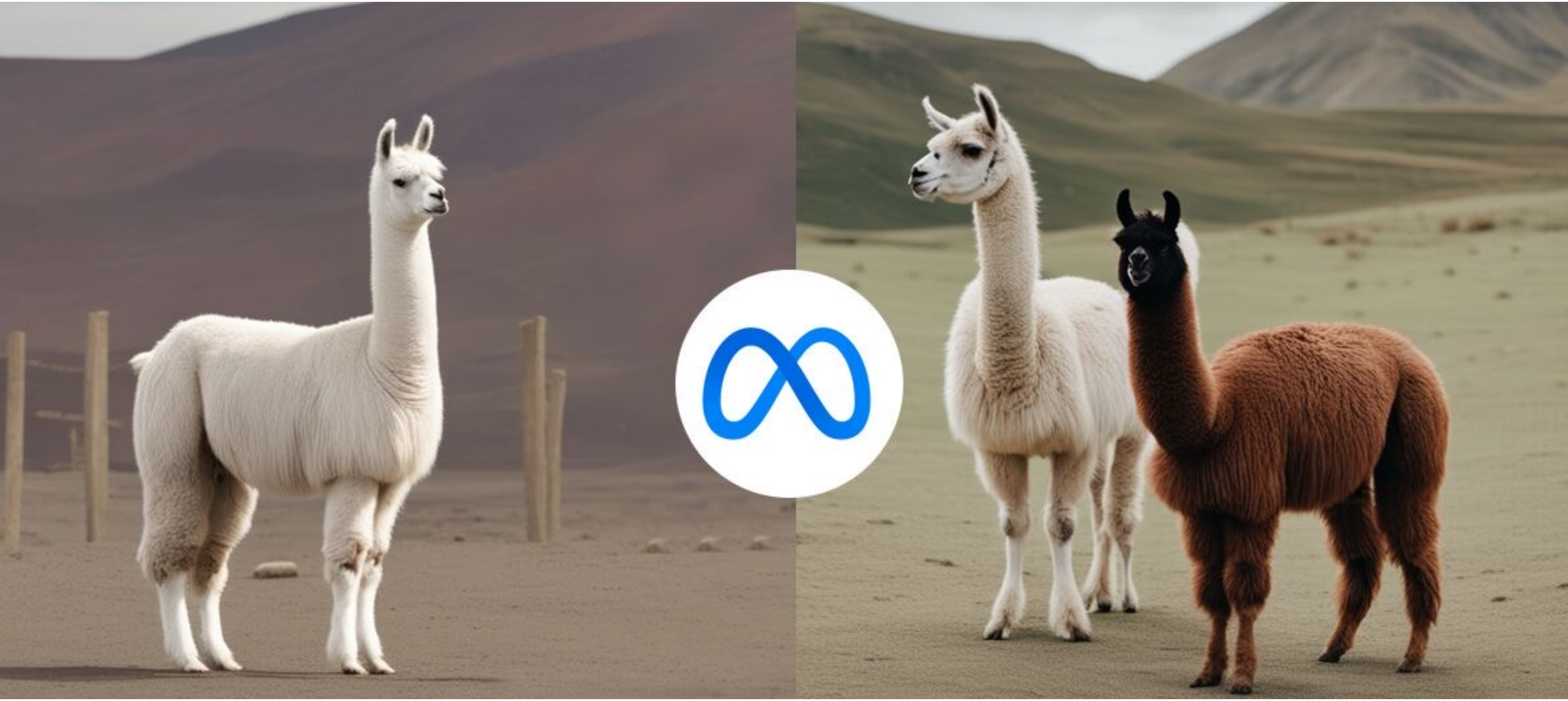
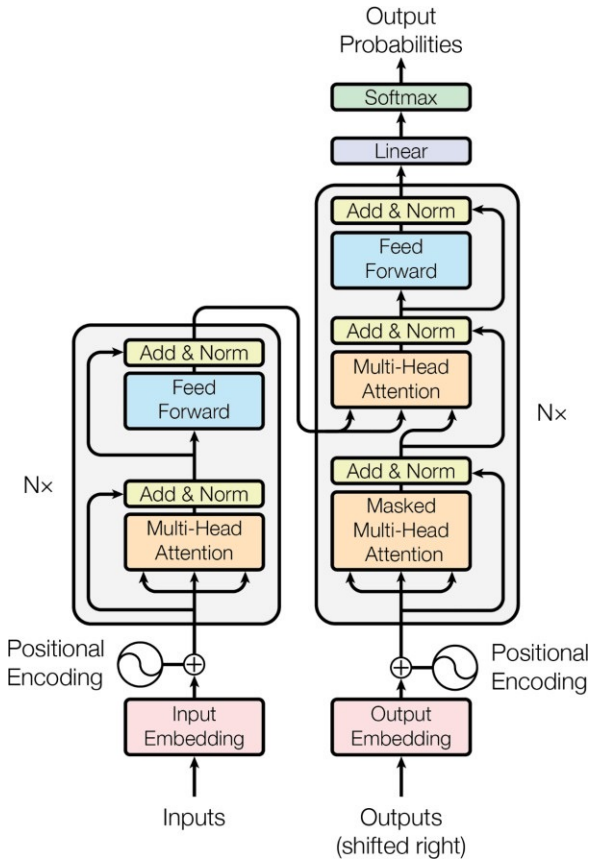
Coding LLaMA2 from scratch



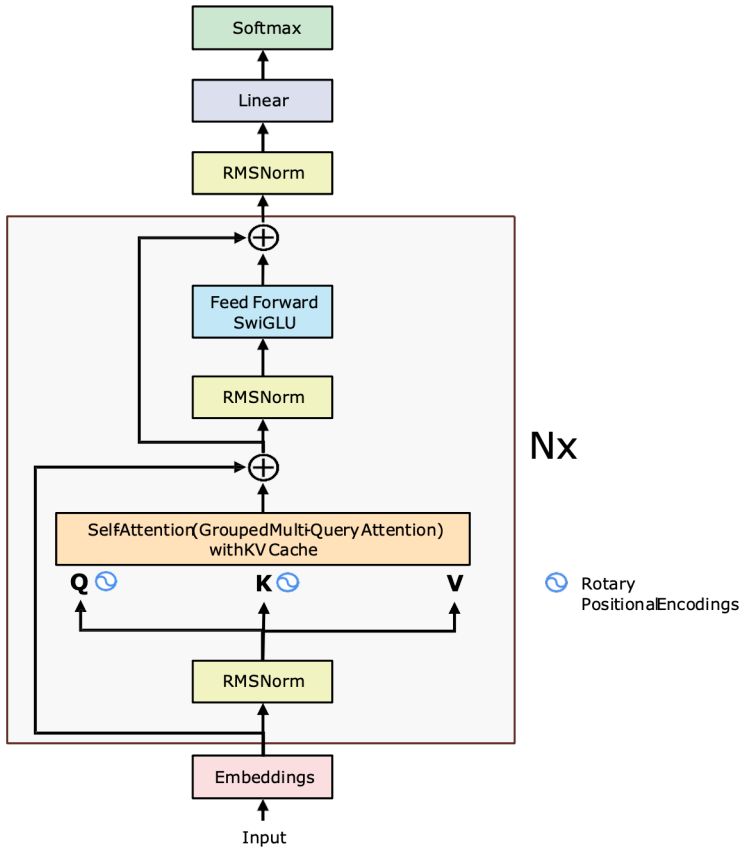
Curriculum

- Architectural differences between Transformer and LLaMA
- RMS Normalization (with review of Layer Normalization)
- Rotary Positional Embeddings
- KV-Cache
- Multi-Query Attention
- Grouped Multi-Query Attention
- SwiGLU Activation Function

Transformer vs LLaMa



Transformer
("Attention is all you need")



LLaMA

LLaMA 1

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

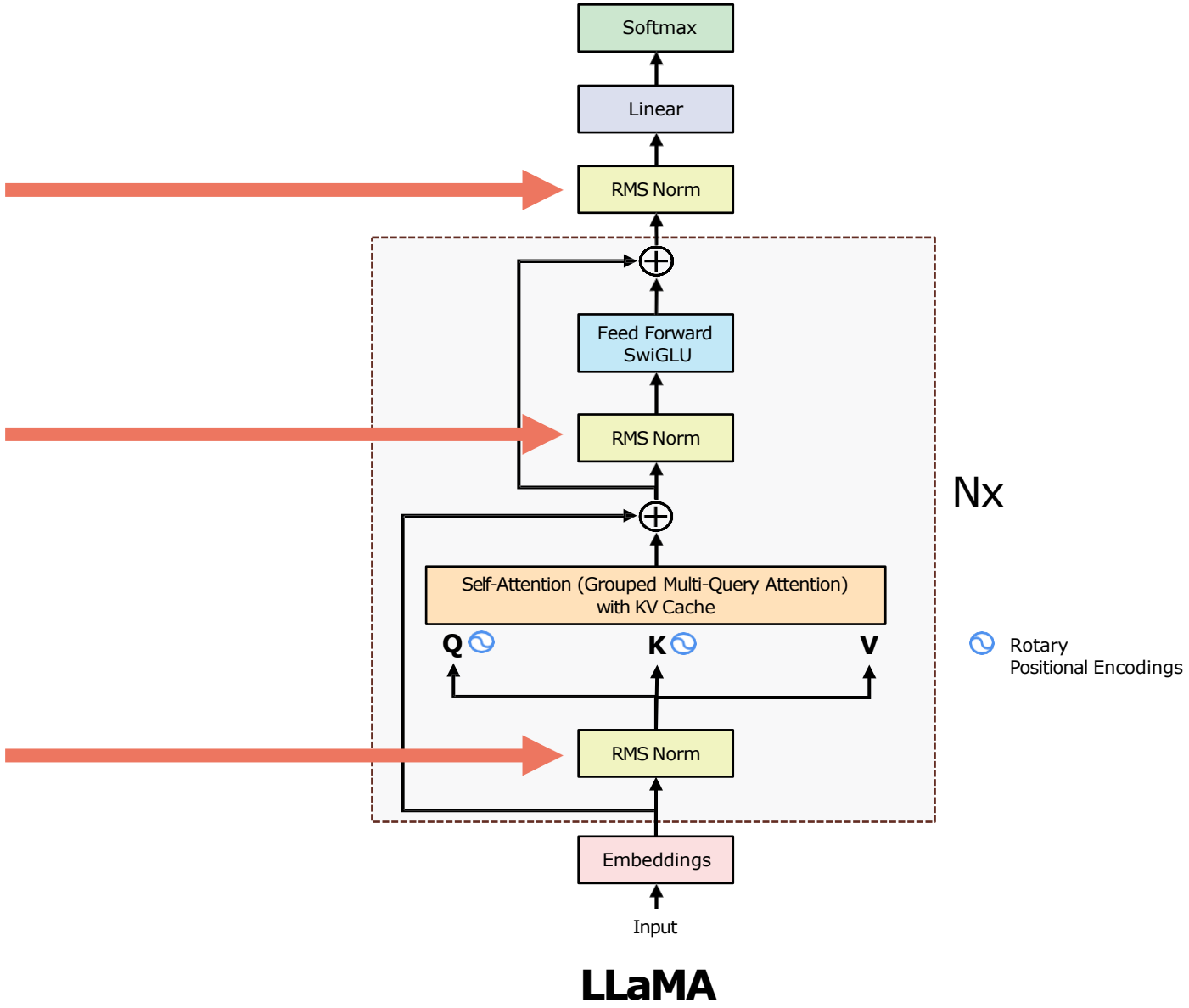
Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

LLaMA 2

	Training Data	Params	Context Length	GQA	Tokens	LR
LLAMA 1	*See Touvron et al. (2023)*	7B	2k	✗	1.0T	3.0×10^{-4}
		13B	2k	✗	1.0T	3.0×10^{-4}
		33B	2k	✗	1.4T	1.5×10^{-4}
		65B	2k	✗	1.4T	1.5×10^{-4}
LLAMA 2	*A new mix of publicly available online data*	7B	4k	✗	2.0T	3.0×10^{-4}
		13B	4k	✗	2.0T	3.0×10^{-4}
		34B	4k	✓	2.0T	1.5×10^{-4}
		70B	4k	✓	2.0T	1.5×10^{-4}

Table 1: LLAMA 2 family of models. Token counts refer to pretraining data only. All models are trained with a global batch-size of 4M tokens. Bigger models — 34B and 70B — use Grouped-Query Attention (GQA) for improved inference scalability.

Normalization



Root Mean Square Normalization

Root Mean Square Layer Normalization

Biao Zhang[1] **Rico Sennrich**[2,1]
[1]School of Informatics, University of Edinburgh
[2]Institute of Computational Linguistics, University of Zurich
B.Zhang@ed.ac.uk, sennrich@cl.uzh.ch

4 RMSNorm

A well-known explanation of the success of LayerNorm is its re-centering and re-scaling invariance property. The former enables the model to be insensitive to shift noises on both inputs and weights, and the latter keeps the output representations intact when both inputs and weights are randomly scaled. In this paper, we hypothesize that the re-scaling invariance is the reason for success of LayerNorm, rather than re-centering invariance.

We propose RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic:

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}. \tag{4}$$

Intuitively, RMSNorm simplifies LayerNorm by totally removing the mean statistic in Eq. (3) at the cost of sacrificing the invariance that mean normalization affords. When the mean of summed inputs is zero, RMSNorm is exactly equal to LayerNorm. Although RMSNorm does not re-center

Just like Layer Normalization, we also have a learnable parameter **gamma** (**g** in the formula on the left) that is multiplied by the normalized values.

Why RMSNorm

- Requires less computation compared to Layer Normalization.
- It works well in practice.

