https://hao-ai-lab.github.io/dsc291-s24/

DSC 291: ML Systems
Spring 2024

LLMs

Parallelization

Single-device Optimization

Basics

Logistics

- Please start preparing your final project talk
  - We will use week 10 (may need additional time) to go through each team's talk
  - TAs will distribute some sample slides / guidelines

Automatic Parallelization Methods

Search-based methods

- ✅ Easy to extend the search space
- ✅ No training cost
- ❌ High inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

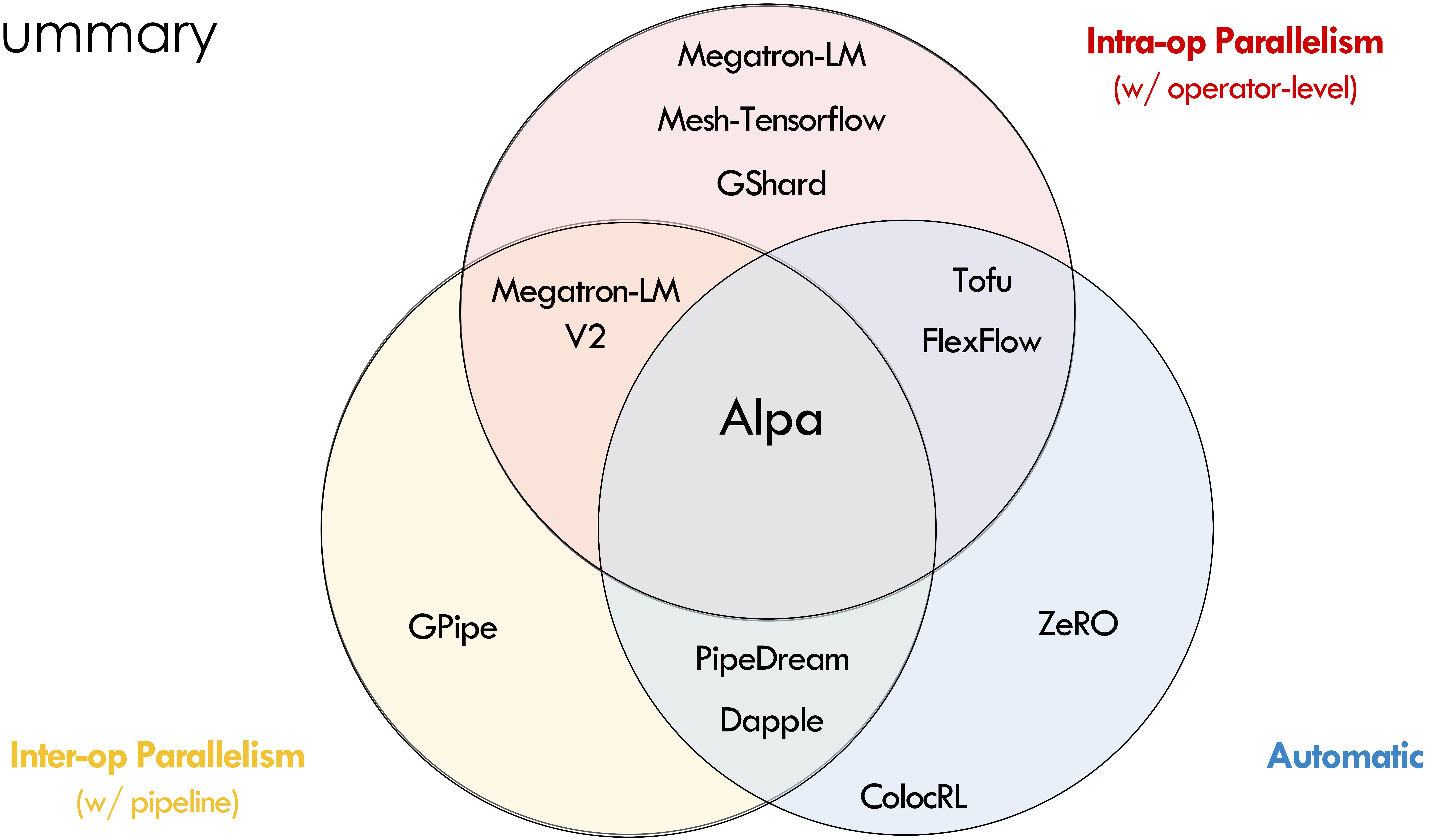
Learning-based methods

- ✅ Easy to extend the search space
- ❌ High training cost
- ✅ Low inference cost
- ❌ Not explainable
- ❌ No optimality guarantee

Optimization-based methods

- ❌ Non-trivial to extend the search space
- ✅ No training cost
- ✅ Medium inference cost
- ✅ Explainable
- ✅ Some optimality guarantee

Summary


Intra-op Parallelism
(w/ operator-level)
Megatron-LM
Mesh-Tensorflow
GShard
Megatron-LM
V2
Tofu
FlexFlow
Alpa
GPipe
PipeDream
Dapple
ZeRO
ColocRL
Inter-op Parallelism
(w/ pipeline)
Automatic

Where We Are: LLMs

- Transformers and Attentions
- LLM Training Optimizations
  - Flash attention
  - 3D parallelism and sequence parallelism
- LLM Inference and Serving
  - Paged attention
  - Continuous batching
  - Speculative decoding
- Scaling Laws
- Long context

Transformer and Attentions

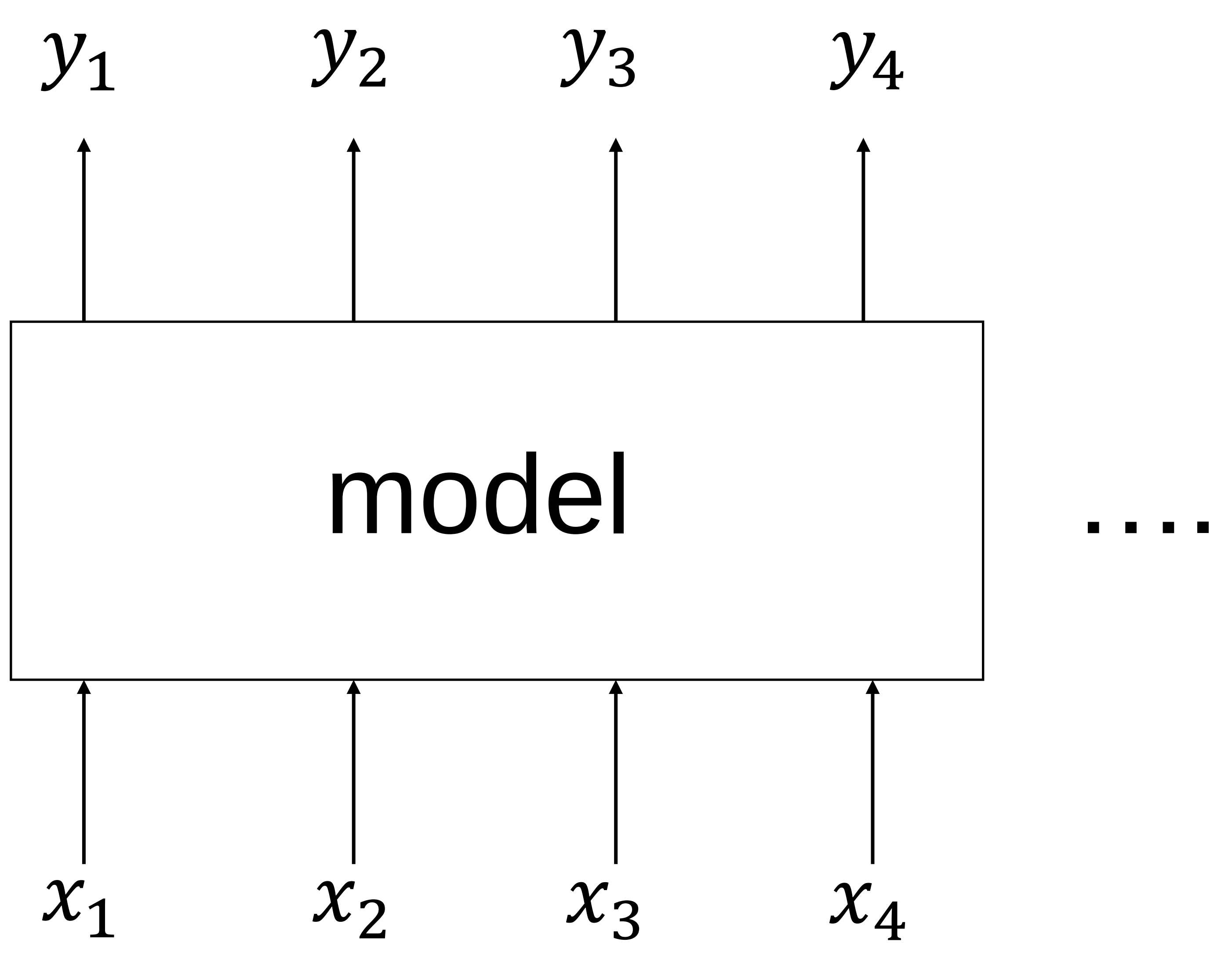
Sequence Prediction

Transformers and Self-Attention

Recursive Attention

Sequence Prediction

Take a set of input sequence, predict the output sequence

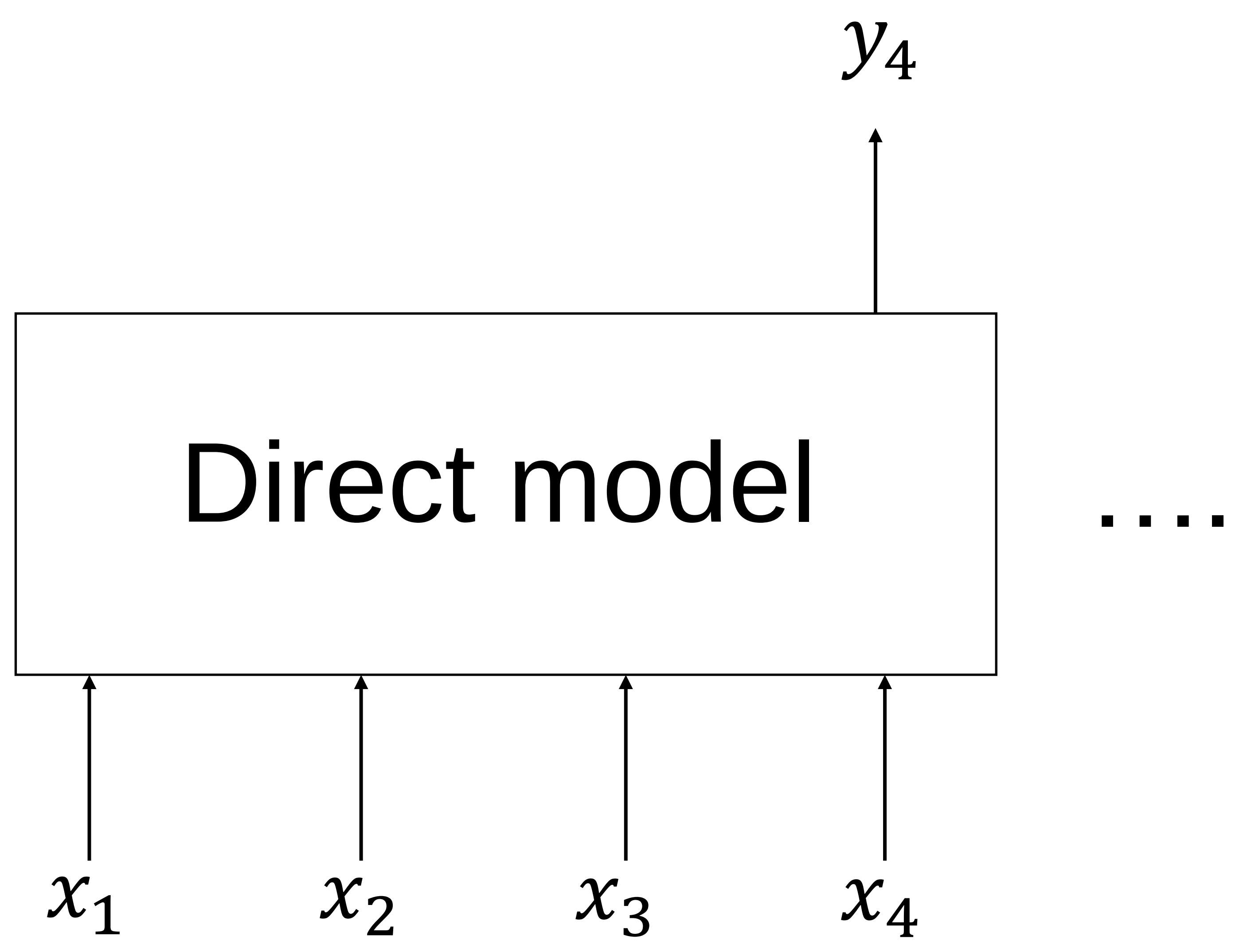


Predict each output based on history $y_t = f_\theta(x_{1:t})$

There are many ways to build up the predictive model

“Direct Prediction”

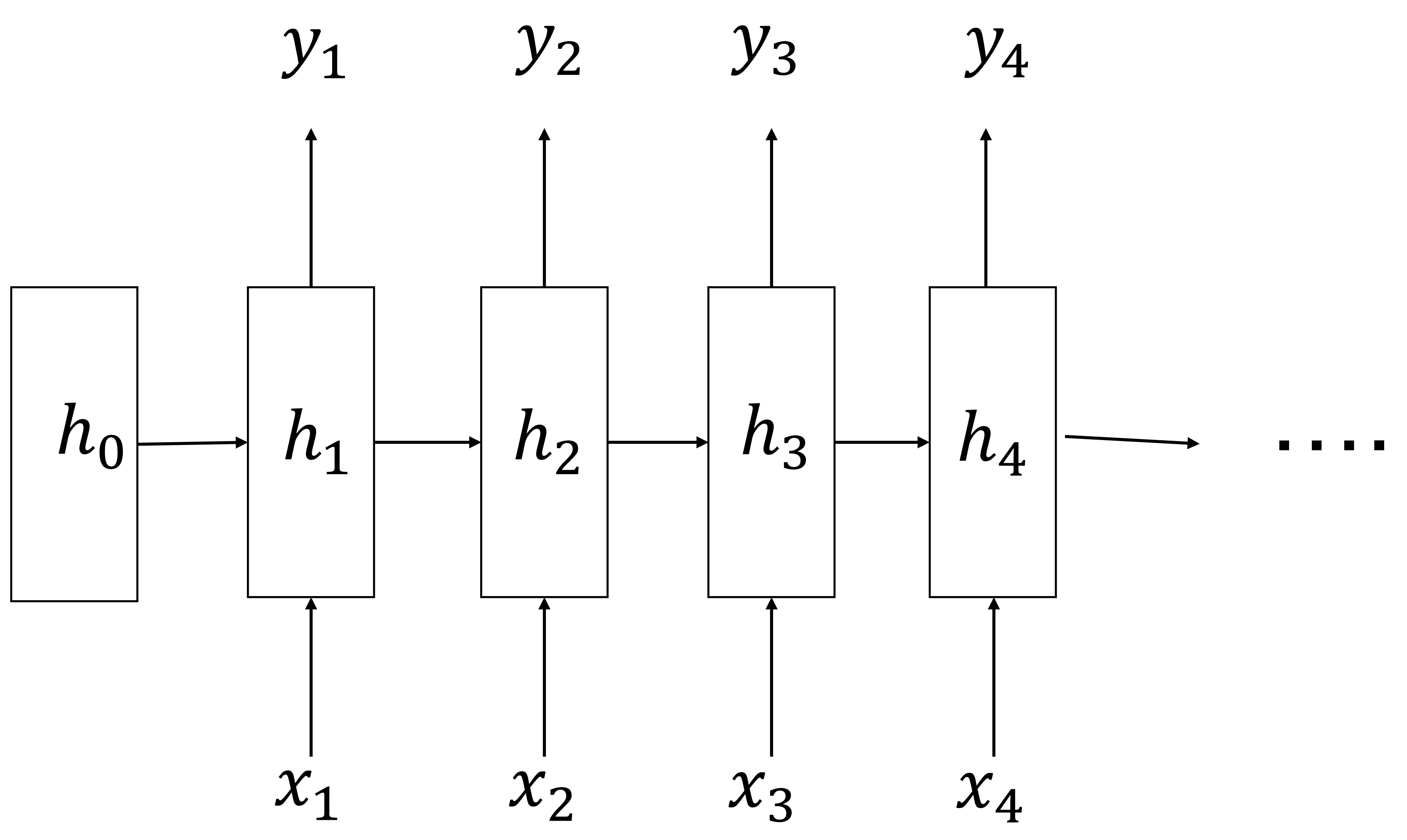
One approach is we can do “direct prediction”



Challenge: the function needs to make prediction based on different size inputs.

