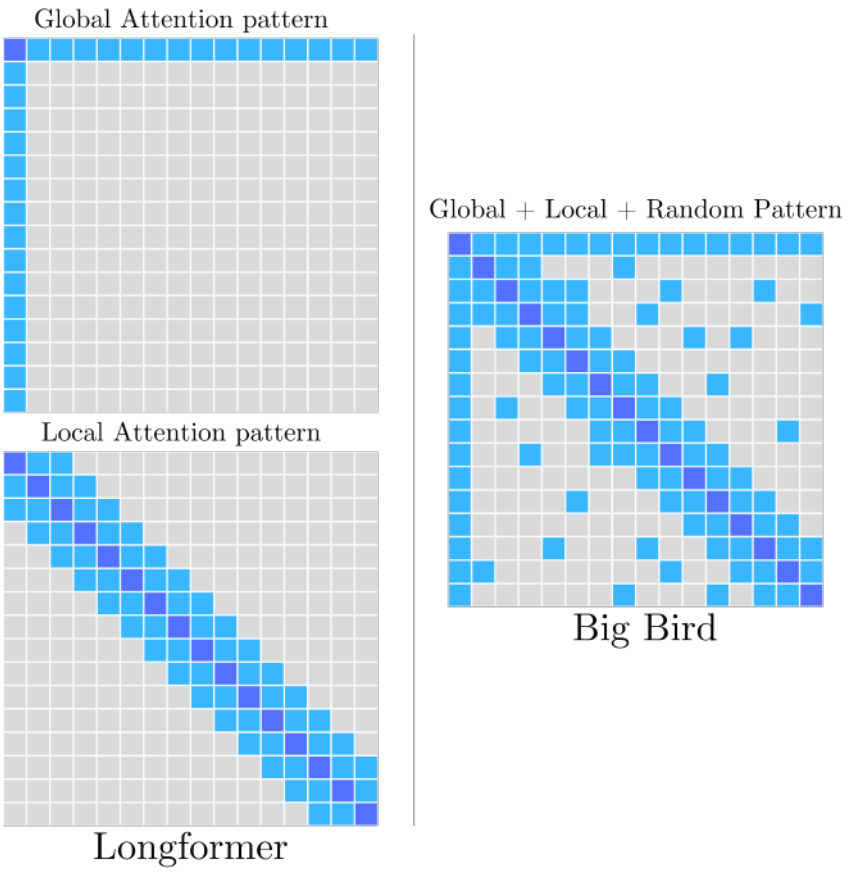

Fig. 28 *On the left, Longformer accumulates the effect of global and local patterns by spreading the patterns across attention heads. On the right, BigBird groups the global and local patterns into the same heads. Additionally, BigBird induces an increased connectivity between tokens by having intermediate random connections in the attention layers.*

Attention Type	**Key Innovations & Benefits**	**Limitations & Downsides**
Sparse Transformer	Strided and fixed patterns reduce computation while maintaining information flow through strategic connectivity	Fixed patterns may miss important connections; implementation complexity with tensor manipulation; strided pattern works poorly for text
Reformer	Uses locality-sensitive hashing to dynamically group similar queries, focusing attention on most relevant keys	LSH bucketing introduces approximation errors; performance degrades with very similar vectors; requires multiple hash rounds for stability
Longformer/BigBird	Combines local windowed, global, and random attention patterns; BigBird is a proven universal approximator	Predetermined sparsity patterns may not be optimal for all tasks; increased implementation complexity; primarily designed for encoder-only models

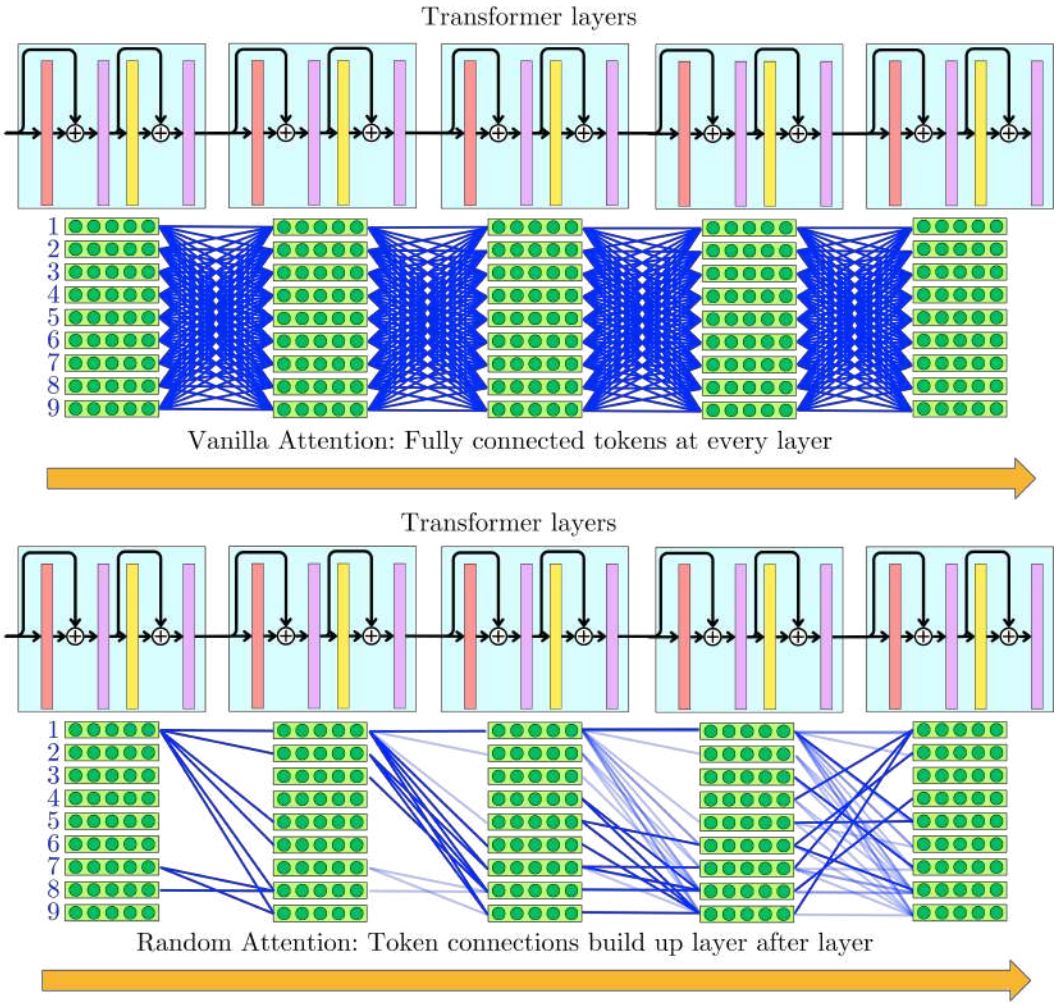

Fig. 29 *In the case of the vanilla attention, every token is connected to every other at each layer. The random attention presented in BigBird leads to sparse connections between tokens at each layer, but induces close to full connectivity over multiple layers. As we progress in the network, tokens are increasingly connected to all the other tokens as the random attention pattern brings diversity in the connections.*

4 Linear Attention Mechanisms

Linear attention mechanisms represent a paradigm shift in transformer architecture by mathematically re-engineering the attention operation to achieve $\mathcal{O}(N)$ complexity while maintaining global context awareness. Unlike sparse attention's pattern restrictions, which preserve quadratic complexity but limit interactions to predefined token subsets, linear attention fundamentally redefines how all tokens interact by reformulating the attention matrix computation rather than pruning token interactions. Where sparse attention sacrifices theoretical completeness for practical speed, linear attention preserves global relationships at the cost of approximating pairwise token influences. This enables native handling of large sequence lengths while avoiding sparse attention's blind spots.

4.1 Low-Rank Projection of Attention Matrices: Linformer

With Sparse attentions, we understood that most of the token interaction information was contained in a small subset of token pairs. *Linformer* [33] introduced the idea that the token-token interaction matrix could be compressed into a smaller representation without too much information loss. Instead of computing the full $N \times N$ interaction $\frac{Q^\top K}{\sqrt{d_{\text{model}}}}$ (ignoring heads for simplicity), we could first project

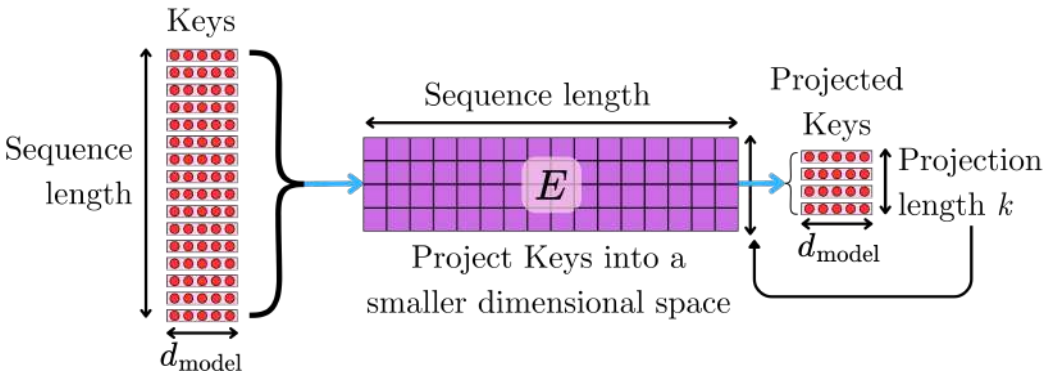

Fig. 30 *The E matrix compresses the keys in the sequence length direction, leading to a resulting key tensor independent of the sequence length.*

the keys K into a lower rank dimension k, and compute the lower rank $N \times k$ approximation:

$$\frac{Q^\top EK}{\sqrt{d_{\text{model}}}} \tag{39}$$

where E is a $N \times k$ projection matrix that project K from the original dimension $d_{\text{model}} \times N$ to $d_{\text{model}} \times k$ (see fig. 30). This leads to $N \times k$ alignment score and attention matrices.

When we project with E, the approximation leads to the error:

$$\text{error} = \left| \frac{Q^\top K}{\sqrt{d_{\text{model}}}} - \frac{Q^\top EK}{\sqrt{d_{\text{model}}}} \right| \tag{40}$$

If the elements of E follow a Gaussian distribution $\mathcal{N}(0, 1/k)$, the Johnson–Lindenstrauss lemma[16] guarantees that:

$$P\left[\text{error} > \epsilon\right] \leq e^{-\gamma \epsilon^2 k}. \tag{41}$$

This means that the probability of choosing E such that the error is greater than ϵ is bounded by $e^{-\gamma \epsilon^2 k}$, where γ is just a scaling constant. If we choose $k \to \infty$, then $P\left[\text{error} > \epsilon\right] \to 0$ for any ϵ. A good choice is $k \propto \log N/\epsilon^2$, yielding:

$$P\left[\text{error} > \epsilon\right] \leq N^{-\gamma}. \tag{42}$$

This means that we can choose an arbitrarily small ϵ such that $P\left[\text{error} > \epsilon\right] \to 0$ as the sequence length increases $N \to \infty$. Understand this as a theoretical guide that tells us that choosing $k \propto \log N$ will guarantee smaller errors as N increases. In practice, k is chosen independently of N, leading to the $\mathcal{O}(N)$ linear complexity while accepting the cost of the approximation error. Additionally, E is chosen as a parameter layer for the model to learn. For example, they showed that choosing $k = 64$ with $N = 512$ leads to slightly worse performance than the full attention.

Since the attention matrix has dimension $N \times k$, we also need to project the values:

$$C = \text{Softmax}\left(\frac{Q^\top EK}{\sqrt{d_{\text{model}}}}\right) FV \tag{43}$$

where F is the $N \times k$ projection matrix for the tensor V. As for E, F is also learned during training.

Projecting the keys and values EK, FV leads to complexity $\mathcal{O}(Nk)$. Computing the alignment scores $Q^\top EK$ and the context vectors $C = AFV$ also follows $\mathcal{O}(Nk)$. Since we fix k, the overall time and space complexity is $\mathcal{O}(N)$.

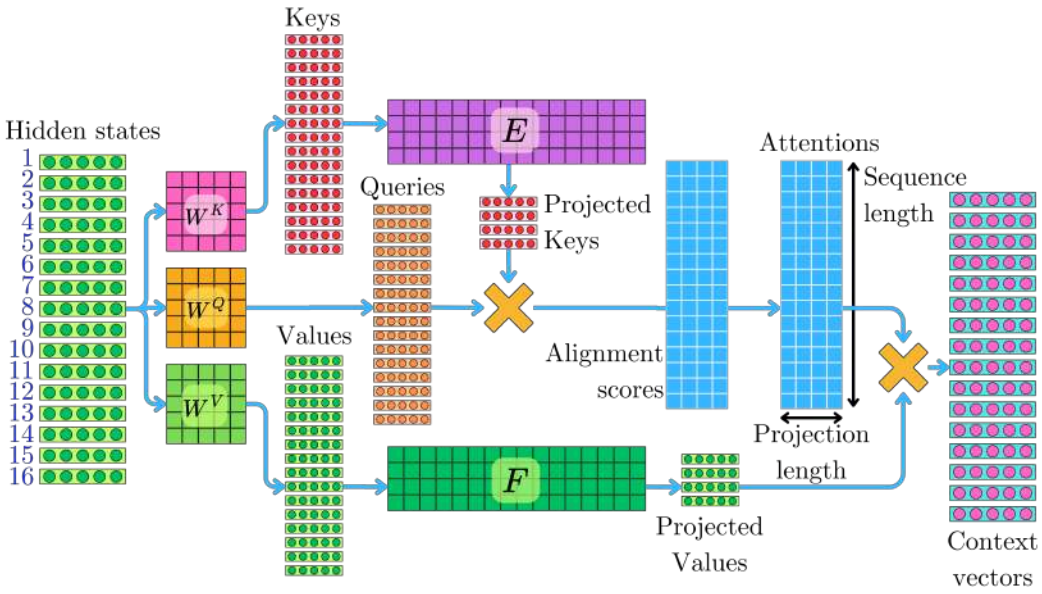

Fig. 31 *The Linformer approach produces a low-rank attention matrix by compressing the keys and the values in the sequence length direction with the E and F model parameter matrices learned during training.*

4.2 Recurrent Attention Equivalence: The Linear Transformer

So far, we have accepted the attention mechanism to be represented by the following computation (again, ignoring heads for simplicity):

$$C = \text{Softmax}\left(\frac{Q^\top K}{\sqrt{d_{\text{model}}}}\right)V \quad \text{or}$$

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)\mathbf{v}_j}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)} \quad \text{for individual vectors} \tag{44}$$

However, this specific analytical choice is not the only one that could be chosen to fulfill the same functional role in capturing pairwise interactions between tokens. Let's review the roles of the different elements in this equation:

- **The dot-product $Q^\top K$: Similarity computation.** For each query vector, it tells you how "compatible" or similar it is to each key vector. This yields a matrix of unnormalized attention scores.
- **Normalizing by $\sqrt{d_{\text{model}}}$: Variance control.** The primary purpose of scaling by $\sqrt{d_{\text{model}}}$ is to control the scale of the attention logits before softmax, ensuring stable gradient flow and preventing the softmax from becoming too "confident" (peaked). Furthermore, extremely large logits can cause numerical instability (e.g., NaN in floating-point arithmetic), and scaling mitigates this.
- **Softmax operation: Normalization and nonlinearity.** The softmax turns the unnormalized similarity scores into a probability distribution, amplifying the effect of the most relevant keys.
- **Multiplication by V: Weighted aggregation.** Each output is a weighted sum of the values, where the weights come from the normalized similarity scores. This is how the model "mixes" information from across the input sequence

Functionally, we need a similarity function that is non-linear and captures the pairwise token interaction:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \text{sim}(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^{N} \text{sim}(\mathbf{q}_i, \mathbf{k}_k)} \tag{45}$$

where the denominator ensures that the similarity function is normalized to 1. If we choose $\text{sim}(\mathbf{q}_i, \mathbf{k}_j) = \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$, we recover the softmax transformation. Katharopoulos et al., with the *Linear Transformer* [17], proposed a new attention mechanism with a different analytical form but similar functional roles. More specifically, they suggested a similarity function where we can factorize the contribution from the keys and the queries as a product:

$$\text{sim}(\mathbf{q}_i, \mathbf{k}_j) = \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j). \tag{46}$$

In the context of kernel methods in machine learning, ϕ is called a "feature map". A feature map is a function that transforms an input vector into a new space, often a higher-dimensional one, so that a kernel function (which measures similarity) can be expressed as an inner product in that space. Essentially, ϕ extracts or "maps" the original features into a new representation where the desired similarity (that mimics the softmax behavior) is computed simply by taking a dot product. In the context of the Linear Transformer, they simply chose ϕ as follows:

$$\phi(x) = \begin{cases} x + 1 & \text{if } x > 0, \\ \exp x & \text{otherwise.} \end{cases} \tag{47}$$

This ensures that $\text{sim}(\mathbf{q}_i, \mathbf{k}_j)$ is always positive and is computationally stable while being non-linear. The main appeal of this linearization of the similarity kernel is the associativity property of the matrix multiplication:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{N} \phi(\mathbf{k}_j)} \tag{48}$$

For one key and one query, $\phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)$ takes $\sim d_{\text{model}}$ operations. Multiplying the resulting scalar alignment score by $\mathbf{v}_j$ takes another $\sim d_{\text{model}}$ operations. Therefore, for all the keys, computing $\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j$ takes $\sim 2Nd_{\text{model}}$ operations and the times complexity is $\mathcal{O}(Nd_{\text{model}})$ per query. Similarly, the denominator $\sum_{j=1}^{N} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)$ follows a time complexity of $\mathcal{O}(Nd_{\text{model}})$ (see fig. 32). Because we have N queries, the total cost is

$$\mathcal{O}(N^2 d_{\text{model}}) \quad \text{(our typical quadratic complexity!)}. \tag{49}$$

If we consider the multiplications in a different order, $\phi(\mathbf{k}_j)\mathbf{v}_j^\top$ is an outer product and results in d_{model}^2 operations. For N keys and values, we end up with Nd_{model}^2 operations for $\sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$. In the denominator, summing the different keys $\sum_{j=1}^{N} \phi(\mathbf{k}_j)$ requires Nd_{model} operations. Let's call $S = \sum_{j=1}^{N} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$ and $\mathbf{z} = \sum_{j=1}^{N} \phi(\mathbf{k}_j)$. S is a matrix of size $d_{\text{model}} \times d_{\text{model}}$ and $\mathbf{z}$ is a vector of size d_{model}. Computing $\phi(\mathbf{q}_i)^\top S$ brings another d_{model}^2 operations, and computing $\phi(\mathbf{q}_i)^\top \mathbf{z}$ takes d_{model} operations. Therefore, the cost of $\frac{\phi(\mathbf{q}_i)^\top S}{\phi(\mathbf{q}_i)^\top \mathbf{z}}$ per query is $\mathcal{O}(d_{\text{model}}^2 + d_{\text{model}}) = \mathcal{O}(d_{\text{model}}^2)$. For N queries, we obtain a total complexity of:

$$\mathcal{O}(Nd_{\text{model}}^2) \quad \text{(linear complexity!)}. \tag{50}$$

By changing the order of the matrix multiplication, we were able to reduce the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ (see fig. 33)! We have here ignored the tensor operations involving the attention heads, but it would result in the same time complexity.

Decoding text with this *Linear Transformer* is also extremely efficient, speed and memory-wise. For example, let's assume we have an initial prompt "How are you doing?". For the vanilla attention, we first need to compute all the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and all the values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ but only the last query $\mathbf{q}_n$ in the sequence since we only need to predict the next token. Let's further assume that the predicted token is 'I', and it is appended to the input sequence "How are you doing? I". In the following decoding iteration we only need the last query in the sequence $\mathbf{q}_{n+1}$, but we still need all

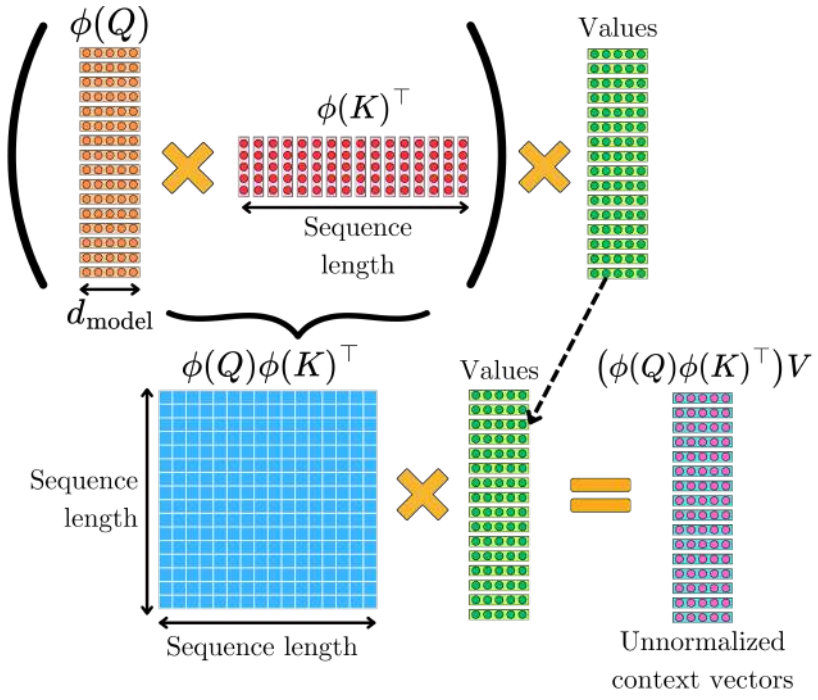

Fig. 32 *Computing $\phi(Q)\phi(K)^\top$ first leads to $\mathcal{O}(N^2)$ time complexity.*

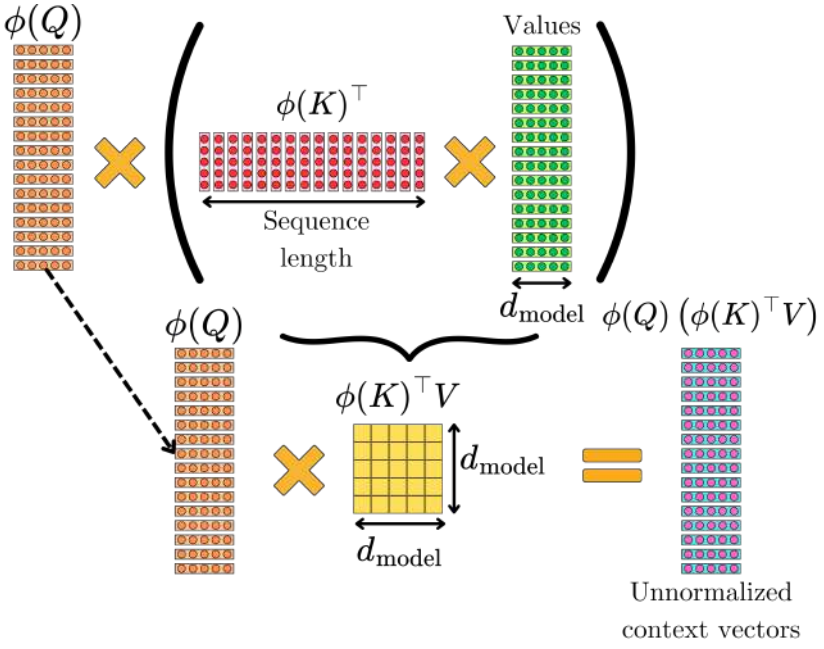

Fig. 33 *Computing $\phi(K)^\top V$ first leads to $\mathcal{O}(N)$ time complexity.*

the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n, \mathbf{k}_{n+1}]$ and all the values $[\mathbf{v}_1, \ldots, \mathbf{v}_n, \mathbf{v}_{n+1}]$. The previous keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ are exactly the same, so we could store them for the next iteration (see fig. 34). Caching the keys and values is called KV-caching. It is computationally efficient, but requires a lot of memory to store them.

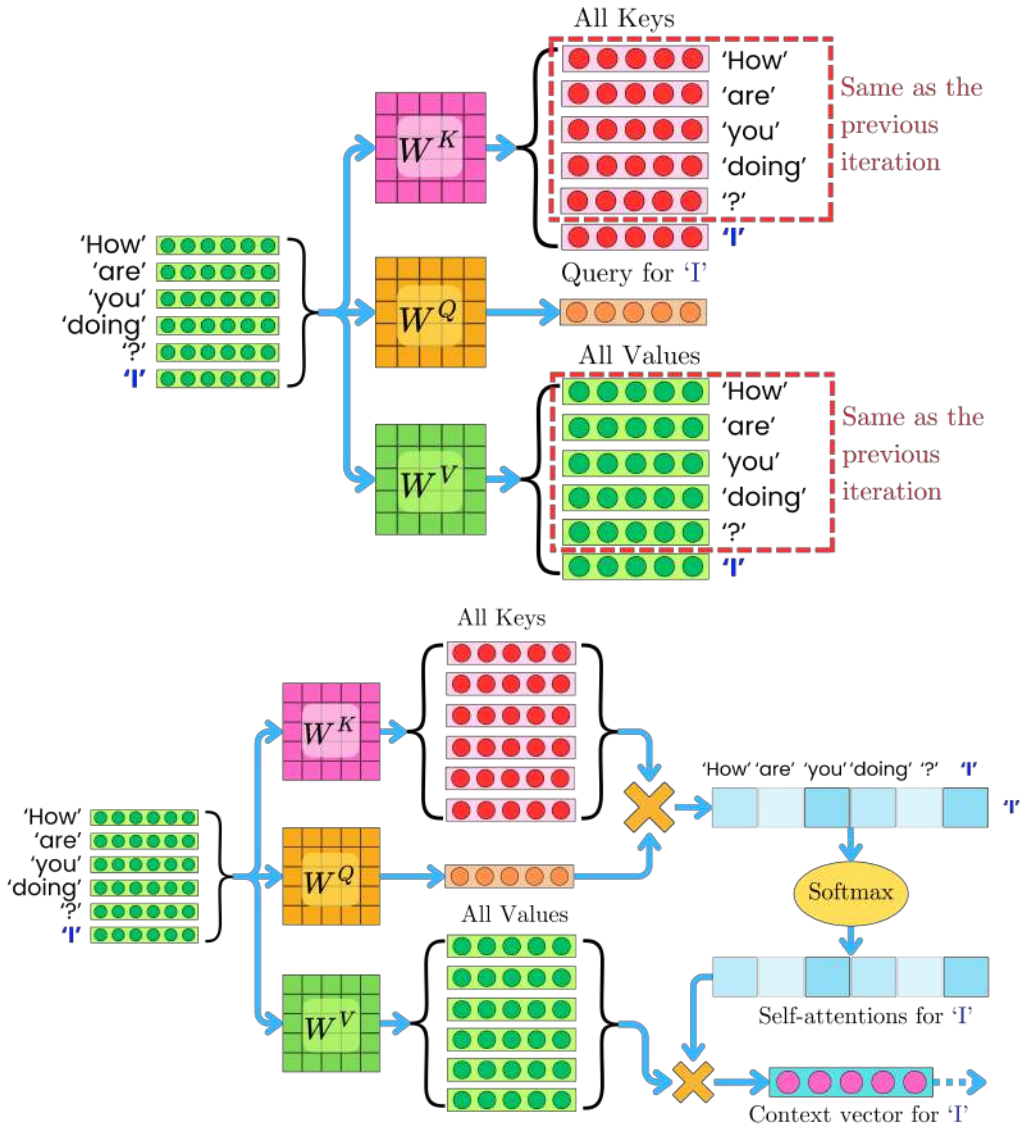

Fig. 34 *For the vanilla attention, we need all the keys and values at every point during the decoding process, but only the last query corresponding to the last token in the sequence. The keys and values at each iteration are exactly the same as those from the previous iteration, with the new key and value corresponding to the latest token. Therefore, those keys and values must be recomputed (which takes time) or cached (which takes memory).*

In the context of the *Linear Transformer*, in the first iteration, we still need to compute all the keys $[\mathbf{k}_1, \dots, \mathbf{k}_n]$, values $[\mathbf{v}_1, \dots, \mathbf{v}_n]$, and the last query $\mathbf{q}_n$ in the sequence to compute $\mathbf{c}_n$:

$$\mathbf{c}_n = \frac{\phi(\mathbf{q}_n)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_n)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)} \tag{51}$$

However, we can now precompute $S_n = \sum_{j=1}^{n} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$ and $\mathbf{z}_n = \sum_{j=1}^{n} \phi(\mathbf{k}_j)$, and store them for the next iteration (see fig. 35). S_n is a $d_{\text{model}} \times d_{\text{model}}$ matrix, and $\mathbf{z}_n$ is a vector of size d_{model}, so a small amount of memory is necessary, and it remains constant for the whole decoding process. In the following iteration, we do not need anymore the previous keys and values, and we only need to compute $\mathbf{q}_{n+1}$, $\mathbf{k}_{n+1}$, and $\mathbf{v}_{n+1}$ (see fig. 36). We can then compute the next context vector in