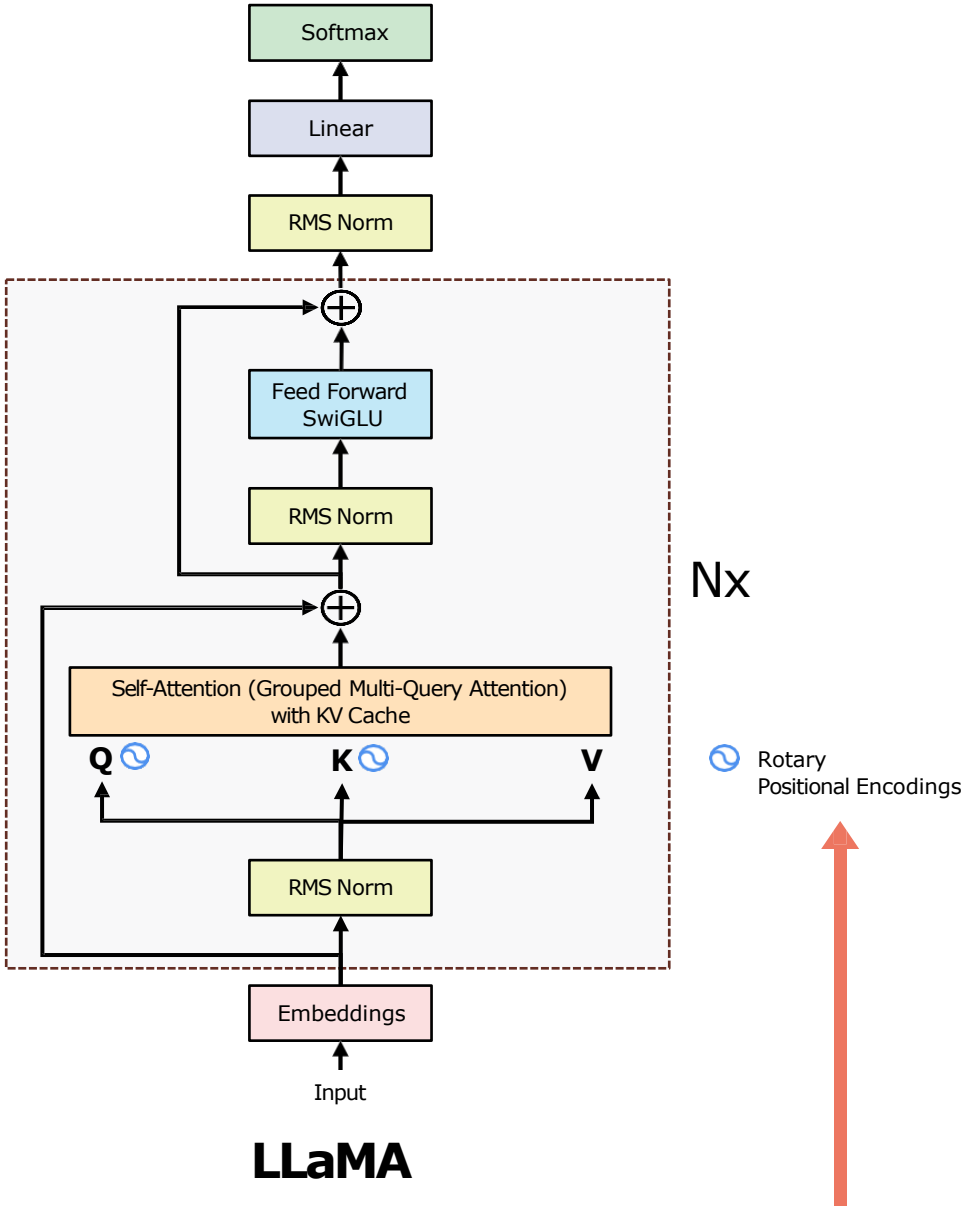
RMSNorm

```python
class RMSNorm(nn.Cell):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        # The gamma parameter
        self.weight = Parameter(ops.ones(dim))

    def _norm(self, x: mindspore.Tensor):
        # (B, Seq_Len, Dim) * (B, Seq_Len, 1) = (B, Seq_Len, Dim)
        # rsqrt: 1 / sqrt(x)
        return x * ops.rsqrt(x.pow(2).mean(-1, keep_dims=True) + self.eps)

    def construct(self, x: mindspore.Tensor):
        # (Dim) * (B, Seq_Len, Dim) = (B, Seq_Len, Dim)
        return self.weight * self._norm(x.float()).astype(x.dtype)
```

Positional Encodings



Introduction of Positional Embedding

位置信息的表示有很多种，如

- **absolute positional embeddings**:
  - 原理：对于第k个位置的向量x_k，添加位置向量p_k（仅依赖于位置编号k），得到x_k+p_k
  - 举例/应用模型：sinusoidal positional embedding（Transformer）、learned absolute positional embedding（BERT/RoBERTa/GPT）
- **relative positional embeddings:**
  - 原理：对于第m个和第n个位置的向量x_m、x_n，将相对位置m-n的信息添加到self-attention matrix中
  - 举例/应用模型：T5
- **rotary positional embeddings**
  - 原理：使用旋转矩阵对绝对位置进行编码，并同时在自注意力公式中引入了显式的相对位置依赖。
  - 举例/应用模型：PaLM/GPT-Neo/GPT-J/LLaMa1&2/ChatGLM1&2

Rotary Position Embeddings

ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING

Jianlin Su
Zhuiyi Technology Co., Ltd.
Shenzhen
bojonesu@wezhuiyi.com

Yu Lu
Zhuiyi Technology Co., Ltd.
Shenzhen
julianlu@wezhuiyi.com

Shengfeng Pan
Zhuiyi Technology Co., Ltd.
Shenzhen
nickpan@wezhuiyi.com

Ahmed Murtadha
Zhuiyi Technology Co., Ltd.
Shenzhen
mengjiayi@wezhuiyi.com

Bo Wen
Zhuiyi Technology Co., Ltd.
Shenzhen
brucewen@wezhuiyi.com

Yunfeng Liu
Zhuiyi Technology Co., Ltd.
Shenzhen
glenliu@wezhuiyi.com

Rotary Position Embeddings: the inner product

- The dot product used in the attention mechanism is a type of inner product, which can be through of as a generalization of the dot product.
- Can we find an inner product over the two vectors q (query) and k (key) used in the attention mechanism that only depends on the two vectors and the relative distance of the token they represent?

Under the case of $d = 2$, we consider two-word embedding vectors $\boldsymbol{x}_q$, $\boldsymbol{x}_k$ corresponds to query and key and their position m and n, respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_q, m), \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_k, n), \end{aligned} \tag{20}$$

where the subscripts of $\boldsymbol{q}_m$ and $\boldsymbol{k}_n$ indicate the encoded positions information. Assume that there exists a function g that defines the inner product between vectors produced by $f_{\{q,k\}}$:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n) \rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, n - m), \tag{21}$$

Rotary Position Embeddings: the inner product

- We can define a function g like the following that only depends on the on the two embeddings vector q and k and their relative distance

$$f_q(\boldsymbol{x}_m, m) = (\boldsymbol{W}_q \boldsymbol{x}_m) e^{im\theta}$$
$$f_k(\boldsymbol{x}_n, n) = (\boldsymbol{W}_k \boldsymbol{x}_n) e^{in\theta}$$
$$g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n) = \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}]$$

Conjugate of the complex number

- Using Euler's formula, we can write it into its matrix form.

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

Rotation matrix in a 2d space, hence the name rotary position embeddings

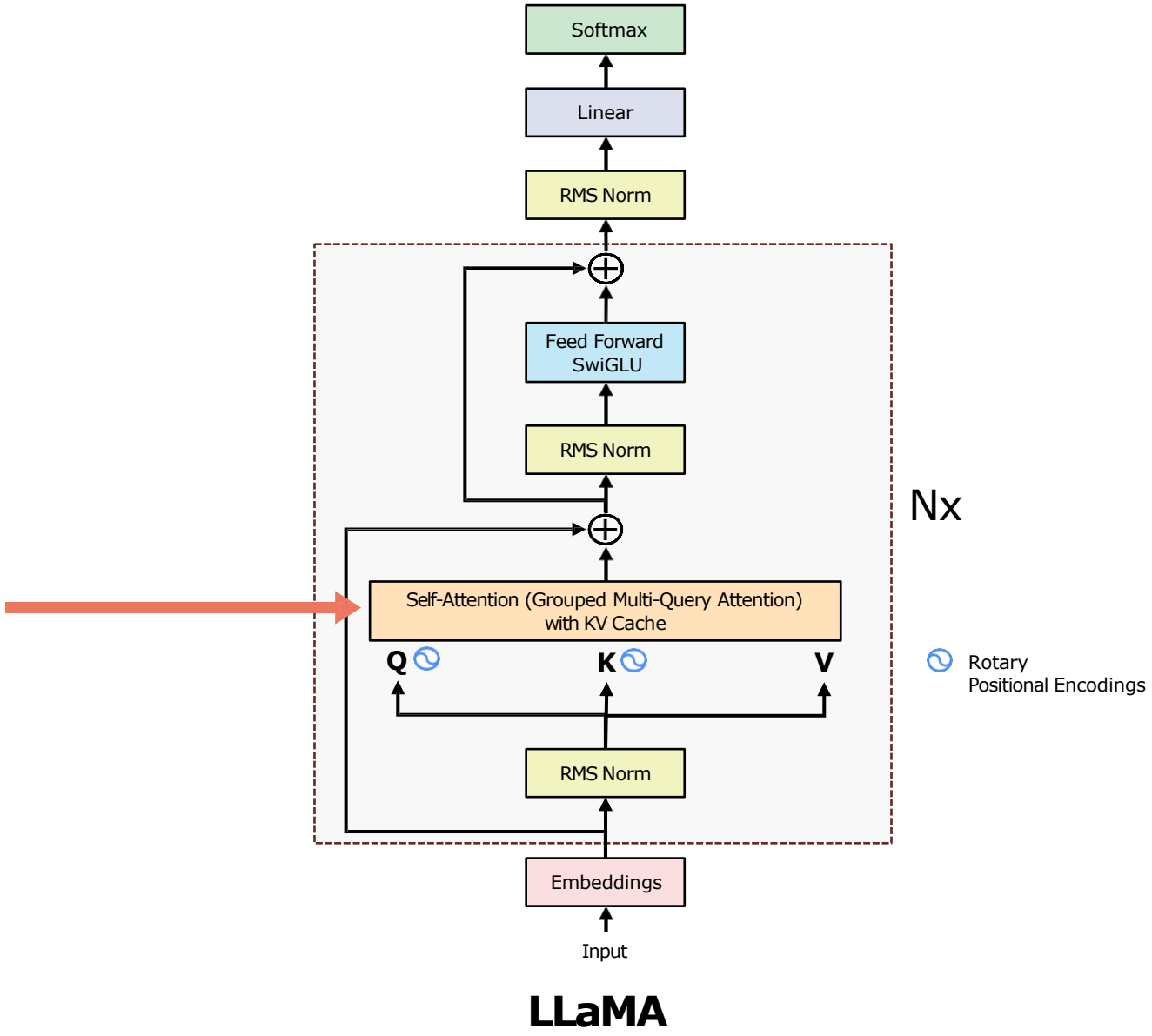
$$f_q(\boldsymbol{x}_m, m) = (\boldsymbol{W}_q \boldsymbol{x}_m) e^{im\theta}$$

$$f_k(\boldsymbol{x}_n, n) = (\boldsymbol{W}_k \boldsymbol{x}_n) e^{in\theta}$$

$$g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n) = \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}]$$

```python
def apply_rotary_embeddings(x: mindspore.Tensor, freqs_complex: mindspore.Tensor):
    # Separate the last dimension pairs of two values, representing the real and imaginary
    # Two consecutive values will become a single complex number
    # (B, Seq_Len, H, Head_Dim) -> (B, Seq_Len, H, Head_Dim/2)
    x_complex = view_as_complex(x.float().reshape(*x.shape[:-1], -1, 2))
    # Reshape the freqs_complex tensor to match the shape of the x_complex tensor. So we n
    # (Seq_Len, Head_Dim/2) --> (1, Seq_Len, 1, Head_Dim/2)
    freqs_complex = freqs_complex.unsqueeze(0).unsqueeze(2)
    # Multiply each complex number in the x_complex tensor by the corresponding complex nu
    # Which results in the rotation of the complex number as shown in the Figure 1 of the
    # (B, Seq_Len, H, Head_Dim/2) * (1, Seq_Len, 1, Head_Dim/2) = (B, Seq_Len, H, Head_Dim
    x_rotated = x_complex * freqs_complex
    # Convert the complex number back to the real number
    # (B, Seq_Len, H, Head_Dim/2) -> (B, Seq_Len, H, Head_Dim/2, 2)
    x_out = ops.view_as_real(x_rotated)
    # (B, Seq_Len, H, Head_Dim/2, 2) -> (B, Seq_Len, H, Head_Dim)
    x_out = x_out.reshape(*x.shape)
    return x_out.astype(x.dtype)
```

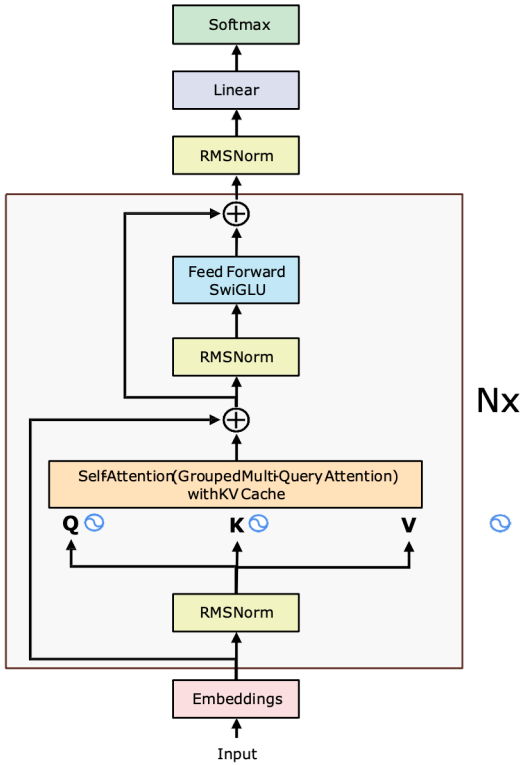
Self-Attention



Next Token Prediction Task

Target: Love that can quickly seize the gentle heart [EOS]

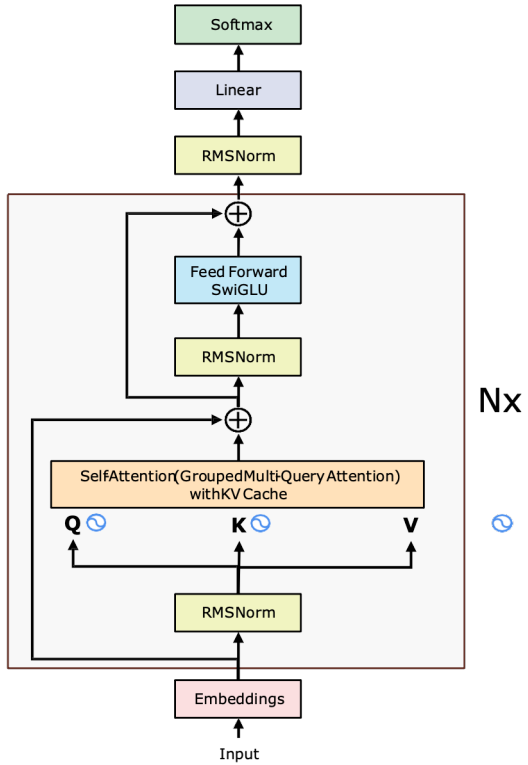
Training



Input: [SOS] Love that can quickly seize the gentle heart

Next Token Prediction Task

Output: **Love**



Inference

T = 1

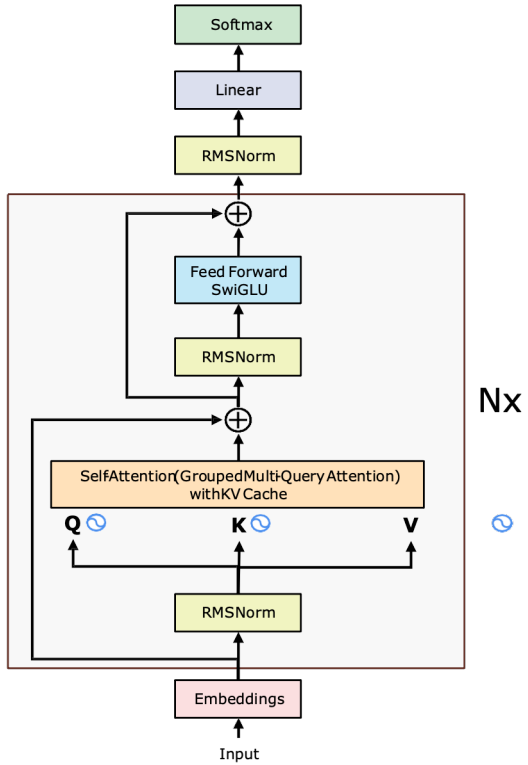
Input: [SOS]