RNN Approach

Try to maintain a “latent state” that is derived from history



The information is carried only through h_t

Outline

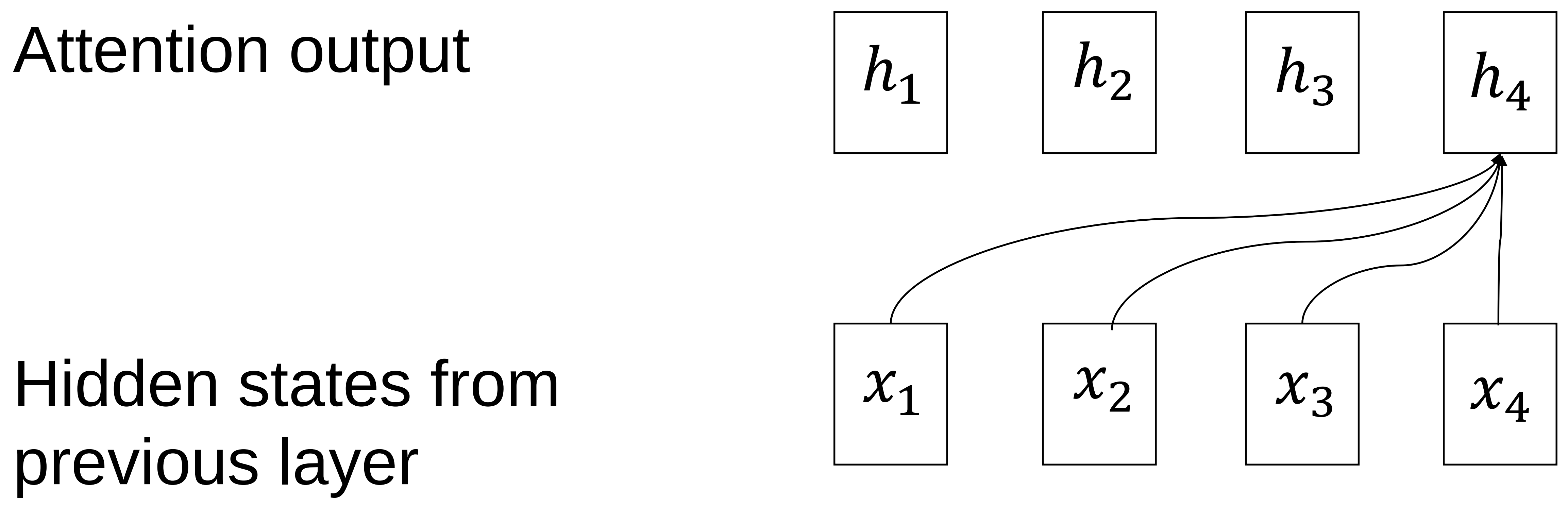
Sequence Prediction

Transformers and Self-Attention

Recursive Attention

“Attention” Mechanism

Generally refers to the approach that weighted combine individual states



$$h_t = \sum_{i=1}^{t} s_i x_t$$

Intuitively s_i is “attention score” that computes how relevant the position i’s input is to this current hidden output

There are different methods to decide how attention score is being computed

Self-Attention Operation

Self attention refers to a particular form of attention mechanism.

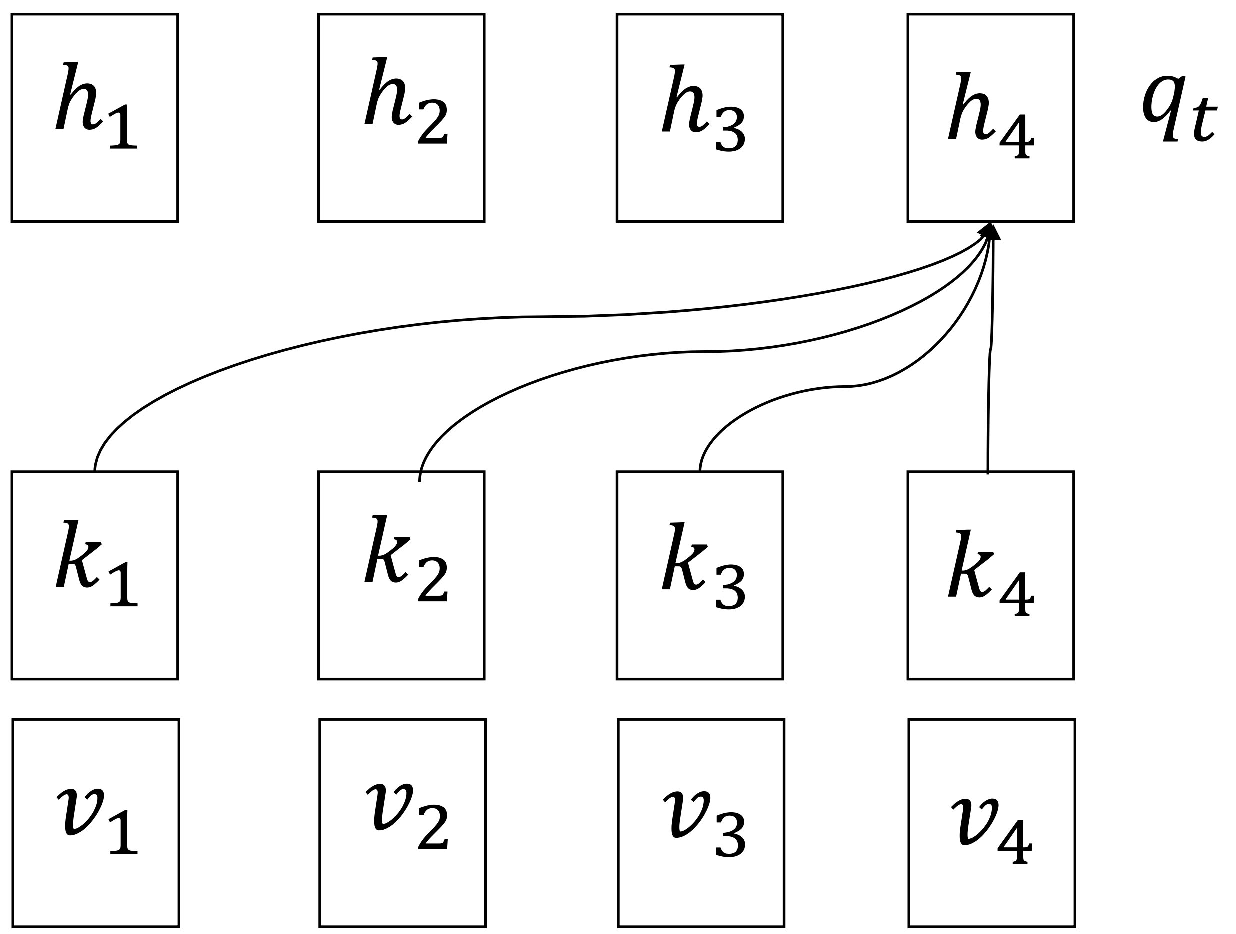
Given three inputs $Q, K, V \in \mathbb{R}^{T \times d}$ ("queries", "keys", "values")

Define the self-attention as:

$$\mathrm{SelfAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

A Closer Look at Self-Attention

Use q_t, k_t, v_t to refers to row t of the K matrix



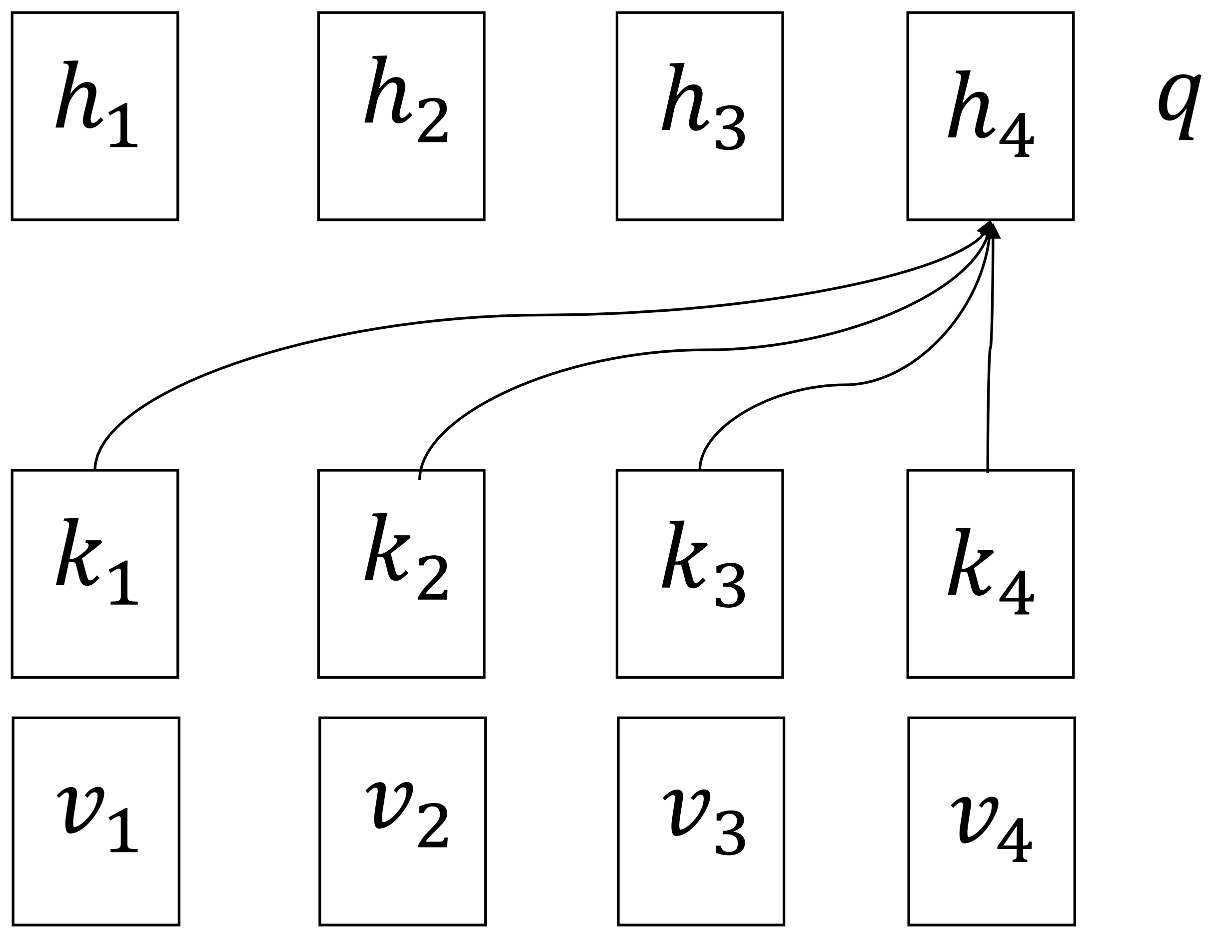
Ask the following question:

How to compute the output h_t, based on q_t, K, V one timestep t

To keep presentation simple, we will drop suffix t and just use q to refer to q_t in next few slide

A Closer Look at Self-Attention

Use q_t, k_t, v_t to refers to row t of the K matrix



Conceptually, we compute the output in the following two steps:

Pre-softmax "attention score"

$$s_i = \frac{1}{\sqrt{d}} q k_i^T$$

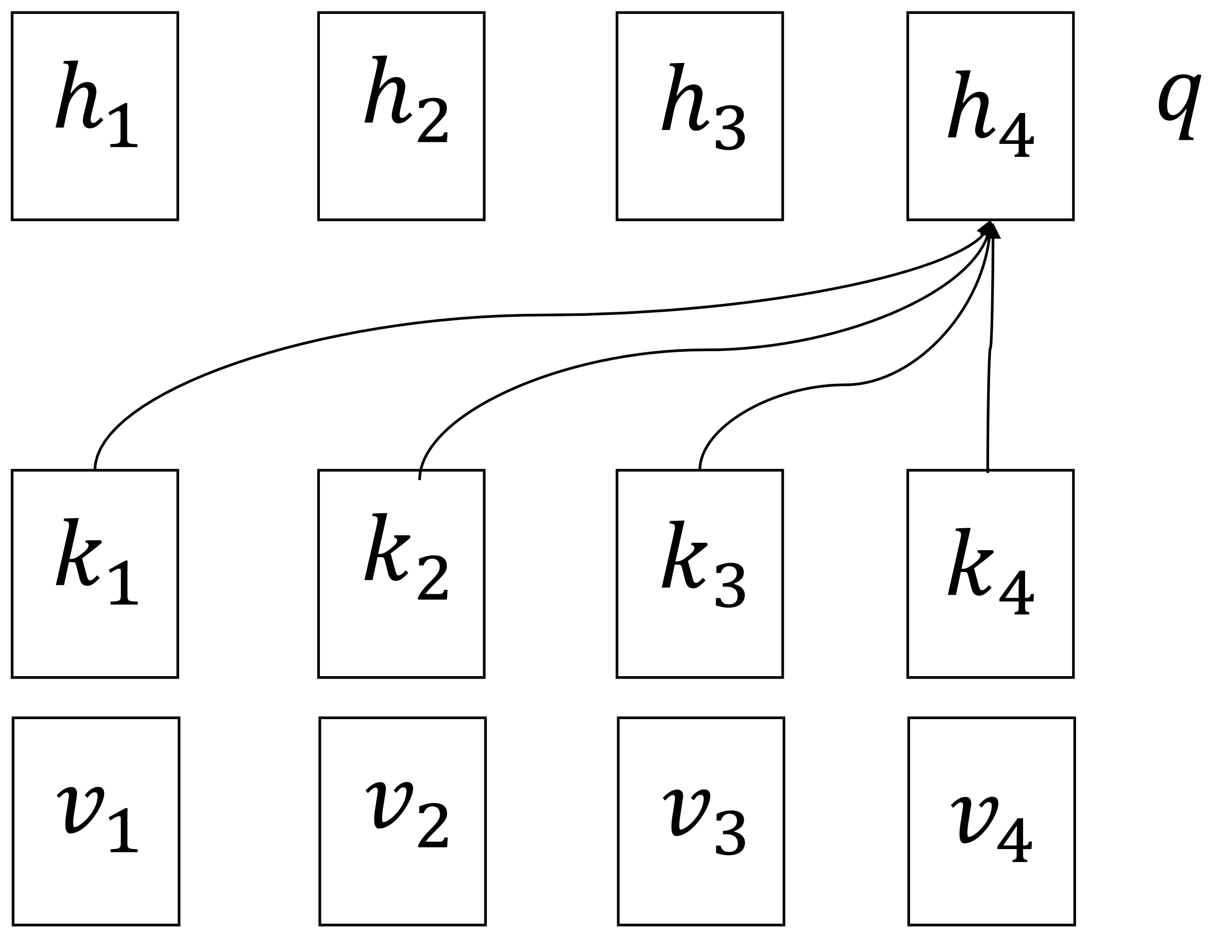
Weighed average via softmax

$$h = \sum_i \mathrm{softmax}(s)_i v_i = \frac{\sum_i \exp(s_i) v_i}{\sum_j \exp(s_j)}$$

Intuition: s_i computes the relevance of k_i to the query q,
then we do weighted sum of values proportional to their relevance

Comparing the Matrix Form and the Decomposed Form

Use q_t, k_t, v_t to refers to row t of the K matrix



$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

Pre-softmax “attention score”

$$S_{ti} = \frac{1}{\sqrt{d}} q_t k_i^T$$

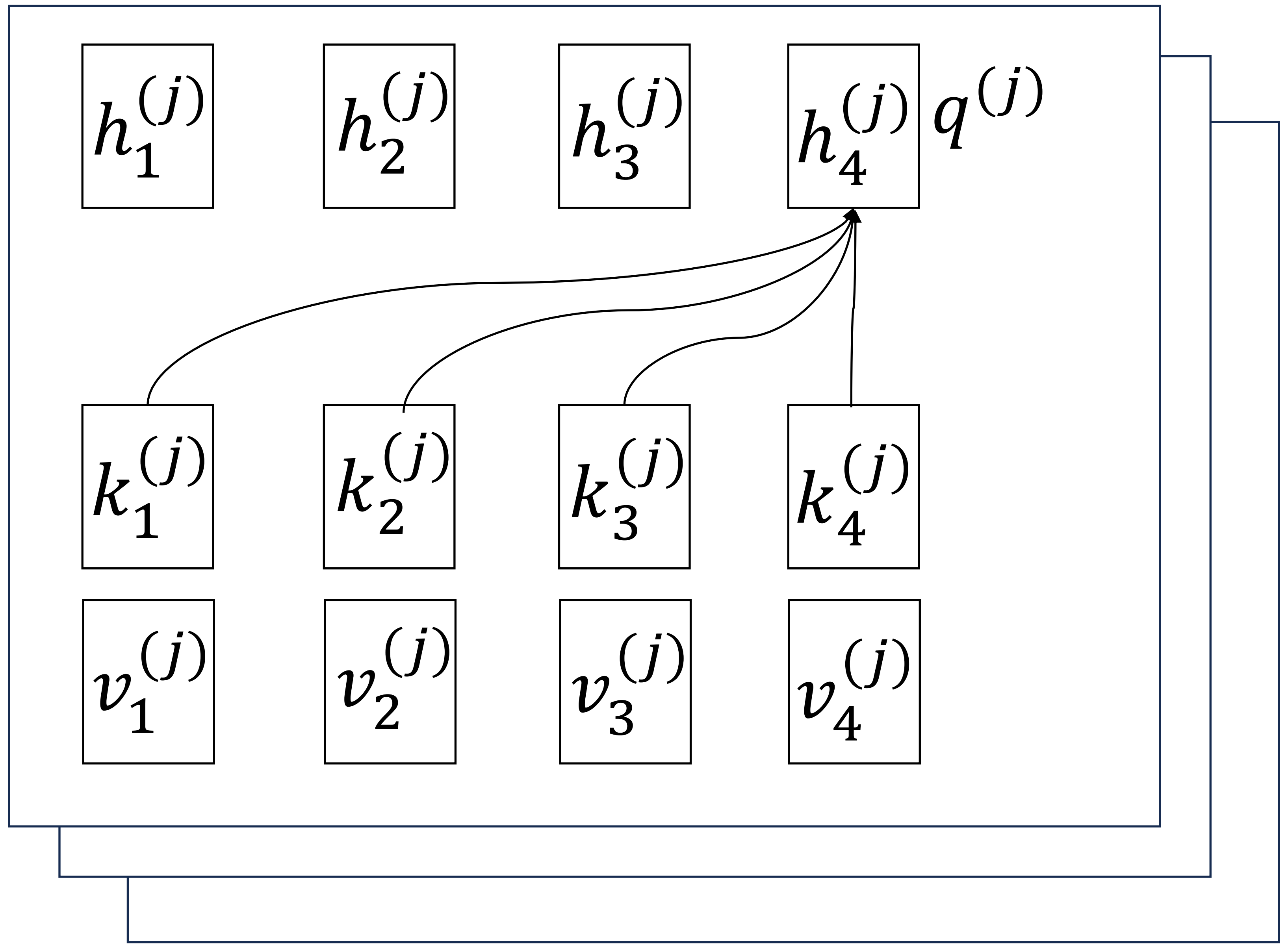
Weighed average via softmax

$$h_t = \sum_i \text{softmax}\left(S_{t,:}\right)_i v_i = \text{softmax}\left(S_{t,:}\right)V$$

Intuition: s_i computes the relevance of k_i to the query q,
then we do weighted sum of values proportional to their relevance

Multi-Head Attention

Have multiple "attention heads" $Q^{(j)}, K^{(j)}, V^{(j)}$ denotes j-th attention head



Apply self-attention in each attention head

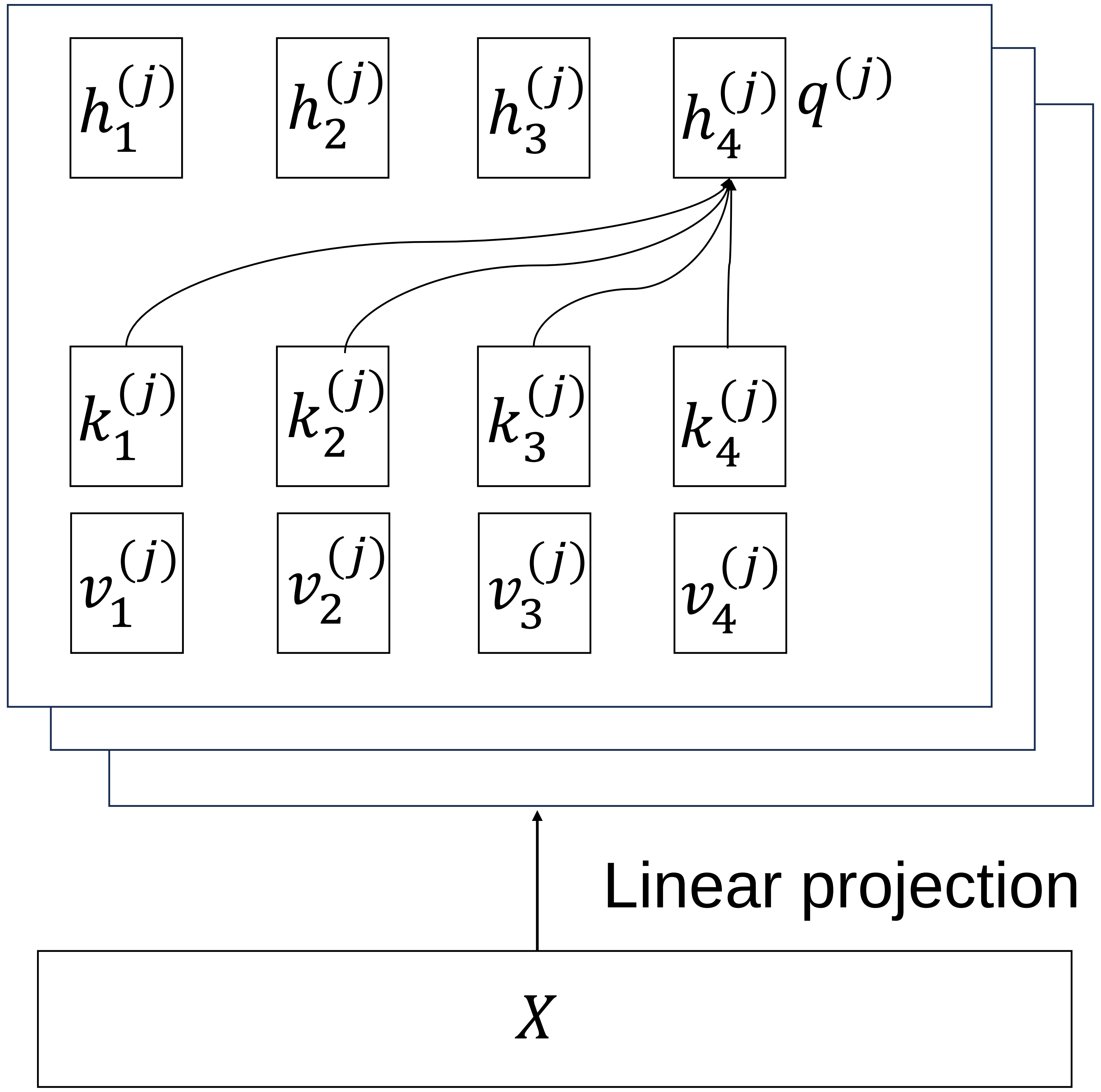
$$\mathrm{SelfAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{d^{1/2}}\right)V$$

Concatenate all output heads together as output

Each head can correspond to different kind of information.
Sometimes we can share the heads: GQA(group query attention) all heads share K, V but have different Q

How to get Q K V?

Obtain Q, K, V from previous layer's hidden state X by linear projection



$$Q = XW_q$$
$$K = XW_K$$
$$V = XW_V$$

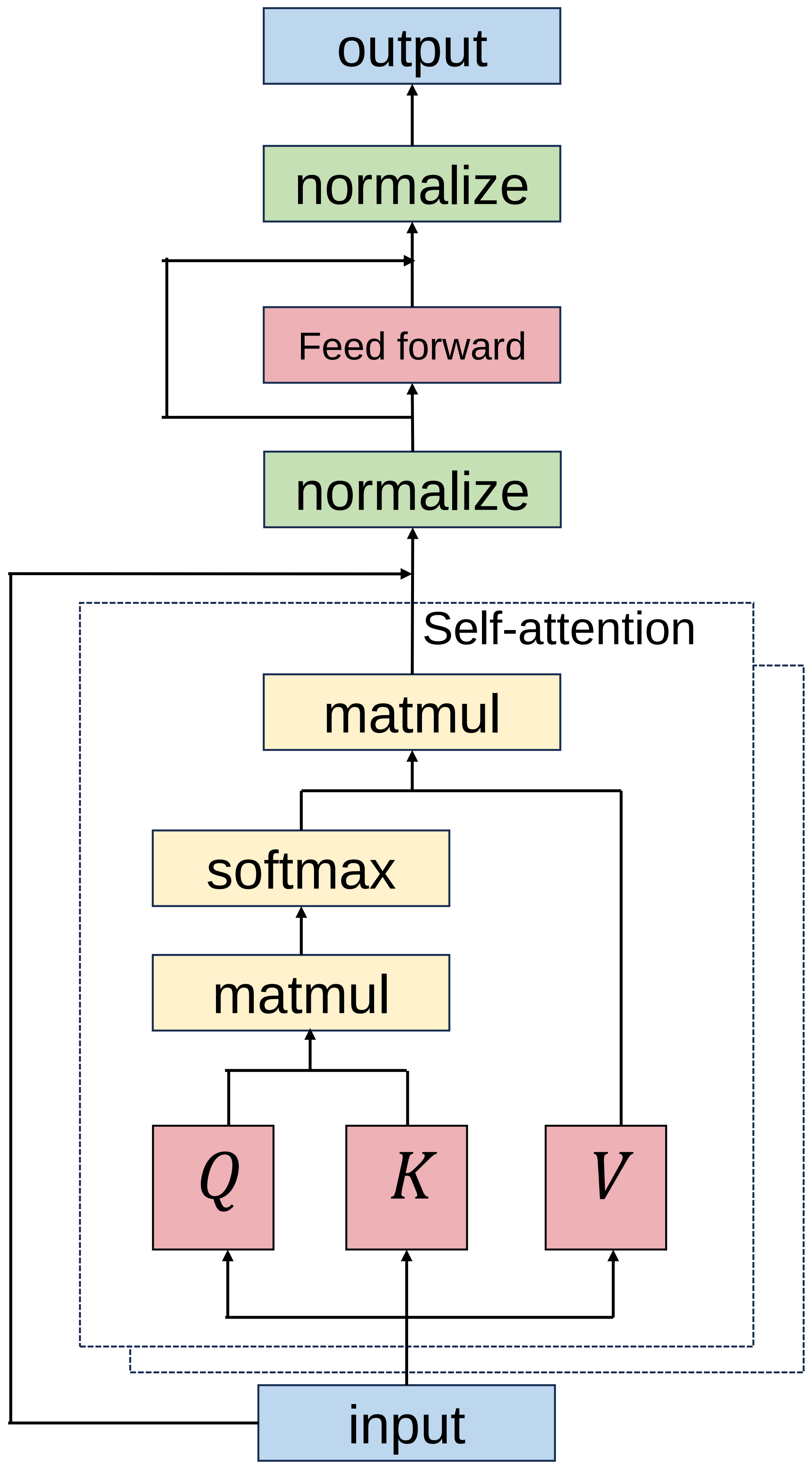
Can compute all heads and Q, K, V together then split/reshape out into individual Q, K, V with multiple heads