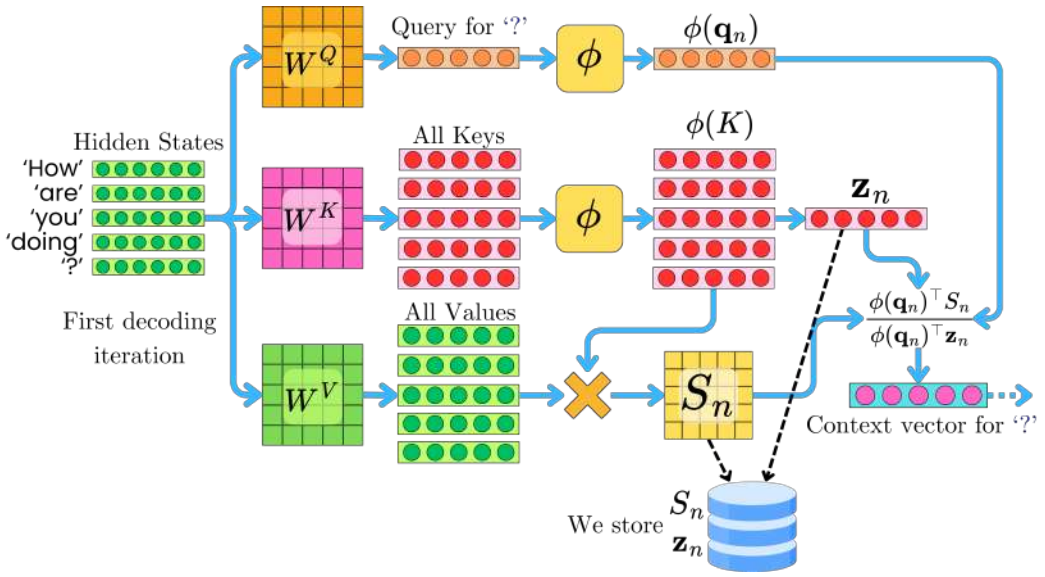

Fig. 35 *During the first iteration, S_n and $\mathbf{z}_n$ are computed and stored. This first iteration is the only one where all the keys and values will be necessary.*

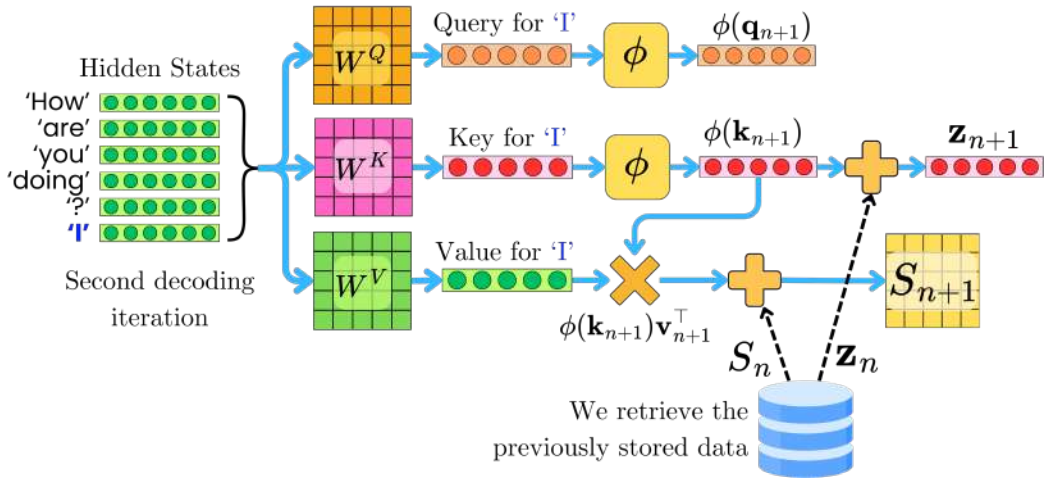

Fig. 36 *After the first interaction, we only need to compute the latest key and value and update the stored memory. This saves a lot of computations and memory during the decoding process.*

constant time:

$$
\begin{aligned}
S_{n+1} &= S_n + \phi(\mathbf{k}_{n+1})\mathbf{v}_{n+1}^\top \\
\mathbf{z}_{n+1} &= \mathbf{z}_n + \phi(\mathbf{k}_{n+1}) \\
\mathbf{c}_{n+1} &= \frac{\phi(\mathbf{q}_n)^\top S_{n+1}}{\phi(\mathbf{q}_n)^\top \mathbf{z}_{n+1}}
\end{aligned}
\tag{52}
$$

A similar trick can also be applied when computing the gradients for the backpropagation algorithm. During the backward pass, we need to compute the gradient of the loss function $\mathcal{L}$ with respect to the model parameters. At the $n-$th iteration, the output $\mathbf{c}_n$ (and thus the loss) depends on the current query $\phi(\mathbf{q}_n)$, all the keys $[\phi(\mathbf{k}_1), \ldots, \phi(\mathbf{k}_n)]$ and all the values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ due to the autoregressive nature of the model. So we need to be able to compute the gradients $\nabla_{\phi(\mathbf{q}_n)}\mathcal{L}$, $[\nabla_{\phi(\mathbf{k}_1)}\mathcal{L}, \ldots, \nabla_{\phi(\mathbf{k}_n)}\mathcal{L}]$ and $[\nabla_{\mathbf{v}_1}\mathcal{L}, \ldots, \nabla_{\mathbf{v}_n}\mathcal{L}]$. Let's consider $\nabla_{\phi(\mathbf{k}_1)}\mathcal{L}$ and let's call

$\tilde{\mathbf{c}}_i = \phi(\mathbf{q}_i)^\top S_i$, the unnormalized context vector. Using the chain rule, we have:

$$\nabla_{\phi(\mathbf{k}_1)}\mathcal{L} = \sum_{i=1}^{n} \nabla_{\tilde{\mathbf{c}}_i}\mathcal{L} \cdot \nabla_{\phi(\mathbf{k}_1)}\tilde{\mathbf{c}}_i \tag{53}$$

This is true because every $\tilde{\mathbf{c}}_i$ depends on $\phi(\mathbf{k}_1)$ for $i > 1$. We can compute $\nabla_{\phi(\mathbf{k}_1)}\tilde{\mathbf{c}}_i$:

$$\begin{aligned}\nabla_{\phi(\mathbf{k}_1)}\tilde{\mathbf{c}}_i &= \nabla_{\phi(\mathbf{k}_1)}\left(\phi(\mathbf{q}_i)^\top \sum_{j=1}^{i} \phi(\mathbf{k}_j)\mathbf{v}_j^\top\right) \\ &= \phi(\mathbf{q}_i)\mathbf{v}_1^\top\end{aligned} \tag{54}$$

Therefore, we have:

$$\begin{aligned}\nabla_{\phi(\mathbf{k}_1)}\mathcal{L} &= \sum_{i=1}^{n} \nabla_{\tilde{\mathbf{c}}_i}\mathcal{L} \cdot \phi(\mathbf{q}_i)\mathbf{v}_1^\top \\ &= \left(\sum_{i=1}^{n} \phi(\mathbf{q}_i)\left(\nabla_{\tilde{\mathbf{c}}_i}\mathcal{L}\right)^\top\right)\mathbf{v}_1\end{aligned} \tag{55}$$

Here, we specifically looked at $\nabla_{\phi(\mathbf{k}_1)}\mathcal{L}$, but the same logic applies to any of the gradients $\nabla_{\phi(\mathbf{k}_j)}\mathcal{L}$:

$$\nabla_{\phi(\mathbf{k}_j)}\mathcal{L} = \left(\sum_{i=j}^{n} \phi(\mathbf{q}_i)\left(\nabla_{\tilde{\mathbf{c}}_i}\mathcal{L}\right)^\top\right)\mathbf{v}_j \tag{56}$$

Similarly, following the same logic for $\nabla_{\mathbf{v}_j}\mathcal{L}$, we obtain:

$$\nabla_{\mathbf{v}_j}\mathcal{L} = \left(\sum_{i=j}^{n} \phi(\mathbf{q}_i)\left(\nabla_{\tilde{\mathbf{c}}_i}\mathcal{L}\right)^\top\right)\phi(\mathbf{k}_j) \tag{57}$$

Computing $\nabla_{\phi(\mathbf{q}_n)}\mathcal{L}$ is a bit simpler since $\tilde{\mathbf{c}}_n$ only depends on the last query:

$$\begin{aligned}\nabla_{\phi(\mathbf{q}_n)}\mathcal{L} &= \nabla_{\tilde{\mathbf{c}}_n}\mathcal{L} \cdot \nabla_{\phi(\mathbf{q}_n)}\tilde{\mathbf{c}}_n \\ &= \nabla_{\tilde{\mathbf{c}}_n}\mathcal{L} \cdot \nabla_{\phi(\mathbf{q}_n)}\left(\phi(\mathbf{q}_n)^\top S_n\right) \\ &= \nabla_{\tilde{\mathbf{c}}_n}\mathcal{L} \cdot S_n^\top\end{aligned} \tag{58}$$

If we call $B_j = \sum_{i=j}^{n} \phi(\mathbf{q}_i)\left(\nabla_{\tilde{\mathbf{c}}_i}\mathcal{L}\right)^\top$, we can use the recurring nature of B_j to backpropagate the gradient to the previous iteration in constant time and memory:

$$\begin{aligned}B_{j-1} &= B_j + \phi(\mathbf{q}_{j-1})\left(\nabla_{\tilde{\mathbf{c}}_{j-1}}\mathcal{L}\right)^\top \\ \nabla_{\phi(\mathbf{k}_{j-1})}\mathcal{L} &= B_{j-1}\mathbf{v}_{j-1} \\ \nabla_{\mathbf{v}_{j-1}}\mathcal{L} &= B_{j-1}\phi(\mathbf{k}_{j-1})\end{aligned} \tag{59}$$

Note that B_j is computed iteratively backward (from $j = n$ to $j = 1$), enabling constant memory usage. This avoids storing all intermediate gradients explicitly. By leveraging reverse cumulative sums to aggregate gradients iteratively, this approach computes gradients for autoregressive transformers in $\mathcal{O}(N)$ time and $\mathcal{O}(1)$ memory, mirroring the efficiency of recurrent neural networks while preserving the expressive power of self-attention.

This formulation is strikingly similar to how recurrent neural networks (RNNs) operate. In an RNN, you update the hidden state $\mathbf{h}_n$ as a function of the previous state $\mathbf{h}_{n-1}$ and the current input $\mathbf{x}_n$:

$$\mathbf{h}_n = f(\mathbf{h}_{n-1}, \mathbf{x}_n) \tag{60}$$

In the *Linear Transformer*, the cumulative sums S_n and $\mathbf{z}_n$ play the role of a hidden state updated with each new key-value pair.

$$(S_n, \mathbf{z}_n) = \left(S_{n-1} + \phi(\mathbf{k}_n)\mathbf{v}_n^\top, \mathbf{z}_{n-1} + \phi(\mathbf{k}_n)\right) \tag{61}$$

Like an RNN's hidden state, the accumulators S_n and $\mathbf{z}_n$ have fixed sizes, independent of the sequence length N. This means that, during inference, you do not need to store all previous keys and values; you only need to maintain these fixed-size summaries. When generating a new token, you use the current query $\phi(\mathbf{q}_n)$ along with the current state $(S_n, \mathbf{z}_n)$ to compute the output $\mathbf{c}_n$.

4.3 Kernel Approximation: Performers

The kernel chosen in the *Linear Transformer* is not an approximation of the softmax kernel but provides many of the same properties. To improve even further, a team at Google and Cambridge University proposed *Performers* [7] in 2022 with feature maps that approximate the softmax kernel. Understand that it is an "improvement" if we assume that the softmax transformation is somehow most optimal in the context of the attention mechanism. Ignoring the $\sqrt{d_{\text{model}}}$ scaling factor, the softmax kernel is simply:

$$SM(\mathbf{q}_i, \mathbf{k}_j) = \exp\left(\mathbf{q}_i^\top \mathbf{k}_j\right). \tag{62}$$

We want to find an approximate kernel expressed as a product of feature maps:

$$\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) = \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j) \tag{63}$$

With Performer, they suggested the following:

$$\phi(\mathbf{q}_i) = \frac{\exp\left(-\frac{\|\mathbf{q}_i\|^2}{2}\right)}{\sqrt{m}} \begin{bmatrix} \exp\left(\omega_1^\top \mathbf{q}_i\right) \\ \exp\left(\omega_2^\top \mathbf{q}_i\right) \\ \vdots \\ \exp\left(\omega_m^\top \mathbf{q}_i\right) \end{bmatrix} \tag{64}$$

$\omega_1, \omega_2, \ldots, \omega_m$ are orthonormal random vectors sampled from $\mathcal{N}(0, \mathbf{I}_{d_{\text{model}}})$. ω_l has a dimension d_{model}, $\exp\left(\omega_l^\top \mathbf{q}_i\right)$ and $\exp\left(-\frac{\|\mathbf{q}_i\|^2}{2}\right)$ are just scalars, and $\left[\exp\left(\omega_1^\top \mathbf{q}_i\right), \exp\left(\omega_2^\top \mathbf{q}_i\right), \ldots, \exp\left(\omega_m^\top \mathbf{q}_i\right)\right]$ is a vector of size m, where m is a hyperparameter. Therefore, ϕ is a transformation that takes a vector of size d_{model} and generates a vector of size m. The full approximated kernel is

$$\begin{aligned} \widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) &= \frac{1}{m} \sum_{l=1}^{m} \exp\left(\omega_l^\top \mathbf{q}_i - \frac{\|\mathbf{q}_i\|^2}{2}\right) \exp\left(\omega_l^\top \mathbf{k}_j - \frac{\|\mathbf{k}_j\|^2}{2}\right) \\ &= \frac{1}{m} \sum_{l=1}^{m} \exp\left(\omega_l^\top (\mathbf{q}_i + \mathbf{k}_j) - \frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right) \end{aligned} \tag{65}$$

$\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$ is a good approximation because we can prove that $SM(\mathbf{q}_i, \mathbf{k}_j)$ is the expected value of $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$ over $\boldsymbol{\omega}_m = [\omega_1, \omega_2, \ldots, \omega_m]$:

$$\mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right] = SM(\mathbf{q}_i, \mathbf{k}_j) \tag{66}$$

Let's prove it! We call $W = \omega_l^\top (\mathbf{q}_i + \mathbf{k}_j)$. Because $\omega_l \sim \mathcal{N}(0, \mathbf{I}_{d_{\text{model}}})$, then $W \sim \mathcal{N}(0, \|\mathbf{q}_i + \mathbf{k}_j\|^2)$, where $\sigma^2 = \|\mathbf{q}_i + \mathbf{k}_j\|^2$ is the variance of W. The expected value of $\exp(W)$ is:

$$\mathbb{E}_{\boldsymbol{\omega}_m}[\exp(W)] = \exp\left(\frac{\sigma^2}{2}\right) = \exp\left(\frac{\|\mathbf{q}_i + \mathbf{k}_j\|^2)}{2}\right) \tag{67}$$

By using the above equation, we can now compute the expected value of $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$:

$$\begin{aligned} \mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right] &= \mathbb{E}_{\boldsymbol{\omega}_m}\left[\frac{1}{m} \sum_{l=1}^{m} \exp\left(\omega_l^\top (\mathbf{q}_i + \mathbf{k}_j) - \frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right)\right] \\ &= \mathbb{E}_{\boldsymbol{\omega}_m}\left[\frac{1}{m} \sum_{l=1}^{m} \exp\left(W - \frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right)\right] \\ &= \exp\left(-\frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right) \mathbb{E}_{\boldsymbol{\omega}_m}\left[\frac{1}{m} \sum_{l=1}^{m} \exp\left(W\right)\right] \\ &= \exp\left(-\frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right) \exp\left(\frac{\|\mathbf{q}_i + \mathbf{k}_j\|^2)}{2}\right) \\ &= \exp\left(\frac{\|\mathbf{q}_i + \mathbf{k}_j\|^2 - \|\mathbf{q}_i\|^2 - \|\mathbf{k}_j\|^2}{2}\right) \\ &= \exp\left(\mathbf{q}_i^\top \mathbf{k}_j\right) = SM(\mathbf{q}_i, \mathbf{k}_j) \end{aligned} \tag{68}$$

This means that $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$ is an unbiased estimator for $SM(\mathbf{q}_i, \mathbf{k}_j)$. To measure how good of an approximation $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$ is, we can use the mean square error (MSE):

$$\text{MSE} = \mathbb{E}_{\boldsymbol{\omega}_m}\left[\left(\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) - SM(\mathbf{q}_i, \mathbf{k}_j)\right)^2\right] \tag{69}$$

Since we have $\mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right] = SM(\mathbf{q}_i, \mathbf{k}_j)$, then the MSE simplifies to the variance of $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)$:

$$\begin{aligned}
\text{MSE} &= \mathbb{E}_{\boldsymbol{\omega}_m}\left[\left(\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) - \mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right]\right)^2\right] \\
&= \text{Var}_{\boldsymbol{\omega}_m}\left[\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right] \\
&= \mathbb{E}_{\boldsymbol{\omega}_m}\left[\left(\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right)^2\right] - SM^2(\mathbf{q}_i, \mathbf{k}_j)
\end{aligned} \tag{70}$$

Let's call $X_l = \exp\left(\omega_l^\top(\mathbf{q}_i + \mathbf{k}_j) - \frac{\|\mathbf{q}_i\|^2 + \|\mathbf{k}_j\|^2}{2}\right)$, such that $\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) = \frac{1}{m}\sum_{l=1}^{m} X_l$. We can expand $\widehat{SM}^2(\mathbf{q}_i, \mathbf{k}_j)$ by using the binomial expansion:

$$\widehat{SM}^2(\mathbf{q}_i, \mathbf{k}_j) = \frac{1}{m^2}\left(\sum_{l=1}^{m} X_l^2 + 2\sum_{k=2}^{m}\sum_{l=1}^{k-1} X_k X_l\right) \tag{71}$$

Taking the expectation, we obtain:

$$\mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}^2(\mathbf{q}_i, \mathbf{k}_j)\right] = \frac{1}{m^2}\left(m\mathbb{E}_{\boldsymbol{\omega}_m}\left[X_l^2\right] + m(m-1)\left(\mathbb{E}_{\boldsymbol{\omega}_m}\left[X_l\right]\right)^2\right) \tag{72}$$

We have:

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\omega}_m}\left[X_l^2\right] &= \exp\left(-\|\mathbf{q}_i\|^2 - \|\mathbf{k}_j\|^2\right)\mathbb{E}_{\boldsymbol{\omega}_m}\left[\exp\left(2W\right)\right] \\
&= \exp\left(-\|\mathbf{q}_i\|^2 - \|\mathbf{k}_j\|^2\right)\exp\left(2\|\mathbf{q}_i + \mathbf{k}_j\|^2\right) \\
&= SM^2(\mathbf{q}_i, \mathbf{k}_j)\exp\left(\|\mathbf{q}_i + \mathbf{k}_j\|^2\right)
\end{aligned} \tag{73}$$

We also know that $\mathbb{E}_{\boldsymbol{\omega}_m}\left[X_l\right] = SM(\mathbf{q}_i, \mathbf{k}_j)$, therefore:

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\omega}_m}\left[\widehat{SM}^2(\mathbf{q}_i, \mathbf{k}_j)\right] &= \frac{1}{m}\left(SM^2(\mathbf{q}_i, \mathbf{k}_j)\exp\left(\|\mathbf{q}_i + \mathbf{k}_j\|^2\right) + (m-1)SM^2(\mathbf{q}_i, \mathbf{k}_j)\right) \\
&= \frac{SM^2(\mathbf{q}_i, \mathbf{k}_j)\left(\exp\left(\|\mathbf{q}_i + \mathbf{k}_j\|^2\right) - 1\right)}{m} + SM^2(\mathbf{q}_i, \mathbf{k}_j)
\end{aligned} \tag{74}$$

Finally, putting everything together to compute the MSE, we obtain:

$$\begin{aligned}
\text{MSE} &= \mathbb{E}_{\boldsymbol{\omega}_m}\left[\left(\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j)\right)^2\right] - SM^2(\mathbf{q}_i, \mathbf{k}_j) \\
&= \frac{SM^2(\mathbf{q}_i, \mathbf{k}_j)\left(\exp\left(\|\mathbf{q}_i + \mathbf{k}_j\|^2\right) - 1\right)}{m} + SM^2(\mathbf{q}_i, \mathbf{k}_j) - SM^2(\mathbf{q}_i, \mathbf{k}_j) \\
&= \frac{SM^2(\mathbf{q}_i, \mathbf{k}_j)\left(\exp\left(\|\mathbf{q}_i + \mathbf{k}_j\|^2\right) - 1\right)}{m}
\end{aligned} \tag{75}$$

We can see that MSE $\to 0$ as $m \to \infty$. This means that the approximation is exact when $m \to \infty$:

$$\widehat{SM}(\mathbf{q}_i, \mathbf{k}_j) \to SM(\mathbf{q}_i, \mathbf{k}_j) \quad \text{if } m \to \infty \tag{76}$$

This guarantees that the approximation is asymptotically accurate for large m. In the *Performer* article, they found that the MSE falls to $\sim 10^{-2}$ when $m \gtrsim 125$ for their specific experimental design. This makes sense when we consider that $1/125 = 0.8 \times 10^{-2}$.

Instead of directly computing the complicated softmax function, we sample a bunch of random "directions" (via random projections) and then average the contributions. This is similar to using

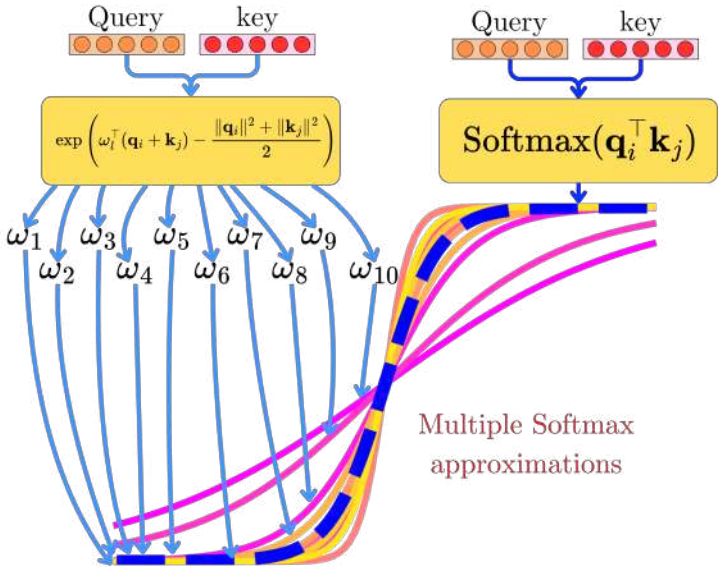

Fig. 37 *Performer projects the keys and queries, where each projection is only a rough approximation to the softmax transformation. Because the keys and queries are projected multiple times, slightly differently for different values of ω_l, it leads, on average, to a high-quality approximation of the softmax function.*

Monte Carlo integration to estimate an expected value. Each random projection gives you a "cheap" estimate of the similarity between $\mathbf{q}_i$ and $\mathbf{k}_j$. When you average enough of these, the law of large numbers ensures that the average converges to the true kernel value. The feature map ϕ transforms each input (query or key) into a new space such that their dot product in that space is, in expectation, equal to $SM(\mathbf{q}_i, \mathbf{k}_j)$. This means that, while each individual random projection is a rough approximation, their combined effect (via averaging) closely mimics the behavior of the exponential function. Even if one random projection is noisy, when you aggregate (or average) many of them, the noise averages out, and you recover the desired value. In high dimensions, many random projections tend to preserve the "geometry" of the data, so although each projection only gives partial information, together, they capture the full non-linear similarity (see fig. 37).

We now have a good approximation of the softmax kernel, and we can use it to compute the context vectors

$$\begin{aligned}\mathbf{c}_i &= \frac{\sum_{j=1}^N SM(\mathbf{q}_i,\mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^N SM(\mathbf{q}_i,\mathbf{k}_j)} \\ &\approx \frac{\sum_{j=1}^N \widehat{SM}(\mathbf{q}_i,\mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^N \widehat{SM}(\mathbf{q}_i,\mathbf{k}_j)} \\ &= \frac{\sum_{j=1}^N \phi(\mathbf{q}_i)^\top\phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_{j=1}^N \phi(\mathbf{q}_i)^\top\phi(\mathbf{k}_j)}\end{aligned} \tag{77}$$

As for the *Linear Transformer*, the associativity property of matrix multiplication allows us to perform the computation in $\mathcal{O}(N)$ instead of $\mathcal{O}(N^2)$:

$$\mathbf{c}_i = \underbrace{\frac{\sum_{j=1}^N \left(\phi(\mathbf{q}_i)^\top\phi(\mathbf{k}_j)\right)\mathbf{v}_j}{\sum_{j=1}^N \phi(\mathbf{q}_i)^\top\phi(\mathbf{k}_j)}}_{\sim\mathcal{O}(N^2)} = \underbrace{\frac{\phi(\mathbf{q}_i)^\top\sum_{j=1}^N \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top\sum_{j=1}^N \phi(\mathbf{k}_j)}}_{\sim\mathcal{O}(N)} \tag{78}$$

4.4 To Summarize

Attention Type	Key Innovations & Benefits	Limitations & Downsides
Linformer	Projects attention matrix to lower rank, requiring minimal information loss while achieving linear scaling	Performance degrades with very long sequences; low-rank approximation loses fine-grained dependencies; projection matrices increase parameter count
Linear Transformer	Replaces softmax with kernel function that allows associative property, enabling recurrent-like computation with constant memory	Feature map choice significantly impacts performance; generally lower quality than softmax attention; less emphasis on key tokens compared to softmax
Performer	Provides unbiased approximation of softmax kernel using orthogonal random features, maintaining better accuracy than Linear Transformer	Requires more random features (higher dimension) for better approximation; training stability issues; sensitivity to initialization and hyperparameters

5 Memory Efficient Attention

So far, we have mainly explored how to reduce the complexity of the attention mechanisms by approximating the vanilla attention. The vanilla attention has a strict $\mathcal{O}(N^2)$ time complexity, but the space complexity doesn't need to be $\mathcal{O}(N^2)$! Computing $Q^\top K$ requires $\sim \mathcal{O}(N^2)$ operations, but the full $N \times N$ alignment scores and attention matrices do not need to be fully materialized all at once in memory. As models and sequence lengths scale, minimizing the memory requirements at training and inference time becomes essential to better utilize the underlying hardware.

5.1 Self-attention Does Not Need $\mathcal{O}(N^2)$ Memory

Let's consider again the computation of the context vectors. Once again, we ignore attention heads for simplicity, as it does not change the analysis:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)}. \tag{79}$$

Here is how we arrive to the typical $\sim \mathcal{O}(N^2)$ space complexity:

1. The typical assumption is that we first compute the dot product between the query $\mathbf{q}_i$ and all the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_N]$:

$$\mathbf{e}_i = \left[\frac{\mathbf{q}_i^\top \mathbf{k}_1}{\sqrt{d_{\text{model}}}}, \ldots, \frac{\mathbf{q}_i^\top \mathbf{k}_N}{\sqrt{d_{\text{model}}}}\right] \tag{80}$$

   where $\mathbf{e}_i$ is the alignment score vector of size N for the query $\mathbf{q}_i$, which leads to the $N \times N$ matrix for the N queries.

2. We then perform the softmax transformation:

$$\mathbf{a}_i = \frac{1}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)} \left[\exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_1}{\sqrt{d_{\text{model}}}}\right), \ldots, \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_N}{\sqrt{d_{\text{model}}}}\right)\right] \tag{81}$$

   Here, $\mathbf{a}_i$ is the attention vector of size N for the query $\mathbf{q}_i$. Again, for N queries, it leads to the typical $N \times N$ attention matrix.

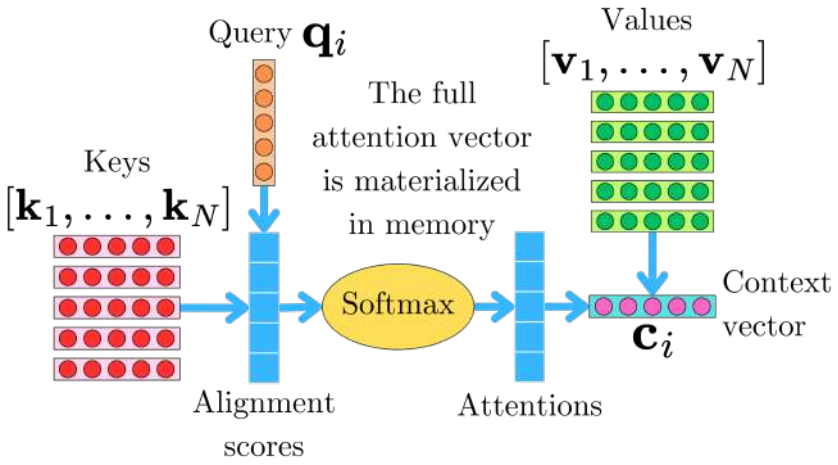

Fig. 38 *Naively computing the dot product between a query $\mathbf{q}_i$ and the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_N]$ leads to an alignment score vector and an attention vector per head of dimension N, with a total $\mathcal{O}(N^2)$ space complexity for N queries.*

3. And finally, we project $\mathbf{a}_i$ onto the different values $V = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$, which leads to $\mathbf{c}_i$:

$$\begin{aligned} \mathbf{c}_i &= \mathbf{a}_i^\top V \\ &= \sum_{j=1}^{N} a_{ij} \mathbf{v}_j \end{aligned} \tag{82}$$

Therefore, naively computing the alignment scores and the attention matrices first forces the materialization of those matrices in memory, which leads to the $\mathcal{O}(N^2)$ space complexity (see fig. 38).