Next Token Prediction Task

Output: Love **that**

Inference

T = 2

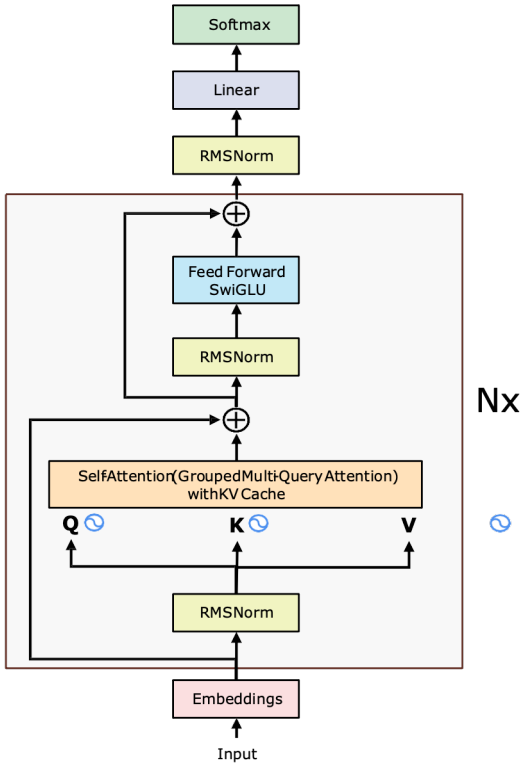


Input: [SOS] love

Next Token Prediction Task

Output: Love that **can**

Inference
T = 3



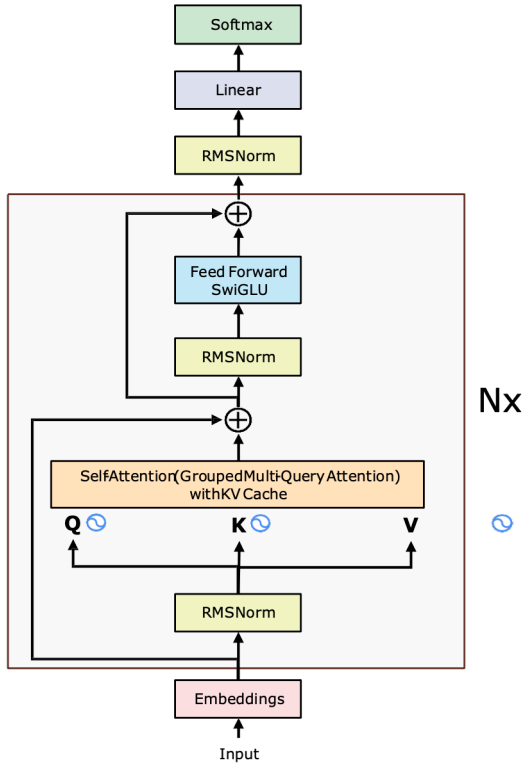
Input: [SOS] Love that

Next Token Prediction Task

Output: Love that can **quickly**

Inference

T = 4



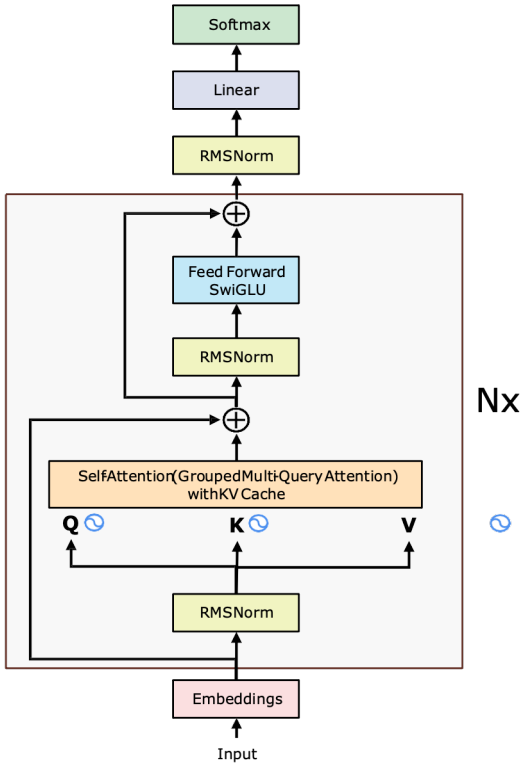
Input: [SOS] Love that can

Next Token Prediction Task

Output: Love that can quickly **seize**

Inference

T = 5



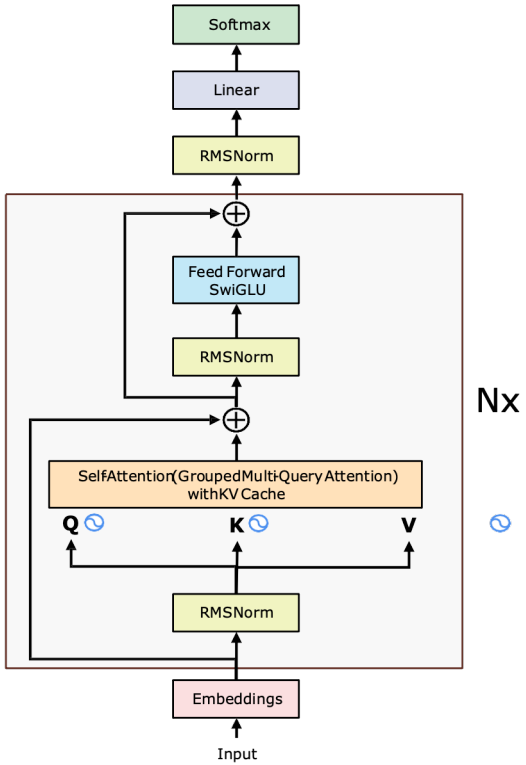
Input: [SOS] Love that can quickly

Next Token Prediction Task

Output: Love that can quickly seize **the**

Inference

T = 6



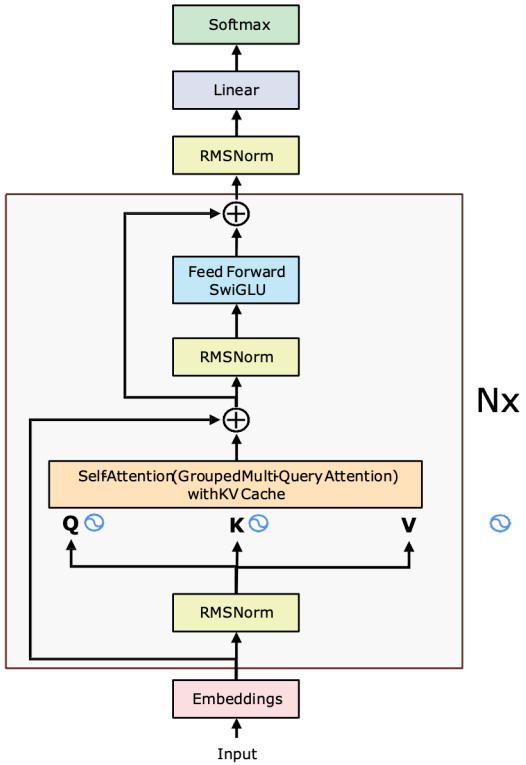
Input: [SOS] Love that can quickly seize

Next Token Prediction Task

Output: Love that can quickly seize the **gentle**

Inference

T = 7



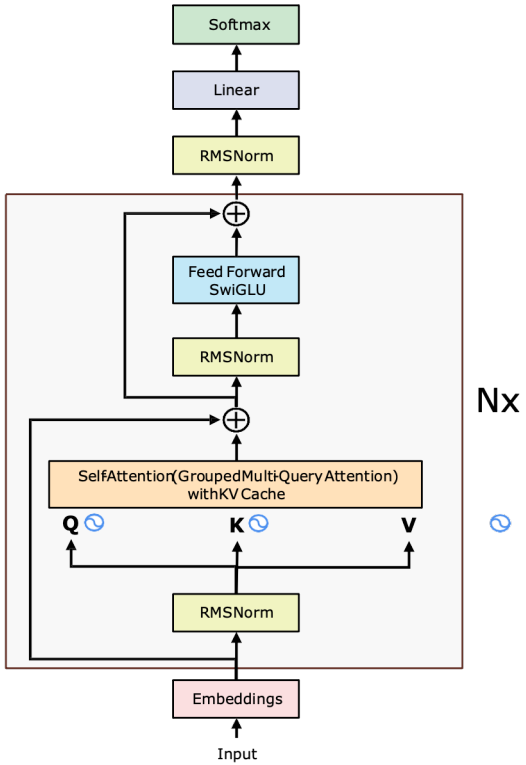
Input: [SOS] Love that can quickly seize the

Next Token Prediction Task

Output: Love that can quickly seize the gentle **heart**

Inference

T = 8

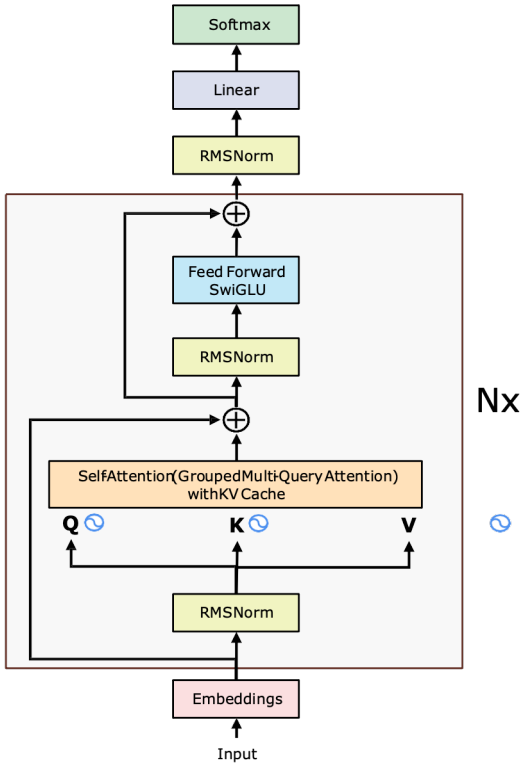


Input: [SOS] Love that can quickly seize the gentle

Next Token Prediction Task

Output: Love that can quickly seize the gentle heart **[EOS]**

Inference
T = 9

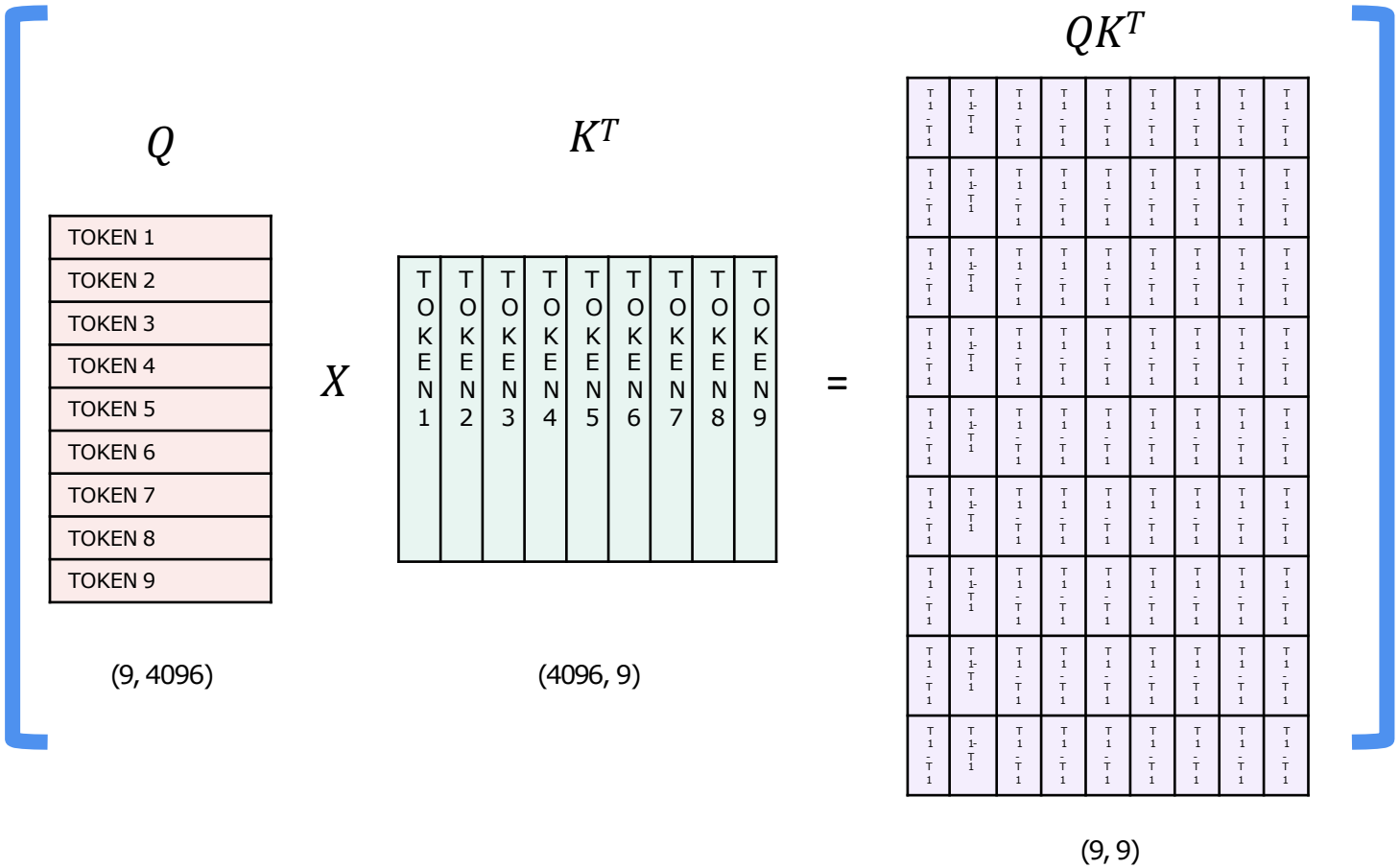


Input: [SOS] Love that can quickly seize the gentle heart

KV Cache

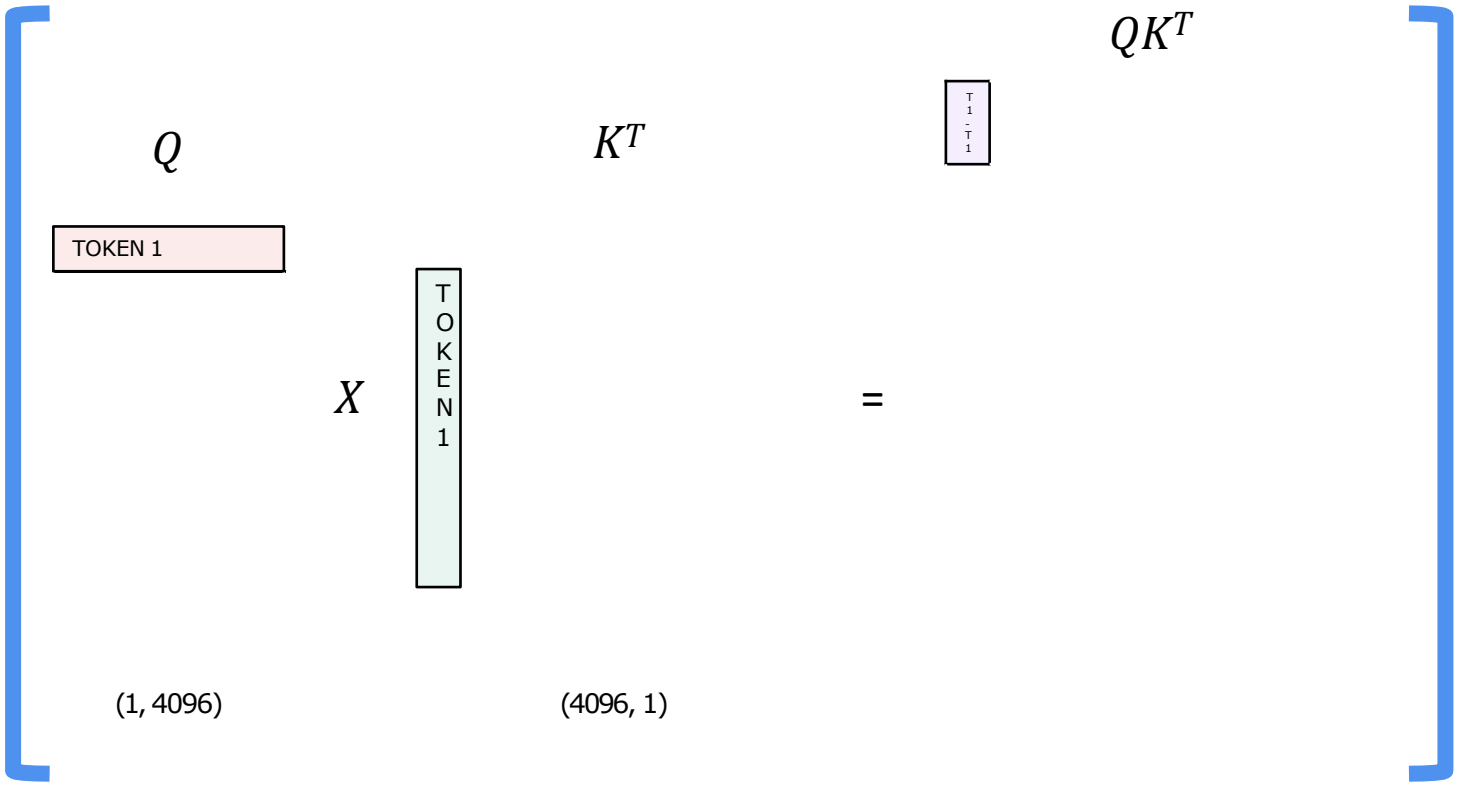
- At every step of the inference, we are only interested in **the last token** output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the “prompt”).
- Is there a way to make the model do less computation on the token it has already seen **during inference**? YES! The solution is the **KV cache**!