Transformer Block

A typical transformer block

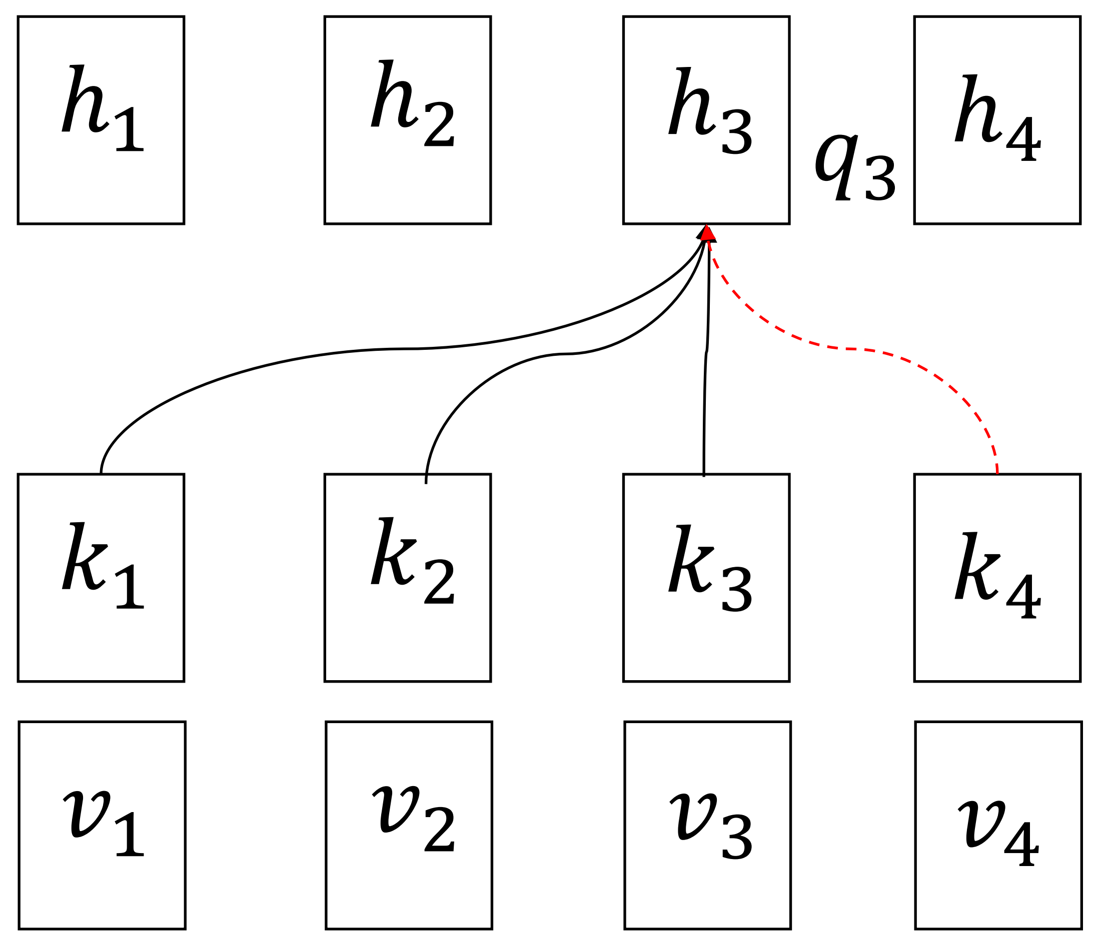
$$Z = \mathrm{SelfAttention}\left(XW_K, XW_Q, XW_V\right)$$
$$Z = \mathrm{LayerNorm}(X + Z)$$
$$H = \mathrm{LayerNorm}(\mathrm{ReLU}(ZW_1)W_2 + Z)$$

(multi-head) self-attention, followed by a linear layer and ReLU and some additional residual connections and normalization



Masked Self-Attention

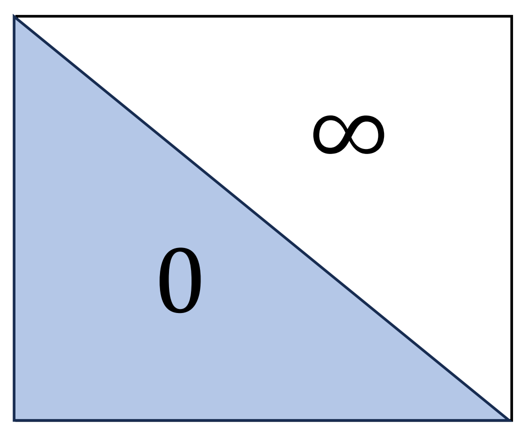
In the matrix form, we are computing weighted average over all inputs



In auto regressive models, usually it is good to maintain casual relation, and only attend to some of the inputs (e.g. skip the red dashed edge on the left). We can add "attention mask"

$$\mathrm{MaskedSelfAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{d^{1/2}} - M\right)V$$

$$M_{ij} = \begin{cases} \infty, j > i \\ 0, j \leq i \end{cases}$$



Only attend to previous inputs. Depending on input structure and model, attention mask can change.

We can also simply skip the computation that are masked out if there is a special implementation to do so

Discussions

What are the advantages of transformers versus RNNs

What are the disadvantages

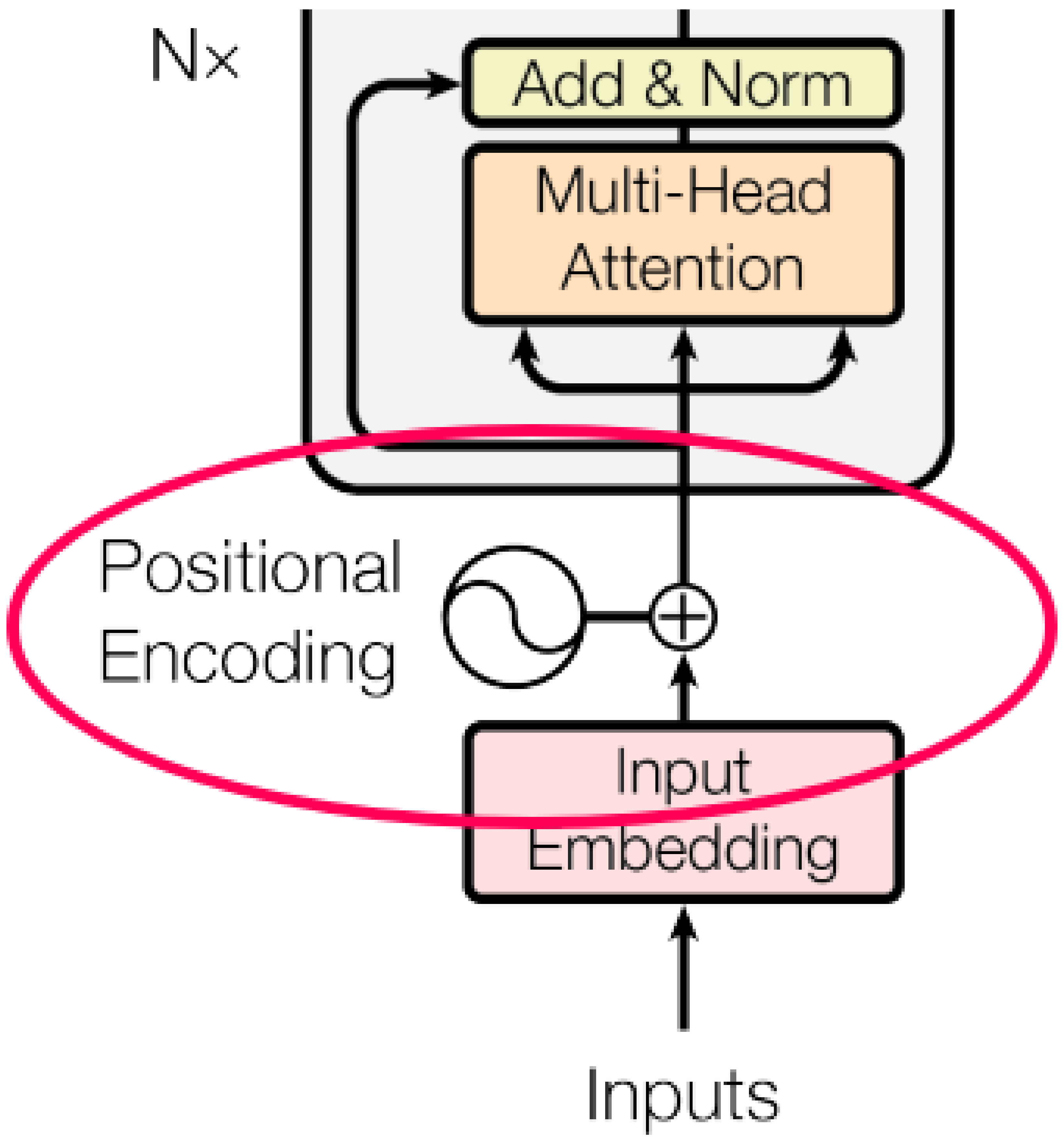
What are other possible ways to apply attention mask

What Components are in LLMs?

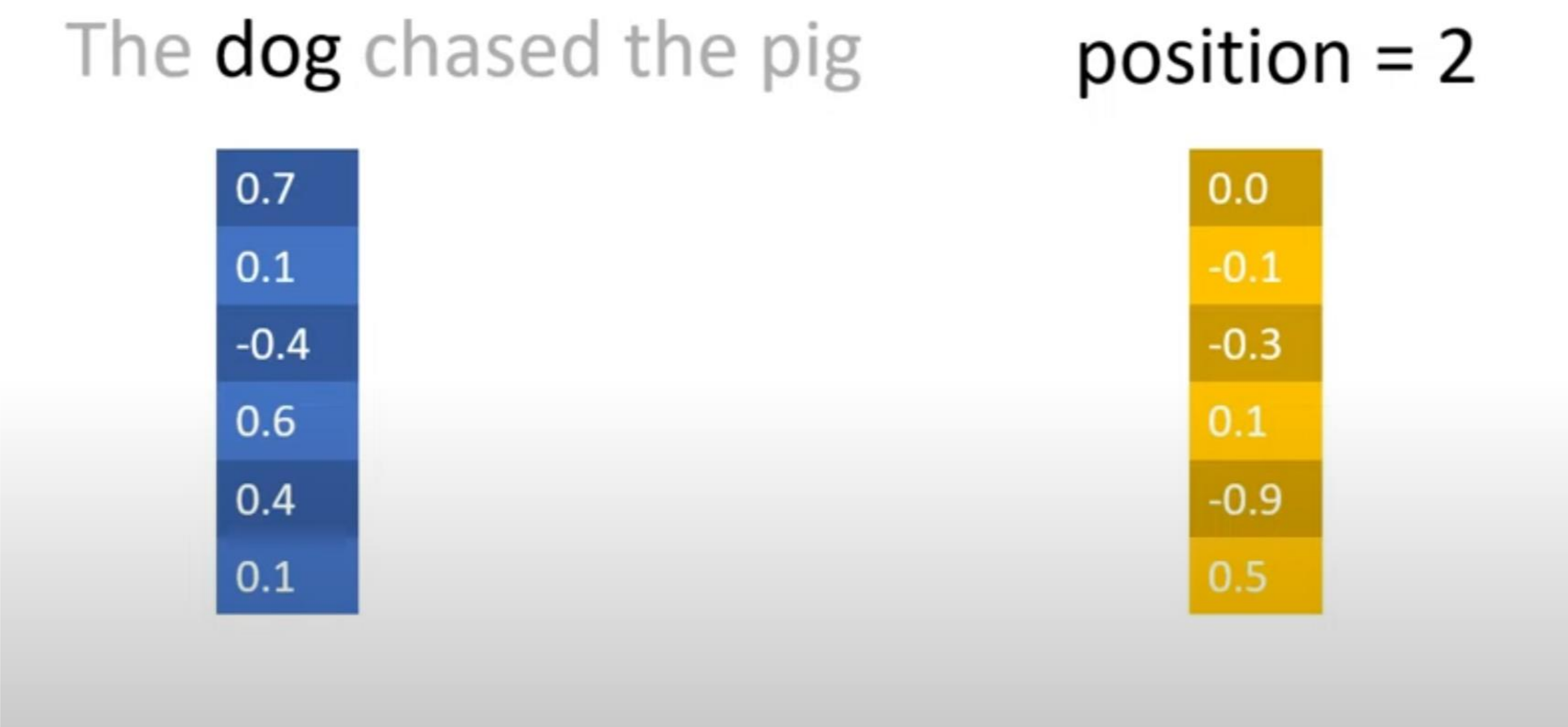
- Transformer decoders
  - Many of them
  - Really just: attentions + layernorm + MLPs + nonlinear + residual
- Word embeddings
- Position embeddings
  - Absolute embedding vs. relative embedding
- Loss function: cross entropy loss over a sequence of words

Position Embedding

- Absolute position embedding
- Relative position embedding
- Rotary position embedding