However, we do not need to order the computations in this manner! In 2021, Rabe and Staats [24] realized that by reordering the operations, we can greatly reduce the requirements on the memory. The idea is to consider the unnormalized context vector $\tilde{\mathbf{c}}_i$ and the normalization constant $\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$ of the softmax transformation separately:

$$\begin{aligned} \tilde{\mathbf{c}}_i &= \sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j \\ s_i &= \sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \\ \mathbf{c}_i &= \frac{\tilde{\mathbf{c}}_i}{s_i} \end{aligned} \tag{83}$$

Because $\tilde{\mathbf{c}}_i$ and s_i are just sums, we can easily loop through the key-value pairs to compute the context vector:

$\tilde{\mathbf{c}}_i = 0, s_i = 0$ ▷ We initialize $\tilde{\mathbf{c}}_i$ and s_i
for $j = 1$ to N **do** ▷ We loop through the key-value pairs $(\mathbf{k}_j, \mathbf{v}_j)$
  $\tilde{\mathbf{c}}_i \leftarrow \tilde{\mathbf{c}}_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j$
  $s_i \leftarrow s_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$
end for
$\mathbf{c}_i = \frac{\tilde{\mathbf{c}}_i}{s_i}$ ▷ We compute the final context vector
Return: $\mathbf{c}_i$

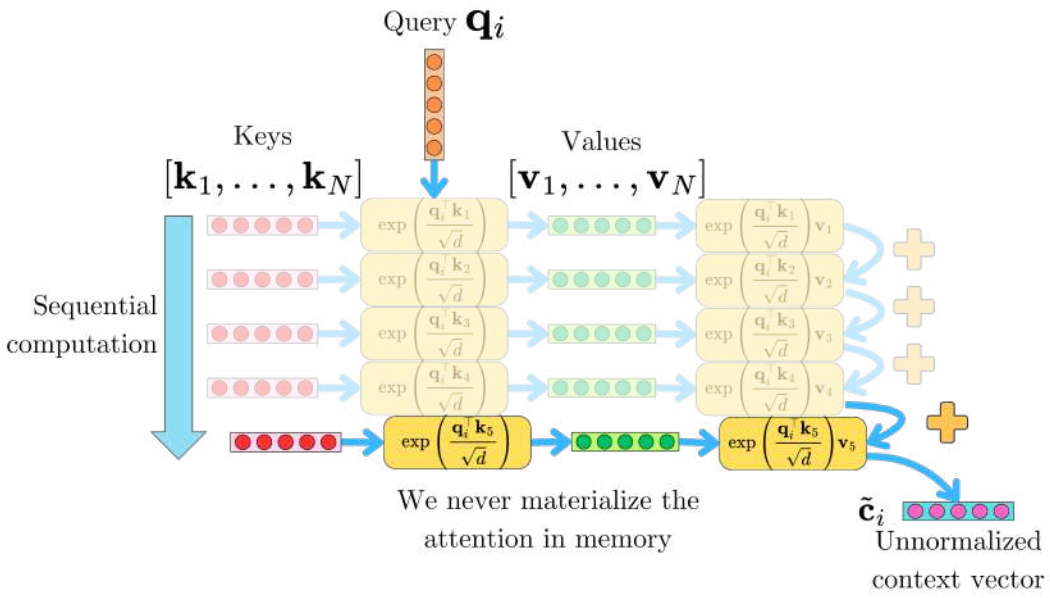

Fig. 39 *Instead of computing the dot product between $\mathbf{q}_i$ and the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_N]$ in one operation, we can iterate through the key-value pairs sequentially without having to materialize the full attention tensor in memory.*

At any point during the for-loop, we only need to store the intermediary values of $\tilde{\mathbf{c}}_i$ and s_i. $\tilde{\mathbf{c}}_i$ is a vector of size d_{model} and s_i is a scalar. Therefore, for one query, we need constant space complexity $\mathcal{O}(1)$ to compute one context vector (see fig. 39). Even iterating through all the queries, we never need to capture more than the intermediary values of $\tilde{\mathbf{c}}_i$ and s_i, so we can compute the full attention mechanism in $\mathcal{O}(1)$ space complexity:

for $i = 1$ to N **do** ▷ We loop through the queries
  $\tilde{\mathbf{c}}_i = 0$
  $s_i = 0$
  for $j = 1$ to N **do** ▷ We loop through the key-value pairs $(\mathbf{k}_j, \mathbf{v}_j)$
    $\tilde{\mathbf{c}}_i \leftarrow \tilde{\mathbf{c}}_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j$
    $s_i \leftarrow s_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$
  end for
  $\mathbf{c}_i = \frac{\tilde{\mathbf{c}}_i}{s_i}$
end for
Return: $[\mathbf{c}_1, \ldots, \mathbf{c}_N]$

In reality, this is not a practical solution because sequential operations are not adapted to the parallelization capability of the CPU, GPU, or TPU hardware commonly used for neural network computations. In practice, the queries, keys, and values are partitioned into chunks to allow for a high degree of parallelization while keeping the memory requirement low (see fig. 40). Let's assume that we partition the queries into n_q chunks and the keys and values into n_k chunks:

$$\begin{aligned} Q &= [Q_1, Q_2, \ldots, Q_{n_q}] \\ K &= [K_1, K_2, \ldots, K_{n_k}] \\ V &= [V_1, V_2, \ldots, V_{n_k}] \end{aligned} \tag{84}$$

where each Q_i is a $\frac{N}{n_q} \times d_{\text{model}}$ matrix and K_i, V_i are $\frac{N}{n_k} \times d_{\text{model}}$ matrices. Let's call $N_q = \frac{N}{n_q}$ the number of queries per chunk, and $N_k = \frac{N}{n_k}$ the number of key-value pairs per chunk. We can now iterate through the chunks exactly in the same way:

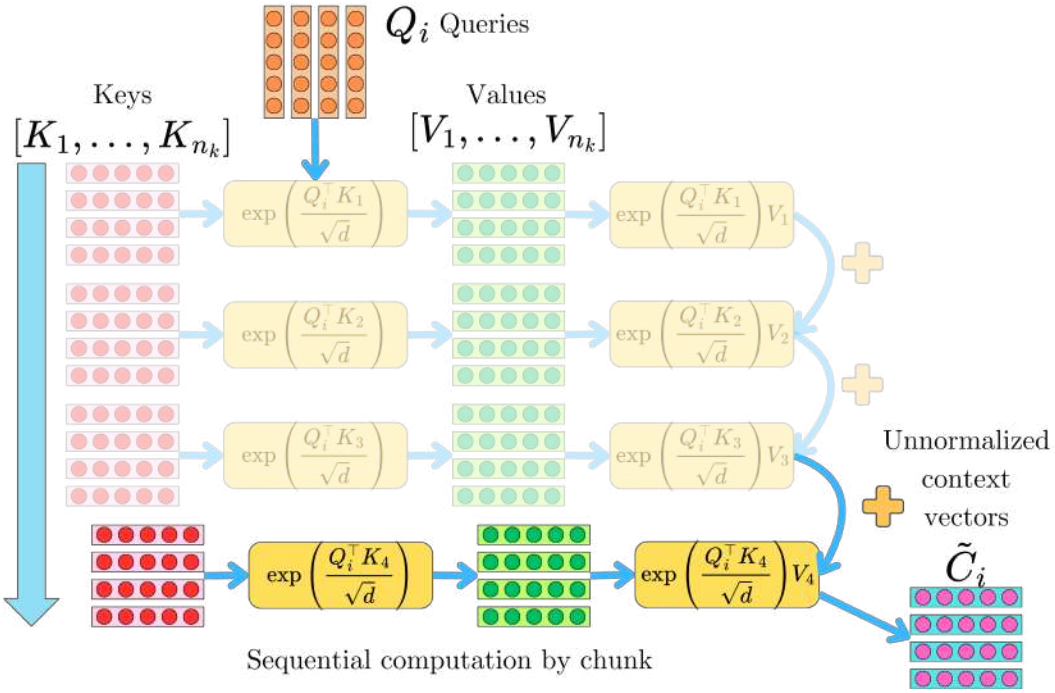

Fig. 40 *Instead of iterating through each key-value pair sequentially as in fig. 39, we can chunk the queries, keys, and values into groups, and iterate through those groups. This prevents us from materializing the full attention tensor in memory while utilizing the parallelization capabilities of the hardware.*

for $i = 1$ to n_q **do** ▷ Process each query chunk
  $\tilde{C}_i \leftarrow \mathbf{0}$ ▷ Initialize output accumulator
  $S_i \leftarrow \mathbf{0}$ ▷ Initialize denominator accumulator
  for $j = 1$ to n_k **do** ▷ Process each key-value chunk
    $A_{ij} \leftarrow \exp\left(\frac{Q_i K_j^\top}{\sqrt{d_{\text{model}}}}\right)$
    $\tilde{C}_i \leftarrow \tilde{C}_i + A_{ij} V j$
    $S_i \leftarrow S_i + \sum_{\text{columns}} A_{ij}$
  end for
  $C_i \leftarrow \tilde{C}_i \oslash S_i$ ▷ Normalize output (element-wise division)
end for
Return: $C = [C_1; C_2; \ldots; C_{n_q}]$ ▷ Concatenate results

As before, we must store intermediary values of $\tilde{C}_i$ and S_i. In this context, $Q_i K_j^\top$ is a matrix of size $N_q \times N_k$, and so is A_{ij}. $\tilde{C}_i$ is a matrix of size $N_q \times d_{\text{model}}$ and S_i is a vector of size N_q. Therefore, the space complexity is $\mathcal{O}(N_q \times N_k + N_q \times d_{\text{model}})$. To balance the number of chunks and the number of key-value pairs per chunk, they chose $N_k = n_k = \sqrt{N}$ and fixed $N_q = 1024$. This results in a space complexity:

$$\mathcal{O}(1024\sqrt{N} + 1024 d_{\text{model}}) = \mathcal{O}(\sqrt{N}) \tag{85}$$

This approach allows for efficient tensor operations within each chunk while dramatically reducing the peak memory requirements. Note that no approximation has been made, and it is mathematically equivalent to the vanilla attention mechanism. However, this approach is slower (8-13% slower during the forward pass and 30-35% slower during the backward pass) due to the sequential computations, but it enables the processing of much longer sequences that would otherwise be impossible due to memory constraints. Along with the *FlashAttention*, it is one of the memory optimization strategies used in the *xFormers* package developed by Meta [19] and used in the development of the Llama models [13, 30, 31].

Until now, we have been ignoring the numerical stability of the softmax computation, but most implementations (PyTorch, TensorFlow, etc.) are using a couple of tricks to ensure its stability. Let's remind ourselves of the softmax function:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{86}$$

Computing e^{x_i} can be tricky because if $x_i \geq 89$, then $e^{x_i} \approx 4.4 \times 10^{38}$, which exceeds the floating-point limit of 3.4×10^{38} for a 32-bit float number, potentially leading to float overflow errors. To prevent this from happening, we typically modify the exponent by finding the maximum x_i value:

$$m = \max\{x_1, x_2, \ldots, x_N\} \tag{87}$$

and we rescale the exponential functions:

$$\begin{aligned}
\text{Softmax}(x_i) &= \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}} \\
&= \frac{e^{-m} e^{x_i}}{e^{-m} \sum_{j=1}^{N} e^{x_j}} \\
&= \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}
\end{aligned} \tag{88}$$

This leads to an equivalent formulation for the softmax function while ensuring stable computation of the exponential function $e^{x_i - m} \leq 1$ for all x_i. This can easily be applied to the vanilla attention, but we need to be careful when using this trick for the chunk-wise attention computation since we cannot know the overall maximum when considering a specific chunk. Consider the first query chunk Q_1 and the first key-value pair (K_1, V_1) chunks. We first find the maximum value for each query:

$$\begin{aligned}
\mathbf{m}_{11} &= \max_{\text{columns}} \frac{Q_1 K_1^\top}{\sqrt{d_{\text{model}}}} \\
&= \frac{1}{\sqrt{d_{\text{model}}}} \begin{pmatrix} \max\left\{\mathbf{q}_1^\top \mathbf{k}_1, , \ldots, \mathbf{q}_1^\top \mathbf{k}_{N_k}\right\} \\ \max\left\{\mathbf{q}_2^\top \mathbf{k}_1, , \ldots, \mathbf{q}_2^\top \mathbf{k}_{N_k}\right\} \\ \vdots \\ \max\left\{\mathbf{q}_{N_q}^\top \mathbf{k}_1, , \ldots, \mathbf{q}_{N_q}^\top \mathbf{k}_{N_k}\right\} \end{pmatrix}
\end{aligned} \tag{89}$$

$\mathbf{m}_{11}$ is a vector of size N_q. We compute the unnormalized attention matrix for those chunks by shifting each row by $\mathbf{m}_{11}$:

$$\begin{aligned}
A_{11}^* &\leftarrow \exp\left(\frac{Q_1 K_1^\top}{\sqrt{d_{\text{model}}}} - \mathbf{m}_{11}\right) \\
\tilde{C}_1^* &\leftarrow A_{11} V_1 \\
S_1^* &\leftarrow \sum_{\text{columns}} A_{11}
\end{aligned} \tag{90}$$

Here, we implicitly assume that $\mathbf{m}_{11}$ is broadcasted to all the rows in $\frac{Q_1 K_1^\top}{\sqrt{d_{\text{model}}}}$. Now, let's look at the second key-value chunk pair (K_2, V_2). We compute the maximum $\mathbf{m}_{12}$ for each query in Q_1:

$$\begin{aligned}
\mathbf{m}_{12} &= \max_{\text{columns}} \frac{Q_1 K_2^\top}{\sqrt{d_{\text{model}}}} \\
&= \frac{1}{\sqrt{d_{\text{model}}}} \begin{pmatrix} \max\left\{\mathbf{q}_1^\top \mathbf{k}_1, , \ldots, \mathbf{q}_1^\top \mathbf{k}_{N_k}\right\} \\ \max\left\{\mathbf{q}_2^\top \mathbf{k}_1, , \ldots, \mathbf{q}_2^\top \mathbf{k}_{N_k}\right\} \\ \vdots \\ \max\left\{\mathbf{q}_{N_q}^\top \mathbf{k}_1, , \ldots, \mathbf{q}_{N_q}^\top \mathbf{k}_{N_k}\right\} \end{pmatrix}
\end{aligned} \tag{91}$$

And we compute again the shifted unnormalized attention matrix for those chunks:

$$A_{12}^* \leftarrow \exp\left(\frac{Q_1 K_2^\top}{\sqrt{d_{\text{model}}}} - \mathbf{m}_{12}\right) \tag{92}$$

We are going to compute a running maximum vector $\mathbf{m}_{\text{running}}$ as we iterate through the different chunks:

$$\mathbf{m}_{\text{running}} = \max(\mathbf{m}_{11}, \mathbf{m}_{12}) \tag{93}$$

We know that $\tilde{C}_1^*$ and S_1^* currently depend on $\mathbf{m}_{11}$ while A_{12} depends on $\mathbf{m}_{12}$, so we correct $\tilde{C}_1^*$ and S_1^* to make sure they depend on $\mathbf{m}_{\text{running}}$ instead:

$$\begin{aligned}\tilde{C}_1^* &\leftarrow \tilde{C}_1^* \exp(\mathbf{m}_{11} - \mathbf{m}_{\text{running}}) \\ \tilde{S}_1^* &\leftarrow \tilde{S}_1^* \exp(\mathbf{m}_{11} - \mathbf{m}_{\text{running}})\end{aligned} \tag{94}$$

We can now accumulate those values:

$$\begin{aligned}\tilde{C}_1^* &\leftarrow \tilde{C}_1^* + A_{12} V_2 \\ S_1^* &\leftarrow S_1^* + \sum_{\text{columns}} A_{12}\end{aligned} \tag{95}$$

At this point, we can easily factor out $\exp(\mathbf{m}_{\text{running}})$:

$$\begin{aligned}\tilde{C}_1^* &= \tilde{C}_1 \exp(\mathbf{m}_{\text{running}}) \\ S_1^* &= S_1 \exp(\mathbf{m}_{\text{running}})\end{aligned} \tag{96}$$

which allows the computation of the current normalized context vector by canceling out the $\exp(\mathbf{m}_{\text{running}})$ from the numerator and denominator:

$$\tilde{C}_1^* \oslash S_1^* = \tilde{C}_1 \oslash S_1 = C_1 \tag{97}$$

In fact, for any key-value chunk pair (K_i, V_i), we can correct $\tilde{C}_1^*$ and S_1^* of the previous iteration:

$$\mathbf{m}_{\text{running,new}} \leftarrow \max(\mathbf{m}_{\text{running, prev}}, \mathbf{m}_{1i}) \tag{98}$$

$$\begin{aligned}\tilde{C}_1^* &\leftarrow \tilde{C}_1^* \exp(\mathbf{m}_{\text{running,prev}} - \mathbf{m}_{\text{running, new}}) \\ \tilde{S}_1^* &\leftarrow \tilde{S}_1^* \exp(\mathbf{m}_{\text{running,prev}} - \mathbf{m}_{\text{running, new}}) \\ \tilde{C}_1^* &\leftarrow \tilde{C}_1^* + A_{1i} V_i \\ S_1^* &\leftarrow S_1^* + \sum_{\text{columns}} A_{1i}\end{aligned} \tag{99}$$

We can now adapt the original chunk-based algorithm to account for the stabilization provided by the running maximum:

for $i = 1$ to n_q **do** ▷ Process each query chunk
  $\tilde{C}_i^* \leftarrow \mathbf{0}$ ▷ Initialize output accumulator
  $S_i^* \leftarrow \mathbf{0}$ ▷ Initialize denominator accumulator
  $\mathbf{m}_{\text{run,prev}} \leftarrow -\infty$ ▷ Initialize running max vector for queries in Q_i
  for $j = 1$ to n_k **do** ▷ Process each key-value chunk
    $E_{ij} \leftarrow \frac{Q_i K_j^\top}{\sqrt{d_{\text{model}}}}$ ▷ Compute scores
    $\mathbf{m}_{ij} \leftarrow \max(E_{ij}, \text{axis} = 1)$ ▷ Compute chunk max
    $\mathbf{m}_{\text{run,new}} \leftarrow \max(\mathbf{m}_{\text{run,prev}}, \mathbf{m}_{ij})$ ▷ Update running max
    $\tilde{C}_i^* \leftarrow \tilde{C}_i^* \odot \exp(\mathbf{m}_{\text{run,prev}} - \mathbf{m}_{\text{run,new}})$ ▷ Element-wise scaling
    $S_i^* \leftarrow S_i^* \odot \exp(\mathbf{m}_{\text{run,prev}} - \mathbf{m}_{\text{run,new}})$
    $\mathbf{m}_{\text{run,prev}} \leftarrow \mathbf{m}_{\text{run,new}}$
    $A_{ij} \leftarrow \exp(E_{ij} - \mathbf{m}_{\text{run,new}})$ ▷ Stabilize scores
    $\tilde{C}_i^* \leftarrow \tilde{C}_i^* + A_{ij} V_j$ ▷ Accumulate
    $S_i^* \leftarrow S_i^* + \sum_{\text{columns}} A_{ij}$ ▷ Accumulate sum
  end for
  Normalize: $C_i \leftarrow \tilde{C}_i^* \oslash S_i^*$ ▷ Normalize
end for

18: **Return:** $C = [C_1; C_2; \ldots; C_{n_q}]$ ▷ Concatenate results

5.2 The FlashAttention

The *FlashAttention*, introduced by Dao et al. in 2022 [10], might be one of the most widely adopted modifications of the vanilla attention mechanism. It is the underlying strategy in the Llama models and all the derivative models available on the Hugging Face website. It uses a similar approach as the one developed by Rabe and Staats [24], but it is specifically designed to optimize the computations on GPU kernels, dramatically slashing costly global memory reads/writes. As a result, *FlashAttention* not only reduces memory usage but is also significantly faster than baseline attention implementations, bridging the gap from a theoretical insight to a practical, hardware-aware algorithm.

5.2.1 The GPU Architecture

Before we can introduce *FlashAttention*, we need to dive into how the computations and memory usage are handled in a GPU card. **CUDA cores** are the basic computational units within NVIDIA GPUs. They are analogous to CPU cores but much more numerous and specialized. Each CUDA core performs arithmetic operations (like floating-point calculations: addition, subtraction, multiplication, division, etc.), and modern GPUs contain thousands of these cores.

GPU	Number of CUDA cores
NVIDIA Tesla V100 (Volta)	5,120
NVIDIA Tesla T4 (Turing)	2,560
NVIDIA A100 (Ampere)	6,912
NVIDIA H100 (Hopper)	16,896
NVIDIA GeForce RTX 3090 (Ampere)	10,496
NVIDIA GeForce RTX 4090 (Ada Lovelace)	16,384

CUDA cores are physically grouped into **Streaming Multiprocessors** (SMs). SMs serve as self-contained computational units that manage and execute parallel workloads. Along CUDA cores, we find in SMs **tensor cores** specializing in matrix multiplication and **special function units** (SFUs) specialized for complex mathematical operations such as sine, cosine, exponential, logarithm, etc. Allocated to each SM, we have a local **static random-access memory (SRAM)** for fast read/write of data and an **L1 cache** and **texture units** to accelerate memory access patterns (see fig. 41).

GPU programming divides work into thousands of **threads** that run in parallel, and the GPU hardware schedules these threads to run on available CUDA cores. Threads are bundled into **thread blocks** (typically 32-1024 threads per block), and threads within the same block can cooperate and share data. In GPU programming (especially with CUDA), a **kernel** is a function that is executed on the GPU in parallel by many threads. When you launch a kernel from the host (CPU), you specify a grid configuration that defines how many thread blocks and threads per block will run the function concurrently. This massive parallelism is one of the key strengths of GPUs for computations like matrix multiplication and deep learning operations. Think about the SMs as the physical partition where the computation happens, and thread blocks as the logical partition of the computation. The CUDA runtime dynamically partitions the SM's physical SRAM among the thread blocks currently running on that SM, and each thread block gets its own dedicated portion of the SM's SRAM (see fig. 42). From a programming perspective, you declare how much shared memory each thread block needs when launching your kernel, and the GPU hardware and CUDA runtime handle the actual allocation. Thread blocks are divided into **warps** (groups of 32 threads) that execute in lockstep, meaning they all execute the same instruction simultaneously, just on different data. This execution model is called SIMT (Single Instruction, Multiple Threads). When you launch a thread block with, say, 256 threads, the GPU hardware automatically divides these into 8 warps of 32 threads each. The warp is the true atomic unit of execution on NVIDIA GPUs.

In addition to the SRAM, the GPU is provided with a global **High Bandwidth Memory (HBM)** that is accessible to all threads across all SMs (see fig. 43). It is where data from the CPU is initially loaded before GPU computation begins, and it is the only way for different thread blocks to share results with each other. It holds input data, intermediate results that exceed SRAM capacity, and

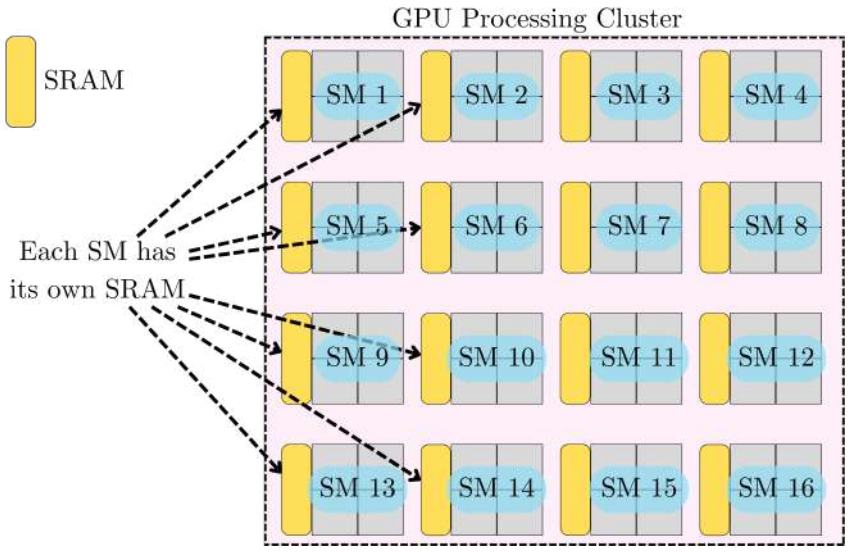

Fig. 41 *The GPU processing cluster is represented by an array of Streaming Messors (SMs), each with its own local static random-access memory (SRAM).*

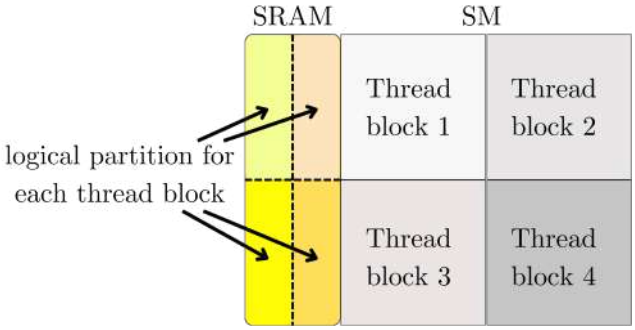

Fig. 42 *Each thread block gets its own logical partition of the local SRAM.*

final outputs. When a thread needs more local storage than can fit in registers (see below), the excess "spills" to global memory. The HBM has a much greater capacity of ∼40-80GB, whereas the SRAM is typically ∼48-192KB per SM. Despite being much larger, the HBM is much slower. The SRAM's bandwidth is about ∼19 TB/s, whereas the HBM bandwidth is around ∼1.5-2.0 TB/s, (ten times slower!).

	Size	Bandwidth
HBM	∼40-80GB	∼1.5-2.0 TB/s
SRAM	∼48-192KB per SM	∼19 TB/s

Besides SRAM and HBM, we have **registers**. Registers are the smallest and fastest type of on-chip memory available to each thread on a GPU. Each thread in a CUDA kernel has its own private set of registers, where it stores temporary variables, intermediate results, and other data that needs to be accessed very quickly. Because registers are extremely fast compared to global memory or even shared memory, they are essential for achieving high performance in GPU computations. However, the number of registers per thread is limited, so efficient use and careful management are imperative in highly optimized kernels.

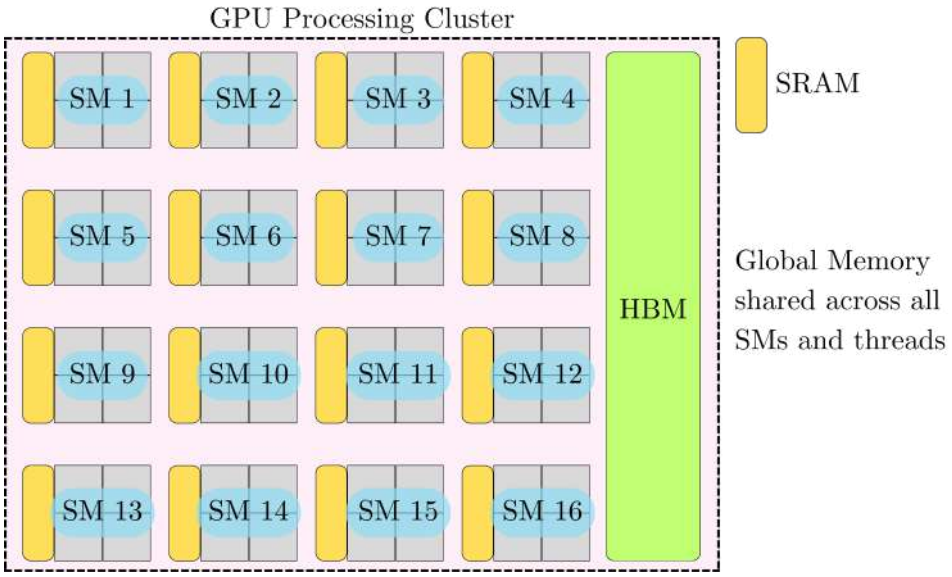

Fig. 43 *The High Bandwidth Memory (HBM) is a larger but with slower memory access available to all the SMs.*

5.2.2 FlashAttention-1

The innovation related to *FlashAttention* is to make use of the accelerated I/O provided by the SRAM between the computations happening on the SMs and the intermediary data writes and reads to and from the memory. Despite being called "High Bandwidth Memory," global memory access represents the primary bottleneck in attention computation because:

- Every thread block can only directly access its portion of SRAM
- Communication between thread blocks must happen through global memory
- The standard attention implementation materializes large matrices in global memory
- Multiple round-trips to global memory create a serious performance penalty

By restructuring the algorithm to minimize global memory access and doing more computation in shared memory instead, *FlashAttention* achieves impressive performance gains.

Let's first consider the vanilla self-attention and the different data access from and to HBM:

1. **From hidden states to queries, keys, and values:**

   $$Q = HW^Q, K = HW^K, V = HW^V.$$

   H, W^Q, W^K, W^V are stored in HBM, moved to the thread blocks, and Q, K, V are moved back to HBM.