Self-Attention during Next Token Prediction Task



$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

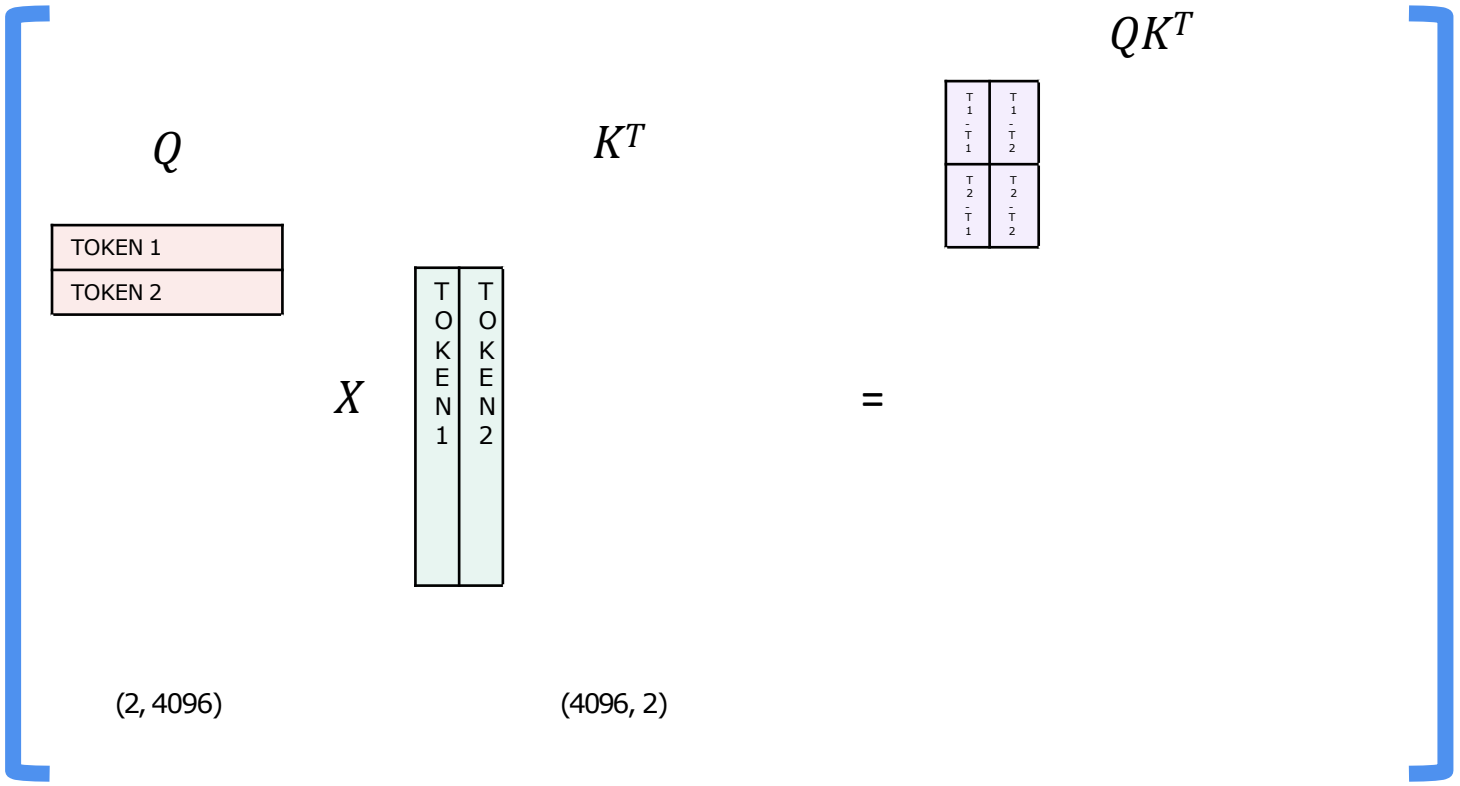
Self-Attention during Next Token Prediction Task



$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Inference
T = 1

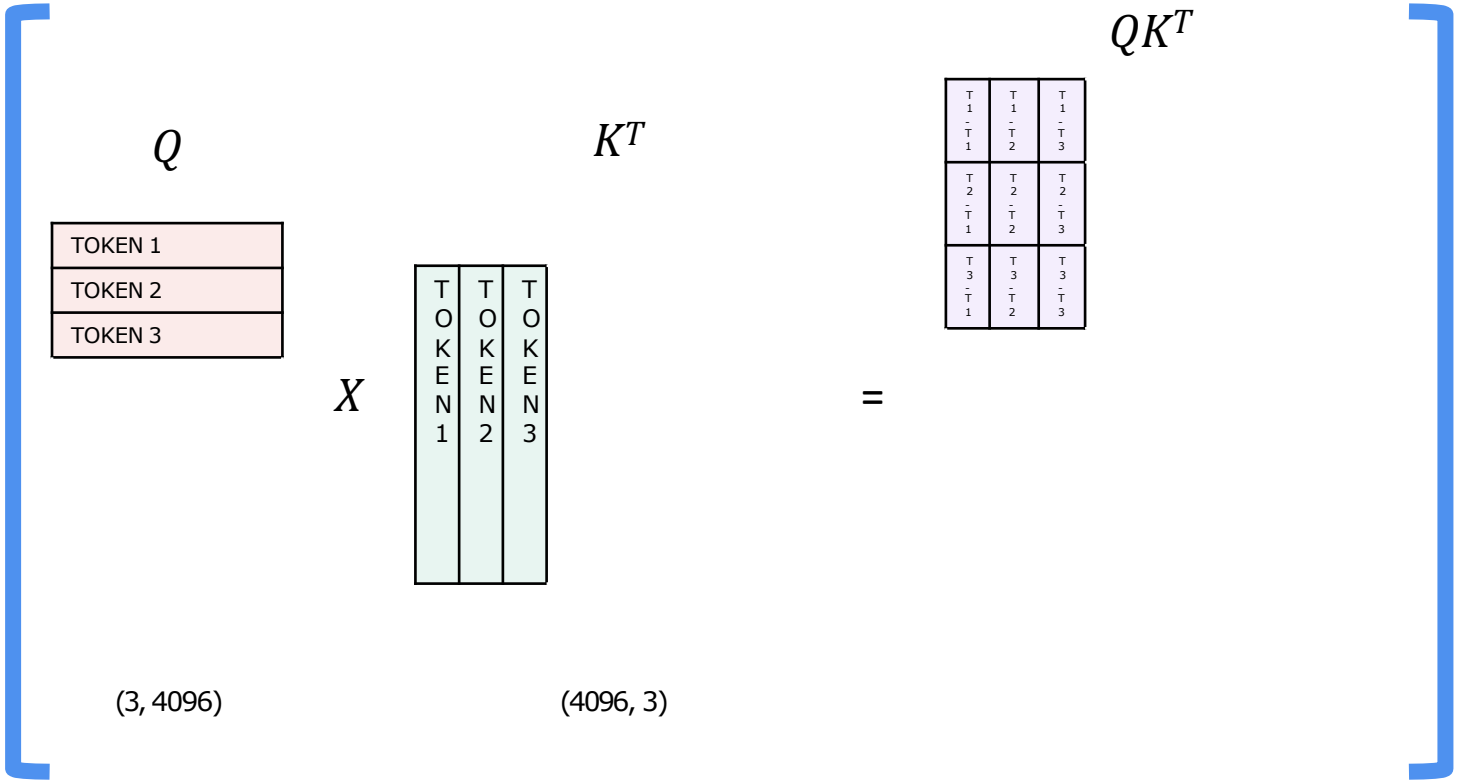
Self-Attention during Next Token Prediction Task

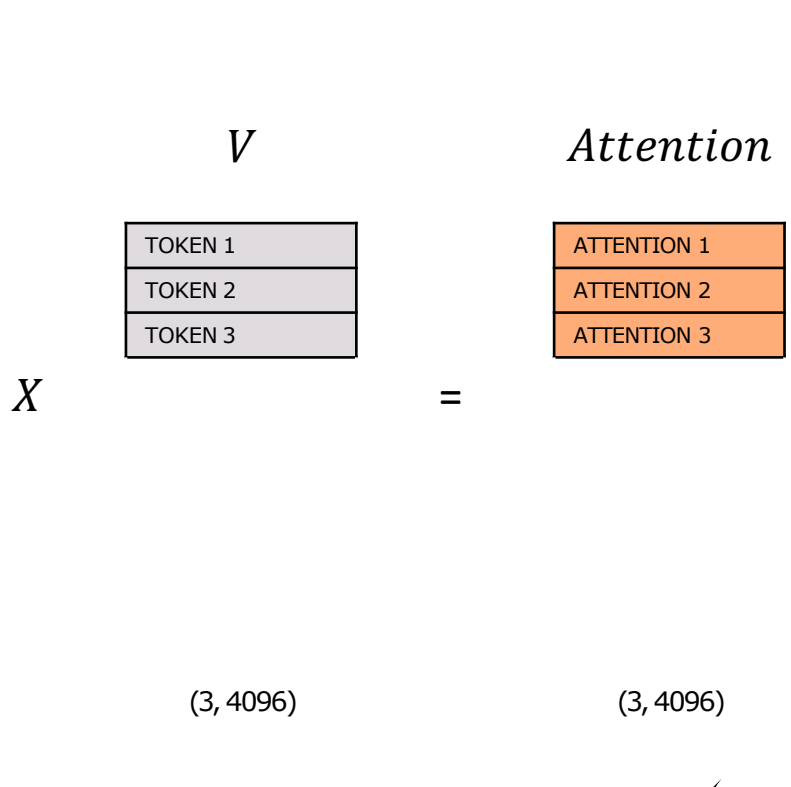


$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Inference
T = 2

Self-Attention during Next Token Prediction Task

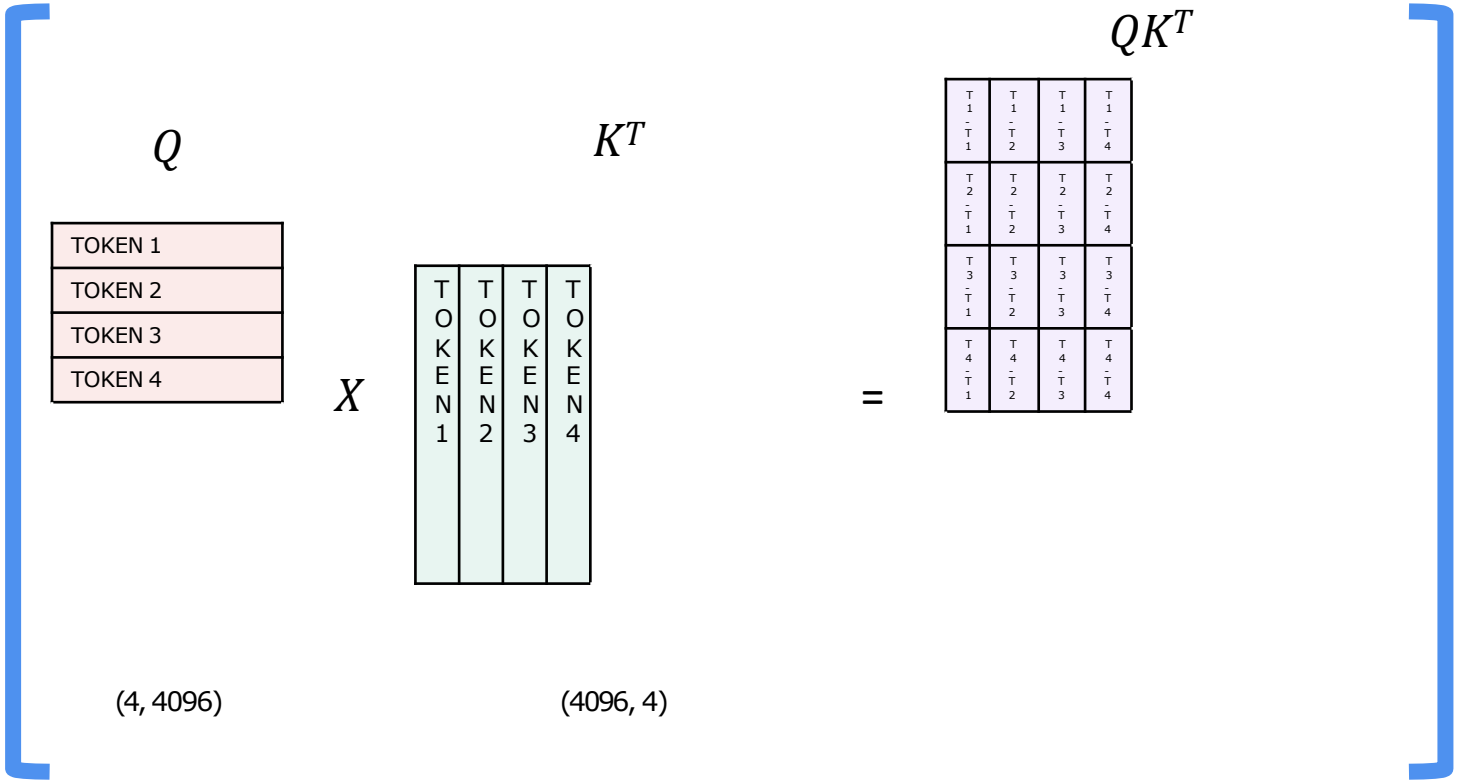




Inference
T = 3

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention during Next Token Prediction Task