Absolute position embedding



Absolute position embedding

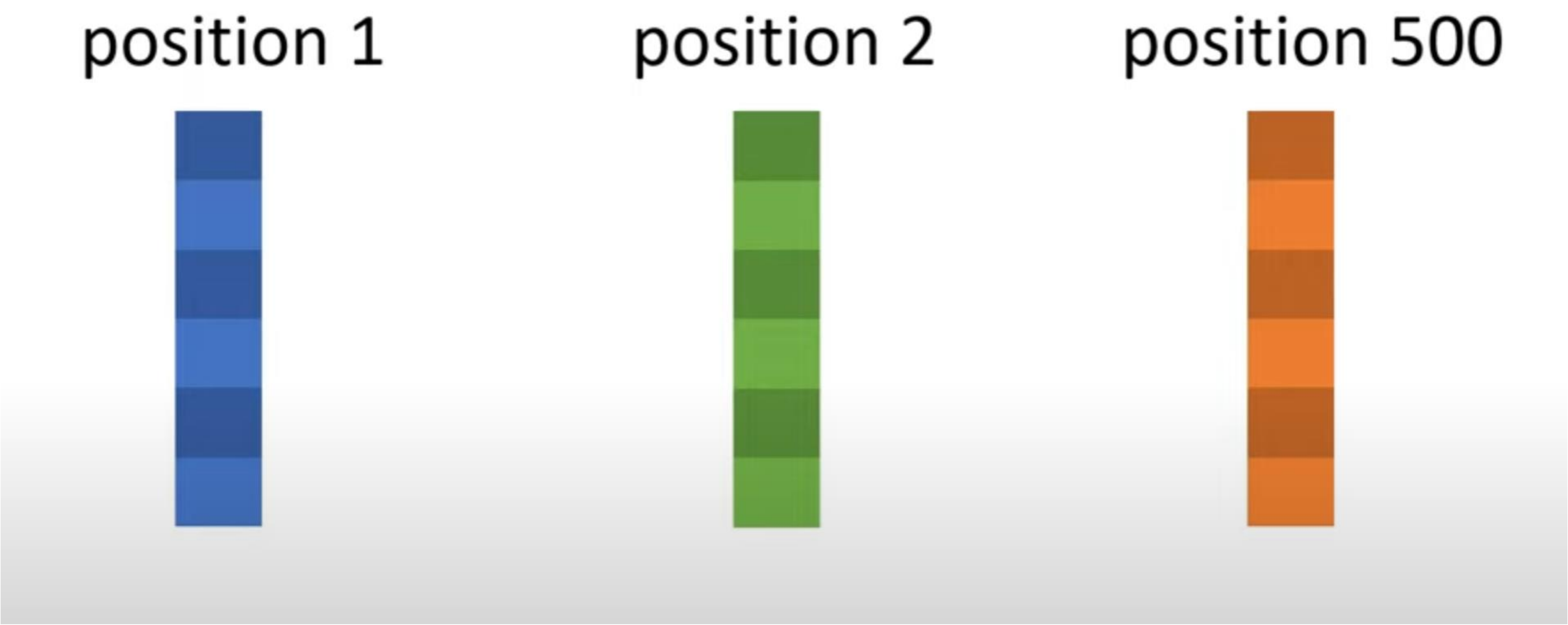
position = 2

0.0
-0.1
-0.3
0.1
-0.9
0.5

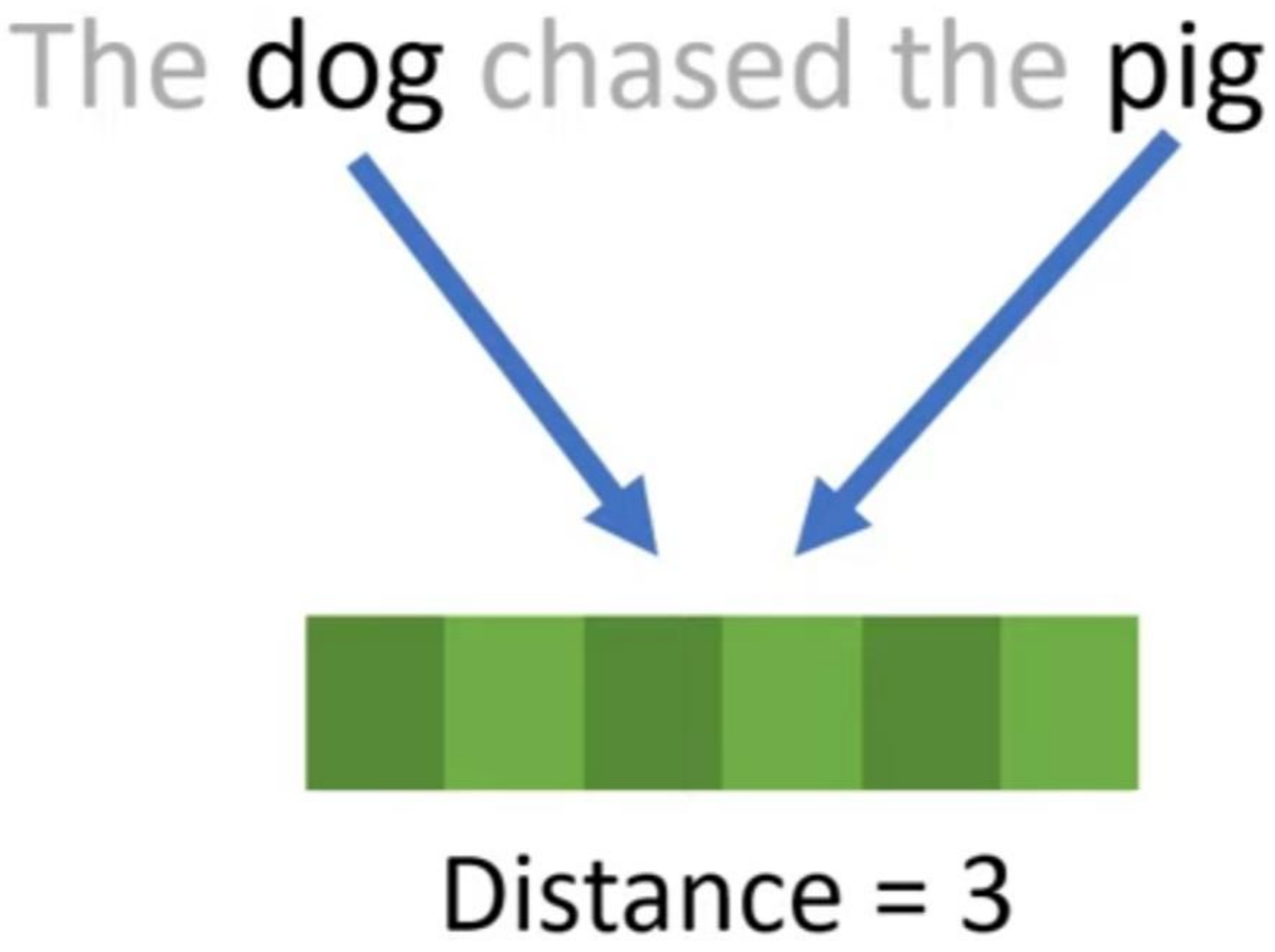
Learned from data

- Position vectors for 1-512
- Max length is bounded

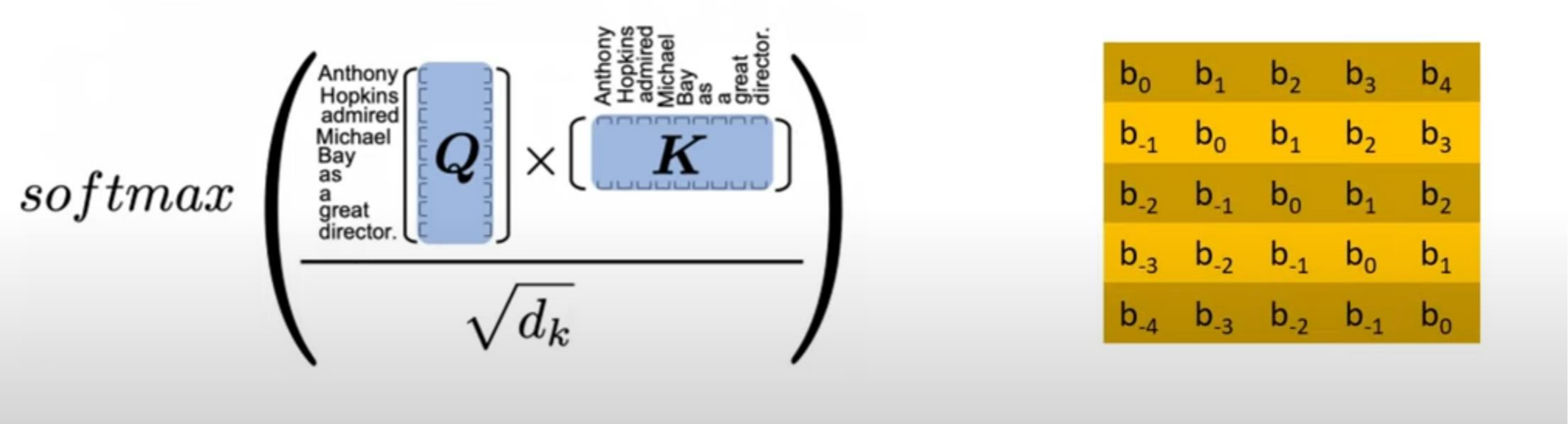
Problem?



Relative Position Embedding

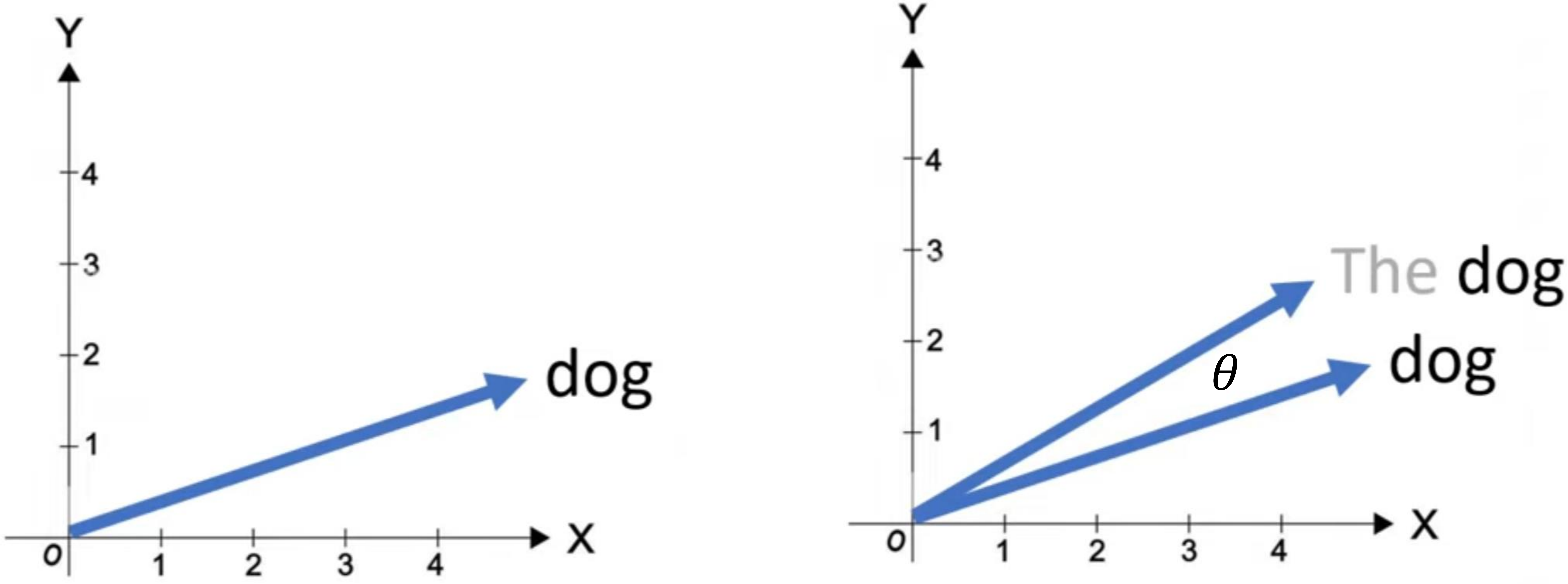


Relative Position Embedding



- Extra step in self attention
- Changes in every new token generated -> no kv cache for inference