2. **From queries and keys to alignment score:**

   $$E = Q^\top K$$

   K and Q are stored in HBM, moved onto the SMs, typically with internal tiling that loads small blocks of Q and K into (SRAM) to perform fast multiplication. However, the entire E matrix is then written back to global memory (fig. 44).

3. **Masking:** If causal or padding masks are applied, the kernel modifies E (for example, setting certain entries to $-\infty$) either in place or into a new buffer, reading and updating in global memory.

4. **From alignment scores to attentions:** The softmax is computed row-wise on E to obtain the probability matrix

   $$A = \text{Softmax}(E)$$

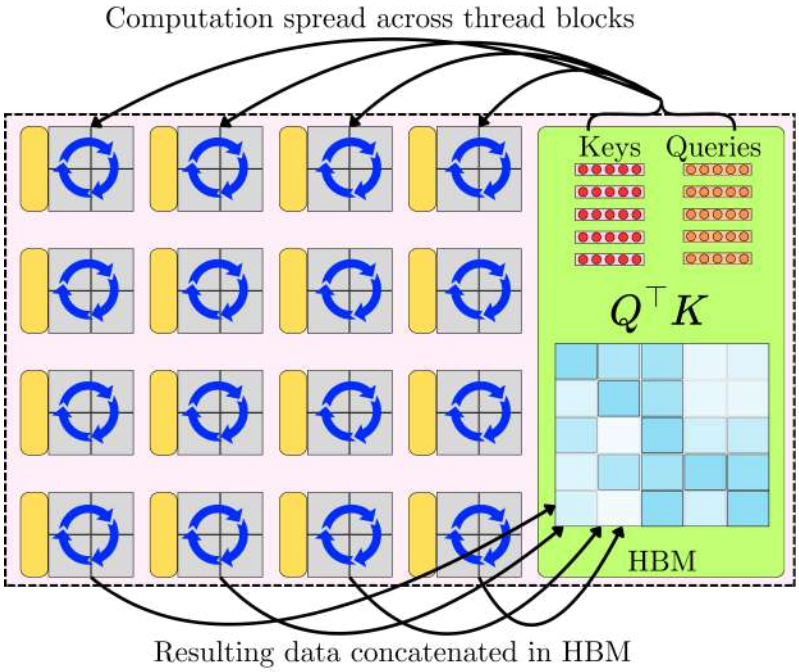

Fig. 44 *To compute the alignment score in the vanilla attention, the keys and queries are distributed across the thread blocks, where the partitioned computing happens. Once the full alignment tensor is computed, it is saved in the global memory.*

This involves first computing the row-wise maximum

$$\mathbf{m} = \max_{\text{columns}}(E)$$

then the row-wise sum,

$$S = \sum_{\text{columns}} \exp(E - \mathbf{m})$$

and finally normalizing,

$$A = \exp(E - \mathbf{m}) \oslash S.$$

E is loaded from global memory, the reduction is performed (often using shared memory SRAM for efficiency), and the result A is written back to global memory (fig. 45).

5. **From attentions to context vectors:**

$$C = AV^\top$$

We load A and V onto the SMs, internally tiling the matrices, loading chunks into shared memory, and writing the final result C back to the global memory (fig. 46).

On the other hand, with *FlashAttention*, we are going to make use of the low-memory computation by chunk of the self-attention introduced by Rabe and Staats:

1. **Input preparation and tiling:** The input matrices Q, K, and V initially reside in global memory (HBM). The algorithm partitions the matrices into blocks:

$$\begin{aligned} Q &= \left[Q_1, Q_2, \ldots, Q_{n_q}\right] \\ K &= \left[K_1, K_2, \ldots, K_{n_k}\right] \\ V &= \left[V_1, V_2, \ldots, V_{n_k}\right] \end{aligned} \tag{100}$$

Before processing a block pair, the corresponding Q_i, K_j, and V_j are explicitly loaded from global memory into on-chip shared memory (SRAM) and registers. This uses CUDA programming techniques to exploit the low-latency, high-bandwidth SRAM (fig. 48).

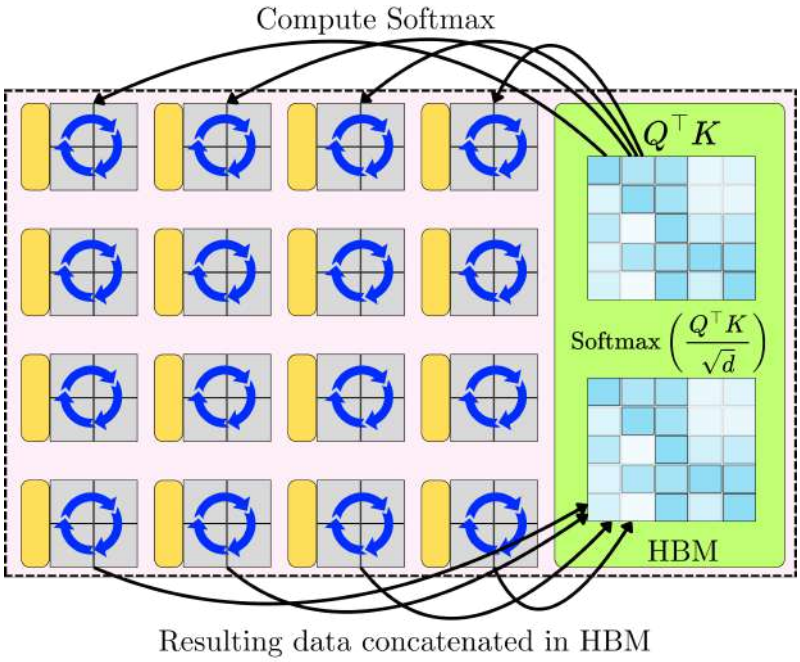

Fig. 45 *To compute the attention tensor, the alignment score tensor is distributed across the thread blocks, potentially with multiple back-and-forth between the SRAM and HBM to compute the different quantities. Ultimately, the resulting attention tensor is written back to the global memory.*

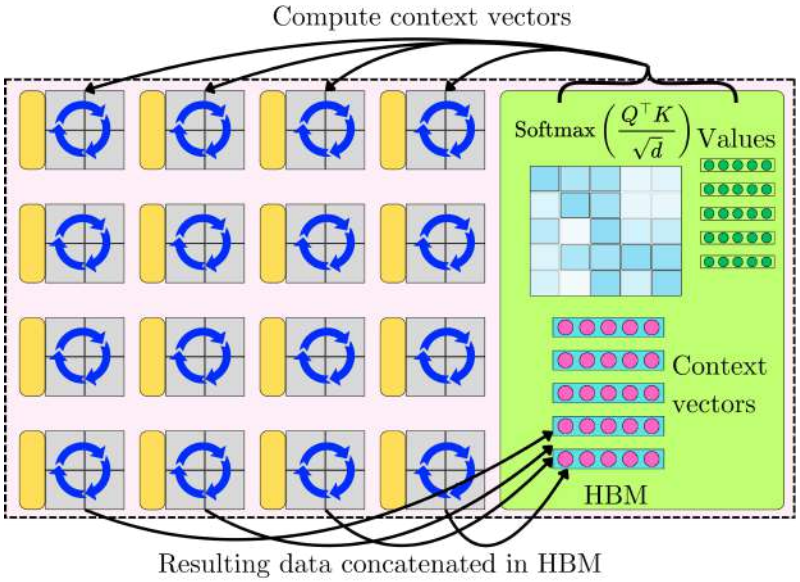

Fig. 46 *Loading the values and the attention matrix from HBM, we can compute the final context vectors that are written back in HBM.*

2. **Computing the partial alignment scores:** For a given pair (i, j) of blocks, the algorithm computes the local score matrix:

$$E_{ij} = Q_i K_j^\top$$

The resulting matrix E_{ij} is stored temporarily in registers or kept in shared memory for immediate use, and it is not written out to global memory (fig. 49).

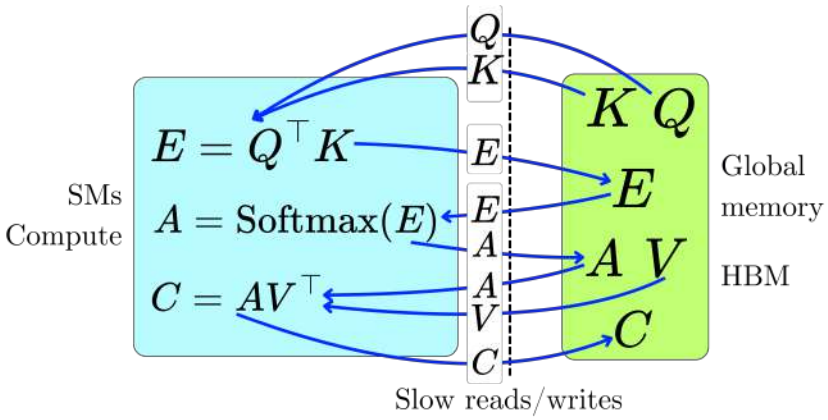

Fig. 47 *The full set of computations happening in the self-attention layer consists of multiple memory access reads/writes of large tensors from global memory. Beyond the time associated with the computations, the overall latency is largely impacted by the slow memory access requirements for the naive vanilla attention.*

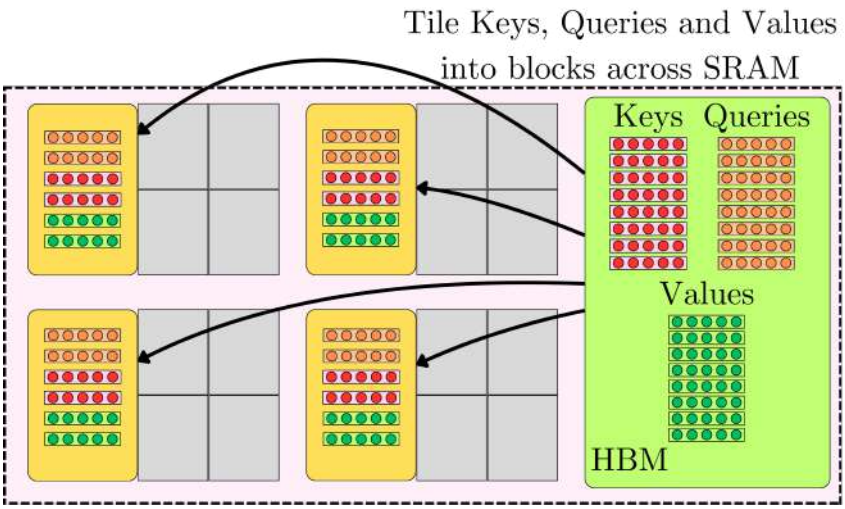

Fig. 48 *For the FlashAttention, the keys, queries, and values are initially tiled and distributed from HBM to SRAM in preparation for the local computation of the attention.*

3. **Local softmax computation:** Within the same tile, the algorithm computes the block's row-wise maximum
$$\mathbf{m}_{ij} = \max_{\text{columns}} E_{ij}$$
then the row-wise sum of exponentials
$$S_{ij} = \sum_{\text{columns}} \exp\left(E_{ij} - \mathbf{m}_{ij}\right)$$
and the unnormalized context vectors
$$C_{ij} = \exp\left(E_{ij} - \mathbf{m}_{ij}\right) V_i.$$
These operations (exponentiation, reduction for the sum, and elementwise multiplications) are performed using on-chip shared memory and registers. The computed $\mathbf{m}_{ij}$, S_{ij}, and C_{ij} are kept on-chip, and they are not stored as full $N \times N$ matrices but rather used immediately in an online update (fig. 50).
4. **Online aggregation across blocks:** Now that we computed $\mathbf{m}_{ij}$, S_{ij}, and C_{ij} for the key-value block j and all the query blocks, we can iterate through the query blocks and

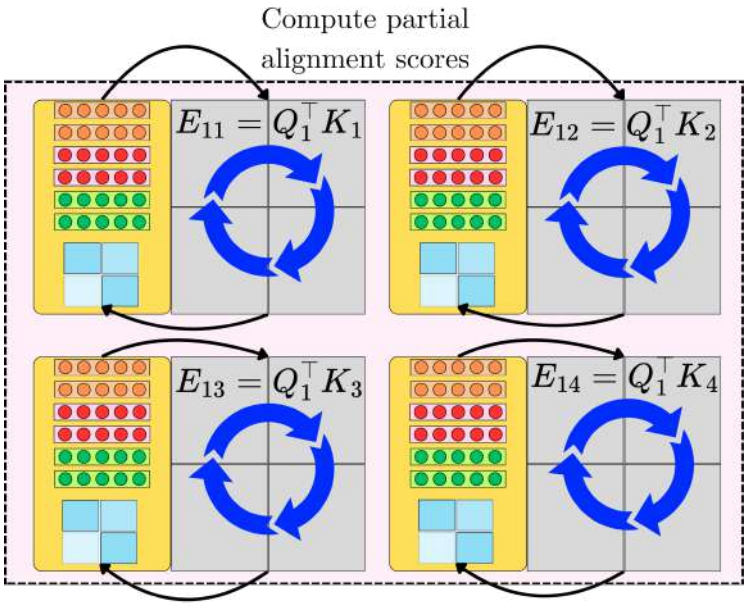

Fig. 49 *Once chunks of keys, queries, and values are made available in the local SRAM, the local alignment score tensor can be computed for those chunks. The size of the chunks is chosen so that the resulting local alignment score tensor can be written to the local SRAM instead of back to HBM.*

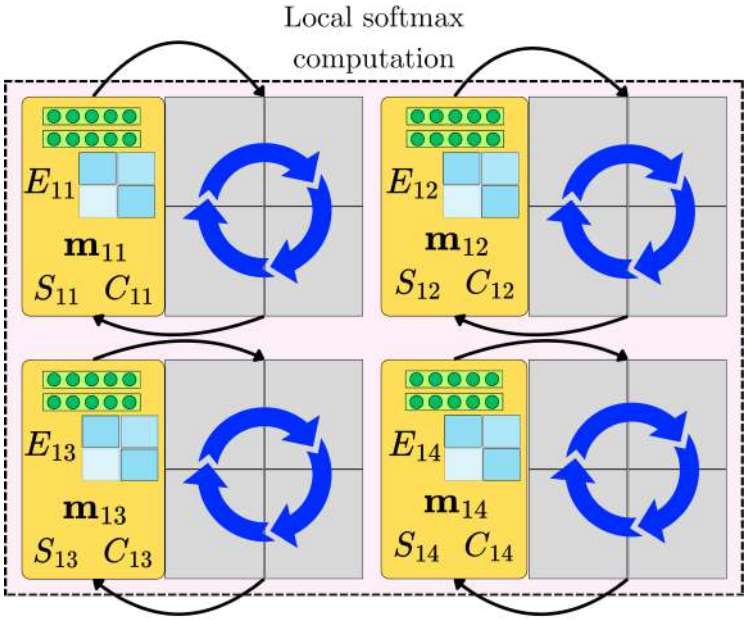

Fig. 50 *By computing the local unnormalized context vectors C_{ij} and the denominator S_{ij}, we can iterate through the different operations without having to write the intermediary tensors back to global memory, saving on precious memory bandwidth.*

accumulate the overall maximum for that key-value block j

$$\mathbf{m}_{\text{run,new}} \leftarrow \max(\mathbf{m}_{\text{run,prev}}, \mathbf{m}_{ij})$$

the overall unnormalized context vector

$$\tilde{C}_i^* \leftarrow \tilde{C}_i^* \odot e^{\mathbf{m}_{\text{run,prev}} - \mathbf{m}_{\text{run,new}}} + C_{ij}$$

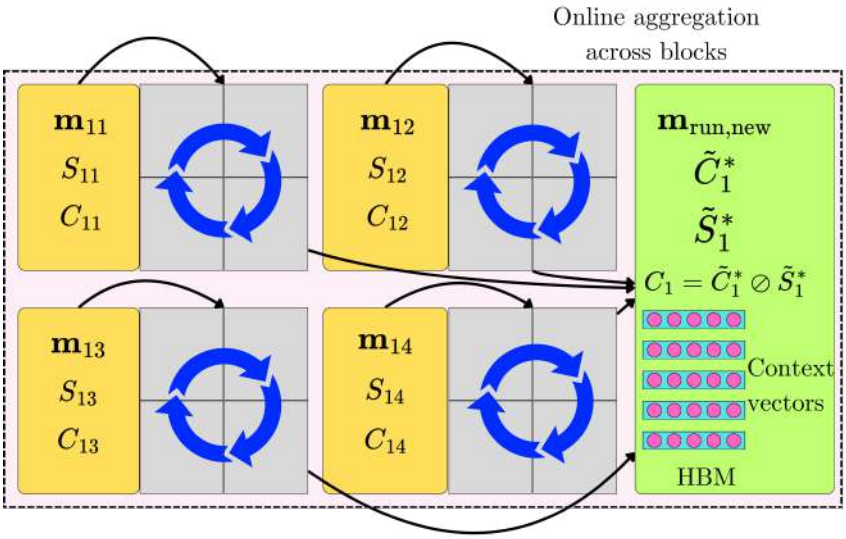

Fig. 51 *For the FlashAttention-1, the context vectors C_i are continuously computed as we iterate through the queries and key-value blocks, and the intermediate tensors are cached in the global memory HBM.*

and the overall denominator

$$\tilde{S}_i^* \leftarrow \tilde{S}_i^* \odot e^{\mathbf{m}_{\text{run,prev}} - \mathbf{m}_{\text{run,new}}} + S_{ij}$$

In the original *FlashAttention*, we also compute the running context vector:

$$C_i = \tilde{C}_i^* \oslash \tilde{S}_i^*$$

These online updates are computed on-chip (in registers/shared memory) as the algorithm processes each block sequentially. Only the running statistics $\mathbf{m}_{\text{run,new}}$, $\tilde{C}_i^*$, $\tilde{S}_i^*$, and C_i (which are much smaller than the full E or A matrices) are occasionally written back to global memory. The heavy lifting (the block-wise computations) remains on-chip (fig. 51).

In *FlashAttention*, the matrices Q, K, and V are partitioned into smaller blocks so that each block can fit into fast on-chip memory. The block sizes N_q for queries and N_k for key-value pairs are chosen so that the data for a block fits within the limited shared memory (SRAM) available per streaming multiprocessor (SM). For example, if the available SRAM is M bytes and each element takes a certain number of bytes (say 4 bytes for FP32 or 2 bytes for FP16), then you might choose

$$N_q = \frac{M}{c \cdot d} \tag{101}$$

where d is the head dimension and c is a constant that accounts for additional storage (such as space for accumulating sums and temporary variables). In practice, many implementations use block sizes on the order of 64 or 128. For instance, a formula sometimes used is

$$N_k = \left\lceil \frac{M}{4 \cdot d} \right\rceil, \quad N_q = \min(N_k, d) \tag{102}$$

where factor 4 accounts for extra temporary storage and safety margins, and $\lceil \cdot \rceil$ is the roundup function. Suppose the GPU uses FP32 (4 bytes per element). Then, the total number of FP32 elements that can be stored in shared memory is roughly $M/4$. K and V have shape $N \times d$, and we want to partition them in blocks of shape $N_k \times d$. For each block, we need to store $N_k \cdot d$ elements in shared memory. To ensure that this block fits into the available memory, we require:

$$N_k \cdot d \lesssim \frac{M}{4} \tag{103}$$

or equivalently:

$$N_k \lesssim \frac{M}{4 \cdot d} \tag{104}$$

Taking the ceiling (to use as many tokens as possible per block without exceeding the capacity), we have:

$$N_k = \left\lceil \frac{M}{4 \cdot d_{\text{head}}} \right\rceil$$

This N_k represents the maximum number of keys (and values) we can pack into shared memory per block, given the head dimension d_{head}. To keep the computation well balanced and to better match the hardware's optimal tile sizes, the query block size is capped by d_{head}

$$N_q = \min(N_k, d_{\text{head}})$$

This means that if the memory allows a very large key block size, you still limit the query block size to d_{head} to maintain computational balance and avoid register spilling or inefficient use of resources. When you orchestrate computations directly at the CUDA level, you gain explicit control over how data is moved and stored across the different memory tiers, which leads to highly optimized memory utilization. *FlashAttention* computes the softmax "online" by processing the keys and values in blocks. As a result, aside from storing the inputs and outputs (which are of size $O(Nd_{\text{head}})$, it only needs to maintain running statistics (like the row-wise maximum and sum of exponentials) for each query, which is $O(N)$. In other words, the additional space complexity of *FlashAttention* is $O(N)$, rather than $O(N^2)$, and this linear extra memory footprint is a key reason why FlashAttention can handle much longer sequences than the vanilla implementation.

FlashAttention can speed up the attention operation by roughly 2–$4\times$ compared to standard vanilla self-attention implementations. For example, experiments have shown that FlashAttention yields about a $3\times$ speedup on models like GPT-2 (with sequence lengths around 1K) and around a 15% end-to-end training speedup on BERT-large (sequence length 512).

5.2.3 FlashAttention-2

The *FlashAttention-2* was published in 2023 by Tri Dao as an improvement on the FlashAttention [9].

Accumulate an Unscaled Numerator. The first improvement comes as a reduction in the number of operations. In *FlashAttention*, we compute a running statistic of the context vector as we aggregate across blocks

$$C_i = \tilde{C}_i^* \oslash \tilde{S}_i^*$$

This means that as we iterate through the different blocks, we constantly move $\tilde{C}_i^*$ and $\tilde{S}_i^*$ from HBM on the thread blocks and write C_i back to HBM. Instead, for *FlashAttention-2*, we delay the normalization and only compute the context vectors once we finish iterating through the different blocks. We have $\tilde{C}^* = [\tilde{C}_1^*, \tilde{C}_2^*, \ldots, \tilde{C}_{n_q}^*]$ and $\tilde{S}^* = [\tilde{S}_1^*, \tilde{S}_2^*, \ldots, \tilde{S}_{n_q}^*]$, then we can compute the final context vectors:

$$C = \tilde{C}^* \oslash \tilde{S}^*.$$

This prevents unnecessary data I/O and division computations.

Parallelization across sequence length. Let's remind ourselves of the simple accumulation formulas:

$$\tilde{C}_i = \sum_{j=1}^{n_k} \exp(Q_i^\top K_j) V_j, \quad S_i = \sum_{j=1}^{n_k} \exp(Q_i^\top K_j)$$

This means we can easily distribute query blocks across thread blocks and sequentially load all the key-value block pairs (K_j, V_j) to compute $\tilde{C}_i$ and S_i. We can then concatenate in memory the contributions from the different queries $\tilde{C} = [\tilde{C}_1, \ldots, \tilde{C}_{n_q}]$ and $S = [S_1, \ldots, S_{n_q}]$. However, it is hard to distribute the key-value block pairs (K_j, V_j) across thread blocks because we need to aggregate the contributions as a sum $\sum_{j=1}^{n_k} \exp(Q_i^\top K_j) V_j$ and $\sum_{j=1}^{n_k} \exp(Q_i^\top K_j)$. In *FlashAttention-1*, we first load the block pair (K_j, V_j) and iterate through the query blocks:

FlashAttention-1

for $j = 1$ to n_k **do** ▷ Process each key-value chunk
  Load K_j, V_j to SRAM
  for $i = 1$ to n_q **do** ▷ Process each query chunk
    Load $Q_i, C_i, S_i^{\text{prev}}, \mathbf{m}_i^{\text{prev}}$ to SRAM ▷ Load current state
    $E_{ij} \leftarrow Q_i K_j^\top$ ▷ Compute alignment scores
    $\mathbf{m}_{ij} \leftarrow \max(E_{ij}, \text{axis} = 1)$ ▷ Compute chunk max
    $\mathbf{m}_i^{\text{new}} \leftarrow \max(\mathbf{m}_i^{\text{prev}}, \mathbf{m}_{ij})$ ▷ Update running max
    $A_{ij} \leftarrow \exp(E_{ij} - \mathbf{m}_{ij})$ ▷ Compute unnormalized attentions
    $S_{ij} \leftarrow \sum_{\text{columns}} A_{ij}$ ▷ Sum of exponentials
    $S_i^{\text{new}} \leftarrow e^{\mathbf{m}_i^{\text{prev}} - \mathbf{m}_i^{\text{new}}} S_i^{\text{prev}} + e^{\mathbf{m}_{ij} - \mathbf{m}_i^{\text{new}}} S_{ij}$ ▷ Update
    $\tilde{C}_i \leftarrow S_i^{\text{prev}} \odot e^{\mathbf{m}_i^{\text{prev}} - \mathbf{m}_i^{\text{new}}} C_i + e^{\mathbf{m}_{ij} - \mathbf{m}_i^{\text{new}}} A_{ij} V_j$ ▷ Update
    $C_i \leftarrow \tilde{C}_i \oslash S_i^{\text{new}}$ ▷ Normalize
    $\mathbf{m}_i^{\text{prev}} \leftarrow \mathbf{m}_i^{\text{new}}, S_i^{\text{prev}} \leftarrow S_i^{\text{new}}$ ▷ Update statistics
    Write $C_i, S_i^{\text{prev}}, \mathbf{m}_i^{\text{prev}}$ back to HBM
  end for
end for
Return: $C = [C_1; C_2; \ldots; C_{n_q}]$

The work is organized into nested loops where each key chunk is processed against every query chunk sequentially. This creates a dependency chain because the inner loop must update running statistics (the accumulated context vectors C_i, denominator, and max values) for the softmax normalization before moving on to the next query chunk. Such sequential updates mean that, even if individual operations (like the matrix multiplications) are parallelized at the lower level, the higher-level loop over query chunks cannot be executed concurrently. This limits the overall GPU occupancy and parallelism because each query block's update must finish before the next can begin, reducing the ability to overlap computation across thread blocks or warps. Additionally, the frequent loading and storing of intermediate states further serializes the process, preventing the algorithm from fully exploiting the GPU's massive parallel resources.

On the other hand, in *FlashAttention-2*, we first load the query block Q_i and iterate through the block pairs (K_j, V_j):

FlashAttention-2

for all $i = 1$ to n_q **do** ▷ Process query chunks in parallel
  Load Q_i to SRAM
  $\tilde{C}_i^{\text{prev}} \leftarrow \mathbf{0}, S_i^{\text{prev}} \leftarrow \mathbf{0}, \mathbf{m}_i^{\text{prev}} \leftarrow -\infty$ ▷ Initialize
  for $j = 1$ to n_k **do** ▷ Process each key–value chunk
    Load K_j, V_j to SRAM
    $E_{ij} \leftarrow Q_i K_j^\top$ ▷ Compute alignment scores
    $\mathbf{m}_{ij} \leftarrow \max(E_{ij}, \text{axis} = 1)$ ▷ Compute chunk max
    $\mathbf{m}_i^{\text{new}} \leftarrow \max(\mathbf{m}_i^{\text{prev}}, \mathbf{m}_{ij})$ ▷ Update running max
    $A_{ij} \leftarrow \exp(E_{ij} - \mathbf{m}_i^{\text{new}})$ ▷ Compute unnormalized softmax
    $S_i^{\text{new}} \leftarrow e^{\mathbf{m}_i^{\text{prev}} - \mathbf{m}_i^{\text{new}}} S_i^{\text{prev}} + \sum_{\text{columns}} A_{ij}$ ▷ Update
    $\tilde{C}_i^{\text{new}} \leftarrow e^{\mathbf{m}_i^{\text{prev}} - \mathbf{m}_i^{\text{new}}} \tilde{C}_i^{\text{prev}} + A_{ij} V_j$ ▷ Update
    $\mathbf{m}_i^{\text{prev}} \leftarrow \mathbf{m}_i^{\text{new}}, S_i^{\text{prev}} \leftarrow S_i^{\text{new}}, \tilde{C}_i^{\text{prev}} \leftarrow \tilde{C}_i^{\text{new}}$ ▷ Update statistics
  end for
  $C_i \leftarrow \tilde{C}_i^{\text{new}} \oslash S_i^{\text{new}}$ ▷ Final normalization
  $L_i \leftarrow \mathbf{m}_i^{\text{new}} + \log(S_i^{\text{new}})$ ▷ Compute LogSumExp
  Write C_i, L_i back to HBM
end for
Return: $C = [C_1; C_2; \ldots; C_{n_q}]$

By organizing the outer loop over query chunks to run in parallel, each query chunk's attention computation is independent. This allows multiple thread blocks to operate concurrently on different query chunks, thereby greatly increasing GPU occupancy. Each query chunk maintains its own accumulators (numerator $\tilde{C}_i$, denominator S_i, $\mathbf{m}_i$). Because these updates are confined to a single

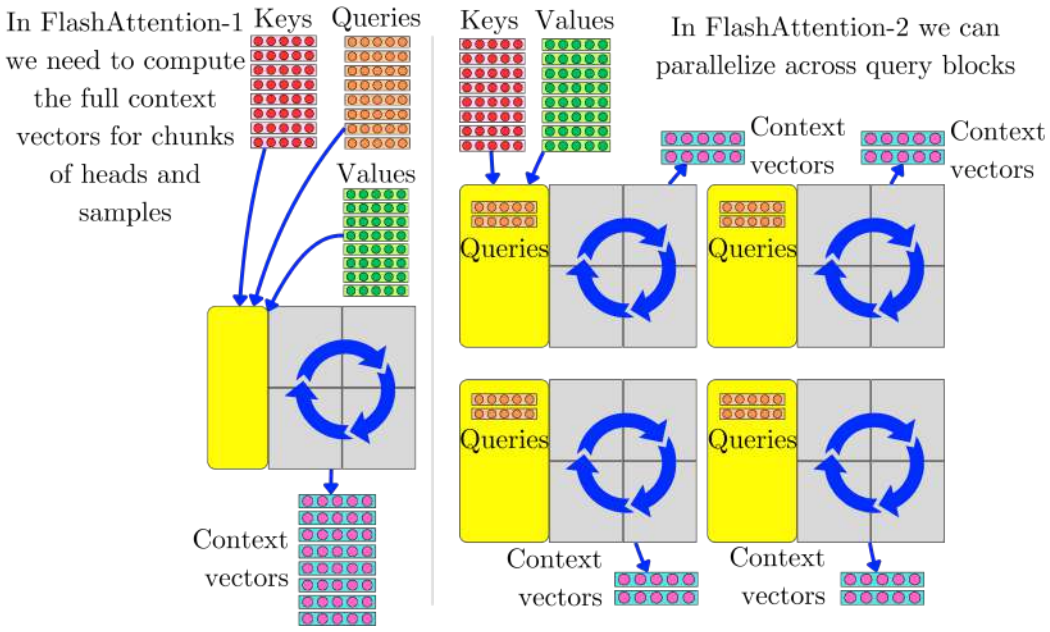

Fig. 52 *On the left, FlashAttention-1 is not set up to be distributed efficiently across the sequence length direction, and the distribution is limited to the batch and heads directions. On the right, the order of computations in FlashAttention-2 allows for a better distribution of the queries across the different SMs and thread blocks.*

thread block (or a group of threads working together), there's less need for global synchronization. This reduction in inter-thread/block dependencies allows for more efficient scheduling and execution. The algorithm assigns the processing of key-value chunks to each query chunk in a way that minimizes sequential dependencies. Each query chunk processes all key-value chunks in an inner loop independently of other query chunks. This more balanced work distribution means that work can be divided among warps within a thread block with less idle time and communication overhead. Due to the limitation in concurrency, the computations in *FlashAttention-1* tend to be parallelized only along heads and batch sample dimensions, whereas in *FlashAttention-2*, they tend to be parallelized along the sequence length dimension as well (fig. 52).