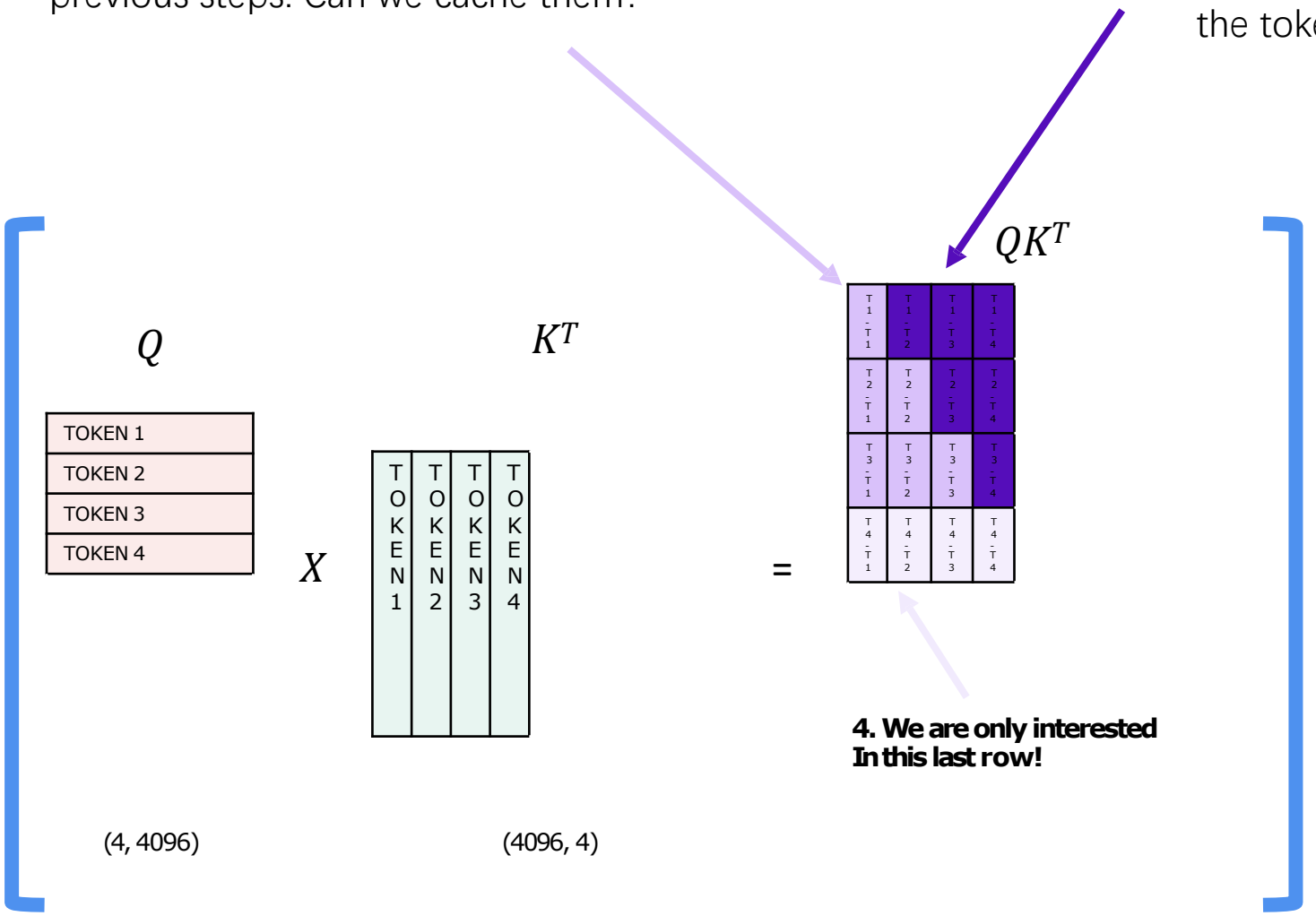
Inference
T = 4

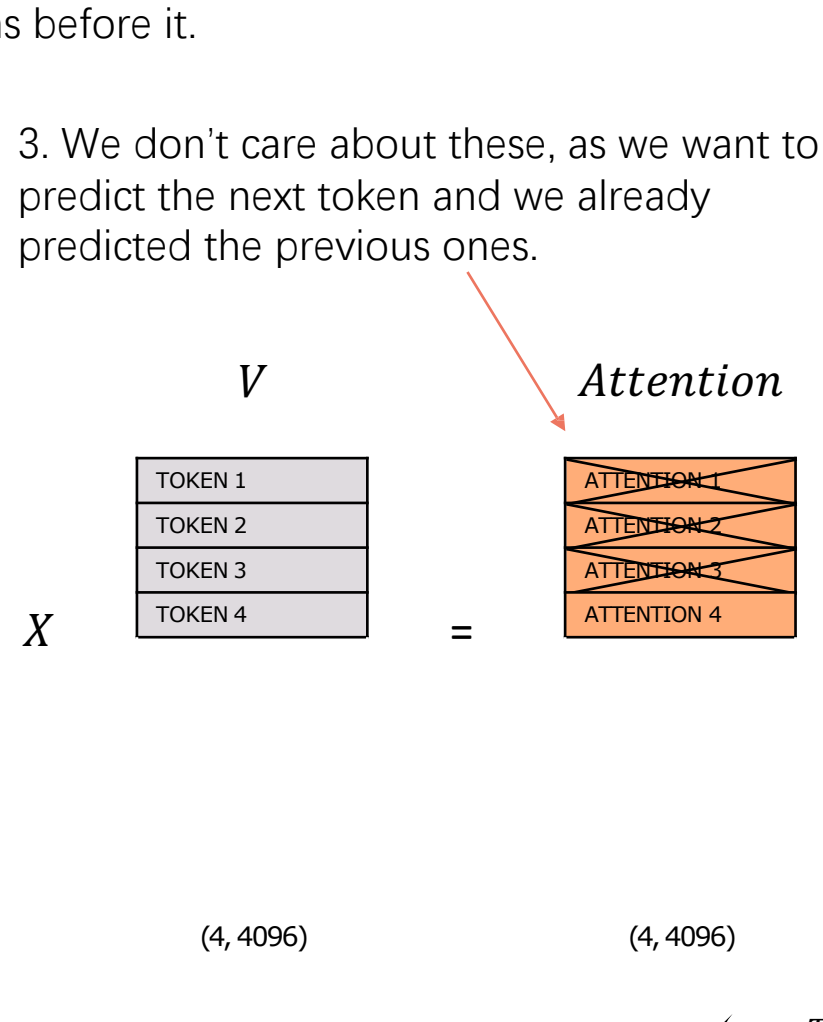
$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

1. We already computed these dot products In the previous steps. Can we cache them?

2. Since the model is causal, we don't care about the attention of a token with its successors, but only with the tokens before it.

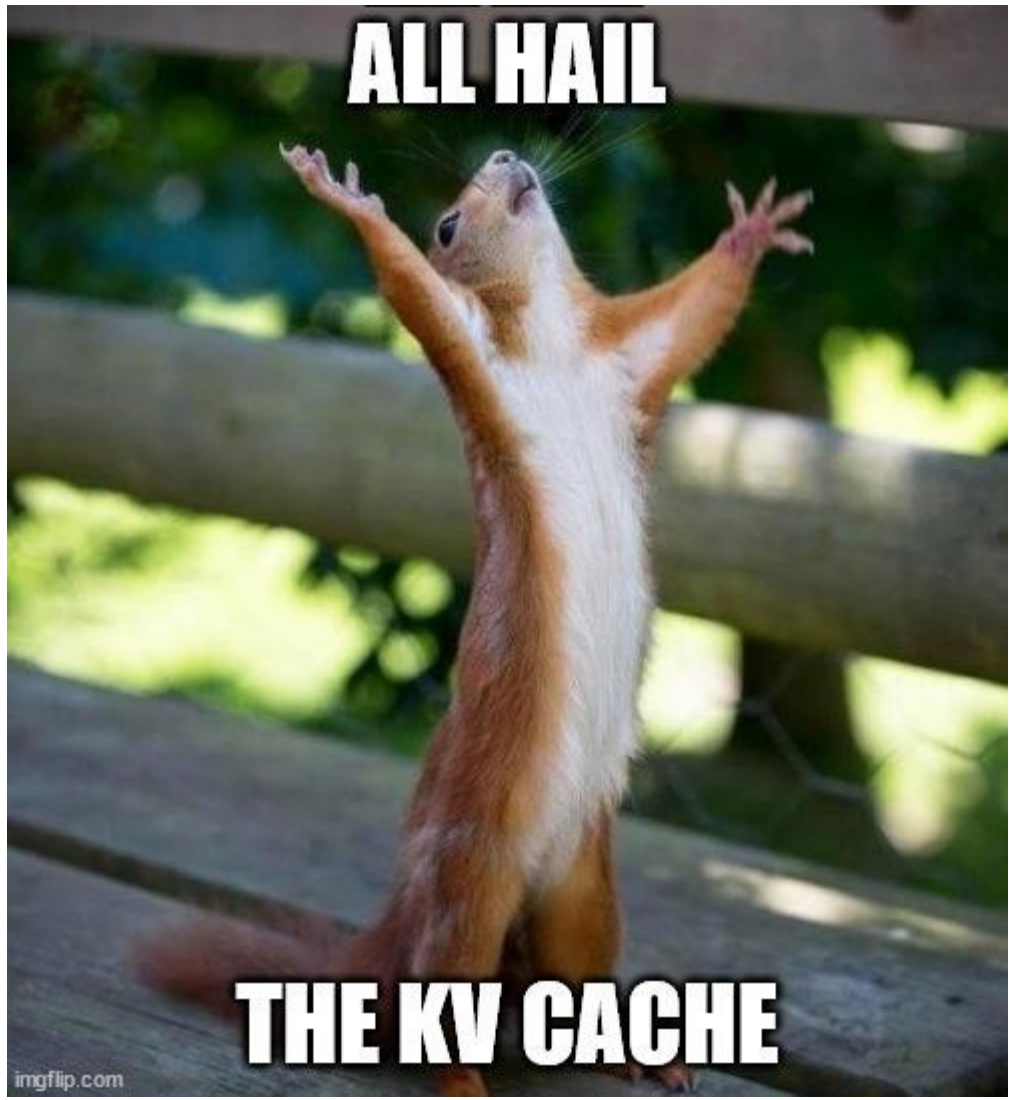
3. We don't care about these, as we want to predict the next token and we already predicted the previous ones.