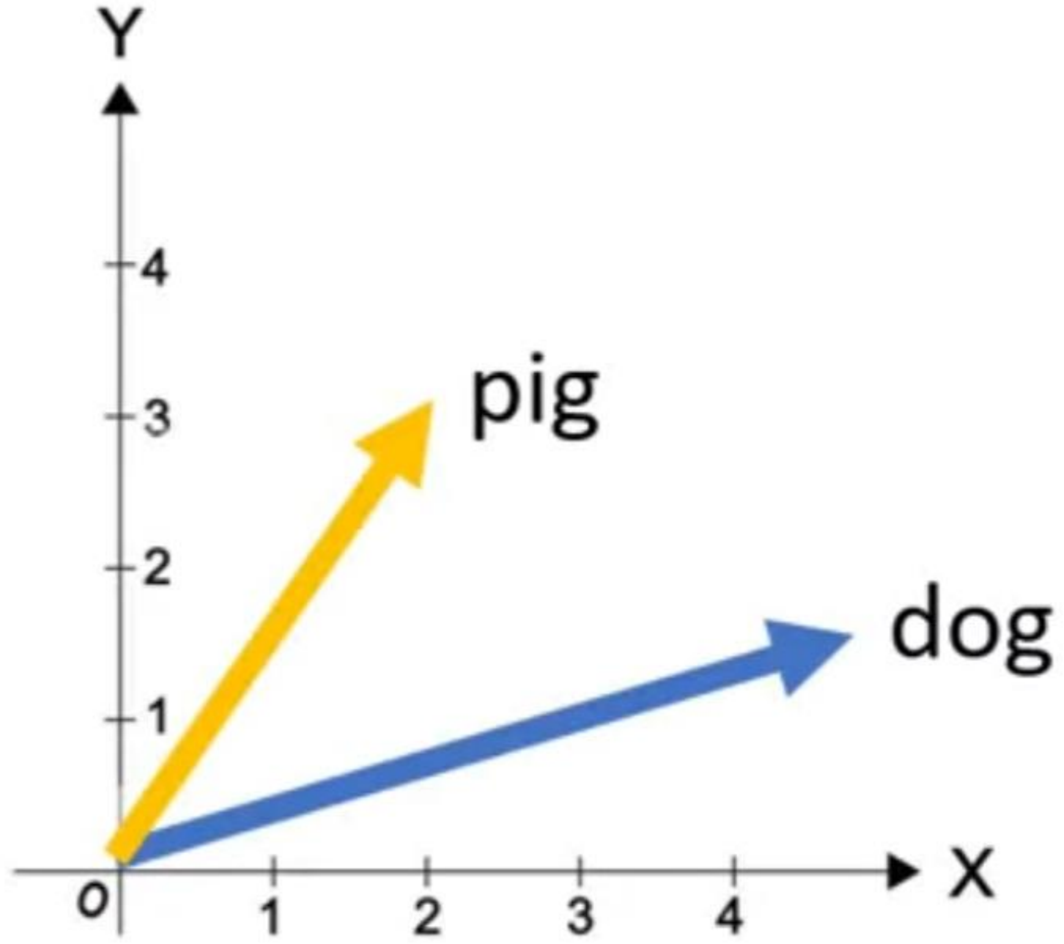
Rotary Embedding



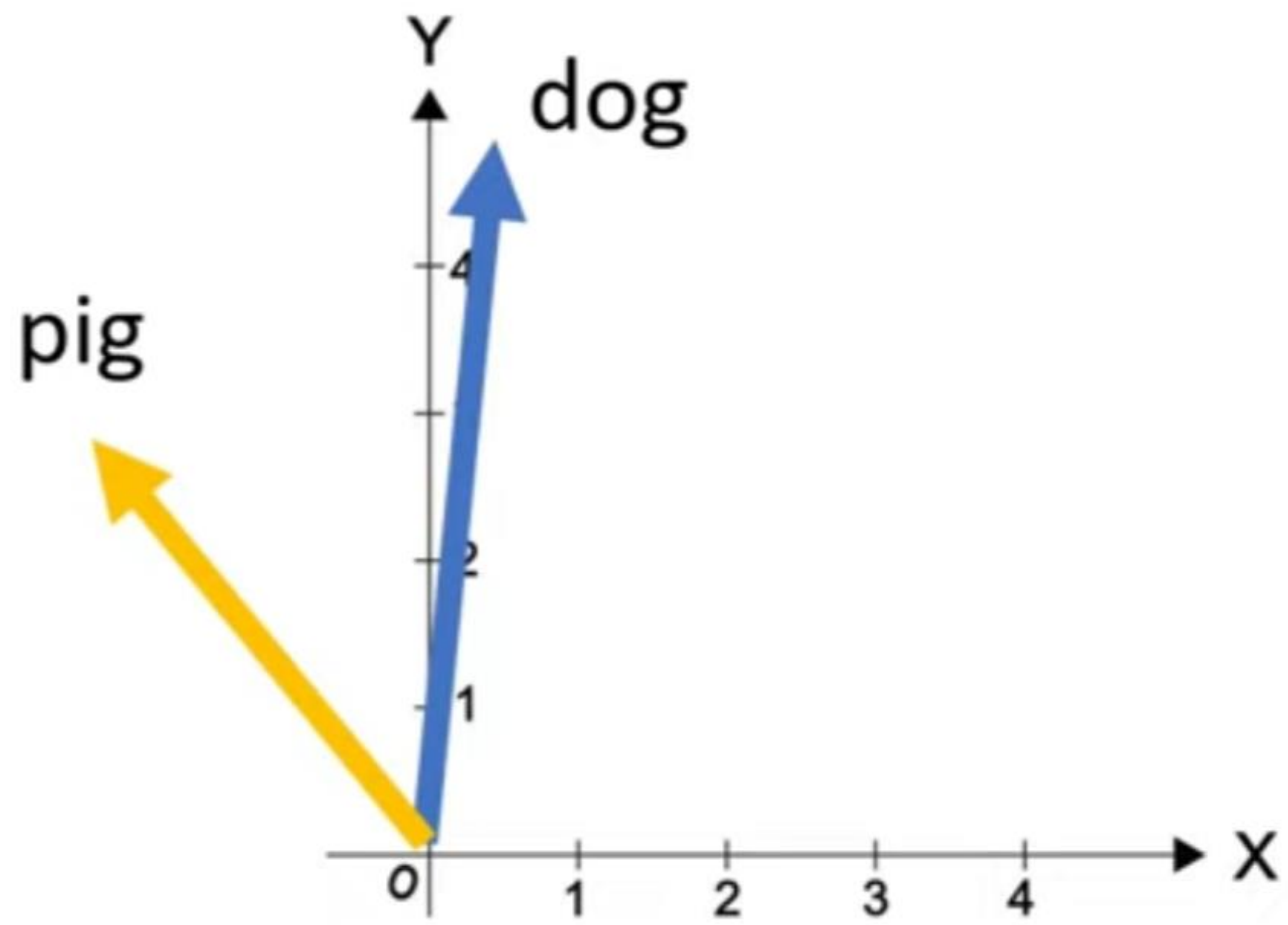
- We can cache: whatever # words can come after “dog”

Rotary Position Embedding

The **pig** chased the **dog**

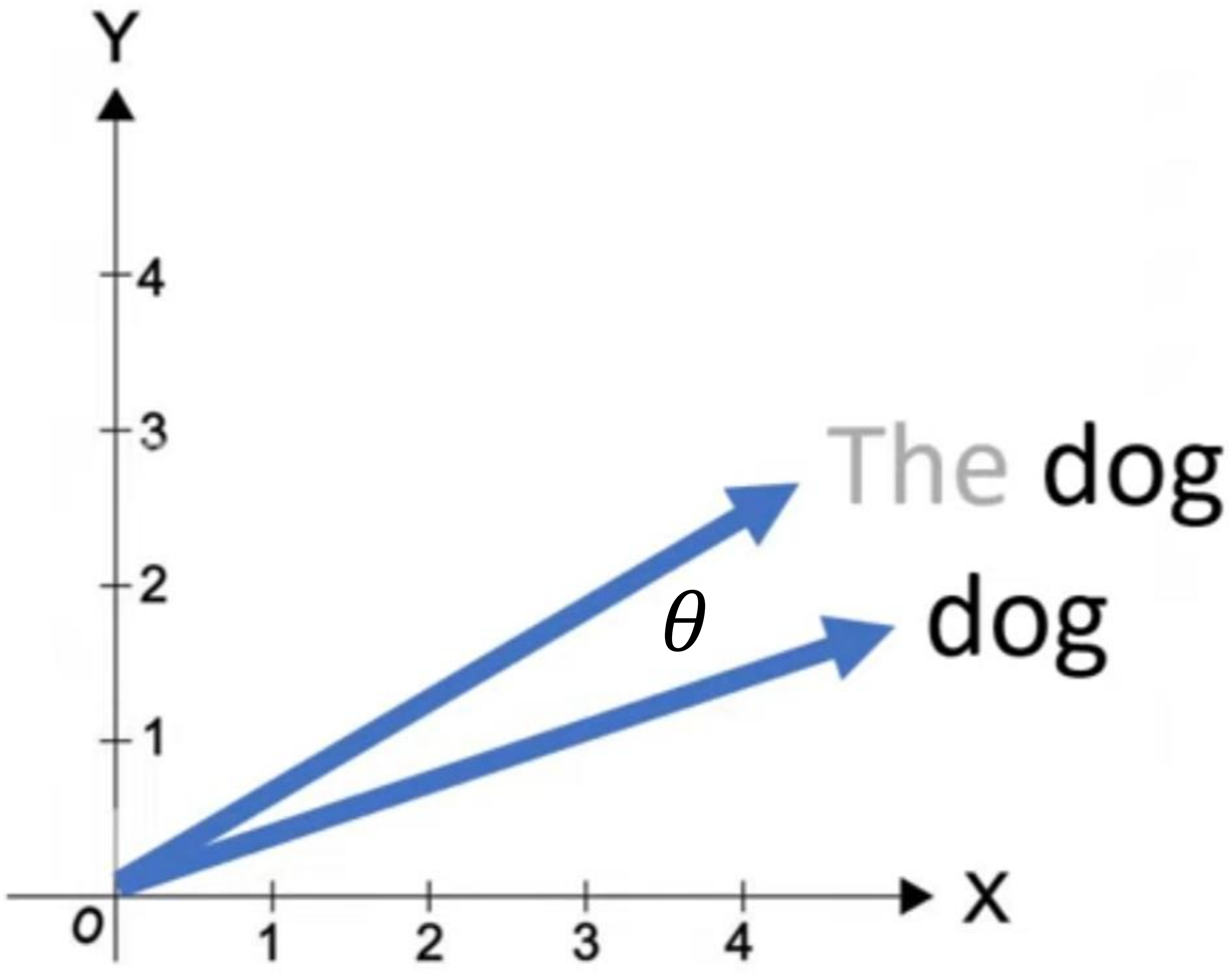


Once upon a time, the **pig** chased the **dog**



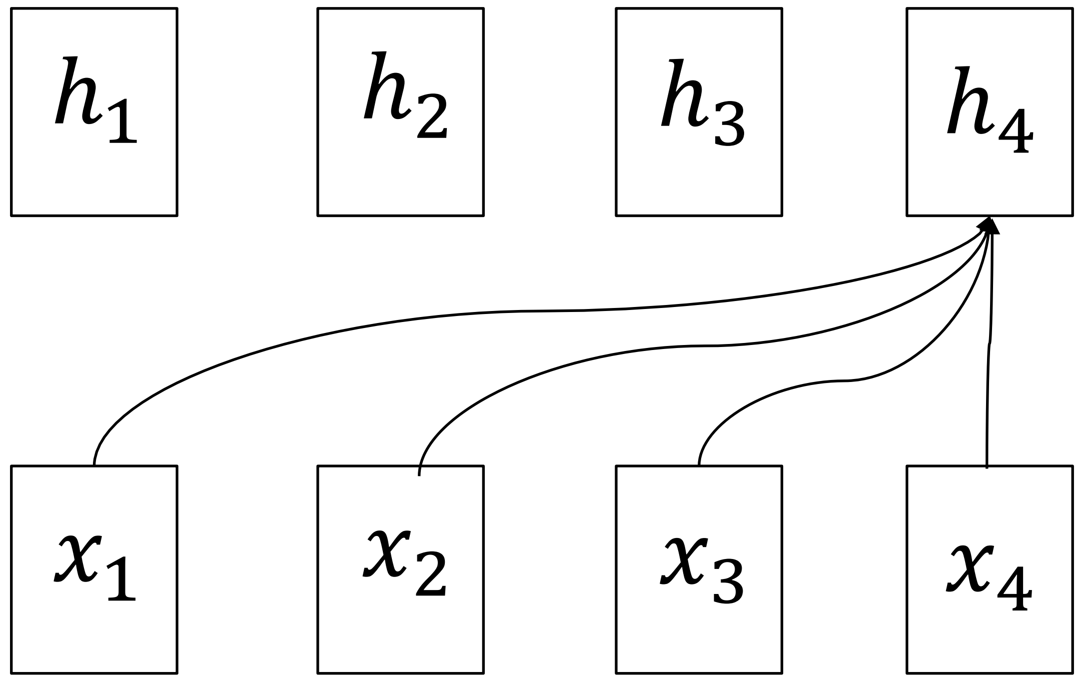
Rotary Position Embedding

- Position Interpolation is important for long sequence



Training LLMs

- Sequences are **known a priori**
- For each poistion, look at [1, 2, …, t-1] words to predict word t, and calculate the loss at t
- Parallelize the computation on all t using masking



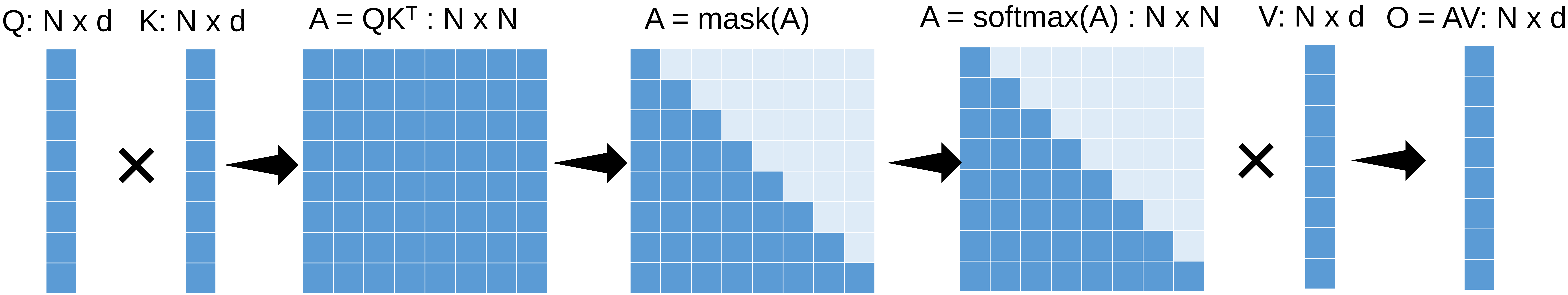
A few Important Problems (will be HW3)

- How to estimate the number of parameters of an LLM?
  - Embedding: position + word
  - Transformers layers:
    - attention Wq,Wk,Wv
    - MLP: up project, down project
    - Layernorm parameters
- How to estimate the flops needed to train an LLM?
- How to estimate the memory needed to train a transformer?

Where We Are: LLMs

- Transformers and Attentions
- LLM Training Optimizations
  - **Flash attention**
  - 3D parallelism
- LLM Inference and Serving
  - Paged attention
  - Continuous batching
  - Speculative decoding
- Scaling Laws
- Long context

Attention: $O = \mathrm{Softmax}(QK^T)\, V$



Attention Computation

Algorithm 0 Standard Attention Implementation

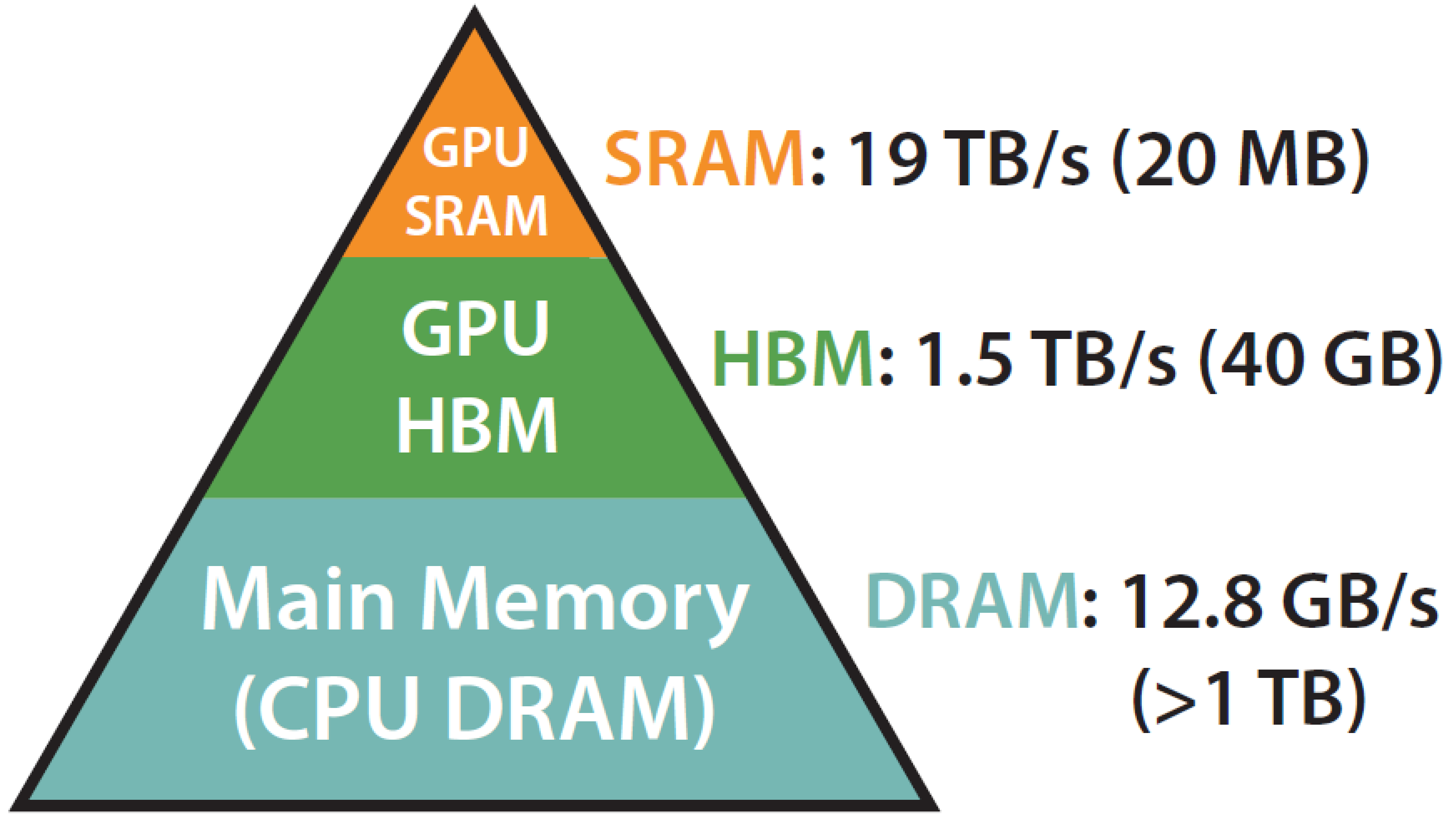
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Challenges:

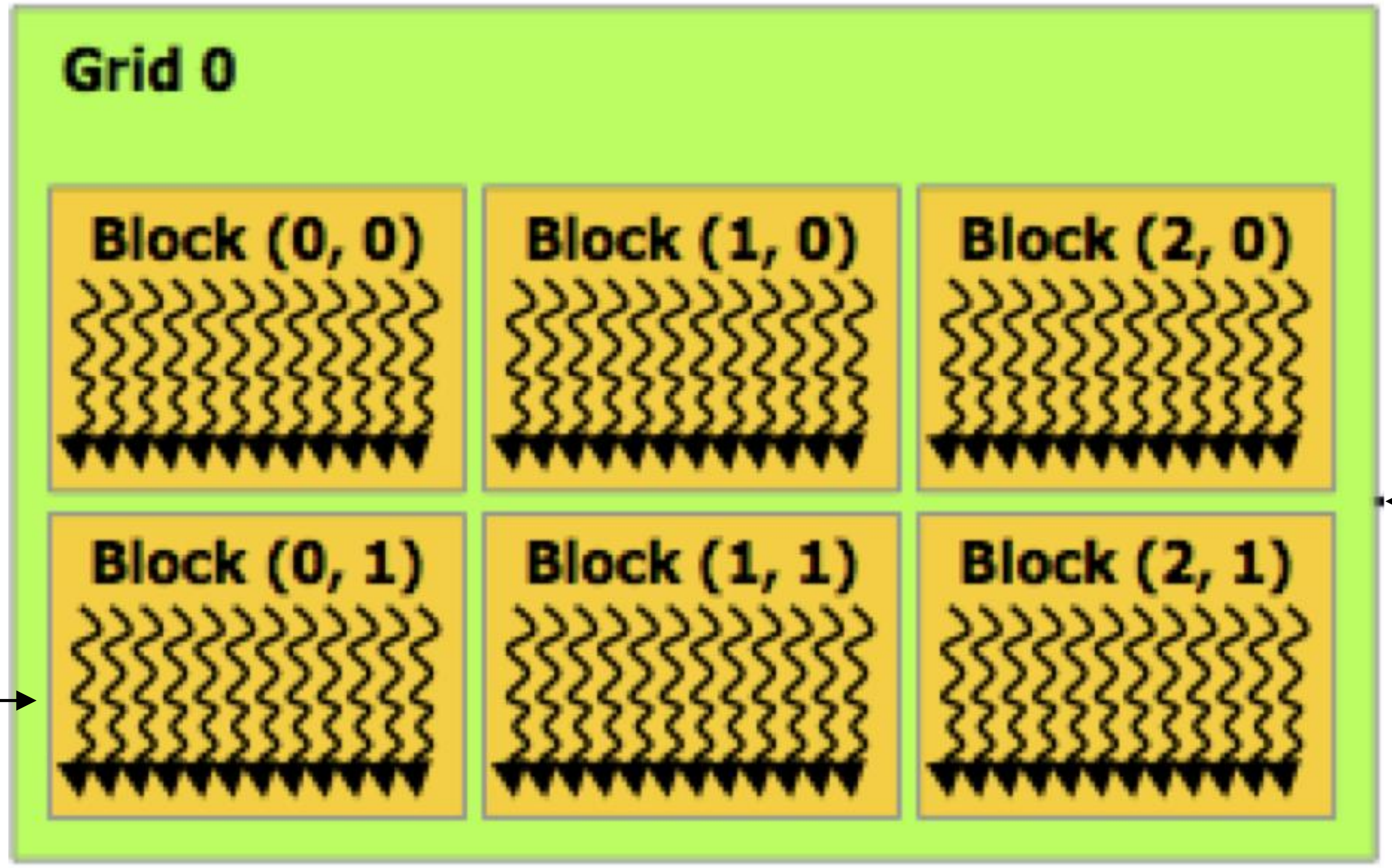
- Large intermediate results
- Repeated reads/writes from GPU device memory
- Cannot scale to long sequences due to O(N^2) intermediate results

Revisit: GPU Memory Hierarchy



Memory Hierarchy with Bandwidth & Memory Size

Revisit: GPU Memory Hierarchy



Per-block **shared memory** (readable/writable by all threads in a block)

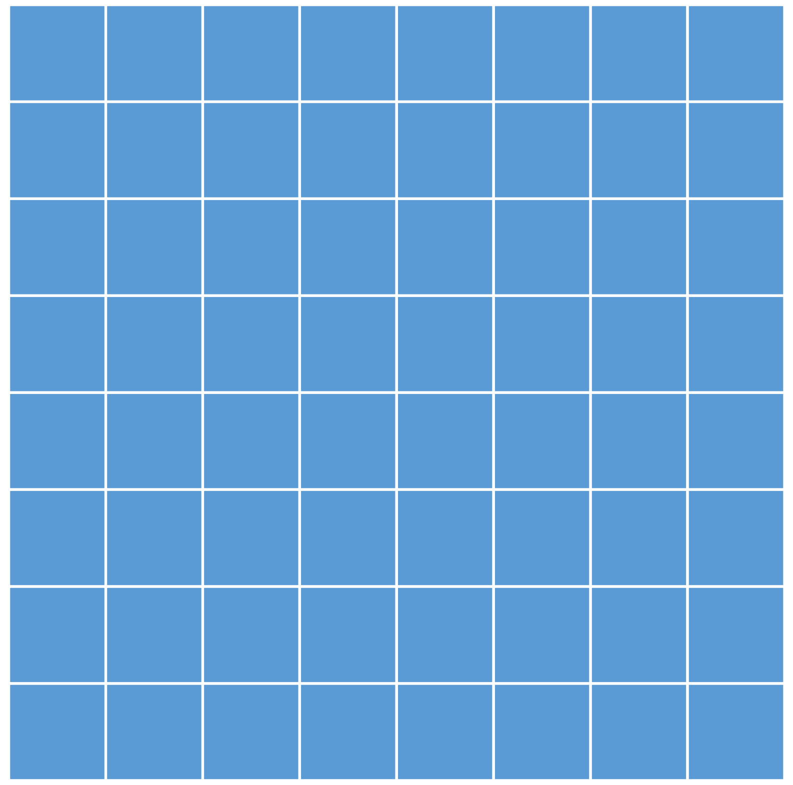
19 TB/s (20 MB)

Device global memory (readable/writable by all threads)

1.5 TB/s (80 GB)

FlashAttention

$A = \text{softmax}(QK^T)$



Key idea: compute attention by blocks to reduce global memory access

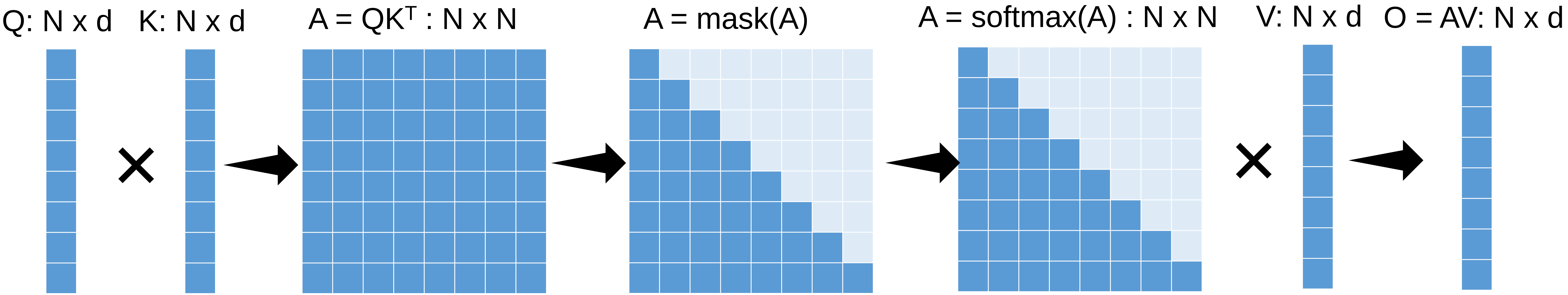
Two main Techniques:

1. **Tiling:** restructure algorithm to load query/key/value block by block from global to shared memory

2. **Recomputation:** don't store attention matrix from forward, recompute it in backward

* FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Problem: How to tile softmax?



Challenges

- We must avoid materializing NxN while still get the precise softmax results
- Compute softmax reduction w/o access to NxN
- Backward without the NxN softmax input

How to Implement Softmax

Algorithm 1 Naive softmax

1: $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do**
3: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do**
6: $\quad y_i \leftarrow \frac{e^{x_i}}{d_V}$
7: **end for**

Problem

- Can easily go overflow because of sum (e^x)

Safe Softmax

$$y_i = \frac{e^{x_i - \max_{k=1}^{V} x_k}}{\sum_{j=1}^{V} e^{x_j - \max_{k=1}^{V} x_k}}$$

Algorithm 2 Safe softmax

$m_0 \leftarrow -\infty$
for $k \leftarrow 1, V$ **do**
    $m_k \leftarrow \max(m_{k-1}, x_k)$
end for
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
    $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
end for
for $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

Online, Safe Softmax

Algorithm 3 Safe softmax with online normalizer calculation

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
    $m_j \leftarrow \max\left(m_{j-1}, x_j\right)$
    $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
end for
for $i \leftarrow 1, V$ **do**
    $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

Online blockwise softmax

Algorithm 3 Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j - m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: **end for**

$$softmax([A_1, A_2]) = [\alpha \times softmax(A_1), \beta \times softmax(A_2)]$$

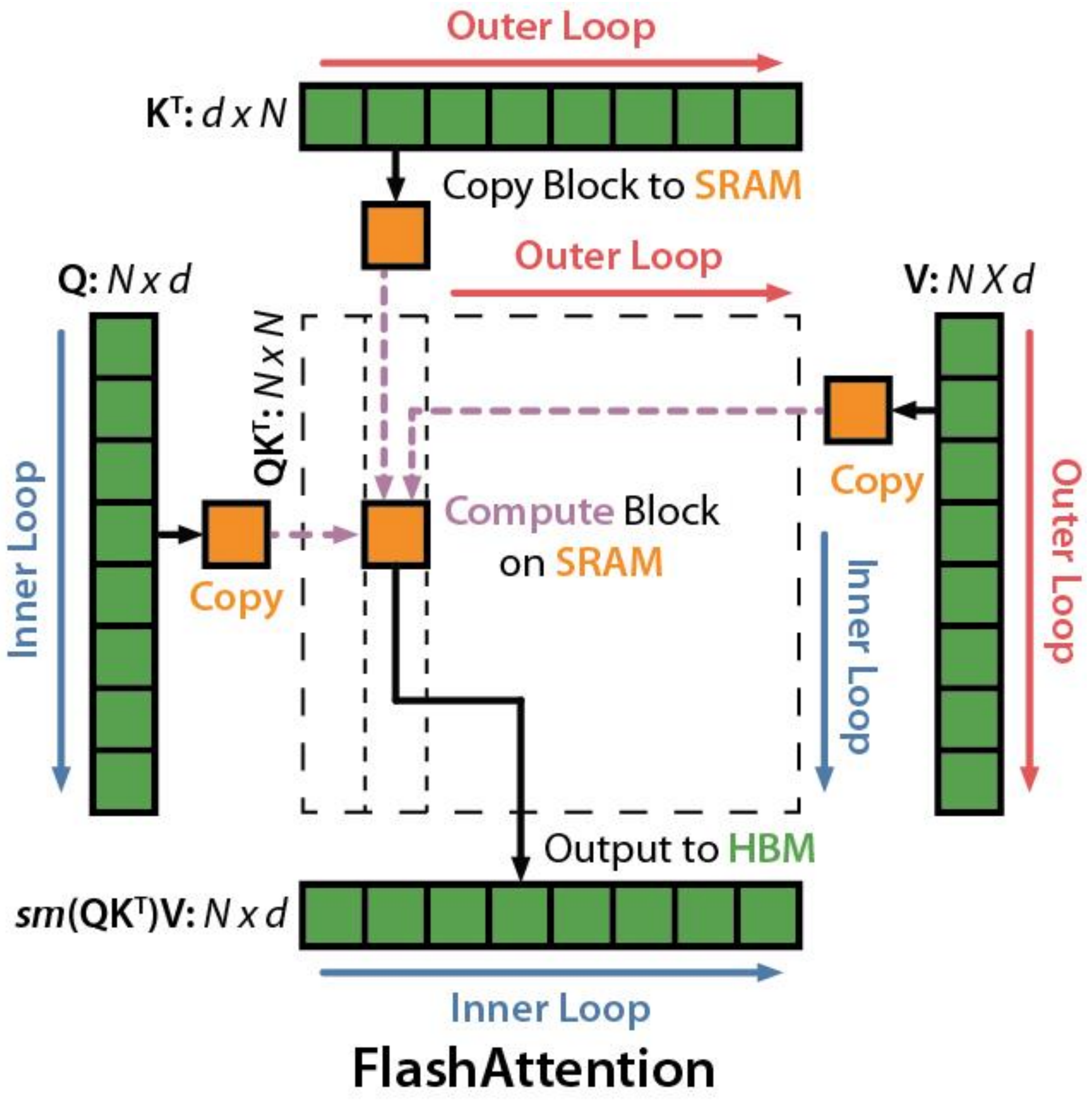
$$softmax([A_1, A_2]) \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha \times softmax(A_1)V_1 + \beta \times softmax(A_2)V_2$$