Merging Statistics. Notice that in *FlashAttention-2* we compute a LogSumExp term:

$$L_i \leftarrow \mathbf{m}_i^{\text{new}} + \log(S_i^{\text{new}}) \tag{105}$$

This term is used as a checkpoint for the backward pass computation. During the backward pass, this stored L_i value enables re-computation of the softmax normalization on the fly without reloading or storing all intermediate results:

$$\begin{aligned} \text{Softmax}(E_{ij}) &= \exp(E_{ij} - L_i) \\ &= \exp(E_{ij} - \mathbf{m}_i^{\text{new}} - \log(S_i^{\text{new}})) \\ &= \exp(E_{ij} - \mathbf{m}_i^{\text{new}}) \oslash S_i^{\text{new}} \end{aligned} \tag{106}$$

This selective checkpointing reduces memory bandwidth usage and overall memory footprint, as we only move L_i instead of $(\mathbf{m}_i^{\text{new}}, S_i^{\text{new}})$ to HBM.

Performance improvements: *FlashAttention-2* ends up to be $2\times$ faster than *FlashAttention-1*. When used in end-to-end GPT-style model training, *FlashAttention-2* provides $1.3\times$ speedup compared to *FlashAttention-1* and $2.8\times$ speedup compared to implementations without *FlashAttention*. The performance improvements from *FlashAttention-2* mean that training models with 16K context length becomes as economically feasible as training 8K context models was previously. Existing models can be trained, fine-tuned, and run inference faster. This advancement represents a significant step toward making longer context models more practical and accessible.

5.2.4 FlashAttention-3

The *FlashAttention-3* was published in 2024 by Jay Shah et al. [27], improving even further on the *FlashAttention* paradigm.

Exploiting hardware asynchrony. Modern GPUs like NVIDIA's H100 are designed with specialized hardware units that can operate independently and concurrently. This hardware-level asynchrony allows different operations to execute simultaneously, dramatically improving performance when properly utilized. *FlashAttention-3* is specifically engineered to exploit these asynchronous capabilities, which *FlashAttention-2* largely overlooked. In the H100 architecture, two particularly important asynchronous components are:

- Tensor Cores: Specialized hardware units for matrix multiplication that can operate independently from other GPU computation
- Tensor Memory Accelerator (TMA): Dedicated hardware for memory transfers between different levels of the memory hierarchy

Think of these components as specialized workers that can perform their tasks in parallel to other operations. *FlashAttention-2* treated these operations more sequentially, requiring one to finish before the next began.

Overlapping operations with pipelining. In FlashAttention-2, operations followed a strict sequential pattern due to data dependencies. For each iteration of the algorithm processing one block:

1. Compute $E = QK^\top$
2. Wait for completion
3. Compute Softmax(E)
4. Wait for completion
5. Proceed to next iteration

This sequential execution created inefficiencies because different operations utilize different hardware units. While one operation ran, specialized hardware units needed for other operations sat idle. *FlashAttention-3*'s main innovation is its 2-stage pipeline that overlaps the computation of Softmax$(E_{i,j+1})$ in one iteration with the matrix multiplication $A_{ij}V_j$ (the output update) from the previous iteration (fig. 53).

FlashAttention-3

for all $i = 1$ to n_q **do** ▷ Process query chunks in parallel
  Load Q_i to SRAM
  $\tilde{C}_i \leftarrow \mathbf{0}, S_i \leftarrow \mathbf{0}, \mathbf{m}_i \leftarrow -\infty$ ▷ Initialize
  Wait for K_0 to load to SRAM
  $E_{i,0} \leftarrow Q_i K_0^\top$ ▷ Initial alignment scores calculation
  $\mathbf{m}_{i,0} \leftarrow \max(E_{i,0}, \text{axis} = 1)$
  $\mathbf{m}_i \leftarrow \mathbf{m}_{i,0}$
  $A_{i,0} \leftarrow \exp(E_{i,0} - \mathbf{m}_i)$ ▷ Compute softmax
  $S_i \leftarrow \sum_{\text{columns}} A_{i,0}$
  for $j = 1$ to $n_k - 1$ **do** ▷ Main pipeline loop
    Wait for K_j to load to SRAM
    $E_{i,j+1} \leftarrow Q_i K_j^\top$ **async** ▷ Next scores calculation (don't wait)
    Wait for V_{j-1} to load to SRAM
    $\tilde{C}_i \leftarrow \tilde{C}_i + A_{i,j-1}V_{j-1}$ **async** ▷ Update output (don't wait)
    Wait for $E_{i,j+1}$ computation to complete
    $\mathbf{m}_{i,j+1} \leftarrow \max(E_{i,j+1}, \text{axis} = 1)$
    $\mathbf{m}_i^{\text{new}} \leftarrow \max(\mathbf{m}_i, \mathbf{m}_{i,j+1})$
    $A_{i,j+1} \leftarrow \exp(E_{i,j+1} - \mathbf{m}_i^{\text{new}})$ ▷ Next softmax
    $S_i^{\text{new}} \leftarrow e^{\mathbf{m}_i - \mathbf{m}_i^{\text{new}}} S_i + \sum_{\text{columns}} A_{i,j+1}$
    Wait for $\tilde{C}_i$ update to complete

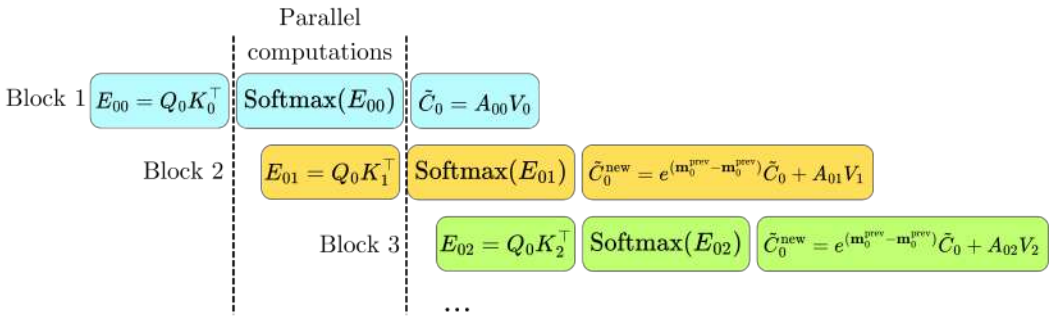

Fig. 53 *When $Q_0 K_0^\top$ is finished being computed, there is no need to wait to start computing $Q_0 K_1^\top$ for the next key-value block. The asynchronous operations allow for a better utilization of the GPU hardware, as the next loop's operations can start as the previous loop is not yet complete.*

$\tilde{C}_i \leftarrow e^{\mathbf{m}_i - \mathbf{m}_i^{\text{new}}} \tilde{C}_i$ ▷ Rescale output
$\mathbf{m}_i \leftarrow \mathbf{m}_i^{\text{new}}, S_i \leftarrow S_i^{\text{new}}$ ▷ Update statistics
$A_{i,j} \leftarrow A_{i,j+1}$ ▷ Prepare for next iteration
end for
Wait for $V_{n_k - 1}$ to load to SRAM
$\tilde{C}_i \leftarrow \tilde{C}_i + A_{i,n_k - 1} V_{n_k - 1}$ ▷ Process final chunk
$C_i \leftarrow \tilde{C}_i \oslash S_i$ ▷ Final normalization
$L_i \leftarrow \mathbf{m}_i + \log(S_i)$ ▷ Compute LogSumExp
Write C_i, L_i back to HBM
end for
Return: $C = [C_1; C_2; \ldots; C_{n_q}]$

This approach minimizes idle time by ensuring that while softmax calculations are being performed, the Tensor Cores are busy with matrix multiplication operations.

Hardware-accelerated low precision. Modern GPUs like NVIDIA's H100 offer specialized hardware support for low-precision computation, most notably FP8 (8-bit floating point). The theoretical advantage is significant: FP8 operations can deliver nearly double the computational throughput compared to FP16/BF16. What makes *FlashAttention-3*'s approach particularly valuable is that it doesn't compromise on accuracy while delivering the performance advantages of low-precision computation. It delivers both speed and numerical stability through clever algorithm design and hardware awareness.

Performance improvements: *FlashAttention-3* delivers substantial performance gains over *FlashAttention-2*:

- $1.5 - 2.0\times$ speedup in the forward pass with FP16
- $1.5 - 1.75\times$ speedup in the backward pass
- Reaches up to 740 TFLOPs/s with FP16 (75% of theoretical maximum on H100)
- Achieves nearly 1.2 PFLOPs/s with FP8
- Outperforms even vendor-optimized implementations (like cuDNN) for medium and long sequences

While *FlashAttention-2* achieved only 35% utilization on H100 GPUs, *FlashAttention-3* is specifically designed to leverage Hopper architecture features. These improvements make *FlashAttention-3* particularly valuable for long-context applications, enabling efficient processing of sequences up to 16K tokens or beyond.

Maximum Sequence Length With and Without FlashAttention: It is worth understanding how many more tokens the *FlashAttention* allows to process. Let's consider, as an example, the GPT-3 [4] models. Here are the specifications:

- 96 layers
- 96 attention heads
- 12,288 hidden dimension
- 16-bit precision (FP16/BF16)

For GPT-3 with vanilla attention, the memory requirement for all layers and heads for attention matrices alone is:

$$96 \text{ layers} \times 96 \text{ heads} \times N^2 \times 2 \text{ bytes} = 18,432 \times N^2 \text{ bytes} \tag{107}$$

For a high-end NVIDIA A100-80GB GPU (assuming model parameters are distributed across multiple GPUs), we have $\sim$ 70GB available for activations after accounting for other overhead. Therefore, we have the following constraint to be satisfied:

$$18,432 \times N^2 < 70 \times 10^9 \tag{108}$$

or equivalently:

$$N \lesssim 1948 \text{ tokens} \tag{109}$$

Consequently, the maximum sequence length that one high-end GPU can handle with the vanilla attention is $\sim 1,948$ tokens, which aligns with GPT-3's initial context window of 2,048 tokens. The quadratic scaling of attention matrices creates a hard ceiling that is difficult to overcome without fundamental changes to the attention mechanism.

With *FlashAttention*, the memory requirement becomes dominated by the hidden states and linear components. The minimum memory needed per token is:

- Hidden states: 96 layers $\times$ 12,288 dimensions $\times$ 2 bytes = 2,359,296 bytes
- Feed-forward activations: 96 layers $\times$ 49,152 $\times$ 2 bytes = 9,437,184 bytes

Additional memory is needed for computation but does not scale quadratically with sequence length. The total per-token memory requirement is approximately 2.4MB + 9.4MB = $\sim$ 11.8MB per token. For an NVIDIA A100-80GB GPU with $\sim$ 70GB available for activations:

$$\frac{70\text{GB}}{11.8\text{MB}} \simeq 5,932 \text{ tokens}$$

Accounting for implementation overhead and other memory requirements, a realistic maximum would be around 4,000-5,000 tokens. With multi-GPU setups (i.e., 8$\times$ A100-80GB) and additional optimizations like gradient checkpointing, we could realistically reach a maximum of 25,000-30,000 tokens (which aligns with the original context size of GPT-3.5-Turbo-16k: 16,384 token context window). Therefore, if we consider additional optimizations, we have a $\sim 2 - 2.5\times$ improvement factor with the bare FlashAttention and $\sim 8 - 15\times$. Changing the scaling behavior from quadratic to linear memory complexity allows for longer context processing that would otherwise be impossible.

5.3 To Summarize

Attention Type	Key Innovations & Benefits	Limitations & Downsides
Self-attention does not need $\mathcal{O}(N^2)$ memory	Block-wise computation with online aggregation reduces memory needs without approximation	Slower computation (8-13% forward, 30-35% backward); complex numerical stabilization; reduced parallelism
FlashAttention-1	GPU-optimized tiling approach minimizes HBM access, $2 - 4\times$ speedup over vanilla attention	Custom CUDA kernels required; hardware-specific optimizations limit portability; implementation complexity

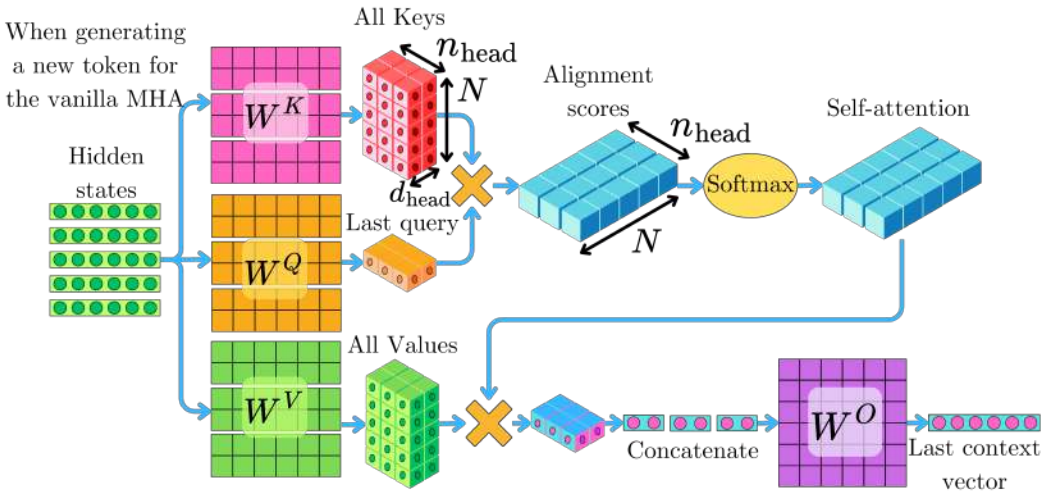

Fig. 54 *In the vanilla Multi-Head Attention, we need to reload or regenerate the same key and value tensors repeatedly as we decode text. In the following sections, we will explore strategies to reduce the size of the keys and values to increase the memory bandwidth for those large tensors and reduce the latency associated with text generation.*

Attention Type	Key Innovations & Benefits	Limitations & Downsides
FlashAttention-2	Improved parallelization across sequence length, $2\times$ faster than FlashAttention-1	More complex implementation; harder to debug and maintain; requires careful tuning of block sizes
FlashAttention-3	Exploits hardware asynchrony and pipelining, $1.5-2\times$ speedup over FlashAttention-2	Requires modern hardware (e.g., H100) for full benefits; highly specialized implementation; focused only on NVIDIA GPUs

6 Faster Decoding Attention Mechanisms

When generating text one token at a time, transformers face a significant performance challenge. In the original transformer architecture with Multi-Head Attention (MHA), each time we generate a new token, we need to:

1. Load the entire history of key (K) and value (V) matrices for each attention head
2. Process them against the new query
3. Generate the next token
4. Repeat

This creates a memory bandwidth bottleneck. For long sequences, we're constantly reloading massive tensors from memory, which becomes the limiting factor in generation speed. Specifically, with n_{head} attention heads and a sequence of length N, the original approach required loading tensors of size approximately $n_{\text{head}} \times d_{\text{head}} \times N$, which could be gigabytes of data for long contexts. Since those K and V tensors are a bottleneck for the decoding process, we will look at strategies to minimize their memory requirements.

Strategies like the *Multi-Query Attention*, the *Grouped-Query Attention*, and *Multi-Head Latent Attention* are highly coupled with the KV-caching technique [23]. With KV-caching, instead of recomputing the same K and V matrices over and over, we cache them in memory and update them

at each iteration. Loading the KV-cache from memory becomes the dominant bottleneck instead of the computation itself.

6.1 Multi-Query Attention

In 2019, Noam Shazeer published a variant of the multi-head attention: the *Multi-Query Attention* (MQA) [28]. He realized the model could maintain most of its capability while sharing a single set of keys and values across all heads. This insight might seem counterintuitive since the whole point of multi-head attention was to have different "perspectives" on the same information. Shazeer discovered that the diversity in the query projections still allows different heads to extract different information, the shared key-value store acts as a common knowledge repository, and each head can still attend to different parts of this shared repository.

Formally, this means that the projection matrix W^Q is still a $d_{\text{model}} \times d_{\text{model}}$ matrix, but the projections W^K and W^V are $d_{\text{model}} \times d_{\text{head}}$ matrices. With an incoming hidden state H of dimension $d_{\text{model}} \times N$, we have at training time the initial projections:

$$\begin{aligned} Q &= W^Q H, \quad \text{Shape: } d_{\text{model}} \times N \\ K &= W^K H, \quad \text{Shape: } d_{\text{head}} \times N \\ V &= W^V H, \quad \text{Shape: } d_{\text{head}} \times N \end{aligned} \tag{110}$$

We then reshape the matrices into tensors to highlight the number of heads:

$$\begin{aligned} Q' &= \text{Reshape}(Q), \quad \text{Shape: } n_{\text{head}} \times d_{\text{head}} \times N \\ K' &= K, \quad \text{Shape: } d_{\text{head}} \times N \\ V' &= V, \quad \text{Shape: } d_{\text{head}} \times N \end{aligned} \tag{111}$$

At inference time, we only consider the last query $\mathbf{q}'_N$ of size $d_{\text{head}} \times n_{\text{head}}$ in the input sequence since we only need to predict the last token. The alignment scores are computed by broadcasting the matrix K' to all the heads:

$$\mathbf{e}'_N = \frac{\mathbf{q}'^{\top}_N K'}{\sqrt{d_{\text{head}}}}, \quad \text{Shape: } n_{\text{head}} \times N \tag{112}$$

We perform the softmax transformation:

$$\mathbf{a}'_N = \text{Softmax}(\mathbf{e}'_N), \quad \text{Shape: } n_{\text{head}} \times N \tag{113}$$

The value is again broadcast to all heads to compute the context vector corresponding to the prediction:

$$\mathbf{c}'_N = \mathbf{a}'_N V'^{\top}, \quad \text{Shape: } n_{\text{head}} \times d_{\text{head}} \tag{114}$$

The context vector is reshaped in the original dimensions:

$$\mathbf{c}_N = \text{Reshape}(\mathbf{c}'_N), \quad \text{Shape: } d_{\text{model}} \times 1 \tag{115}$$

Finally, the context vector is projected one last time to mix the information from the different heads:

$$\mathbf{c}^{\text{final}}_N = W^O \mathbf{c}_N, \quad \text{Shape: } d_{\text{model}} \times 1 \tag{116}$$

Let's consider the memory access complexity (or memory bandwidth complexity). It measures the total amount of data that must be transferred between memory and compute units during the entire sequence of operations. This measures bandwidth requirements. For MHA, we need to load W^Q, W^K, W^V, and W^O at each decoding step. They are all $d_{\text{model}} \times d_{\text{model}}$ matrices, so over N decoding steps the memory access complexity is $\sim \mathcal{O}(N d^2_{\text{model}})$. As we generate each token, we must reload the entire history. For the i-th token, we load keys and values of size $i \times d_{\text{model}}$. Summing over all N steps:

$$\sum_{i=1}^{N} i \cdot d_{\text{model}} = d_{\text{model}} \frac{N(N+1)}{2} \sim \mathcal{O}(d_{\text{model}} N^2)$$

We must also load the N input hidden states of size d_{model}. So, the overall memory access complexity for MHA is

$$\mathcal{O}(d_{\text{model}} N + N d^2_{\text{model}} + d_{\text{model}} N^2)$$

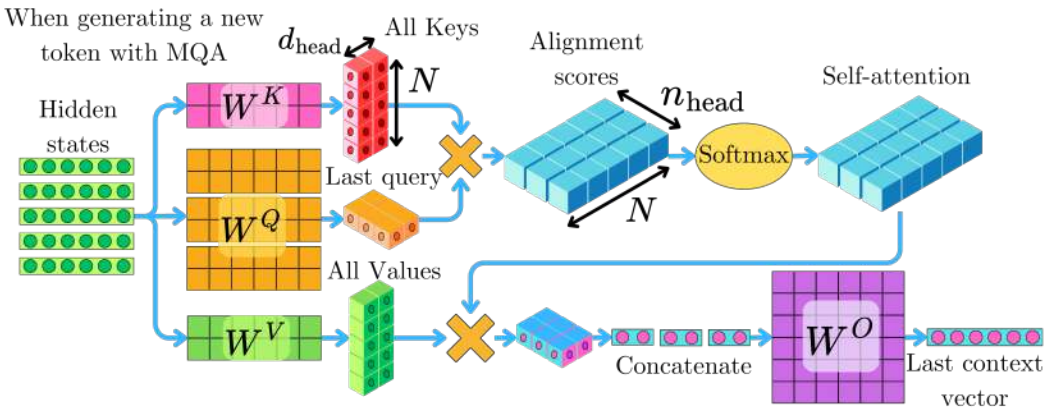

Fig. 55 *For MQA, the key and value projections W^K and W^V are reduced to only one head and the resulting key and value tensors are squished on the heads direction. This greatly reduces the memory allocated for those tensors.*

For long sequences, $d_{\text{model}}N^2$ dominates, which is the problematic bottleneck.

For MQA, loading W^Q, W^K, W^V, and W^O is the same asymptotic behavior $\sim \mathcal{O}(Nd_{\text{model}}^2)$ and the input hidden states as well $\sim \mathcal{O}(d_{\text{model}}N)$. However, the K and V matrices have a size $i \times d_{\text{head}}$ at the i-th decoding step, which leads to:

$$\sum_{i=1}^{N} i \cdot d_{\text{head}} = d_{\text{head}} \frac{N(N+1)}{2} \sim \mathcal{O}(d_{\text{head}}N^2)$$

So the overall memory access complexity is

$$\mathcal{O}(d_{\text{model}}N + Nd_{\text{model}}^2 + d_{\text{head}}N^2)$$

As the $d_{\text{head}}N^2$ term dominates for long sequences, it has a substantial impact because memory access increase leads to higher latency. For the specific experiments run in Shazeer's paper, he found that MQA achieves $\sim 12\times$ faster decoder inference with minimal performance loss. For inference with a sequence length of 128 tokens, MHA's decoding time per token was $46\mu s$ whereas MQA's was $3.8\mu s$.

6.2 Grouped-Query Attention

With *Multi-Query Attention*, we gain in decoding speed, but we lose in performance compared to Multi-Head Attention. The *Grouped-Query Attention* [1] (GQA) provides a middle ground between MQA and MHA to keep performance high while improving decoding speed performance. GQA divides query heads into G groups, each sharing a single key head and value head. This creates a configurable spectrum:

- When $G = 1$, it is equivalent to MQA (single key-value head for all queries)
- When $G = n_{\text{head}}$, it is equivalent to standard MHA (separate key-value for each query)
- When $1 < G < n_{\text{head}}$, we have the GQA sweet spot that balances efficiency and quality (fig. 56)

It is important to understand that latency is not linearly proportional to memory access. Memory-level parallelism refers to a computer system's ability to process multiple memory operations simultaneously rather than sequentially. There is a regime where the memory-level parallelism of the hardware is underutilized when we are trying to load too few groups, and there is a regime where we are saturating it when we are loading too many groups. Because of the parallelism, it is quite likely that having $G = 1$ (MQA) induces as much latency as $G = 4$, for example, despite the increased predictive performance. This relationship between groups and latency is highly hardware-specific, so empirical testing is necessary to find the optimal configuration for any given system. The TPUs

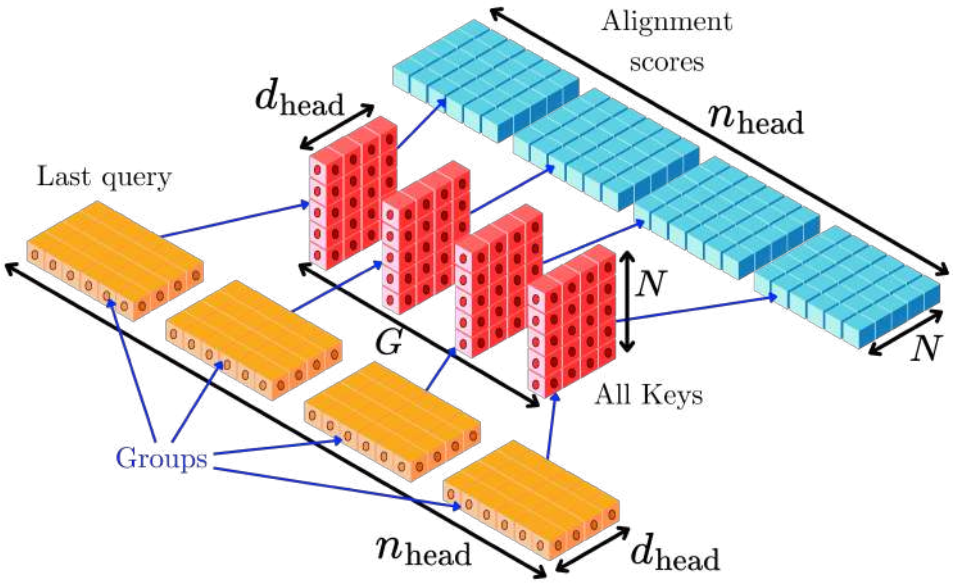

Fig. 56 *For GQA, each group of heads per query is associated with one $N \times d_{head}$ slice of keys. The key slice is broadcast to all the heads in the group, and the alignment score $Q^T K$ is computed as usual.*

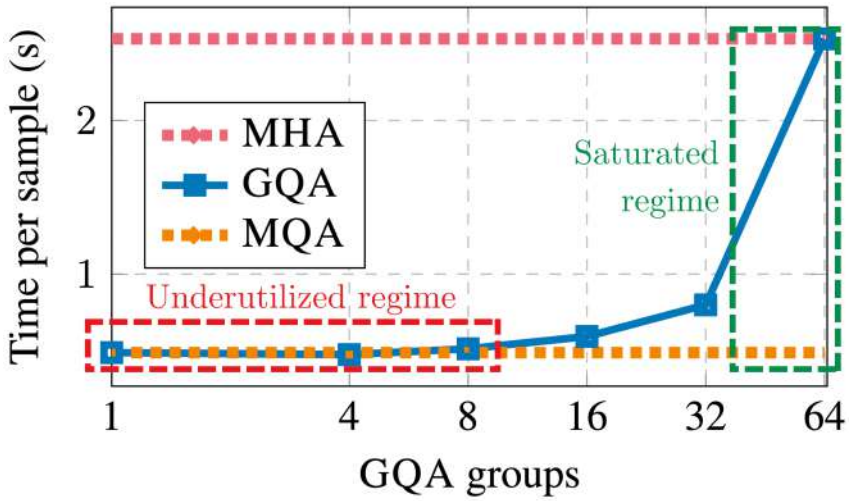

Fig. 57 *This graph is extracted from the original GQA paper [1] (figure 6 in the original paper). It shows that the time per sample is not linearly proportional to the number of groups. From 1 to 8 groups, the time per sample remains constant, while it almost triples when we double the number of groups from 32 to 64. For a small number of groups, the memory bandwidth for moving the key and value tensors is not fully utilized. Moving additional data does not necessarily increase the latency, as everything is done in parallel.*

used in the original paper showed this saturation around $G = 8$ (see Fig. 57), but different architectures might have different saturation points.