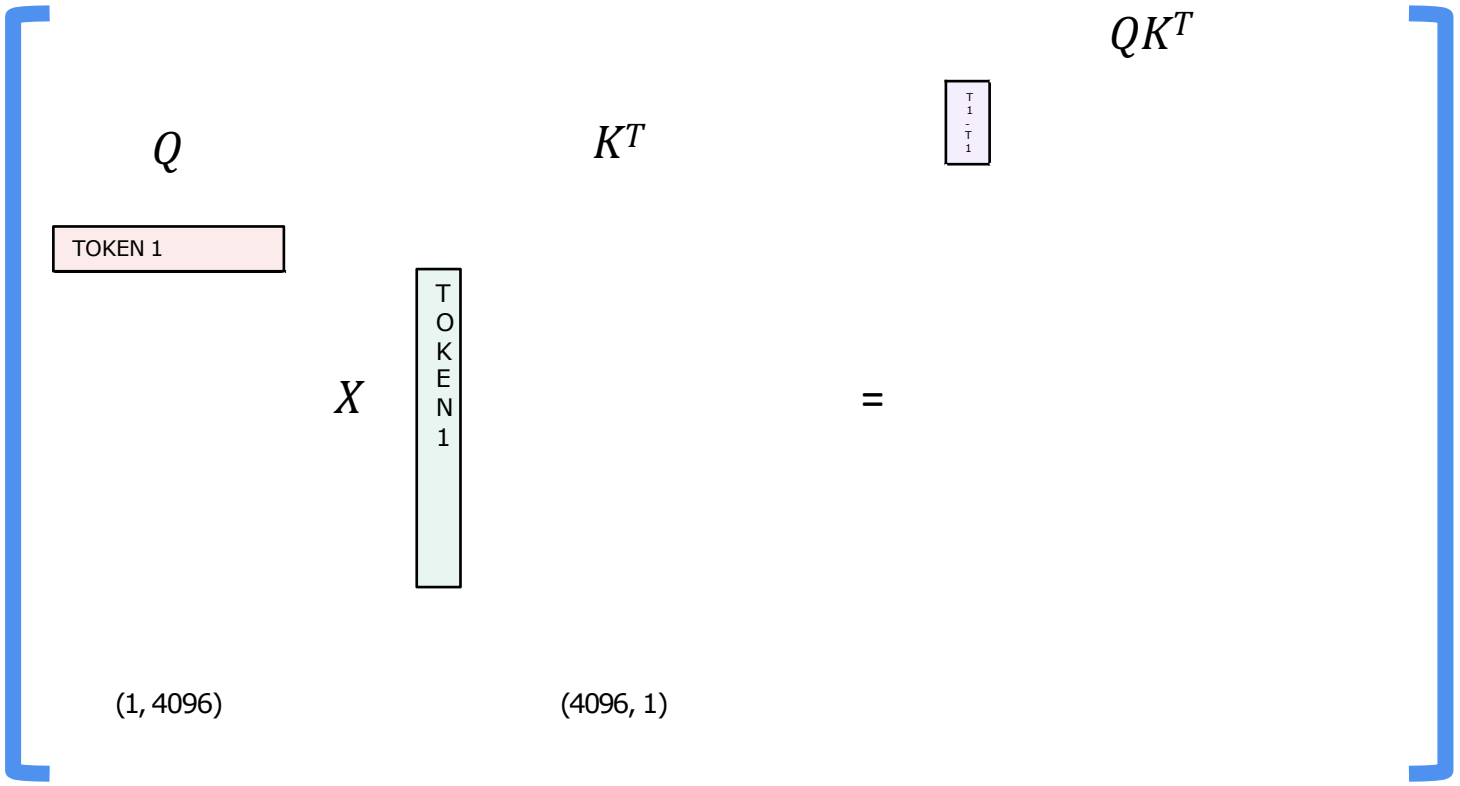


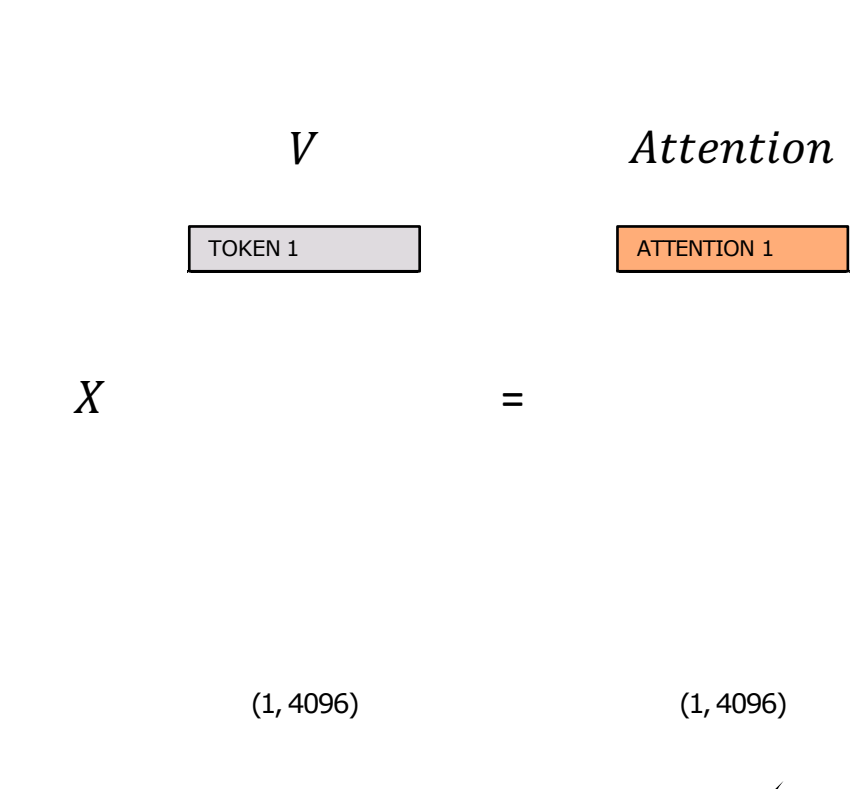
Inference
T = 4

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$


ALL HAIL
THE KV CACHE
imgflip.com

Self-Attention with KV-Cache

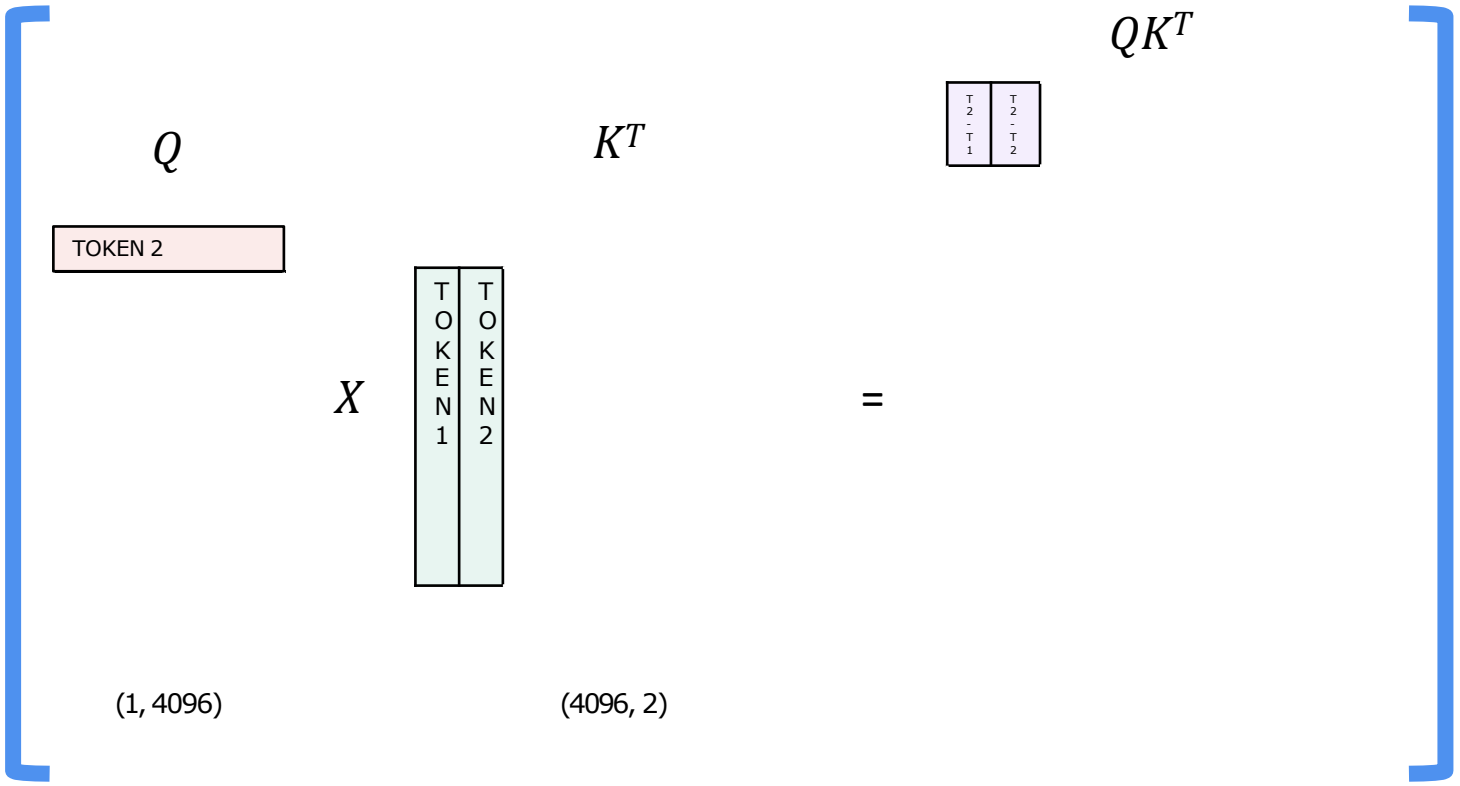


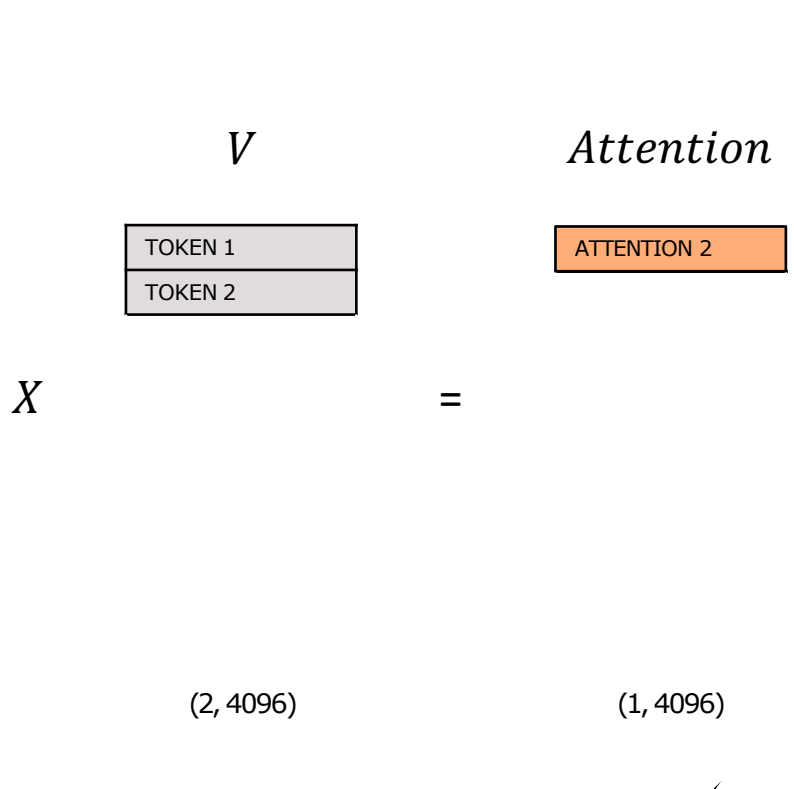


Inference
T = 1

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention with KV-Cache

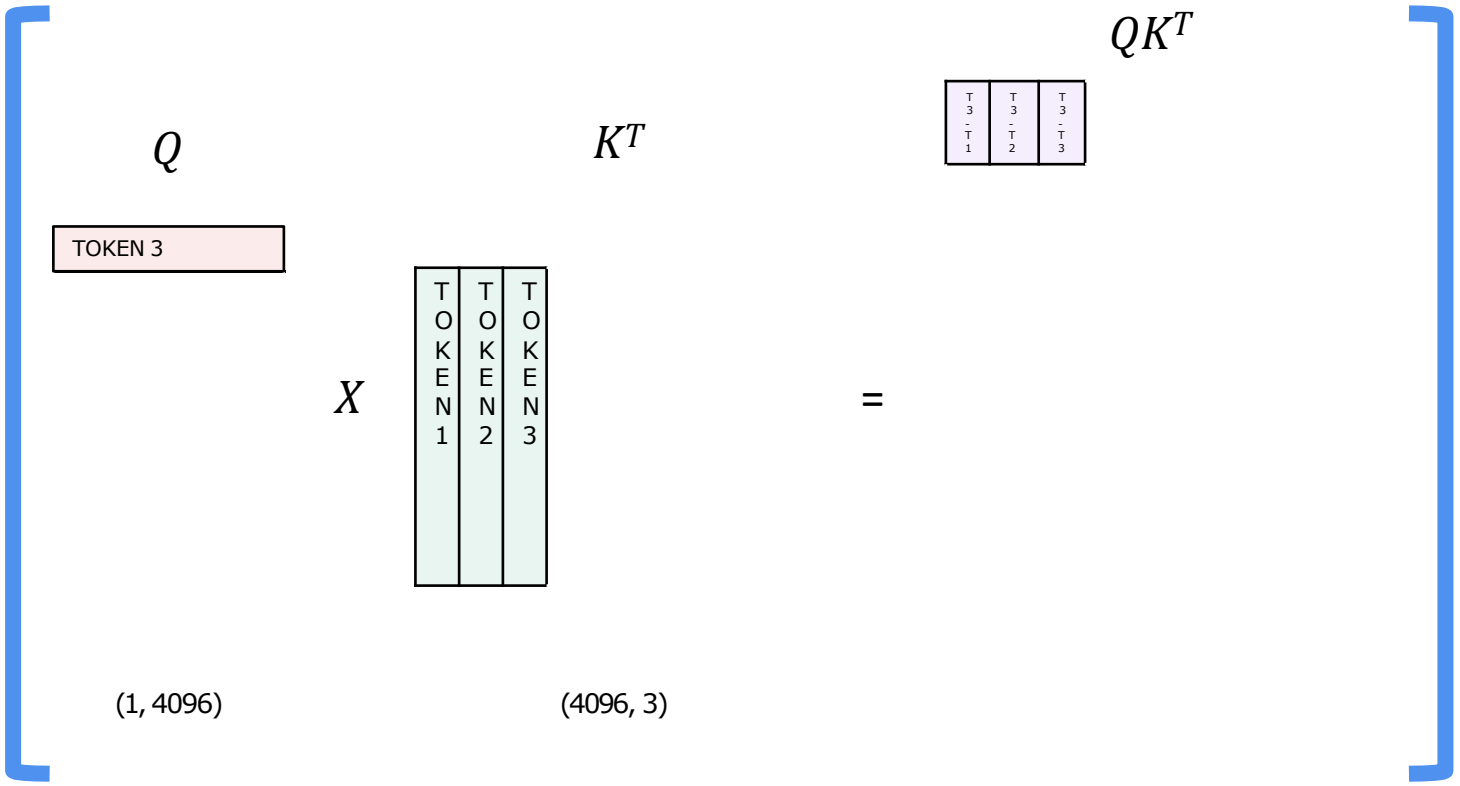


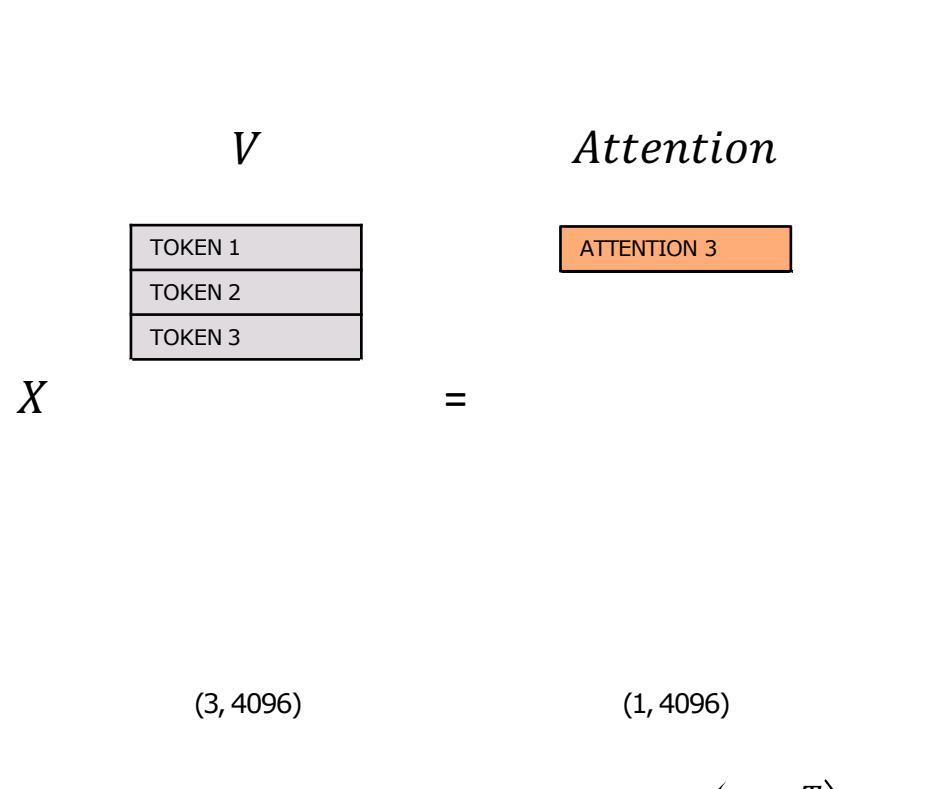


$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Inference
T = 2

Self-Attention with KV-Cache

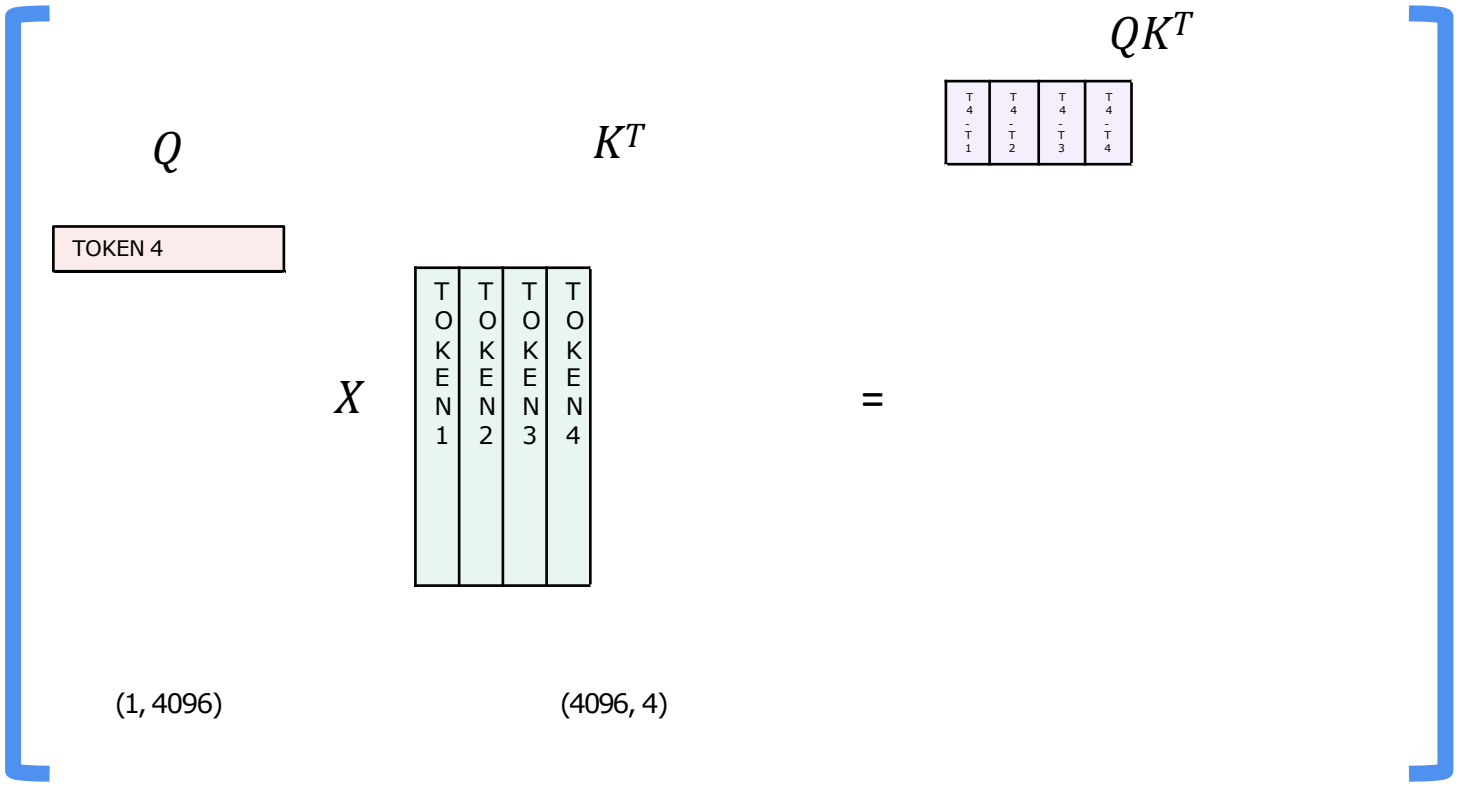


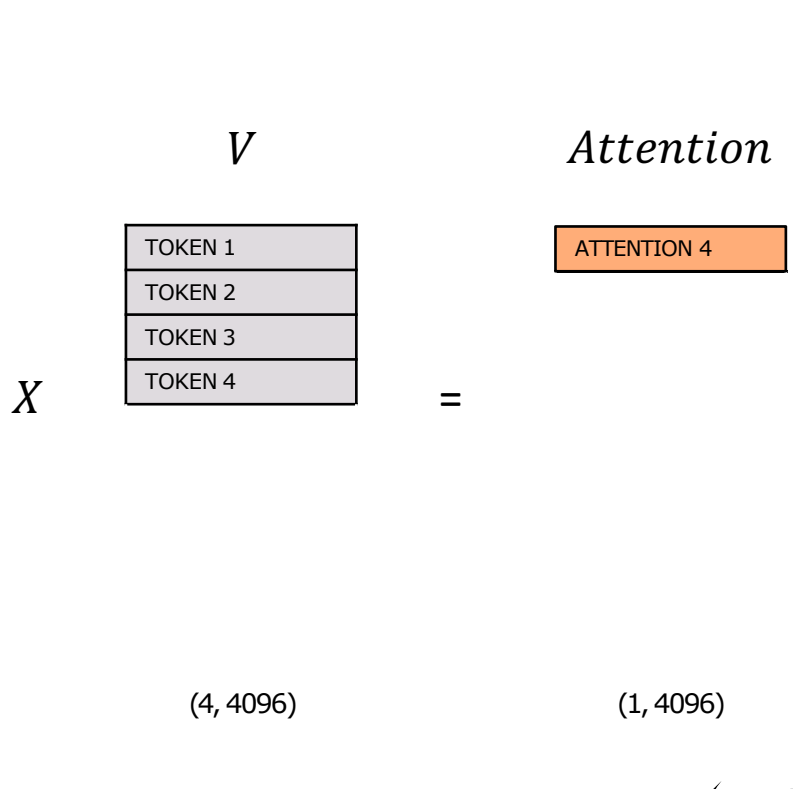


$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Inference
T = 3

Self-Attention with KV-Cache

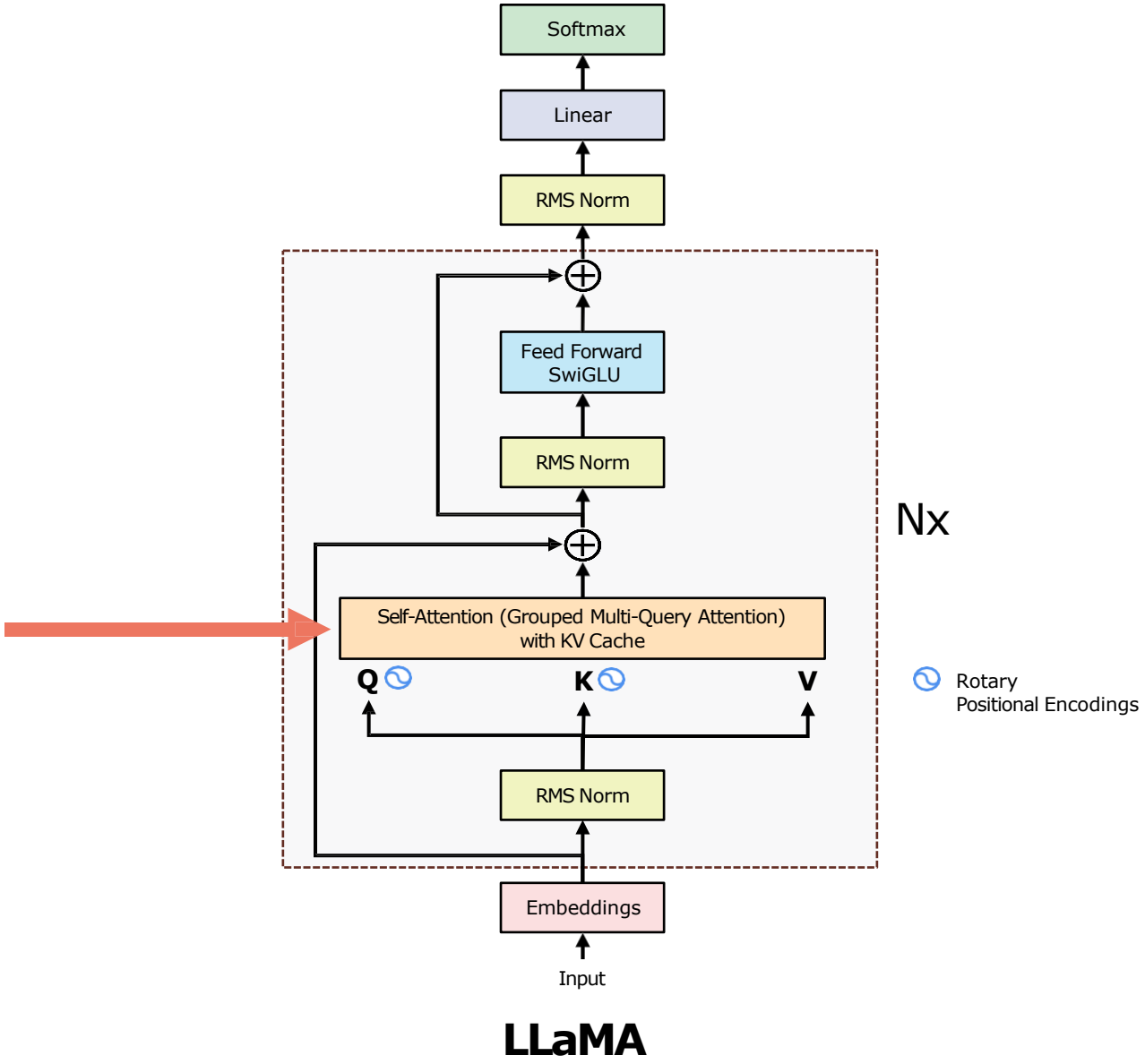




Inference
T = 4

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Grouped Multi-Query Attention



GPUs have a “problem”: they’re too fast.

- In recent years, GPUs have become very fast at performing calculations, insomuch that the speed of computation (FLOPs) is much higher than the memory bandwidth (GB/s) or speed of data transfer between memory areas. For example, an NVIDIA A100 can perform 19.5 TFLOPs while having a memory bandwidth of 2TB/s.
- This means that sometimes the bottleneck is not how many operations we perform, but how much data transfer our operations need, and that depends on the size and the quantity of the tensors involved in our calculations.
- For example, computing the same operation on the same tensor N times may be faster than computing the same operation on N different tensors, even if they have the same size, this is because the GPU may need to move the tensors around.
- This means that our goal should not only be to optimize the number of operations we do, but also minimize the memory access/transfers that we perform.

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Multi-Query Attention

Fast Transformer Decoding: One Write-Head is All You Need

Noam Shazeer
Google
noam@google.com

November 7, 2019

Vanilla batched multi-head attention

- Multihead Attention as presented in the original paper "Attention is all you need".
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bhn^2 + d^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{k} + \frac{1}{bn})$
- In this case, the ratio is much smaller than 1, which means that the number of memory access we are performing is much less than the number of arithmetic operations, so the memory access is **not** the bottleneck here.

```python
from mindspore import ops

def MultiheadAttentionBatched():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    X = ops.rand(b, n, d) # Query
    M = ops.rand(b, m, d) # Key and Value
    mask = ops.rand(b, h, n, m)
    P_q = ops.rand(h, d, k) # W_q
    P_k = ops.rand(h, d, k) # W_k
    P_v = ops.rand(h, d, k) # W_v
    P_o = ops.rand(h, d, k) # W_o

    Q = ops.einsum("bnd, hdk -> bhnk", X, P_q)
    K = ops.einsum("bmd, hdk -> bhmk", X, P_k)
    V = ops.einsum("bmd, hdk -> bhmv", X, P_v)

    logits = ops.einsum("bhnk, bhmk, bhnm", Q, K)
    weights = ops.softmax(logits + mask, axis=-1)

    O = ops.einsum("bhnm, bhmv -> bhnv", weights, V)
    Y = ops.einsum("bhnv, hdv -> bnd", O, P_o)
    return Y
```

Batched multi-head attention with KV cache

- Uses the KV cache to reduce the number of operations performed.
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O\left(bn^2d + nd^2\right)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{n}{d} + \frac{1}{b})$
- When $n \approx d$ (the sequence length is close to the size of the embedding vector) or $b \approx 1$ (the batch size is 1), the ratio becomes 1 and the memory access now becomes the bottleneck of the algorithm. For the batch size is not a problem, since it is generally much higher than 1, while for the $\frac{n}{d}$ term, we need to reduce the sequence length. **But there's a better way…**

```python
from mindspore import ops

def MultiheadAttentionIncremental():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    m = 5