GQA achieves quality close to MHA with speeds comparable to MQA. GQA is more stable during training than pure MQA, which can sometimes exhibit training instability. For example, on the T5-XXL model, the GQA-8 version presented in the original paper achieved an average performance of 47.1 across key benchmarks compared to 47.2 for MHA and 46.6 for MQA, while maintaining inference speeds much closer to MQA.

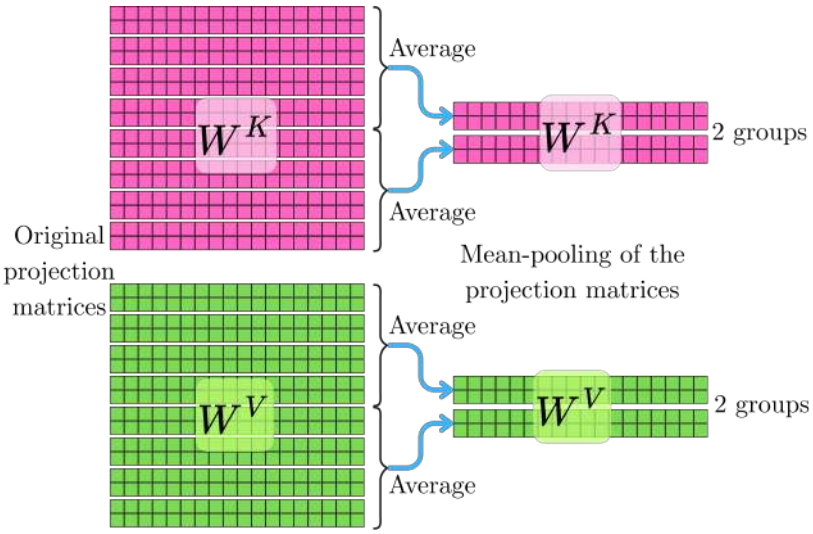

Fig. 58 *With the original W^K and W^V projection matrices, each slice of the matrices is associated with one head. As a first step to convert a MHA to a GQA, we average the different matrix slices together per group, leading to key and value tensors with exactly G groups instead of n_{head}.*

One of the main appeals of GQA is the ability to convert a vanilla MHA into GQA with minimal effort. The ability to convert existing models rather than train new ones has accelerated the adoption of GQA across the industry. Large models like PaLM 2 [2] and LLaMA 2 [31] have incorporated GQA, partly because this conversion process made it practical to do so. Converting from MHA to GQA is a two-step process:

1. **Checkpoint Conversion:** We first convert the model's weights by mean-pooling the key and value projection matrices within each group (see fig. 58).
2. **Additional Pre-training:** After conversion, we continue pre-training the model for a small fraction (about 5%) of the original training steps, using the same pre-training dataset and objectives.

This approach requires only about 5% of the original training compute to achieve comparable performance to the original MHA model. For large models that cost millions to train, this represents enormous savings.

6.3 Multi-Head Latent Attention

Multi-Head Latent Attention (MLA) was introduced in DeepSeek-V2 [11] as a way to optimize for training and inference speed while preserving the predictive performance. Even if GQA keeps performance high, it must be balanced with performance. MLA provides a solution that improves on both fronts at the same time.

6.3.1 Compressing the Keys, Values, and Queries

In MQA and GQA, the main goal is to reduce the number of keys and values that need to be transferred back and forth between the HBM and SRAM to make better use of the memory bandwidth. Consequently, we also reduce the necessary cache memory to cache the keys and values reused at every decoding step. Instead, in MLA, we compute a compressed latent representation of the keys and values for all the heads. So, as much as possible, we attempt to preserve the information related to the different heads. We first use the down-projection matrix W^{DKV} (***D***own ***K***eys ***V***alues) on the incoming hidden state $\mathbf{h}_i$ to reshape it from d_{model} to d_{latent}:

$$\mathbf{l}_i^{KV} = W^{DKV}\mathbf{h}_i, \quad \text{From } d_{\text{model}} \text{ to } d_{\text{latent}} \tag{117}$$

The original paper uses the notation $\mathbf{c}_i^{KV} \equiv \mathbf{l}_i^{KV}$, but here we use $\mathbf{l}_i^{KV}$ to prevent confusion with the context vector notation. The subscript KV highlights that it is the latent representation specifically

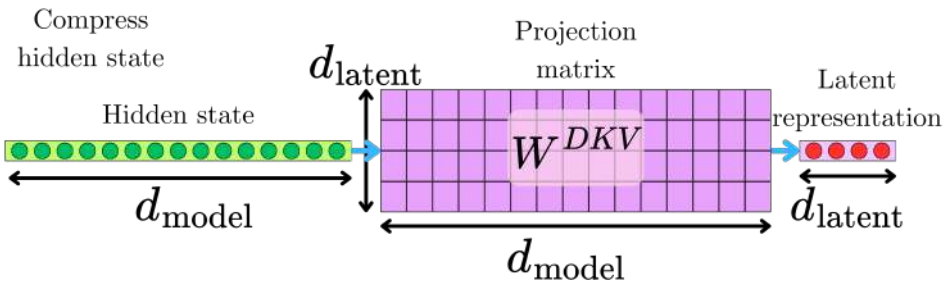

Fig. 59 *The W^{DKV} matrix takes a hidden state of size d_{model} and project it into a smaller latent representation of size d_{latent}.*

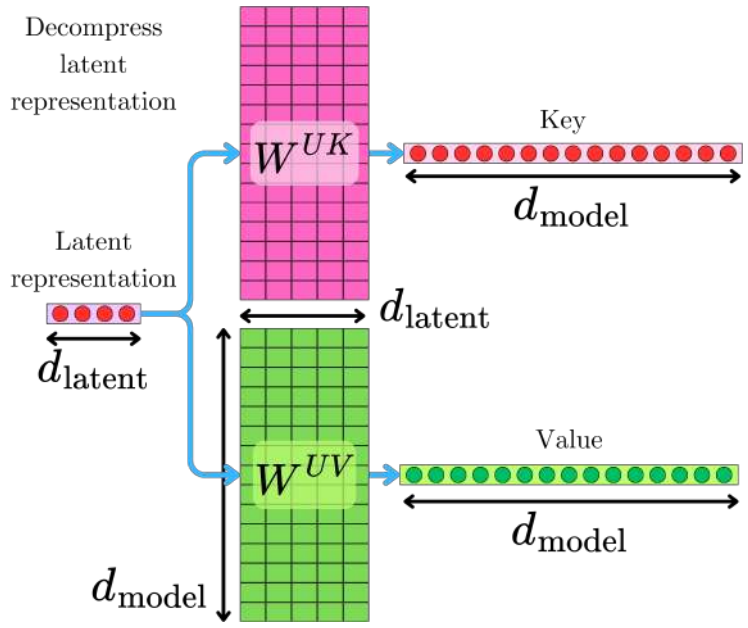

Fig. 60 *The latent representation $\mathbf{l}_i^{KV}$ is just a compressed vector representation of the original hidden state. To specialize the latent representation into the typical key and value representation, we project $\mathbf{l}_i^{KV}$ to the original d_{model} dimension through two projection matrices W^{UK} and W^{UV}.*

used for the keys and values. The idea is to choose d_{latent} such that $d_{\text{model}} \gg d_{\text{latent}}$ to compress the information (fig. 59).

The actual keys and values can be recovered by decompressing the latent representation (fig. 60):

$$\begin{aligned} \mathbf{k}_i &= W^{UK}\mathbf{l}_i^{KV}, \quad \text{From } d_{\text{latent}} \text{ to } d_{\text{model}} \\ \mathbf{v}_i &= W^{UV}\mathbf{l}_i^{KV}, \quad \text{From } d_{\text{latent}} \text{ to } d_{\text{model}} \end{aligned} \tag{118}$$

W^{UK} (***U***p ***K***ey) and W^{UV} (***U***p ***V***alue) are up-projection matrices of size $d_{\text{model}} \times d_{\text{latent}}$.

The computations can be performed with the decompressed Keys and Values, but the memory accesses can be done using the compressed representation (see fig. 61). In the typical MHA, we have N keys and values of size d_{model} with a memory access complexity $\sim \mathcal{O}(d_{\text{model}}N)$ for each global memory to local memory (HBM $\rightarrow$ SRAM) inbound or outbound, and for the MLA, we have N latent representations of size d_{latent} with $\sim \mathcal{O}(d_{\text{latent}}N)$ for each inbound/outbound. This allows for moving more data at once to and from the local SRAM to utilize the high GPU parallelism better.

To improve memory bandwidth, we also compress the query representation with a dedicated down-projection matrix W^DQ (***D***own ***Q***uery):

$$\mathbf{l}_i^Q = W^{DQ}\mathbf{h}_i, \quad \text{From } d_{\text{model}} \text{ to } d_{\text{latent}} \tag{119}$$

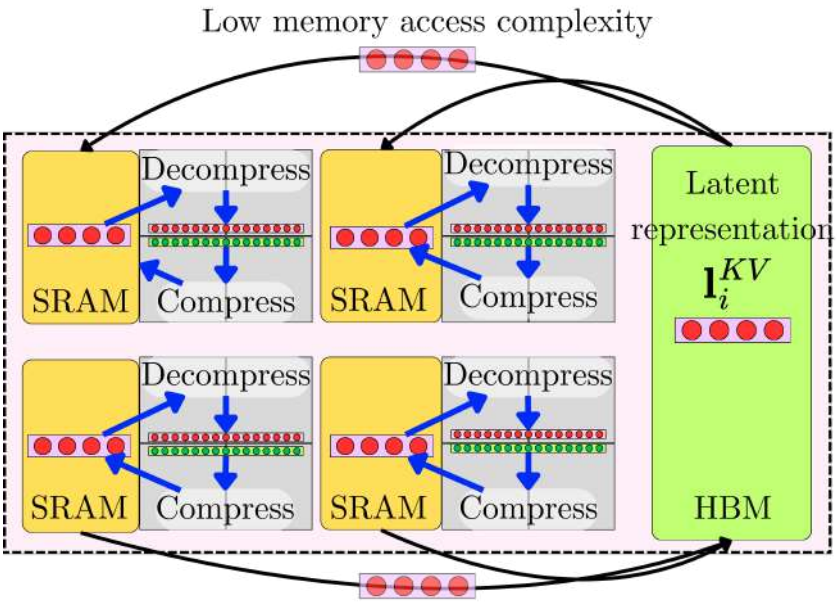

Fig. 61 *The advantage of the latent representation for MLA is the lower amount of data that needs to be moved from global memory to SRAMs. This compressed representation allows for faster memory access while adding computational latency for compressing and decompressing it. d_{latent} has to be carefully chosen to balance memory access vs additional computations to reduce the overall latency.*

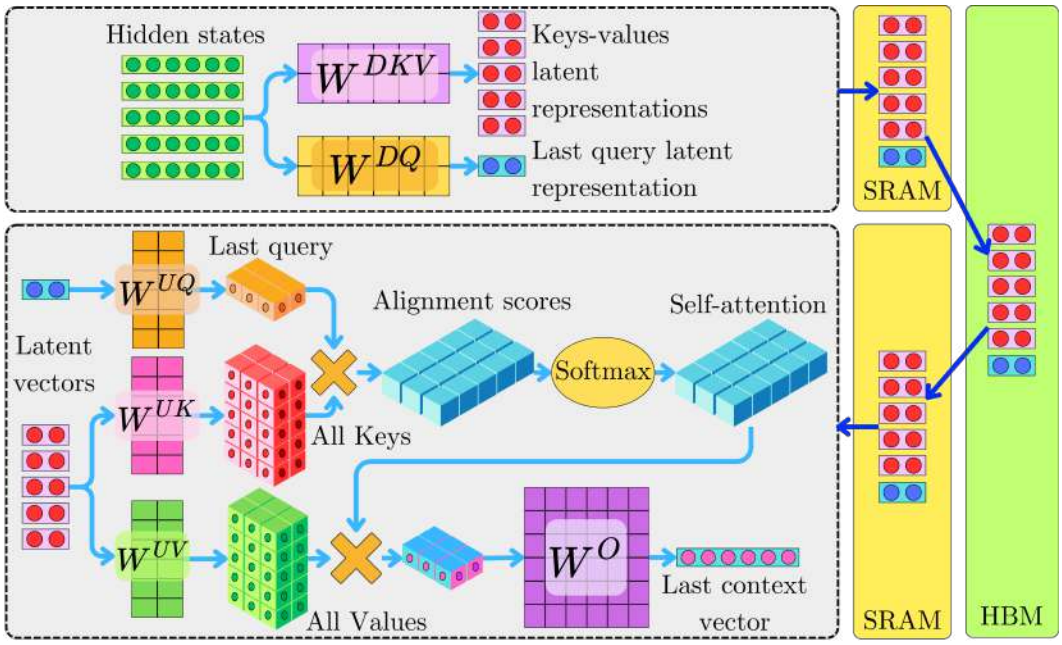

Fig. 62 *This is the full process of the MLA during training. The hidden states are compressed in their latent representations for faster memory access between HBM and SRAMs. Before computing the local alignment score tensor, the latent representations are decompressed, and the context vectors are computed as usual.*

with the opposite operation to decompress the latent representation:

$$\mathbf{q}_i = W^{UQ}\mathbf{l}_i^Q, \quad \text{From } d_{\text{latent}} \text{ to } d_{\text{model}} \tag{120}$$

Again, W^{UQ} (***U***p ***Q***uery) is a up-projection matrix of size $d_{\text{model}} \times d_{\text{latent}}$. The keys, queries, and values are only compressed to minimize the memory access complexity, and once decompressed, the context vectors can be computed as usual (see fig. 62):

$$\mathbf{c_i} = W^O \cdot \sum_{j=1}^{N} \text{Softmax}\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}}\right) \mathbf{v}_j \tag{121}$$

6.3.2 Optimizing the Computations

During training, all those matrices are learned separately, but at inference time we can actually simplify the computations by combining some of them together. In the softmax transformation, we have:

$$\begin{aligned} \mathbf{q}_i^\top \cdot \mathbf{k}_j &= \left(W^{UQ}\mathbf{l}_i^Q\right)^\top \cdot \left(W^{UK}\mathbf{l}_j^{KV}\right) \\ &= \mathbf{l}_i^{Q\top} W^{UQ\top} W^{UK} \mathbf{l}_j^{KV} \end{aligned} \tag{122}$$

Using the above equation, we can define an absorbed query-key matrix:

$$W^{QK}_{\text{absorbed}} = W^{UQ\top} W^{UK} \tag{123}$$

and we can precompute an alternative latent representation of the query $\mathbf{l}_i^{*Q}$:

$$\mathbf{l}_i^{*Q} = W^{QK}_{\text{absorbed}} \mathbf{l}_i^Q \tag{124}$$

W^{QK}_{absorbed} has a dimension $d_{\text{latent}} \times d_{\text{latent}}$, therefore $\mathbf{l}_i^{*Q}$ also has dimension d_{latent}. As a consequence, we can overwrite the original network by precomputing $W^{QK}_{\text{absorbed}} = W^{UQ\top} W^{UK}$ and directly operate on the latent representations of the keys and queries for the alignment scores. Effectively, we actually compute the alignment scores $\mathbf{l}_i^{*Q^\top} \cdot \mathbf{l}_j^{KV}$, and their related attentions in the latent space where the heads are "squished" together.

Similarly, we can simplify the computations related to the values. We have:

$$\begin{aligned} \mathbf{c_i} &= W^O \cdot \sum_{j=1}^{N} \text{Softmax}\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}}\right) \mathbf{v}_j \\ &= W^O \cdot \sum_{j=1}^{N} a_{ij} \mathbf{v}_j \\ &= W^O \cdot \sum_{j=1}^{N} a_{ij} W^{UV} \mathbf{l}_j^{KV} \\ &= \left(W^O \cdot W^{UV}\right) \sum_{j=1}^{N} a_{ij} \mathbf{l}_j^{KV} \\ &= W^{OV}_{\text{absorbed}} \cdot \sum_{j=1}^{N} a_{ij} \mathbf{l}_j^{KV} \end{aligned} \tag{125}$$

The size of W^{OV}_{absorbed} is $d_{\text{model}} \times d_{\text{latent}}$ and it can be precomputed as well. The number of computations is greatly reduced while preserving the low memory bandwidth (see fig. 63).

6.3.3 Including Positional Information

Many modern LLMs use Rotary Position Embedding(RoPE) [29] to replace the original positional encoding. RoPE is a rotational transformation that "injects" the positional information directly into the keys and queries:

$$\begin{aligned} \mathbf{q}_i^R &= \text{RoPE}(\mathbf{q}_i) = R_i \mathbf{q}_i \\ \mathbf{k}_j^R &= \text{RoPE}(\mathbf{k}_j) = R_j \mathbf{k}_j \end{aligned} \tag{126}$$

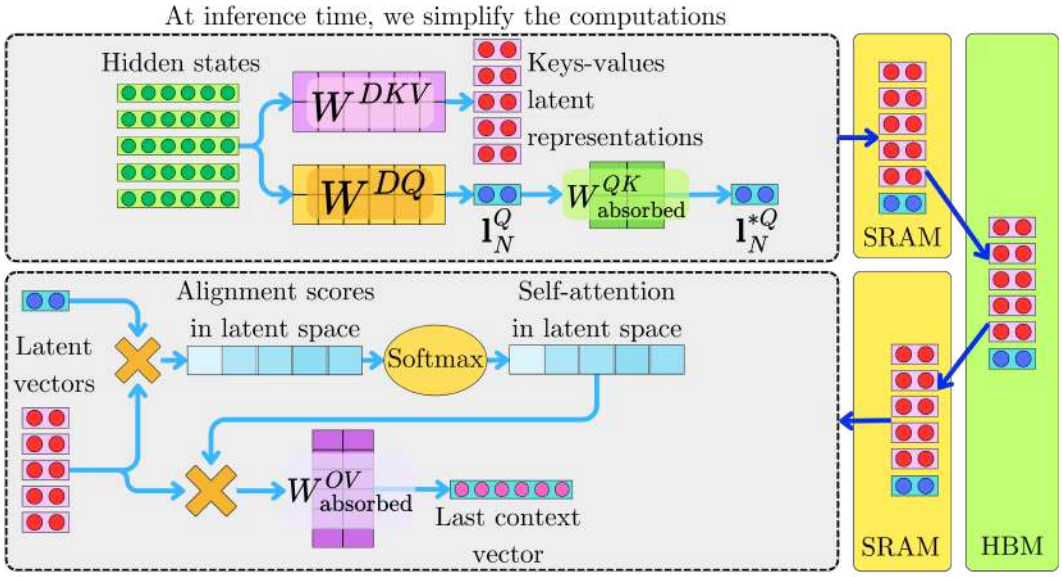

Fig. 63 *At inference time, the model parameters are fixed, and the amount of computation can be greatly reduced (to be compared to fig. 62) by computing the alignment and the attention matrices in the latent space.*

where R_i and R_j are position-specific rotation matrices for positions i and j respectively. They typically are $d_{\text{model}} \times d_{\text{model}}$ matrices and $\mathbf{q}_i^R, \mathbf{k}_j^R$ are the resulting rotated query and key.

In the context of MLA, directly applying RoPE on the queries and keys prevents us from using the matrix absorption tricks presented in the previous section. If we consider the alignment score, for example:

$$\begin{aligned}\mathbf{q}_i^{R\top} \cdot \mathbf{k}_j^R &= \left(R_i W^{UQ} \mathbf{l}_i^Q\right)^\top \cdot \left(R_j W^{UK} \mathbf{l}_j^{KV}\right) \\ &= \mathbf{l}_i^{Q\top} W^{UQ\top} R_i^\top R_j W^{UK} \mathbf{l}_j^{KV}\end{aligned} \tag{127}$$

The position-dependent rotation matrices R_i and R_j are now between the terms we want to absorb $\mathbf{l}_i^{Q\top}\left(W^{UQ\top} R_i^\top R_j W^{UK}\right)\mathbf{l}_j^{KV}$. We cannot precompute a fixed absorbed matrix W^{QK}_{absorbed} because R_i and R_j depend on the positions. This means for each new token i, we would need to compute a new R_i, compute $R_i^\top R_j$ for each previous position j, and multiply by the up-projection matrices for each token pair. This would require $\mathcal{O}(N^2)$ operations as the sequence length grows, destroying the efficiency advantage of MLA.

Instead, we are going to treat the positional encoding separately. From the latent representation of the queries $\mathbf{l}_i^Q$, we are mapping them to a d_{rotate} dimension with a W^{QR} (***Q***uery ***R***otate) matrix and applying the RoPE transformation. We are doing similar for the keys, but instead, we start from the hidden states $\mathbf{h}_j$ by using a W^{KR} (***K***ey ***R***otate) matrix:

$$\begin{aligned}\mathbf{q}_i^R &= \text{RoPE}\left(W^{QR}\mathbf{l}_i^Q\right) \\ \mathbf{k}_j^R &= \text{RoPE}\left(W^{KR}\mathbf{h}_j\right)\end{aligned} \tag{128}$$

W^{QR} has a dimension $d_{\text{rotate}} \times d_{\text{latent}}$ and W^{KR} has a dimension $d_{\text{rotate}} \times d_{\text{model}}$. d_{rotate} is usually chosen small enough to keep the memory bandwidth efficient, and in the original article, they chose $d_{\text{rotate}} = \frac{d_{\text{head}}}{2} = 64$. During inference, we only compute queries for the new token we generate. Since we're already computing the latent representation $\mathbf{l}_i^Q$ for the content path of this token, it is computationally efficient to derive the positional information from this same latent representation. The key side is handled differently for a crucial reason related to caching. By applying W^{KR} directly to the hidden states, we compute the position-aware key once for each token and then cache it. The position-aware key does not change as new tokens are generated, and we avoid having to

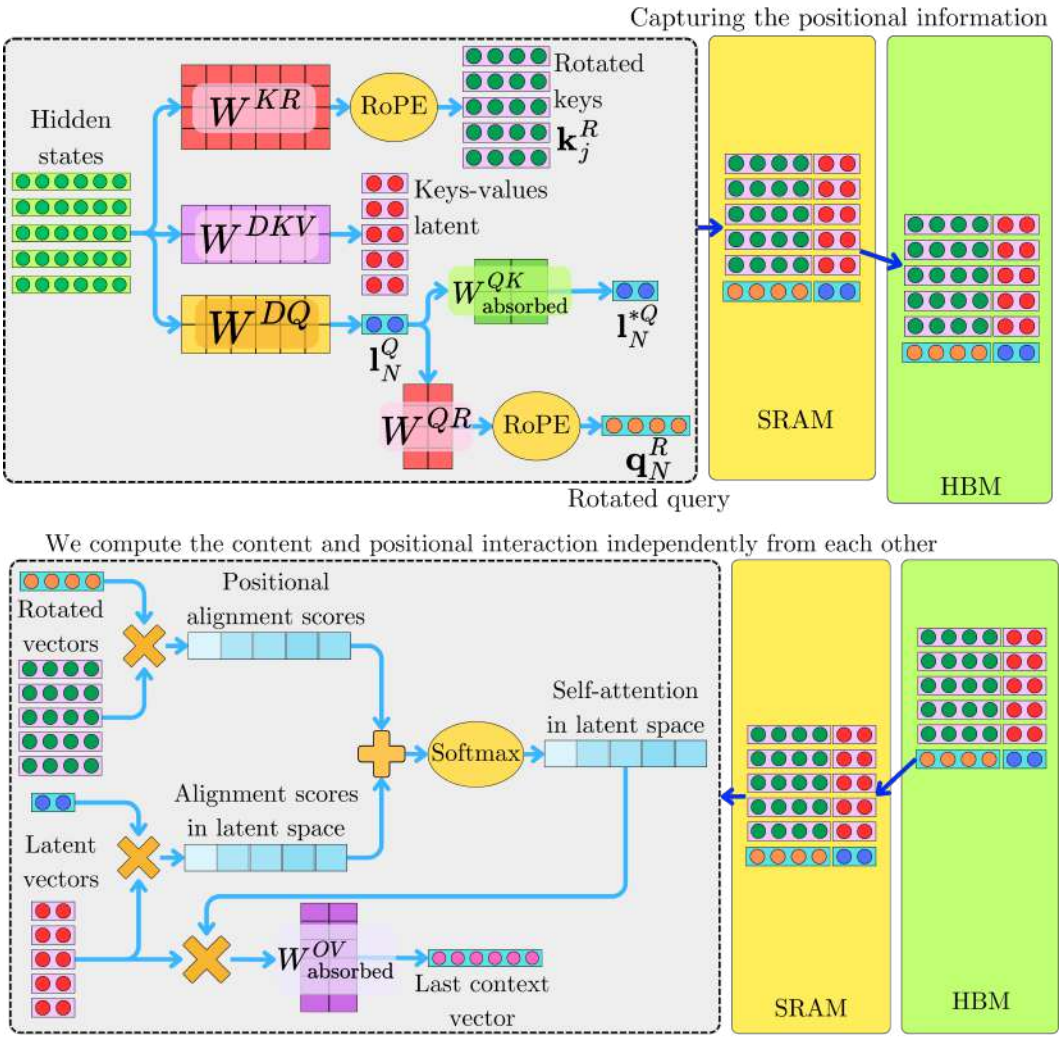

Fig. 64 *To account for the positional encoding, the process is slightly more complex than presented in fig. 63. We have dedicated matrices W^{KR} and W^{QR} to prepare the queries and keys for the RoPE transformation. The resulting vectors are cached and participate additively in the alignment score computation.*

regenerate keys from the latent representation during each inference step. If we had applied W^{KR} to the latent representation instead, we would need to either regenerate the position-aware keys at each step (inefficient) or cache both the latent representation and the position-aware keys (more memory).

The trick is to concatenate the rotated queries $\mathbf{q}_i^R$ and keys $\mathbf{k}_j^R$ to their content level counterparts $\mathbf{q}_i$ and $\mathbf{k}_j$:

$$\begin{aligned} \tilde{\mathbf{q}}_i &= \left[\mathbf{q}_i; \mathbf{q}_i^R\right], \quad \text{shape: } d_{\text{head}} + d_{\text{rotate}} \\ \tilde{\mathbf{k}}_j &= \left[\mathbf{k}_j; \mathbf{k}_j^R\right], \quad \text{shape: } d_{\text{head}} + d_{\text{rotate}} \end{aligned} \tag{129}$$

and to modify the context vectors computation:

$$\mathbf{c}_i = W^O \cdot \sum_{j=1}^{N} \text{Softmax}\left(\frac{\tilde{\mathbf{q}}_i^\top \cdot \tilde{\mathbf{k}}_j}{\sqrt{d_{\text{head}} + d_{\text{rotate}}}}\right) \mathbf{v}_j \tag{130}$$

Because we have:

$$\begin{aligned}\tilde{\mathbf{q}}_i^\top \cdot \tilde{\mathbf{k}}_j &= \left[\mathbf{q}_i; \mathbf{q}_i^R\right]^\top \cdot \left[\mathbf{k}_j; \mathbf{k}_j^R\right] \\ &= \underbrace{\mathbf{q}_i^\top \cdot \mathbf{k}_j}_{\text{score}_{\text{content}}} + \underbrace{\mathbf{q}_i^{R\top} \cdot \mathbf{k}_j^R}_{\text{score}_{\text{position}}}\end{aligned} \tag{131}$$

we can separate the alignment score into the contributions from the content $\mathbf{q}_i^\top \cdot \mathbf{k}_j$ and the position $\mathbf{q}_i^{R\top} \cdot \mathbf{k}_j^R$ parts. We can apply the matrix absorption trick to $\mathbf{q}_i^\top \cdot \mathbf{k}_j$ and handle independently the position piece $\mathbf{q}_i^{R\top} \cdot \mathbf{k}_j^R$ (see fig. 64).

For MHA, we have a $d_{\text{model}} \times N$ matrix for the keys and values. If we need to cache those in the KV-cache, it is $2 \times d_{\text{model}} \times N \times L$ elements for each token generated, where L is the number of layers in the model. For MLA, we have the $d_{\text{latent}} \times N$ latent representations $\mathbf{l}_j^{KV}$ and the $d_{\text{rotate}} \times N$ rotated representations $\mathbf{k}_j^R$. We then have a total of $(d_{\text{latent}} + d_{\text{rotate}}) \times N \times L$ elements to cache. To put numbers on those, DeepSeek-V2 uses:

- $d_{\text{model}} = 5120$
- $d_{\text{latent}} = 512$
- $d_{\text{rotate}} = 64$

We obtain:

$$\begin{aligned}2 \times 5120 \times 128000 \times 60 &\simeq 78.6 \text{ trillion elements for MHA} \\ (512 + 64) \times 128000 \times 60 &\simeq 4.4 \text{ trillion elements for MLA}\end{aligned} \tag{132}$$

That is a 94.4% reduction in memory access complexity! By reducing memory access requirements, it was able to utilize GPU computational resources much more efficiently, leading to the observed $5.76\times$ increase in generation throughput. What makes MLA particularly remarkable is that it does not just improve memory efficiency, it actually enhances predictive performance compared to standard Multi-Head Attention. This "win-win" outcome is relatively rare in neural network optimization, where efficiency gains typically come at the cost of performance. According to the DeepSeek-V2 paper and related ablation studies, MLA outperforms standard MHA on most benchmarks, rather than just matching it. This is counterintuitive since MLA uses compression techniques. The paper reports that on key benchmarks, MLA variants showed approximately 0.5-1.5% higher accuracy than equivalent MHA models with the same parameter count.

6.4 To Summarize

Attention Type	Key Innovations & Benefits	Limitations & Downsides
Multi-Query Attention (MQA)	Shares single set of keys/values across all heads, reducing KV cache by n_{head}	Quality degradation compared to MHA (0.5-1% perplexity); training instability issues; performance varies by task
Grouped-Query Attention (GQA)	Balances MQA and MHA by dividing heads into G groups sharing KV pairs, configurable trade-off	Requires conversion and additional pre-training from MHA models; optimal group size depends on hardware; slight quality loss
Multi-head Latent Attention (MLA)	Compresses queries, keys, and values into latent space, achieving up to $5.76\times$ throughput improvement	Complex implementation with separate content/positional paths; compression requires careful tuning; sensitive to latent space dimensionality

7 Long Sequence Attentions

In the previous sections, we examined methods to reduce attention's computational complexity. In this section, we will focus on designing attention mechanisms specifically optimized for processing extremely long contexts. The fundamental difference lies in the objective. Low-complexity attention methods primarily aim to approximate standard attention more efficiently, whereas long-sequence attention mechanisms fundamentally rethink how information flows across distant positions. Rather than merely making attention more computationally feasible, we will look at strategies to make distant contextual information meaningfully accessible and useful to the model.

7.1 Transformer-XL

Transformer-XL [8] was proposed in 2019 as a way to process sequences of virtually unlimited length while maintaining coherent information flow across the entire document. The main limitations to handling sequences of any length are:

- The typical time complexity $\mathcal{O}(N^2)$ of the attention layers. However, we have already seen strategies to reduce that complexity to $\mathcal{O}(N)$. For an autoregressive process, we are bounded from below by, at least, a linear decoding process in the sequence length. Therefore, we can never do better than this theoretical constraint.
- The absolute positional encoding proposed in the original "Attention is all you need" paper is the leading blocker for encoding arbitrary sequence lengths. We would need a way to encode any possible positions, which is challenging in practice.
- Another important blocker is the memory constraint. Longer sequences take more space in memory, and we would reach an upper bound in length when the GPU memory becomes saturated. We saw when discussing the FlashAttention that a high-end NVIDIA A100-8GB could realistically handle a maximum sequence length of 5,932 for a GPT-3 model.

With *Transformer-XL*, we will design an attention mechanism that processes sequences with linear time complexity, constant memory complexity, and a novel relative positional encoding that captures the relative distance between tokens instead of their absolute positions. We will delay diving into the relative positional encoding until the next chapter and focus here on how to process arbitrary sequence lengths in linear time with bounded memory constraints.

To illustrate how it works, let's consider the following toy example of an input sequence:

> "Teaching computers to see the world makes every colorful dataset an adventure"

The strategy with Transformer-XL is to handle the incoming tokens in segments. We break down the incoming sequence in segments, typically of size $\sim$128-512 tokens during training, and up to 1600 tokens during evaluation. For our toy example, let's assume that our segments are four tokens long:

- Segment 1: ['Teaching', 'computers', 'to', 'see']
- Segment 2: ['the', 'world', 'makes', 'every']
- Segment 3: ['colorful', 'dataset', 'an', 'adventure']

Formally, we divide the incoming sequence into $T = N/n$ segments, where n is the number of tokens per segment. We are going to create a segment-level recurrence to generate the output vectors from the model:

1. Generate all the hidden states $[H_1^1, H_1^2, \ldots, H_1^L]$ related to the first segment $\tau = 1$. $l \in \{1, \ldots, L\}$ is the index of the layer l and L is the total number of layers in the model. The full attention is computed within segments, so its time and memory complexity is $\mathcal{O}(n^2)$. We are going to cache the intermediate representations of the tokens $[H_1^1, H_1^2, \ldots, H_1^L]$ for the next iteration (fig. 65).
2. In the next iteration, we will consider the second segment and its interaction with the first segment (see fig. 66). At each layer, we retrieve the hidden states of segment 1 and append

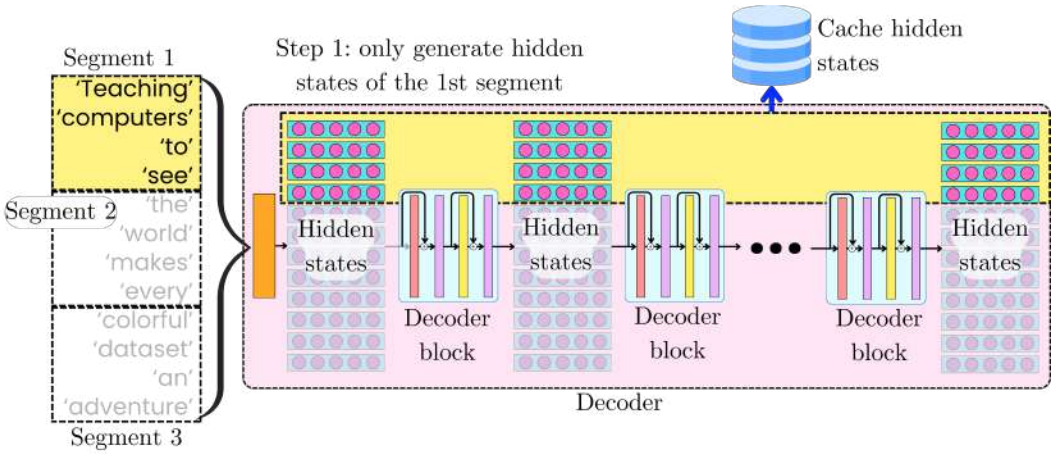

Fig. 65 *For Transformer-XL, the whole sequence is partitioned into fixed-sized segments. We iterate through the segments in sequence and cache the underlying hidden states to compute the consecutive segments' interactions during the following iteration.*

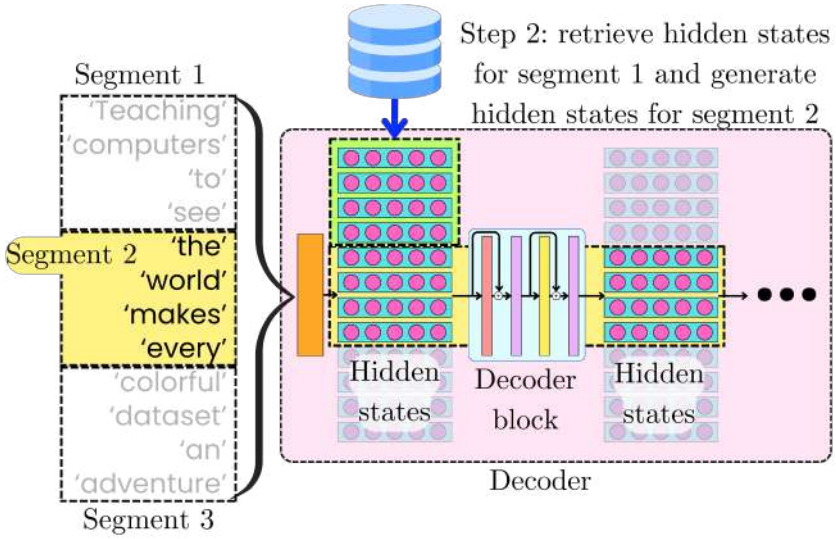

Fig. 66 *In the second iteration, we process the second segment in the sequence. We retrieve the hidden states computed for the previous segment to compute the local attention between the first and second segments.*

them to the hidden states of segment 2

$$\tilde{H}_2^l = \left[H_1^l; H_2^l\right]$$

and we compute the next hidden states by passing them through the different layers:

$$H_2^{l+1} = \text{Layer}_l\left(\tilde{H}_2^l\right)$$

Here, the attention matrix is computed across segments 1 and 2 and is therefore of size $2n \times 2n$, which still follows a quadratic complexity $\sim \mathcal{O}(n^2)$.

3. In general, for any segment τ, we retrieve the computed hidden states $[H_{\tau-1}^1, H_{\tau-1}^2, \ldots, H_{\tau-1}^L]$ for the previous segment $\tau - 1$, and append them to the hidden states of the current segment H_τ^l for layer l:

$$\tilde{H}_\tau^l = \left[H_{\tau-1}^l; H_\tau^l\right]$$

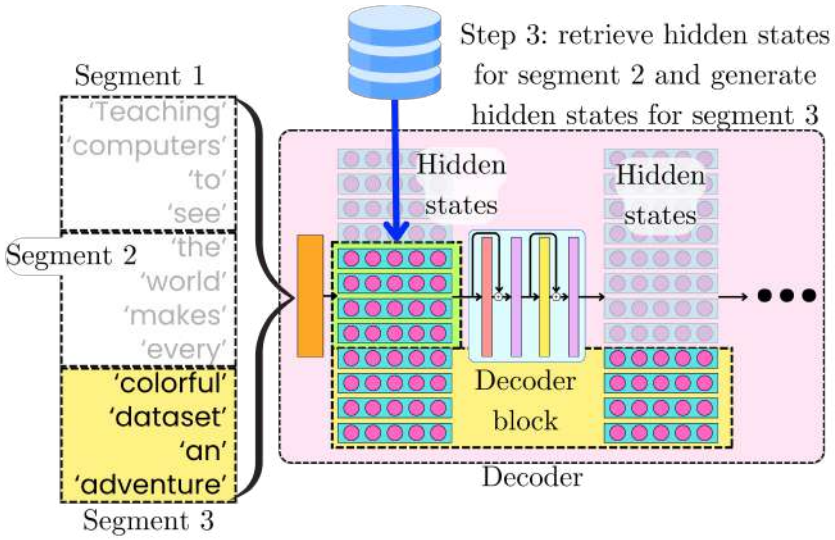

Fig. 67 *At any point during the process, we process the current segment while retrieving the hidden states computed for the previous segment. The hidden states related to the current segment capture the interaction with the previous segment.*

and compute the next hidden states:

$$H_\tau^{l+1} = \text{Layer}_l\left(\tilde{H}_\tau^l\right)$$

At every point during this recurring process, the time and space complexity is at most $\sim \mathcal{O}(n^2)$, and we iterate this process until we reach the last segment in the sequence (fig. 67).

The recurrence mechanism effectively creates a form of "memory" that allows the model to maintain coherent understanding across very long texts while keeping computational requirements manageable. Transformer-XL's ability to maintain coherence across long contexts directly relates to its depth. H_τ^{l+1} depends on $H_{\tau-1}^l$ and H_τ^l, which means that the generation of the n hidden states in H_τ^{l+1} depend on the $2 \times n$ hidden states in $\left[H_{\tau-1}^l; H_\tau^l\right]$. The hidden states $H_{\tau-1}^l$ also depend on $H_{\tau-2}^{l-1}$ and $H_{\tau-1}^{l-1}$, therefore H_τ^{l+1} depend on the $3 \times n$ hidden states in H_τ^l, $H_{\tau-1}^{l-1}$, and $H_{\tau-2}^{l-1}$. If we go back k layers in the model, then the hidden states in H_τ^{l+1} depend on the $(k+1) \times n$ hidden states in $[H_\tau^{l-k+1}, H_{\tau-1}^{l-k+1}, \ldots, H_{\tau-k}^{l-k+1}]$. By unrolling the recurrence of the hidden states' dependency, we establish the functional relationship:

$$H_\tau^{l+1} = f_k(H_\tau^{l-k+1}, H_{\tau-1}^{l-k+1}, \ldots, H_{\tau-k}^{l-k+1}) \tag{133}$$

This creates a recurrence relation where information from segment $\tau - k$ can reach the final layer L in segment τ only if $k + 1 \leq L$. If we want to cover all the N tokens in the sequence, with $k = N/n - 1$, this means we need:

$$\frac{N}{n} \leq L$$
$$\text{or equivalently } N \leq nL \tag{134}$$

The theoretical dependency length is therefore $\mathcal{O}(n \times L)$ (see fig. 68). If the network is not deep enough, its ability to propagate information across segments would be limited. This is because information flows through layers within a segment before being passed to the next segment. A shallow Transformer-XL would still have the unbounded context mechanism but might not effectively utilize information from distant parts of the sequence.

So far, we have only considered the forward pass, but it becomes quite messy when considering the backward pass! When we compute the gradient of the loss function $\mathcal{L}$, we need to compute its

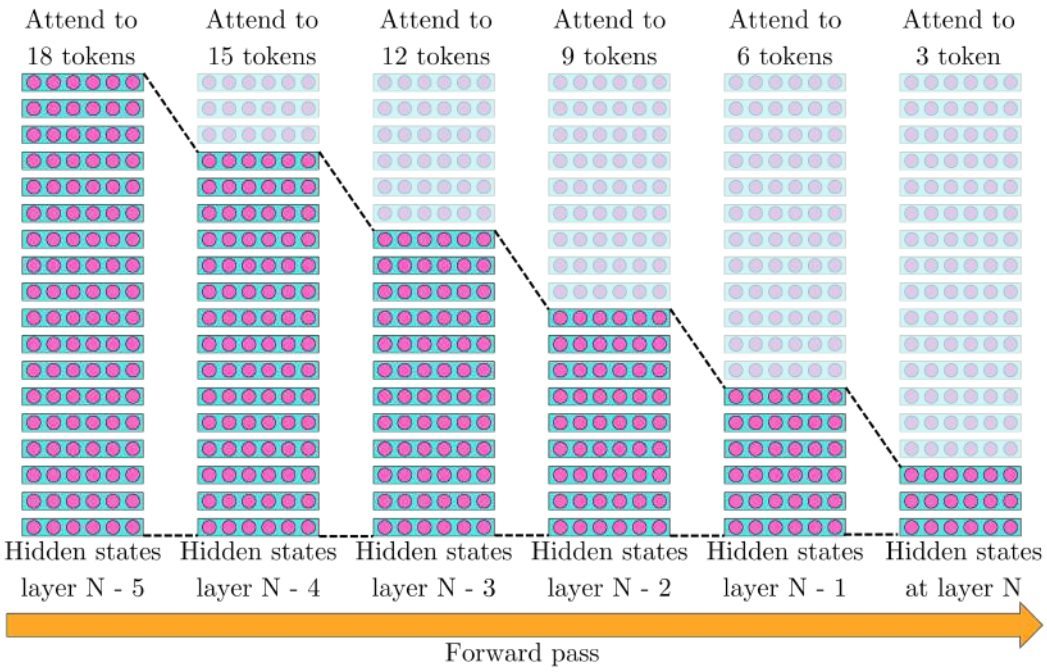

Fig. 68 *For the vanilla MHA, an outbound hidden state is a linear combination of all the inbound hidden states due to the full attention mechanism. For the Transformer-XL, from one layer to the next, the resulting hidden states for a specific segment only carry information related to the current and previous segments. By compounding multiple layers, the hidden states of a specific segment can capture the information of as many previous segments as there are layers, ensuring long-range coherence for a deep transformer model.*

relation to the hidden states H_{τ}^{l+1}:

$$\frac{\partial \mathcal{L}}{\partial H_{\tau-k}^{l-k+1}} = \sum_{i=l-k+1}^{l+1} \sum_{j=\tau-k}^{\tau} \frac{\partial \mathcal{L}}{\partial H_j^i} \cdot \frac{\partial H_j^i}{\partial H_{\tau-k}^{l-k+1}} \tag{135}$$

The layer-wise summation $\sum_{i=l-k+1}^{l+1}$ must consider all layers from $l-k+1$ up to $l+1$ and the segment-wise summation $\sum_{j=\tau-k}^{\tau}$ must consider all segments from $\tau-k$ up to τ. This creates $(k+1)^2$ gradient paths for a single hidden state segment. Now, for a sequence of length N divided into N/n segments, if we extend this to consider all possible hidden states that influence the loss, we get approximately:

$$\sum_{k=1}^{N/n} (k+1)^2 \approx \frac{(N/n)^3}{3} \sim \mathcal{O}(N^3) \tag{136}$$

gradient paths to compute. This cubic growth makes training infeasible for long sequences. We are going to modify the segment-level dependency by introducing the Stop-Gradient $SG(\cdot)$ operator:

$$\tilde{H}_{\tau}^{l} = \left[SG(H_{\tau-1}^{l}); H_{\tau}^{l}\right] \tag{137}$$

This operation prevents gradients from flowing backward through the cached states during back-propagation. This means:

$$\begin{aligned} SG(H_{\tau-1}^{l}) &= H_{\tau-1}^{l} && \text{During the forward pass} \\ \frac{\partial \tilde{H}_{\tau}^{l}}{\partial SG(H_{\tau-1}^{l})} &= 0 && \text{During the backward pass} \end{aligned} \tag{138}$$

By applying $SG(\cdot)$, the previous segment hidden states $H_{\tau-1}^{l}$ are treated as constant during the backward pass. When we apply $SG(\cdot)$ to prevent gradient flow across segment boundaries, we introduce a profound asymmetry:

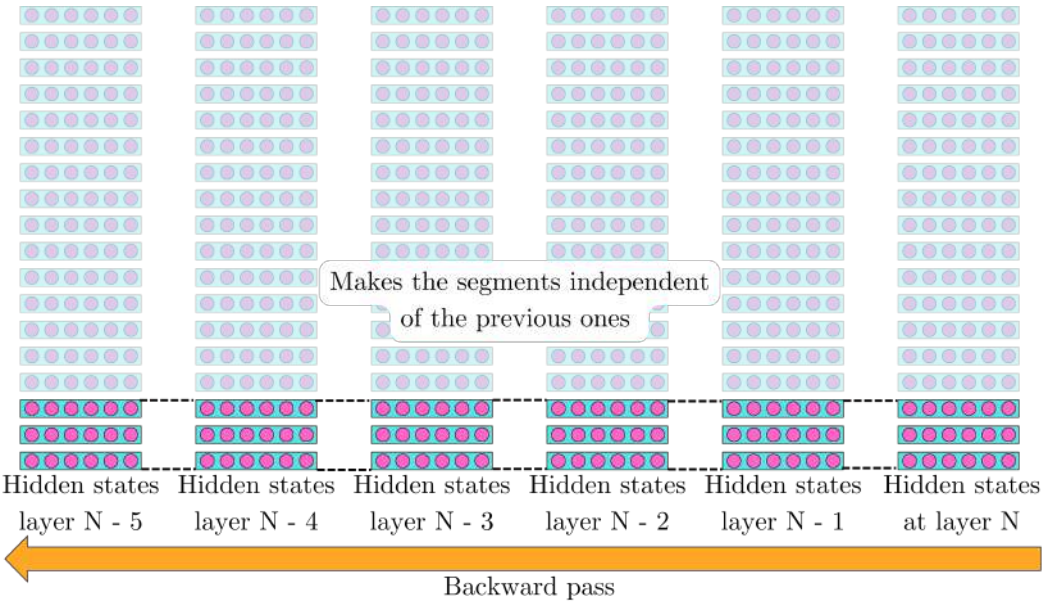

Fig. 69 *For Transformer-XL, during the backward pass of the backpropagation algorithm, we prevent the gradients from flowing to the previous segments to reduce the memory required to store the whole dependency graph for long sequences. This means that the learned behavior resulting from the previous segment will not be affected by the mistakes happening in the later segments.*

- Information Asymmetry: During the forward pass, the model can access and use information from previous segments, but during backpropagation, it cannot receive gradient signals from future segments.
- Truncated Credit Assignment: The model cannot directly attribute credit or blame to decisions made in previous segments, even though those decisions influence future outcomes.

This creates a unique learning paradigm where the model must learn to encode useful information in its hidden states without direct optimization signals for long-range dependencies. Interestingly, despite not being directly optimized for very long dependencies, *Transformer-XL* still develops impressive long-range capabilities because the recurrence mechanism creates paths for information to propagate forward (see fig. 69).

For any two layers a and b, we have

$$\frac{\partial H_{\tau'}^{a}}{\partial SG(H_{\tau}^{b})} = 0 \quad \text{if } \tau' \neq \tau \tag{139}$$

Therefore, the loss backpropagation simplifies to

$$\frac{\partial \mathcal{L}}{\partial H_{\tau-k}^{l-k+1}} = \sum_{i=l-k+1}^{l+1} \frac{\partial \mathcal{L}}{\partial H_{j}^{i}} \cdot \frac{\partial H_{j}^{i}}{\partial H_{\tau-k}^{l-k+1}} \tag{140}$$

which reduces the computational complexity from the cubic behavior $\mathcal{O}((N/n)^3)$ to the linear one $\mathcal{O}(N/n)$.

Each segment has length n, and when processing a single segment, attention is computed over the current segment plus the cached previous segment. The time complexity of the computation per segment is therefore $\sim \mathcal{O}(4n^2) = \mathcal{O}(n^2)$. For a sequence of length N, we process approximately $T = N/n$ segments, and each segment requires $\mathcal{O}(n^2)$ operations. Therefore, the complexity of the total operations is:

$$\mathcal{O}\left(\frac{N}{n} \times n^2\right) = \mathcal{O}(Nn) \tag{141}$$

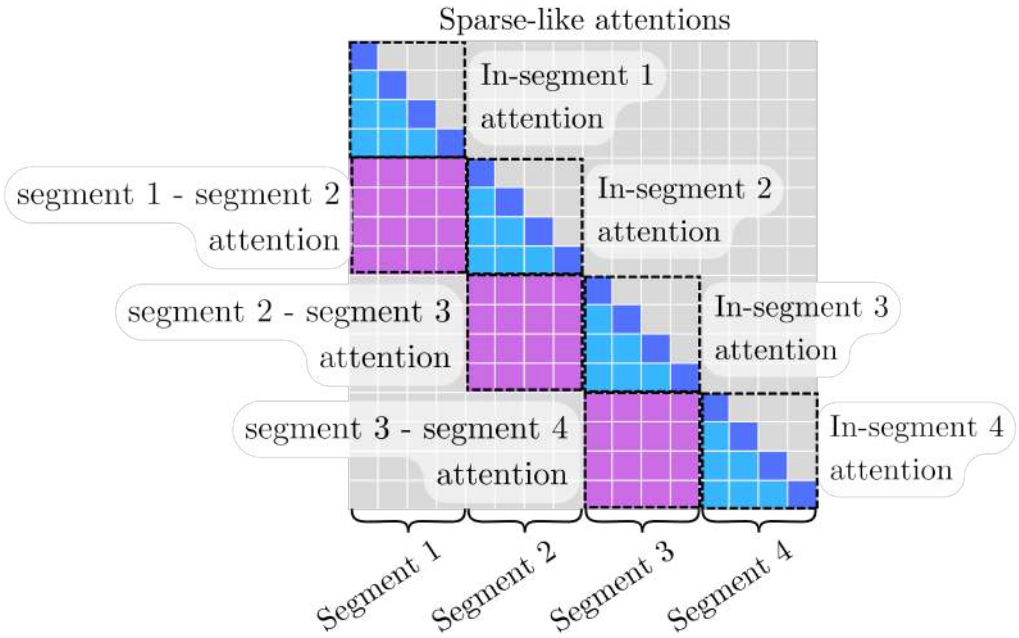

Fig. 70 *The connectivity pattern between the different tokens in Transformer-XL is similar to the sparse attention patterns we have seen previously. But, as opposed to the sparse attention, the memory requirements do not grow with the sequence size.*

which is linear in the sequence size. Regarding how attention is computed, it is similar to the sparse attention with a sliding window. Beyond the current and past segments, the hidden states are blind to the other tokens (see fig. 70).

However, with sparse attention, the memory requirements grow linearly with the sequence size, but with *Transformer-XL*, the space complexity is bounded by $\mathcal{O}(n^2) = \mathcal{O}(1)$ as n is a fixed number. We still need to choose n large enough to use the high GPU parallelism efficiently.

7.2 Memorizing Transformers

In *Transformer-XL*, the long-range coherence is captured by the successive layers in the model, but the direct interaction between tokens is lost beyond a two-segment window. Google introduced the *Memorizing Transformers* in 2022 [34] that extended the long-range coherence by caching previous key-value pairs for selective retrieval. The attention computation is broken down into two parts:

- **The local attention:** As before, we partition the input sequence into segments (usually 512 tokens) and compute the token-token interactions within each segment:

$$C_{\text{local}} = \text{Softmax}\left(\frac{Q_{\text{local}} K_{\text{local}}^{\top}}{\sqrt{d}}\right) V_{\text{local}} \tag{142}$$

  where Q_{local}, K_{local}, and V_{local} are the local queries, keys, and values within the segment.
- **The memory attention:** After processing a segment, the related keys and values are added to a local cache. This cache acts as a memory of the previous segments. Once a new segment is processed, besides the in-segment attention, we also retrieve key-value pairs $(K_{\text{mem}}, V_{\text{mem}})$ from the memory to provide context from previous segments and capture longer-range dependencies (see fig. 71):

$$C_{\text{mem}} = \text{Softmax}\left(\frac{Q_{\text{local}} K_{\text{mem}}^{\top}}{\sqrt{d_{\text{head}}}}\right) V_{\text{mem}} \tag{143}$$

The two resulting sets of context vectors are then simply combined with a weighted average:

$$C = C_{\text{local}} \cdot \sigma(\mathbf{b}) + C_{\text{mem}} \cdot (1 - \sigma(\mathbf{b})) \tag{144}$$

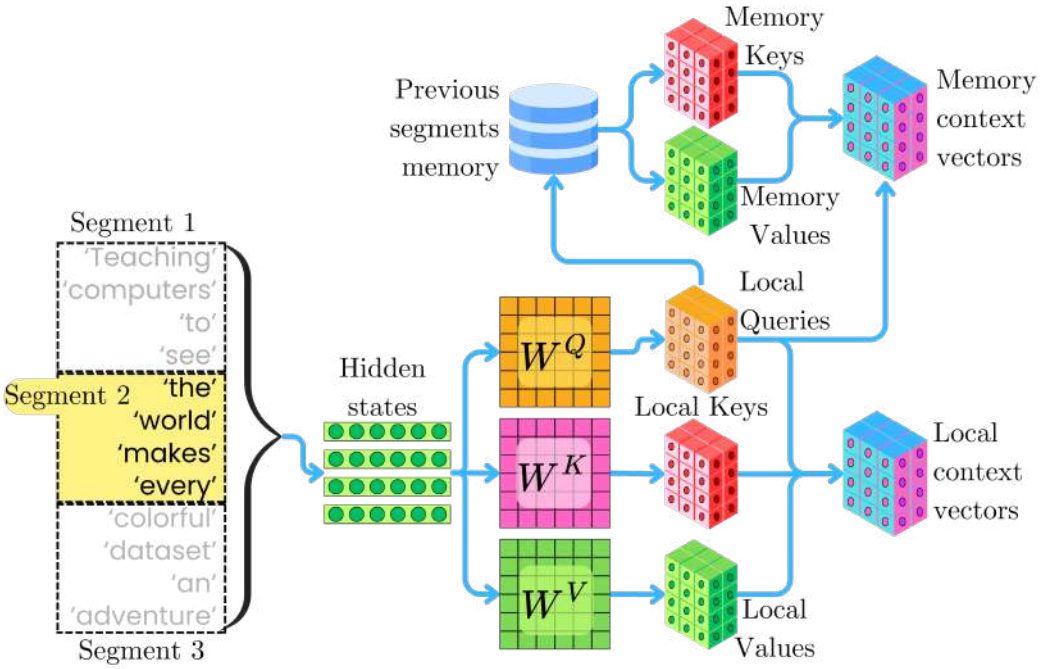

Fig. 71 *With the Memorizing Transformer, the keys and queries are constantly stored in a cache as a memory of the previous segments. We compute two sets of context vectors: the interaction between the queries of the current segment and a subset of the keys and values retrieved from the memory, and the local interaction between the tokens within the current segment.*

where $\sigma(\mathbf{b})$ is the sigmoid function that acts as a gate for each head, and $\mathbf{b}$ is a learned vector of size n_{head} that will put more or less weight on local vs memory contributions. This gating mechanism allows each head to learn whether to focus more on local context or long-range memory. Once combined, the context vectors are reshaped from $d_{\text{head}} \times n_{\text{head}} \times n$ to $d_{\text{model}} \times n$, where n is the number of tokens per segment, and passed through the last projection matrix W^O as usual (see fig. 72):

$$C_{\text{final}} = W^O \cdot \text{Reshape}(C)$$

One important architectural detail is that this caching mechanism is done for only one of the attention layers in the model. Applying the memory retrieval mechanism to every layer would significantly increase computation time and resource requirements. Even with just one memory-augmented layer, the paper reports step time increasing from 0.2s to 0.6s when scaling memory from 8K to 65K tokens. In their experiments, they mention using the 9th layer of a 12-layer transformer as the memory-augmented attention layer. Placing the memory layer too early means the representations are not rich enough to form effective queries, while placing it too late means the retrieved information has less opportunity to influence the final predictions.