    prev_K = ops.rand(b, h, m, k)
    prev_V = ops.rand(b, h, m, v)

    X = ops.rand(b, n, d) # Query
    M = ops.rand(b, m, d) # Key and Value
    mask = ops.rand(b, h, n, m)
    P_q = ops.rand(h, d, k) # W_q
    P_k = ops.rand(h, d, k) # W_k
    P_v = ops.rand(h, d, v) # W_v
    P_o = ops.rand(h, d, v) # W_o

    q = ops.einsum("bd, hdk -> bhk", X, P_q)
    new_K = ops.concat([prev_K, ops.einsum("bd, hdk -> bhk", M, P_k).unsqueeze(2)], axis=2)
    new_V = ops.concat([prev_V, ops.einsum("bd, hdv -> bhv", M, P_v).unsqueeze(2)], axis=2)
    logits = ops.einsum("bhk, bhmk, bhm", q, new_K)
    weights = ops.softmax(logits + mask, axis=-1)

    O = ops.einsum("bhm, bhmv -> bhv", weights, new_V)
    Y = ops.einsum("bnv, hdv -> bd", O, P_o)
    return Y
```

Multi-query attention with KV cache

- We remove the h dimension from the K and the V, while keeping it for the Q. This means that all the different query heads will share the same keys and values.
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O\left(bnd + bn^2k + nd^2\right)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b})$
- Comparing with the previous approach, we have reduced the expensive term $\frac{n}{d}$ by a factor of h.
- The performance gains are important, while the model's quality degrades only a little bit.

```python
from mindspore import ops

def MultiheadAttentionIncremental():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    m = 5

    prev_K = ops.rand(b, h, m, k)
    prev_V = ops.rand(b, h, m, v)

    X = ops.rand(b, n, d) # Query
    M = ops.rand(b, m, d) # Key and Value
    mask = ops.rand(b, h, n, m)
    P_q = ops.rand(h, d, k) # W_q
    P_k = ops.rand(d, k) # W_k
    P_v = ops.rand(d, v) # W_v
    P_o = ops.rand(h, d, v) # W_o

    q = ops.einsum("bd, hdk -> bhk", X, P_q)
    new_K = ops.concat([prev_K, ops.einsum("bd, hdk -> bhk", M, P_k).unsqueeze(1)], axis=1)
    new_V = ops.concat([prev_V, ops.einsum("bd, hdv -> bhv", M, P_v).unsqueeze(1)], axis=1)
    logits = ops.einsum("bhk, bmk -> bhm", q, new_K)
    weights = ops.softmax(logits + mask, axis=-1)

    O = ops.einsum("bhm, bhmv -> bhv", weights, new_V)
    Y = ops.einsum("bnv, hdv -> bd", O, P_o)
    return Y
```

Speed & Quality comparisons

Table 1: WMT14 EN-DE Results.

Attention Type	h	d_k, d_v	d_{ff}	ln(PPL) (dev)	BLEU (dev)	BLEU (test) beam 1 / 4
multi-head	8	128	4096	**1.424**	**26.7**	27.7 / 28.4
multi-query	8	128	5440	1.439	26.5	27.5 / **28.5**
multi-head local	8	128	4096	1.427	26.6	27.5 / 28.3
multi-query local	8	128	5440	1.437	26.5	27.6 / 28.2
multi-head	1	128	6784	1.518	25.8	
multi-head	2	64	6784	1.480	26.2	26.8 / 27.9
multi-head	4	32	6784	1.488	26.1	
multi-head	8	16	6784	1.513	25.8	

Table 2: Amortized training and inference costs for WMT14 EN-DE Translation Task with sequence length 128. Values listed are in TPUv2-microseconds per output token.

Attention Type	Training	Inference enc. + dec.	Beam-4 Search enc. + dec.
multi-head	13.2	1.7 + 46	2.0 + 203
multi-query	**13.0**	1.5 + 3.8	1.6 + 32
multi-head local	13.2	1.7 + 23	1.9 + 47
multi-query local	**13.0**	**1.5 + 3.3**	**1.6 + 16**

To demonstrate that local-attention and multi-query attention are orthogonal, we also trained "local" versions of the baseline and multi-query models, where the decoder-self-attention layers (but not the other attention layers) restrict attention to the current position and the previous 31 positions.

Grouped Multi-Query Attention

Multi-Head Attention

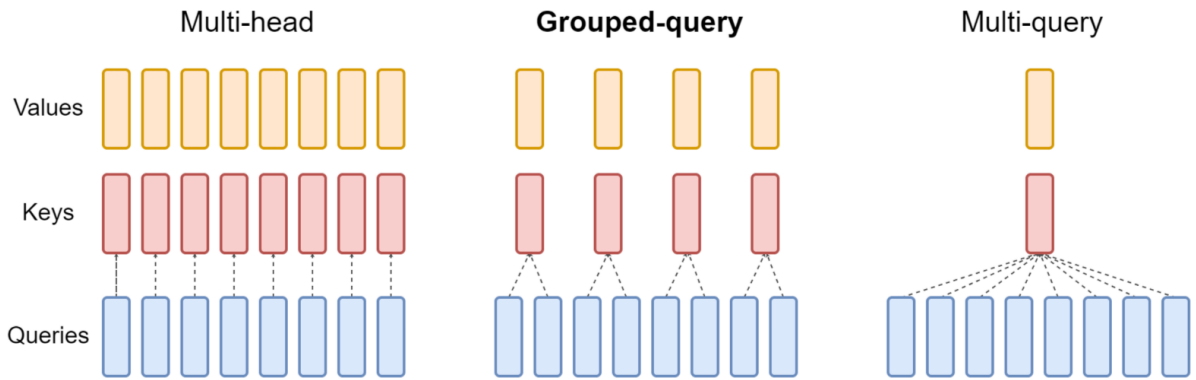
- High quality
- Computationally slow

Grouped Multi-Query Attention

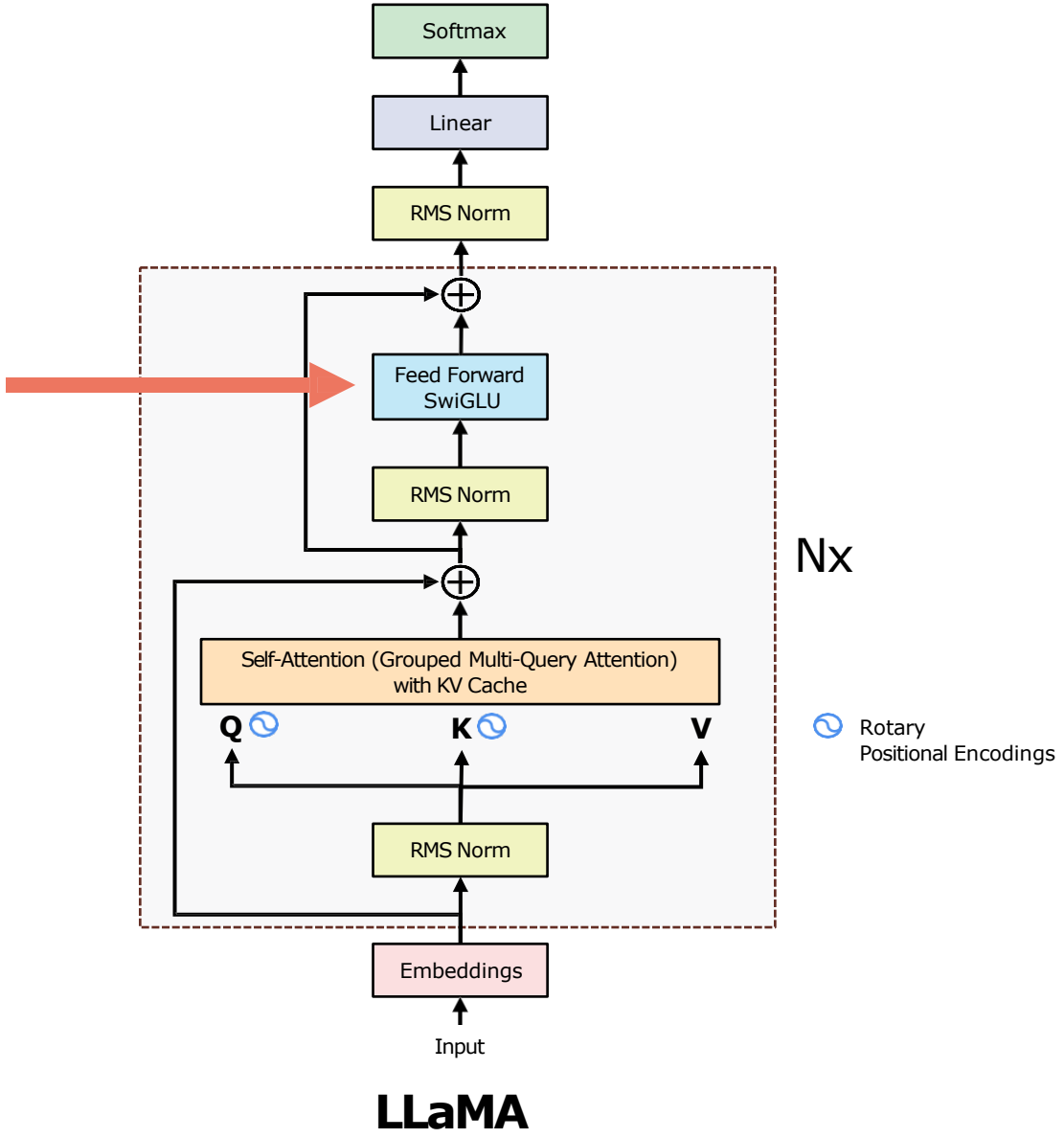
- A good compromise between quality and speed

Multi-Query Attention

- Loss in quality
- Computationally fast



SwiGLU activation



SwiGLU Activation Function

GLU Variants Improve Transformer

Noam Shazeer
Google
noam@google.com

February 14, 2020

SwiGLU Activation Function

- The author compared the performance of a Transformer model by using different activation functions in the Feed-Forward layer of the Transformer architecture.