Once a segment is processed, the related keys $[\mathbf{k}_1, \ldots, \mathbf{k}_n]$ and values $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ are stored in the memory:

$$M_{\text{new}} = M_{\text{old}} \cup \{(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)\} \tag{145}$$

In their experiments, the memory size ranged from 8K to 262K tokens. If the memory exceeds its maximum size, the oldest entries are removed. When processing a new segment, for each query $\mathbf{q}_i$, we find the k most similar keys $[\mathbf{k}_1^{\text{mem}}, \ldots, \mathbf{k}_k^{\text{mem}}]$ in memory, where similarity is measured by the dot-product metric:

$$\text{sim}\left(\mathbf{q}_i, \mathbf{k}_j^{\text{mem}}\right) = \mathbf{q}_i^\top \cdot \mathbf{k}_j^{\text{mem}} \tag{146}$$

During training, as the segments are processed, the model parameters are updated through the backpropagation algorithm, and the queries, keys, and values representations evolve. To mitigate the

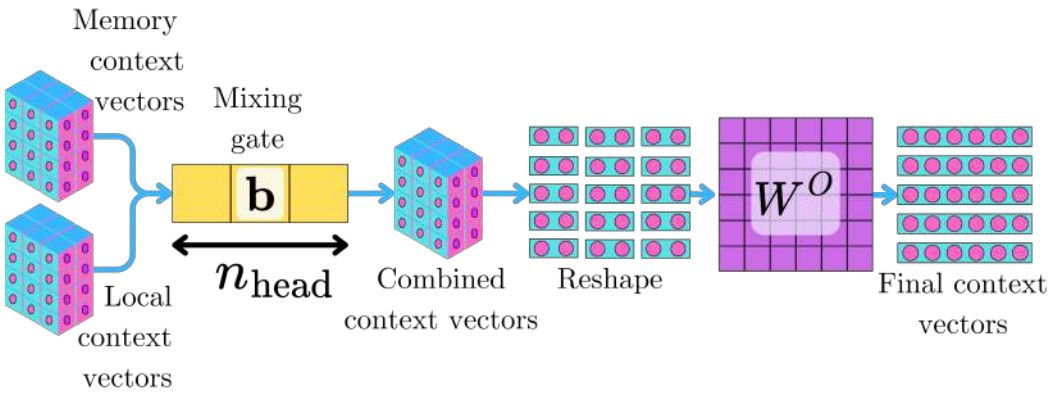

Fig. 72 *Once the two sets of context vectors are generated, they are combined through a mixing gate that will learn to put more or less importance on the local context or the long-range memory.*

effects of the distributional shift, the queries and keys are normalized:

$$\hat{\mathbf{q}}_i = \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|}, \quad \hat{\mathbf{k}}_j^{\text{mem}} = \frac{\mathbf{k}_j^{\text{mem}}}{\left\|\mathbf{k}_j^{\text{mem}}\right\|} \tag{147}$$

Normalization constrains all vectors to live on the unit hypersphere, creating a more stable similarity space even as parameters evolve. By normalizing, we emphasize the relative patterns in the vectors rather than their absolute values. Recall that the softmax transformation exacerbates the largest values; therefore, the computation of context vectors is dominated by the terms with the highest $\mathbf{q}_i^\top \cdot \mathbf{k}_j^{\text{mem}}$ product:

$$\mathbf{c}_i^{\text{mem}} = \sum_{j=1}^{k} \text{Softmax}\left(\hat{\mathbf{q}}_i^\top \cdot \hat{\mathbf{k}}_j^{\text{mem}}\right) \mathbf{v}_j^{\text{mem}} \tag{148}$$

motivating the search for the top-k similar keys in the memory. They found that $k = 128$ led to the best results, while $k = 32$ provided less computational requirements with minimal loss in performance. To retrieve the top-k similar keys, they used an approximate k-Nearest Neighbor (kNN) approximate algorithm such as ScaNN [14] and Faiss [15] guaranteeing a retrieval time $\sim \mathcal{O}(\log M)$ for 90% recall performance, where M is the memory size.

As for the *Transformer-XL*, the gradients are not backpropagated into the memory to prevent the unbounded growth of the memory during the backward pass. The backward pass does not require storing computation graphs for the entire memory, only for the k retrieved items per query. The external memory content (keys and values) is treated as non-differentiable constants (we do not update W^K and W^V), and the gradients flow through the operations performed using those retrieved items to update the query computation (to update W^Q).

When processing a new segment, the first token only has access to the external memory, and its query vector is formed with minimal context, potentially reducing retrieval quality. This problem can be addressed by combining the *Memorizing Transformer* with *Transformer-XL*'s caching mechanism, where we cache the hidden states of the previous segment to provide better context to the initial tokens. Overall, this led to better coherence than *Transformer-XL* over very long documents. One of the most significant findings from the Memorizing Transformer research was the dramatic parameter efficiency it achieved. The experiments demonstrated that adding even a modest memory component (8K tokens) to a 200M parameter model yielded better perplexity than a vanilla transformer with 1B parameters, representing a 5$\times$ parameter efficiency gain. This efficiency held across model scales, suggesting that explicitly storing and retrieving information provides a fundamentally different advantage than simply implicitly encoding knowledge in more model parameters. This finding challenges the scaling paradigm in language modeling, indicating that architectural innovations focused on memory and retrieval may offer more efficient paths to performance improvement than simply increasing model size.

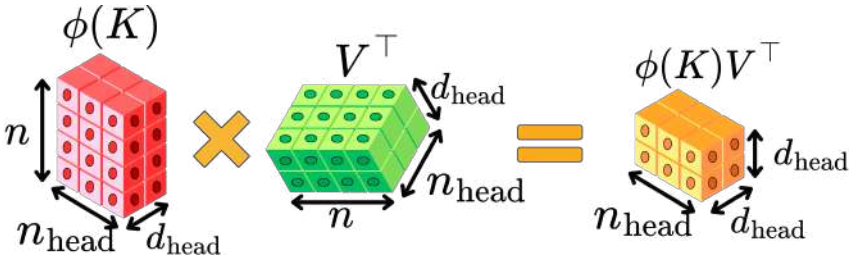

Fig. 73 *Graphical representation of the outer product $\phi(K)V^\top$ generating a tensor independent of the sequence size.*

7.3 Infini-Attention

The problem with the *Memorizing Transformers* is the need to store a growing memory of keys and values as we progress through the sequence, and the memory constraint prevents us from implementing this retrieval mechanism within multiple layers in the model. In 2024, Google introduced the *Infini-attention* [21] with constant-size memory for long-sequence processing. To create a memory independent of the sequence length, they utilized the trick developed in the *Linear Transformer*. We replace the softmax function with a linear similarity kernel:

$$\text{Softmax}\left(\mathbf{q}_i^\top \mathbf{k}_j^{\text{mem}}\right) \to \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j^{\text{mem}}) \tag{149}$$

where ϕ is a feature map. In this specific case, ϕ is chosen as:

$$\phi(x) = \text{ELU}(x) + 1 = \begin{cases} x+1 & \text{if } x > 0, \\ \exp x & \text{otherwise.} \end{cases} \tag{150}$$

with ELU being the Exponential Linear Unit function often used as activation function within neural networks. By using this replacement, the computation of a context vector associated with a current token and its interaction with keys and values from previous sequence segments becomes:

$$\begin{aligned} \mathbf{c}_i^{\text{mem}} &= \frac{\sum_{j=1}^{tn} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j^{\text{mem}}) \mathbf{v}_j^{\text{mem}}}{\sum_{j=1}^{tn} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j^{\text{mem}})} \\ &= \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}}) \mathbf{v}_j^{\top\text{mem}}}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}})} \end{aligned} \tag{151}$$

assuming t is the current number of segments processed so far and n is the number of tokens per segment. $\sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}}) \mathbf{v}_j^{\top\text{mem}}$ is a $n_{\text{head}} \times d_{\text{head}} \times d_{\text{head}}$ tensor and is independent of the sequence length (fig. 73), and $\sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}})$ is a $n_{\text{head}} \times d_{\text{head}}$ tensor, also independent of the sequence length.

By keeping track of $\sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}}) \mathbf{v}_j^{\top\text{mem}}$ and $\sum_{j=1}^{tn} \phi(\mathbf{k}_j^{\text{mem}})$, we effectively obtain a memory of all the past tokens. As for the *Memorizing Transformer*, as we iterate through the segments, we break down the contribution of the different tokens in the whole sequence into two parts:

- We have the local attention computed within the current segment:

$$C_{\text{local}} = \text{Softmax}\left(\frac{Q_{\text{local}} K_{\text{local}}^\top}{\sqrt{d}}\right) V_{\text{local}} \tag{152}$$

  where Q_{local}, K_{local}, and V_{local} represent the queries, keys, and values corresponding to the tokens of the segment currently being processed.

- We have the attention computed between the tokens of the current segment $t+1$, and the tokens of all the previous segments $\{1, \ldots, t\}$:

$$C_{\text{mem}} = \frac{\phi(Q_{\text{local}}) M_t}{\phi(Q_{\text{local}}) z_t} \tag{153}$$

  where M_t and z_t are the memory accumulation over all the previous segments (fig. 74):

$$M_t = \sum_{j=1}^{t} \phi(K_j) V_j^\top, \quad z_t = \sum_{j=1}^{t} \phi(K_j) \tag{154}$$

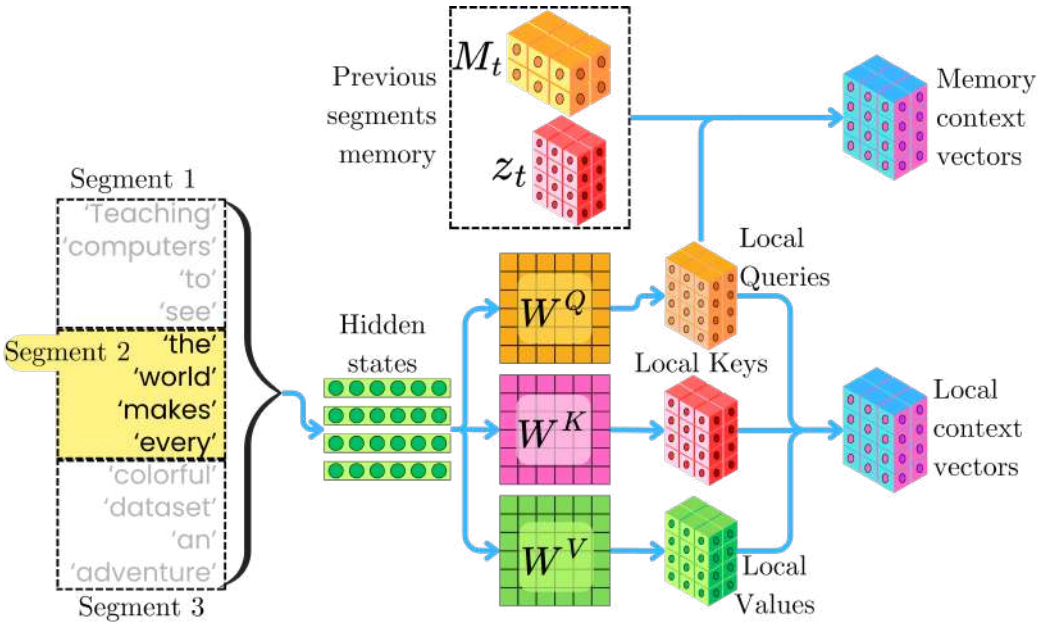

Fig. 74 *As opposed to the caching memory strategy of the Memorizing Transformer, the Infini-Attention captures the memory of previous segments in the form of the M_t and z_t tensors. M_t and z_t have a fixed size so that the strategy can be applied at every layer.*

The two attention contributions are once again combined as a weighted average:

$$C = C_{\text{local}} \cdot \sigma(\mathbf{b}) + C_{\text{mem}} \cdot (1 - \sigma(\mathbf{b})) \tag{155}$$

where σ is the sigmoid function and $\mathbf{b}$ is a learning parameter of size n_{head} that will help the different heads to specialize more or less on local context or long-range memory. Once a segment is processed by the model (i.e, the latest K_{local}, Q_{local}, and V_{local} have been generated), we can update the memory with the latest keys and values (fig. 75):

$$\begin{aligned} M_{t+1} &\leftarrow M_t + \phi(K_{\text{local}})V_{\text{local}}^\top \\ z_{t+1} &\leftarrow z_t + \phi(K_{\text{local}}) \end{aligned} \tag{156}$$

As opposed to *Transformer-XL* and *Memorizing Transformers*, we can backpropagate the gradients through the memory as the memory constraint remains bounded. When we consider the influence of M_t on the loss function $\mathcal{L}$, we need to compute $\partial\mathcal{L}/\partial M_t$. Since M_t is used as input in the following operations:

$$\begin{aligned} M_{t+1} &= M_t + \phi(K_{\text{local}})V_{\text{local}}^\top \\ C_{\text{mem}} &= \frac{\phi(Q_{\text{local}})M_t}{\phi(Q_{\text{local}})z_t} \end{aligned} \tag{157}$$

the gradient of the loss function can flow through C_{mem} and M_{t+1}:

$$\frac{\partial\mathcal{L}}{\partial M_t} = \frac{\partial\mathcal{L}}{\partial C_{\text{mem}}}\frac{\partial C_{\text{mem}}}{\partial M_t} + \frac{\partial\mathcal{L}}{\partial M_{t+1}}\frac{\partial M_{t+1}}{\partial M_t} \tag{158}$$

So at any point during the backpropagation algorithm, for segment $t+1$, we compute $\frac{\partial\mathcal{L}}{\partial M_{t+1}}$ and use it to compute $\frac{\partial\mathcal{L}}{\partial M_t}$. Once $\frac{\partial\mathcal{L}}{\partial M_t}$ is computed, $\frac{\partial\mathcal{L}}{\partial M_{t+1}}$ can be discarded before we iterate to the previous segment. The sequential chain of dependencies resembles the one seen with recurrent networks, and it prevents the memory requirements from growing unbounded.

The memory update presented in Eq. 156 is a direct consequence of modeling the attention as a linear similarity kernel

$$\text{Softmax}\left(\mathbf{q}_i^\top \mathbf{k}_j^{\text{mem}}\right) \rightarrow \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j^{\text{mem}})$$

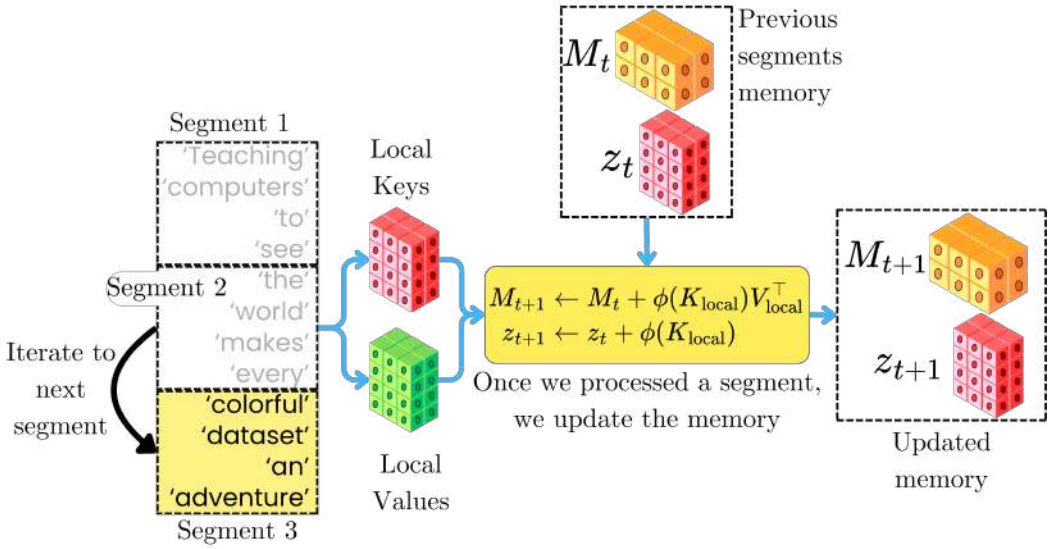

Fig. 75 *Once we computed the context vectors for the latest segment, we can update the memory by adding the latest keys and values.*

However, there are potentially better approaches to capture the relation between queries and keys. When we apply a query vector $\mathbf{q}_i$ to the memory (M_t, z_t), we want to compute a weighted average of the previous segment values $[\mathbf{v}_1^{\text{mem}}, \dots, \mathbf{v}_{tn}^{\text{mem}}]$ such that the weights a_{ij} capture the meaningful interaction between the query $\mathbf{q}_i$ and the previous segment keys $[\mathbf{k}_1^{\text{mem}}, \dots, \mathbf{k}_{tn}^{\text{mem}}]$:

$$\mathbf{c}_i^{\text{mem}} = \frac{\phi(\mathbf{q}_i)M_t}{\phi(\mathbf{q}_i)z_t} \approx \sum_{j=1}^{tn} a_{ij}\mathbf{v}_j^{\text{mem}} \tag{159}$$

Therefore, we want to design a memory that returns the values vectors with their associated weights when applied a query $\mathbf{q}_i$. The memory can be understood as a retrieval operator that correctly captures the relationship between the keys and values. When we apply a query, we assess its similarity to the keys and return the associated values as a fuzzy retrieval system. However, when we apply a key $\mathbf{k}_j^{\text{mem}}$ to the memory, we should have an exact match with itself if the key has already been added to the memory. Because we have an exact match, the memory should only return the value vector $\mathbf{v}_j^{\text{mem}}$ associated with $\mathbf{k}_j^{\text{mem}}$. More explicitly, we want to design a memory that captures the original meaning behind the queries, keys, and values within the context of information retrieval (see fig. 76).

Formally, when we apply $\mathbf{k}_j^{\text{mem}}$ to the memory, we want:

$$\frac{\phi(\mathbf{k}_j^{\text{mem}})M_t}{\phi(\mathbf{k}_j)z_t} \approx \mathbf{v}_j^{\text{mem}} \tag{160}$$

Let's reframe this problem as an optimization problem where we want to learn M_t^* such that:

$$M_t^* = \arg\min_{M_t} \left\| V - \frac{\phi(K_{\text{mem}})M_t}{\phi(K_{\text{mem}})z_t} \right\|^2 \tag{161}$$

If we apply gradient descent to this optimization problem with a learning rate of 1, we get:

$$M_t^* \leftarrow M_t^* - \alpha \frac{\partial}{\partial M_t} \left(\left\| V - \frac{\phi(K_{\text{mem}})M_t}{\phi(K_{\text{mem}})z_t} \right\|^2 \right) \tag{162}$$

Instead of solving the gradient descent problem exactly, they replace it by a heuristic approach where the strength of the gradient update is approximated by the strength of the key activations $\phi(K_{\text{mem}})$

$$\frac{\partial}{\partial M_t} \left(\left\| V - \frac{\phi(K_{\text{mem}})M_t}{\phi(K_{\text{mem}})z_t} \right\|^2 \right) \approx -\phi(K_{\text{mem}})^\top \left(V - \frac{\phi(K_{\text{mem}})M_t}{\phi(K_{\text{mem}})z_t} \right) \tag{163}$$

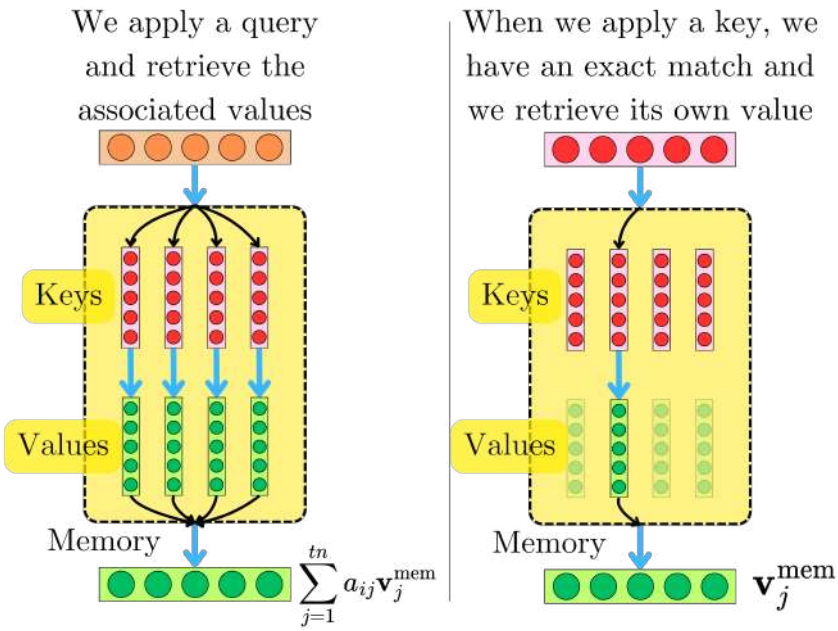

Fig. 76 *We want to engineer a memory that acts as a retriever operator. If we apply a query, we retrieve all the values weighted by how similar the keys are to the query. However, if we apply a key that has already been added to the memory, it should match exactly and retrieve only the associated value.*

The weighting by $\phi(K_{\text{mem}})$ ensures that the strength of association in memory updates matches the strength of the keys. This prevents weakly activated keys from introducing noise into the memory system while allowing strongly activated keys to make significant corrections to the memory update. In the attention mechanism, weakly activated keys would not lead to significant contributions in the similarity $\text{sim}(Q, K_{\text{mem}})$ between keys and queries. This leads to the following delta update rule once the latest segment has been processed:

$$\begin{aligned} M_{t+1} &\leftarrow M_t + \phi(K_{\text{local}})^\top \left(V_{\text{local}} - \frac{\phi(K_{\text{local}})M_t}{\phi(K_{\text{local}})z_t} \right) \\ z_{t+1} &\leftarrow z_t + \phi(K_{\text{local}}) \end{aligned} \tag{164}$$

The difference $\left(V_{\text{local}} - \frac{\phi(K_{\text{local}})M_t}{\phi(K_{\text{local}})z_t} \right)$ represents new information that needs to be learned. If the prediction is wrong, the memory updates proportionally to the error. If the memory already correctly maps K_{local} to V_{local}, the error term becomes zero, and no redundant update occurs. Only updating what is necessary avoids disrupting existing associations that are already correctly stored. This approach directs the memory's attention to what is genuinely new or different, and helps maintain older memories that might otherwise get overwritten.

The *Infini-Attention* achieves a $114\times$ compression ratio compared to *Memorizing Transformers* while maintaining better performance. It showed an extraordinary ability to generalize to much longer contexts than it was trained on. A 1B parameter model fine-tuned on just 5K length inputs could effectively handle inputs up to 1 million tokens in length. This is one of the most dramatic examples of length generalization in any transformer architecture. The passkey retrieval task (finding a specific number hidden among a long text) showcased *Infini-Attention*'s long-range memory capabilities. It achieved 100% accuracy on retrieving passkeys hidden at the beginning, middle, or end of sequences and successfully handled context lengths of 32K, 128K, 256K, 512K, and even 1M tokens. This is consistent with the current 2M tokens context size of Google's Gemini models, as they are likely utilizing similar and improved strategies to handle long sequences.

7.4 To Summarize

Attention Type	Key Innovations & Benefits	Limitations & Downsides
Transformer-XL	Segment-level recurrence with cached representations enables virtually unlimited context length	Stop-gradient limits true backpropagation through time; segment boundary effects; theoretical dependency length limited by network depth
Memorizing Transformers	External cache of KV pairs with kNN retrieval, outperforms larger models with smaller memory	Large memory requirements; kNN retrieval overhead increases with memory size; typically applied to only one layer due to computational cost
Infini-Attention	Constant-size memory with linear similarity kernel enables processing sequences up to 1M tokens	Linear kernel approximation limits expressiveness; potential memory staleness with long sequences; heuristic memory update may not be optimal

8 To Conclude

As we've seen throughout this article, the evolution of attention mechanisms reveals several important insights about efficient transformer architectures:

The Importance of Memory Access. Many innovations focus not just on reducing computational complexity but specifically on minimizing memory bandwidth bottlenecks. FlashAttention demonstrated that on modern hardware, the cost of moving data often exceeds the cost of computation itself. Future attention mechanisms will likely continue to optimize for specific hardware characteristics.

Approximation vs. Exact Computation. We have observed a spectrum of approaches from exact computation (FlashAttention) to various approximations (Linformer, Performer). While approximations offer theoretical efficiency, exact computations that leverage hardware optimizations often deliver better practical performance. The best approach depends on the specific use case and sequence length requirements.

Specialization for Training vs. Inference. Attention optimizations increasingly diverge between training and inference priorities. While training benefits from bidirectional attention and parallelism, inference (especially for autoregressive generation) requires optimizations for KV-caching and incremental processing. Mechanisms like MQA, GQA, and MLA specifically target inference efficiency.

Unbounded Context Length. Long-sequence attention mechanisms demonstrate that transformers can process virtually unlimited context with architectural innovations like recurrence, retrieval, and constant-memory representations. These approaches enable applications requiring extremely long contexts without proportionally increasing computational resources.

Complementary Techniques. Many of these attention variants can be combined. For example, FlashAttention can be applied to implement MQA or GQA more efficiently, or Infini-Attention could incorporate FlashAttention's tiling approach for better hardware utilization.

The rapid pace of innovation in attention mechanisms continues to expand the practical capabilities of transformer models. Modern implementations typically combine multiple techniques from this chapter, selecting the specific optimizations that best match their application requirements, hardware constraints, and desired quality-efficiency trade-offs.

Attention Type	Time Complexity	Space Complexity
Sparse Attention Mechanisms		
Sparse Transformer	$\mathcal{O}(N\sqrt{N})$ or $\mathcal{O}(N \log N)$	$\mathcal{O}(N)$
Reformer	$\mathcal{O}(N \log N)$	$\mathcal{O}(N)$
Longformer/BigBird	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Linear Attention Mechanisms		
Linformer	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Linear Transformer	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Performer	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Memory Efficient Attention		
Self-attention Does Not Need $\mathcal{O}(N^2)$ Memory	$\mathcal{O}(N^2)$	$\mathcal{O}(\sqrt{N})$
FlashAttention	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
Decoding Latency Optimizations		
Multi-Query Attention (MQA)	$\mathcal{O}(N^2)$	$\mathcal{O}(d_{\text{head}}N)$
Grouped-Query Attention (GQA)	$\mathcal{O}(N^2)$	$\mathcal{O}(Gd_{\text{head}}N)$
Multi-head Latent Attention (MLA)	$\mathcal{O}(N^2)$	$\mathcal{O}(d_{\text{latent}}N)$
Long Sequence Attention		
Transformer-XL	$\mathcal{O}(Nn)$	$\mathcal{O}(1)$
Memorizing Transformers	$\mathcal{O}(N \log M)$	$\mathcal{O}(M)$
Infini-Attention	$\mathcal{O}(Nn)$	$\mathcal{O}(1)$

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL: https://arxiv.org/abs/2305.13245, arXiv:2305.13245.

[2] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. Palm 2 technical report, 2023. URL: https://arxiv.org/abs/2305.10403, arXiv:2305.10403.

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL: https://arxiv.org/abs/2004.05150, arXiv:2004.05150.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL: https://arxiv.org/abs/2005.14165, arXiv:2005.14165.

[5] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading, 2016. URL: https://arxiv.org/abs/1601.06733, arXiv:1601.06733.

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

[7] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022. URL: https://arxiv.org/abs/2009.14794, arXiv:2009.14794.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL: https://arxiv.org/abs/1901.02860, arXiv:1901.02860.

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL: https://arxiv.org/abs/2307.08691, arXiv:2307.08691.

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL: https://arxiv.org/abs/2205.14135, arXiv:2205.14135.

[11] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL: https://arxiv.org/abs/2405.04434, arXiv:2405.04434.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL: https://arxiv.org/abs/1810.04805, arXiv:1810.04805.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang,

Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, and Danny Wyatt ... Zhiyu Ma. The llama 3 herd of models, 2024. URL: https://arxiv.org/abs/2407.21783, arXiv:2407.21783.

[14] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization, 2020. URL: https://arxiv.org/abs/1908.10396, arXiv:1908.10396.

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus, 2017. URL: https://arxiv.org/abs/1702.08734, arXiv:1702.08734.

[16] William B. Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into hilbert space. *Contemporary mathematics*, 26:189–206, 1984. URL: https://api.semanticscholar.org/CorpusID:117819162.

[17] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020. URL: https://arxiv.org/abs/2006.16236, arXiv:2006.16236.

[18] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020. URL: https://arxiv.org/abs/2001.04451, arXiv:2001.04451.

[19] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[20] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding, 2017. URL: https://arxiv.org/abs/1703.03130, arXiv:1703.03130.

[21] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention, 2024. URL: https://arxiv.org/abs/2404.07143, arXiv:2404.07143.

[22] Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference, 2016. URL: https://arxiv.org/abs/1606.01933, arXiv:1606.01933.

[23] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference, 2022. URL: https://arxiv.org/abs/2211.05102, arXiv:2211.05102.

[24] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022. URL: https://arxiv.org/abs/2112.05682, arXiv:2112.05682.

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

[26] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992. doi:10.1162/neco.1992.4.1.131.

[27] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL: https://arxiv.org/abs/2407.08608, arXiv:2407.08608.

[28] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. URL: https://arxiv.org/abs/1911.02150, arXiv:1911.02150.

[29] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL: https://arxiv.org/abs/2104.09864, arXiv:2104.09864.

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

[31] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL: https://arxiv.org/abs/2307.09288, arXiv:2307.09288.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL: https://arxiv.org/abs/1706.03762, arXiv:1706.03762.

[33] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020. URL: https://arxiv.org/abs/2006.04768, arXiv:2006.04768.

[34] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers, 2022. URL: https://arxiv.org/abs/2203.08913, arXiv:2203.08913.

[35] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021. URL: https://arxiv.org/abs/2007.14062, arXiv:2007.14062.