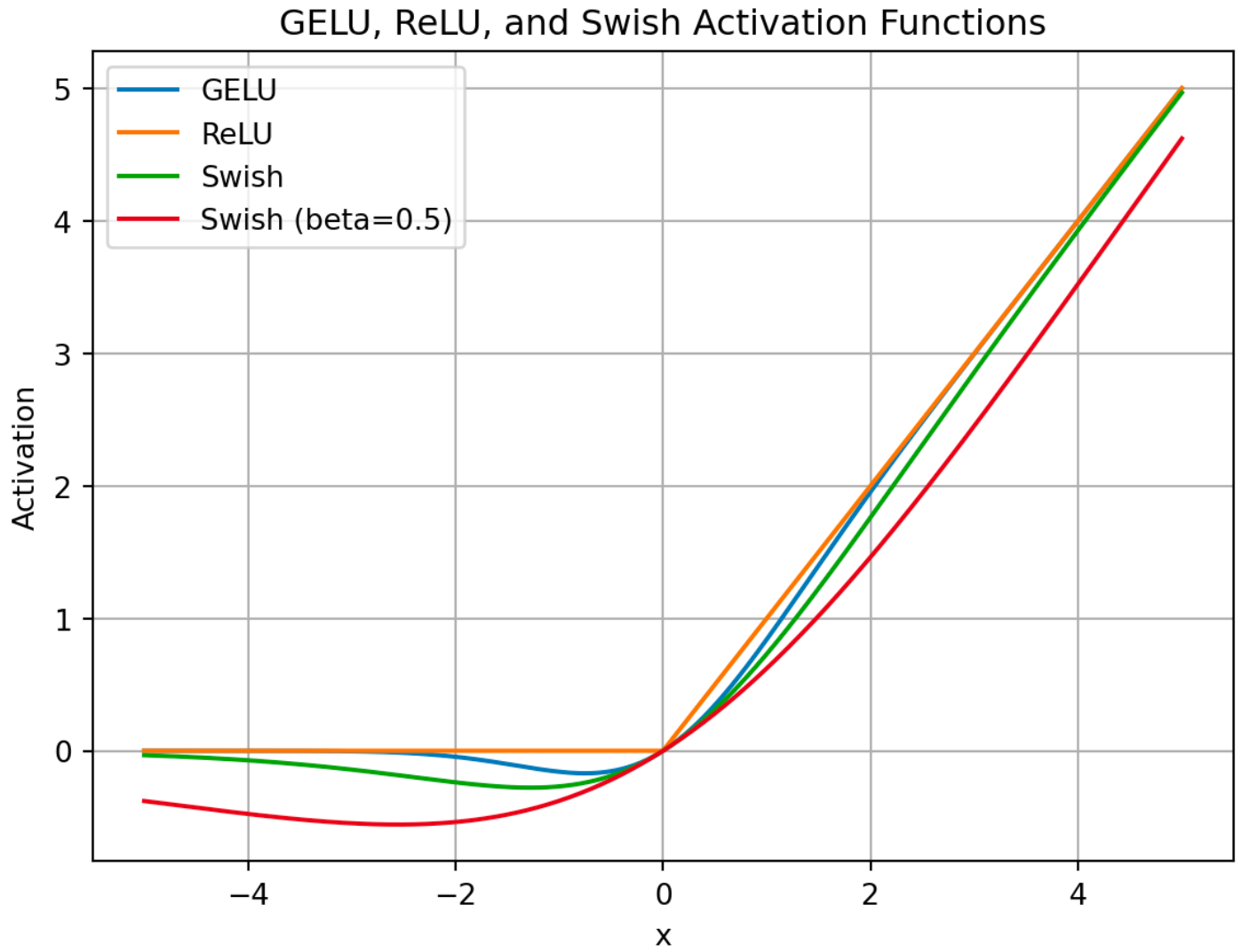
$$
\begin{aligned}
\mathrm{ReGLU}(x, W, V, b, c) &= \max(0, xW + b) \otimes (xV + c) \\
\mathrm{GEGLU}(x, W, V, b, c) &= \mathrm{GELU}(xW + b) \otimes (xV + c) \\
\mathrm{SwiGLU}(x, W, V, b, c, \beta) &= \mathrm{Swish}_\beta(xW + b) \otimes (xV + c)
\end{aligned} \tag{5}
$$

In this paper, we propose additional variations on the Transformer FFN layer which use GLU or one of its variants in place of the first linear transformation and the activation function. Again, we omit the bias terms.

$$
\begin{aligned}
\mathrm{FFN}_{\mathrm{GLU}}(x, W, V, W_2) &= (\sigma(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{Bilinear}}(x, W, V, W_2) &= (xW \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{ReGLU}}(x, W, V, W_2) &= (\max(0, xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{GEGLU}}(x, W, V, W_2) &= (\mathrm{GELU}(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) &= (\mathrm{Swish}_1(xW) \otimes xV)W_2
\end{aligned} \tag{6}
$$

All of these layers have three weight matrices, as opposed to two for the original FFN. To keep the number of parameters and the amount of computation constant, we reduce the number of hidden units d_{ff} (the second dimension of W and V and the first dimension of W_2) by a factor of $\frac{2}{3}$ when comparing these layers to the original two-matrix version.

SwiGLU Activation Function



业界LLM结构对比

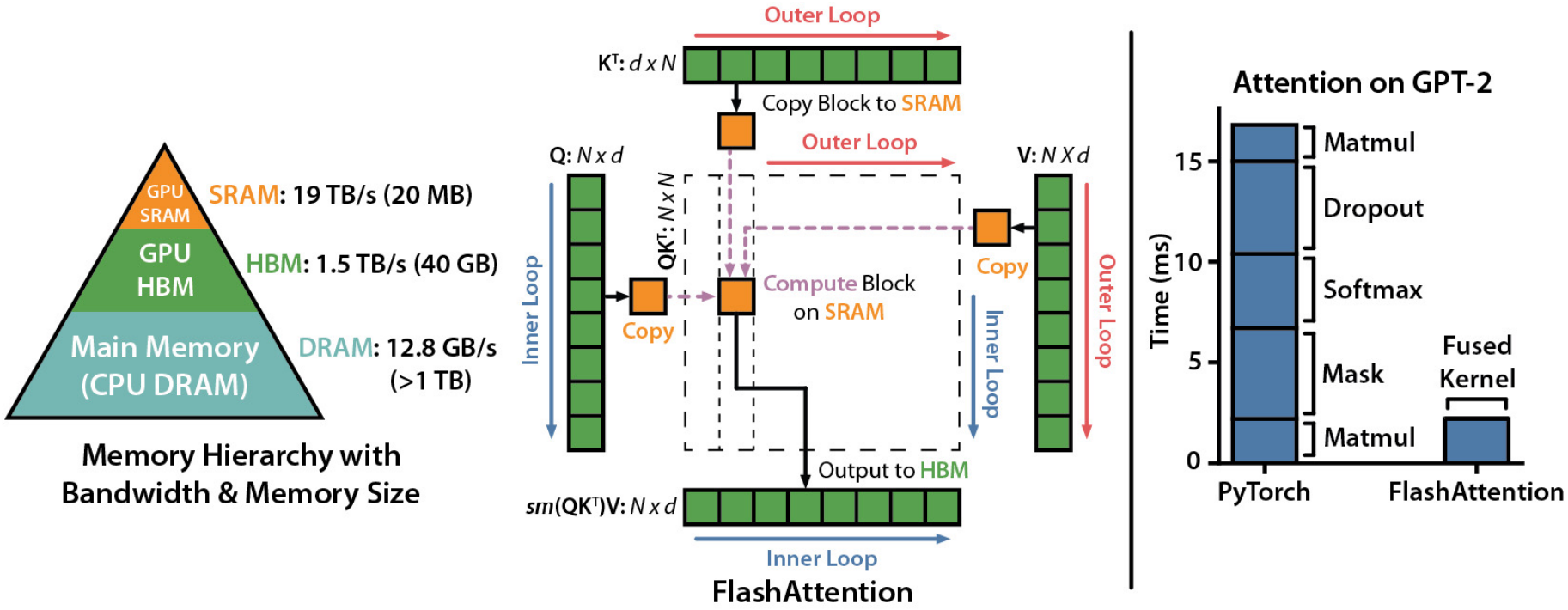
模型名称	参数	隐藏层维度	层数	注意力头数	训练数据	位置编码	激活函数	归一化方法	注意力机制	词表大小	最大长度
LLAMA	6.7B	4096	32	32	1T	RoPE	SwiGLU	RMSNorm（pre-norm）Attention Layer和MLP的输入上使用	多头注意力机制(MHA)	32000	2048
	13.0B	5120	40	40	1T	RoPE	SwiGLU	RMSNorm（pre-norm）Attention Layer和MLP的输入上使用	多头注意力机制(MHA)	32000	2048
	32.5B	6656	60	52	1.4T	RoPE	SwiGLU	RMSNorm（pre-norm）Attention Layer和MLP的输入上使用	多头注意力机制(MHA)	32000	2048
	65.2B	8192	80	64	1.4T	RoPE	SwiGLU	RMSNorm（pre-norm）Attention Layer和MLP的输入上使用	多头注意力机制(MHA)	32000	2048
LLAMA2	7B	4096	32	32	2.0T	RoPE	SwiGLU	RMSNorm（pre-norm）Attention Layer和MLP的输入上使用	多头注意力机制(MHA)	32000	4096
	13B	5120	40	40	2.0T	RoPE	SwiGLU	RMSNorm	多头注意力机制(MHA)	32000	4096
	70B	8192	80	64	2.0T	RoPE	SwiGLU	RMSNorm	Group Query Attention group=8	32000	4096
chatglm-6B	6.2B	4096	28	32	1T	RoPE 2d位置编码	GELU	layer norm（post-norm)	多头注意力机制(MHA)	130528	2048
chatglm2-6B	6.2B	4096	28	32	1.4T	RoPE 推理时，舍弃2d位置编码，回归decoder-only	SwiGLU	RMSNorm（post-norm)	Multi-Query Attention （MQA)	65024	32768
baichuan-7b	7B	4096	32	32	1.2T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	64,000	4096
baichuan-13b	13B	5120	40	40	1.4T	ALiBi	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	64,000	4096
baichuan2-7b	7B	4096	32	32	2.6T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	125,696	4096
baichuan2-13b	13B	5120	40	40	2.6T	ALiBi	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	125,696	4096
Qwen-7B	7B	4096	32	32	2.4T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	151851	2048
Qwen-14B	14B	5120	40	40	3T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	152064	8192
Yi-6B	6B	4096	32	32	3T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	64000	4096
Yi-34B	34B	7168	56	60	3T	RoPE	SwiGLU	RMSNorm（pre-norm)	多头注意力机制(MHA)	64000	4096

LLM改进

模型升级之路	训练Token数	序列长度	算子改进	核心点
ChatGLM->ChatGLM2	1T->1.4T	2K->8K/32K	FlashAttention & Multi Query Attention	Prefix-LM->Decoder-Only
LLAMA->LLAMA2	1.4T->2T	2K->4K	-	更高质量的SFT&RLHF
baichuan->baichuan 13b	1.2T->1.4T	4K(RoPE)->4K(ALiBi)	FlashAttention	参数量升级
baichuan->baichuan2	1.2T->2.6T	4K	-	Tokenizer/NormHead/Max-z Loss

改进点1: 性能优化

- Flash Attention、Multi-Query Attention提高训练&推理的速度



改进点2: 增加训练数据

Llama 2 was trained on **40% more data** than Llama 1,
and has double the context length.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:**	**Data collection for helpfulness and safety:**
13B	**Pretraining Tokens:** 2 Trillion	**Supervised fine-tuning:** Over 100,000
70B	**Context Length:** 4096	**Human Preferences:** Over 1,000,000

改进点3: 加大context长度

Llama 2 was trained on **40% more data** than Llama 1,
and has double the context length.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:**	**Data collection for helpfulness and safety:**
13B	**Pretraining Tokens:** 2 Trillion	**Supervised fine-tuning:** Over 100,000
70B	**Context Length:** 4096	**Human Preferences:** Over 1,000,000

改进点4: 提高训练的稳定性

- NormHead:主要用于对输出嵌入进行归一化处理，有助于稳定训练动态，并降低了L2距离在计算logits时的影响
- Max-z Loss: 引入了最大z损失，用于规范模型输出的logit值，从而提高训练的稳定性并使推断更加鲁棒

```python
class NormHead(nn.Module):
    def __init__(self, hidden_size, vocab_size, bias=False):
        super().__init__()
        self.weight = nn.Parameter(torch.empty((vocab_size, hidden_size)))
        nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        self.first_flag = True

    def forward(self, hidden_states):
        if self.training:
            norm_weight = nn.functional.normalize(self.weight)
        elif self.first_flag:
            self.first_flag = False
            self.weight = nn.Parameter(nn.functional.normalize(self.weight
            norm_weight = self.weight
        else:
            norm_weight = self.weight
        return nn.functional.linear(hidden_states, norm_weight)
```

```python
hidden_states = outputs[0]
logits = self.lm_head(hidden_states)
loss = None
if labels is not None:
    # Shift so that tokens < n predict n
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
    # Flatten the tokens
    loss_fct = CrossEntropyLoss()
    shift_logits = shift_logits.view(-1, self.config.vocab_size)
    shift_labels = shift_labels.view(-1)
    softmax_normalizer = shift_logits.max(-1).values ** 2
    z_loss = self.config.z_loss_weight * softmax_normalizer.mean()
    # Enable model parallelism
    shift_labels = shift_labels.to(shift_logits.device)
    loss = loss_fct(shift_logits, shift_labels) + z_loss
```