Tiling: Decompose Large Softmax into smaller ones by Scaling

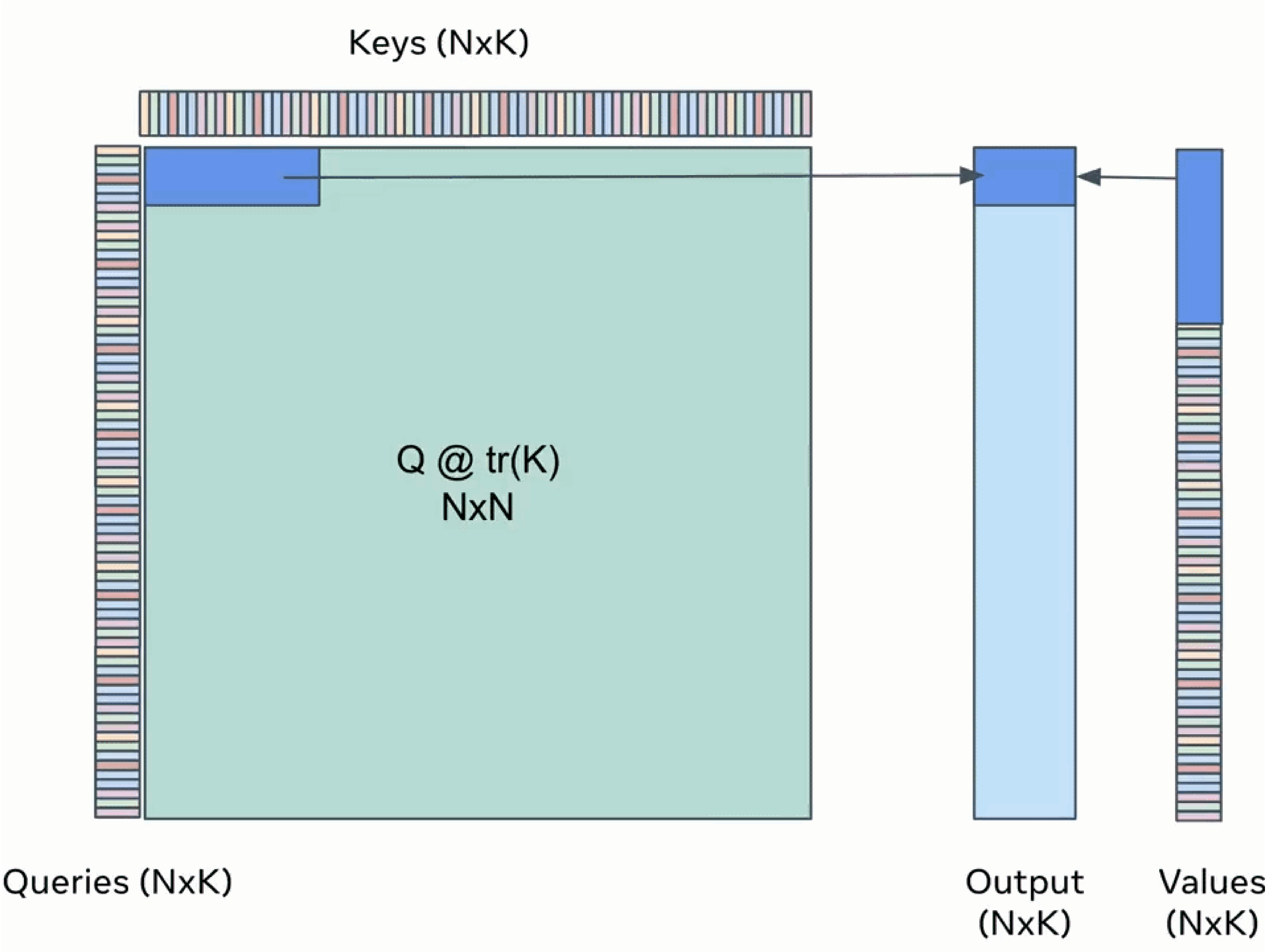
1. Load inputs by blocks from global to shared memory
2. On chip, compute attention output wrt the block
3. Update output in device memory by scaling

$$softmax([A_1, A_2])\begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$$
$$= \alpha \times softmax(A_1)V_1 + \beta \times softmax(A_2)V_2$$

$$softmax([A_1, A_2]) = [\alpha \times softmax(A_1), \beta \times softmax(A_2)]$$



Tiling

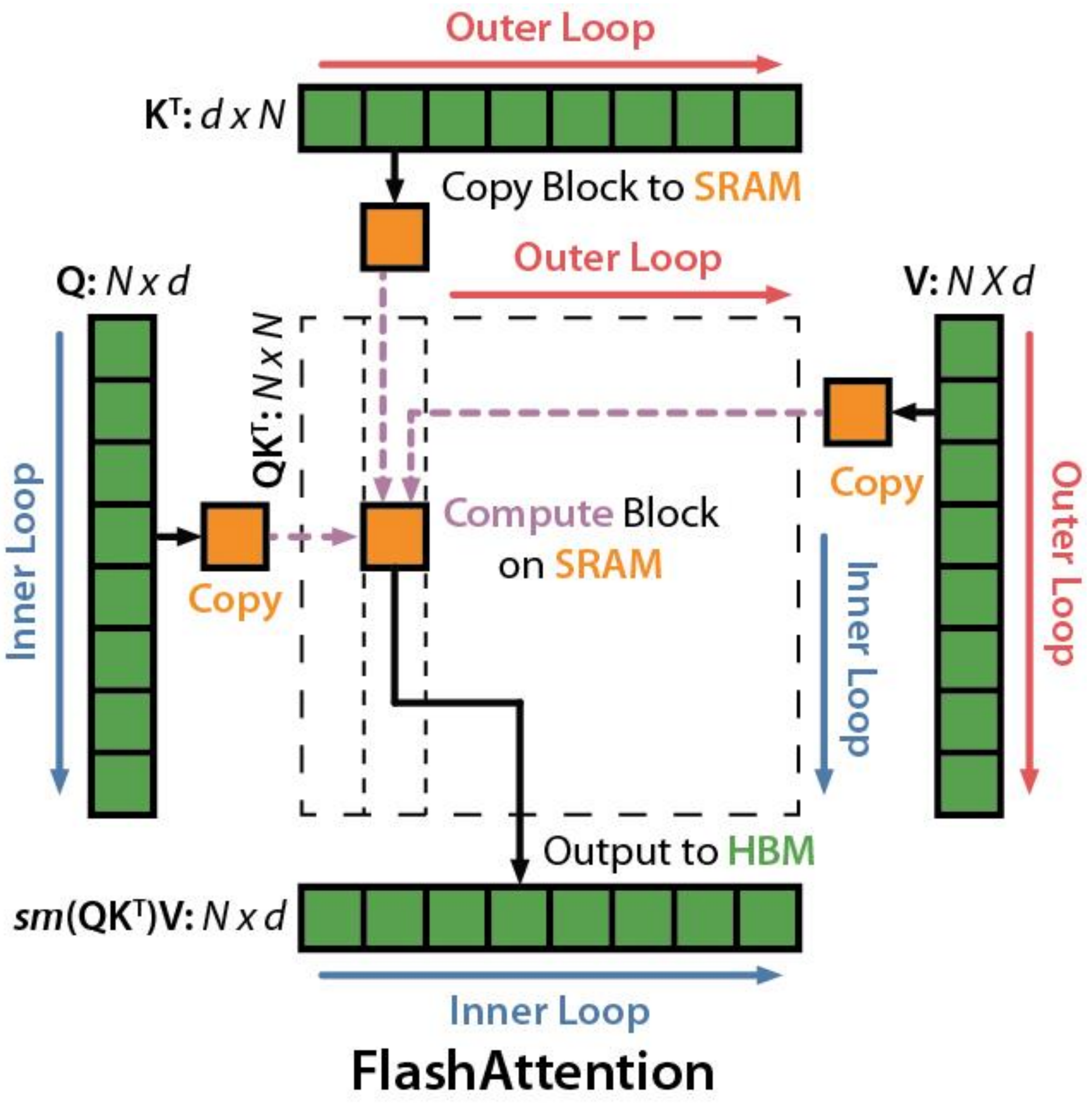


Animation credit: Francisco Massa

Recomputation: Backward Pass

By storing softmax normalization factors from forward (size N), recompute attention in the backward from inputs in shared memory

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
Global mem access	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms



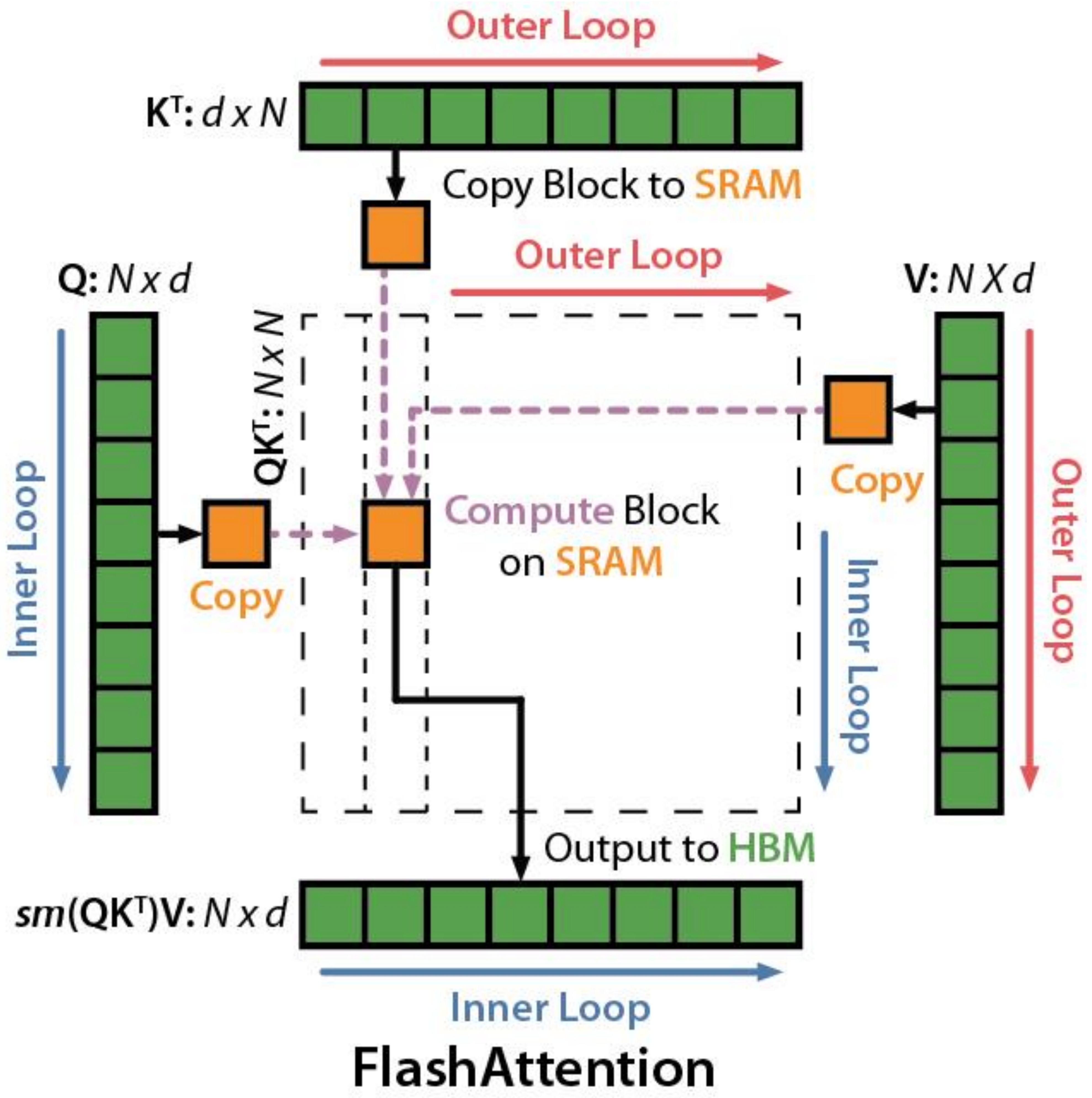
Speed up backward pass with increased FLOPs

FlashAttention: Threadblock-level Parallelism

How to partition FlasshAttention across thread blocks?

(An A100 has 108 SMMs -> 108 thread blocks)

- Step 1: assign different heads to different thread blocks (16-64 heads)

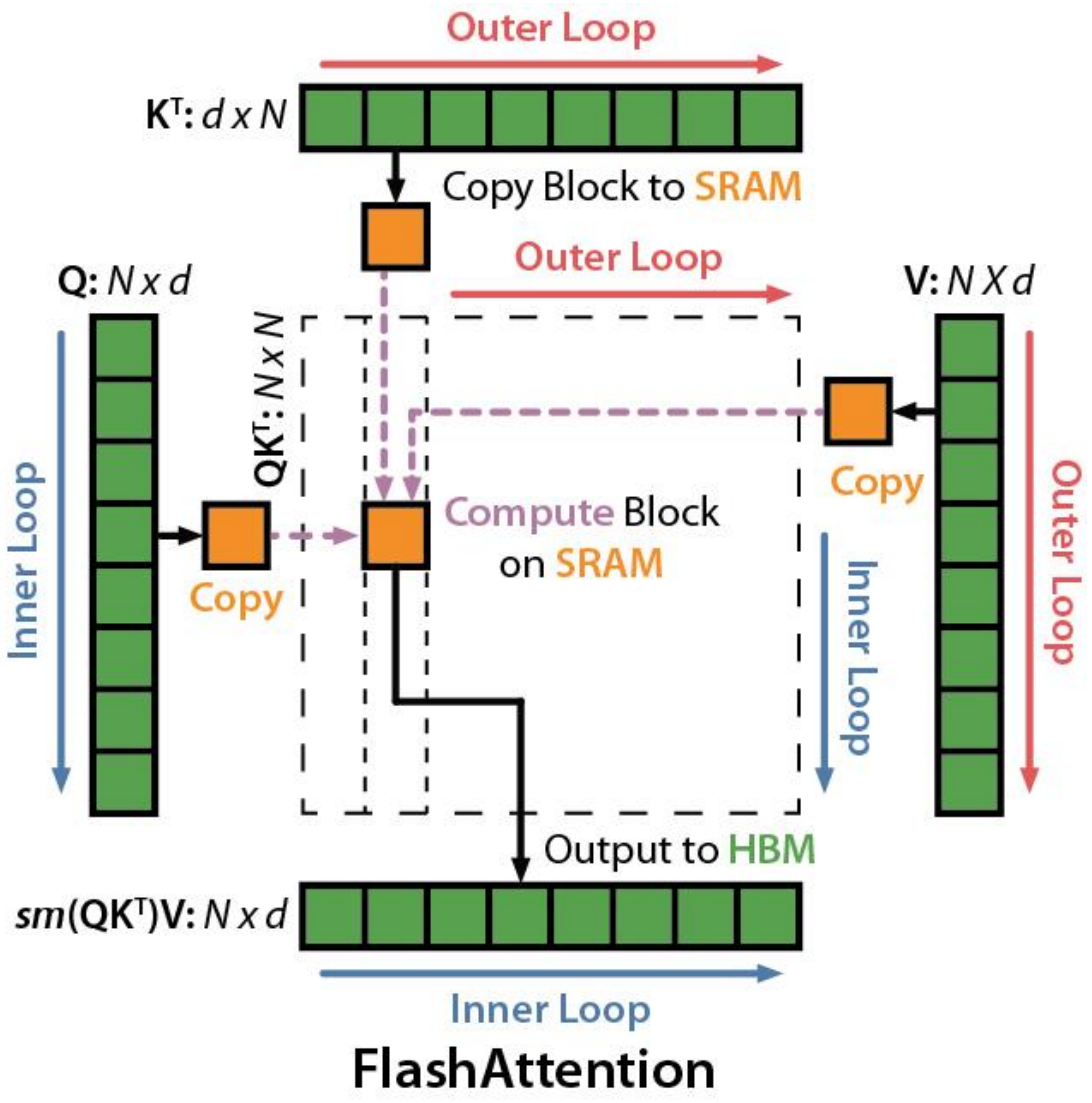


FlashAttention: Threadblock-level Parallelism

How to partition FlasshAttention across thread blocks?

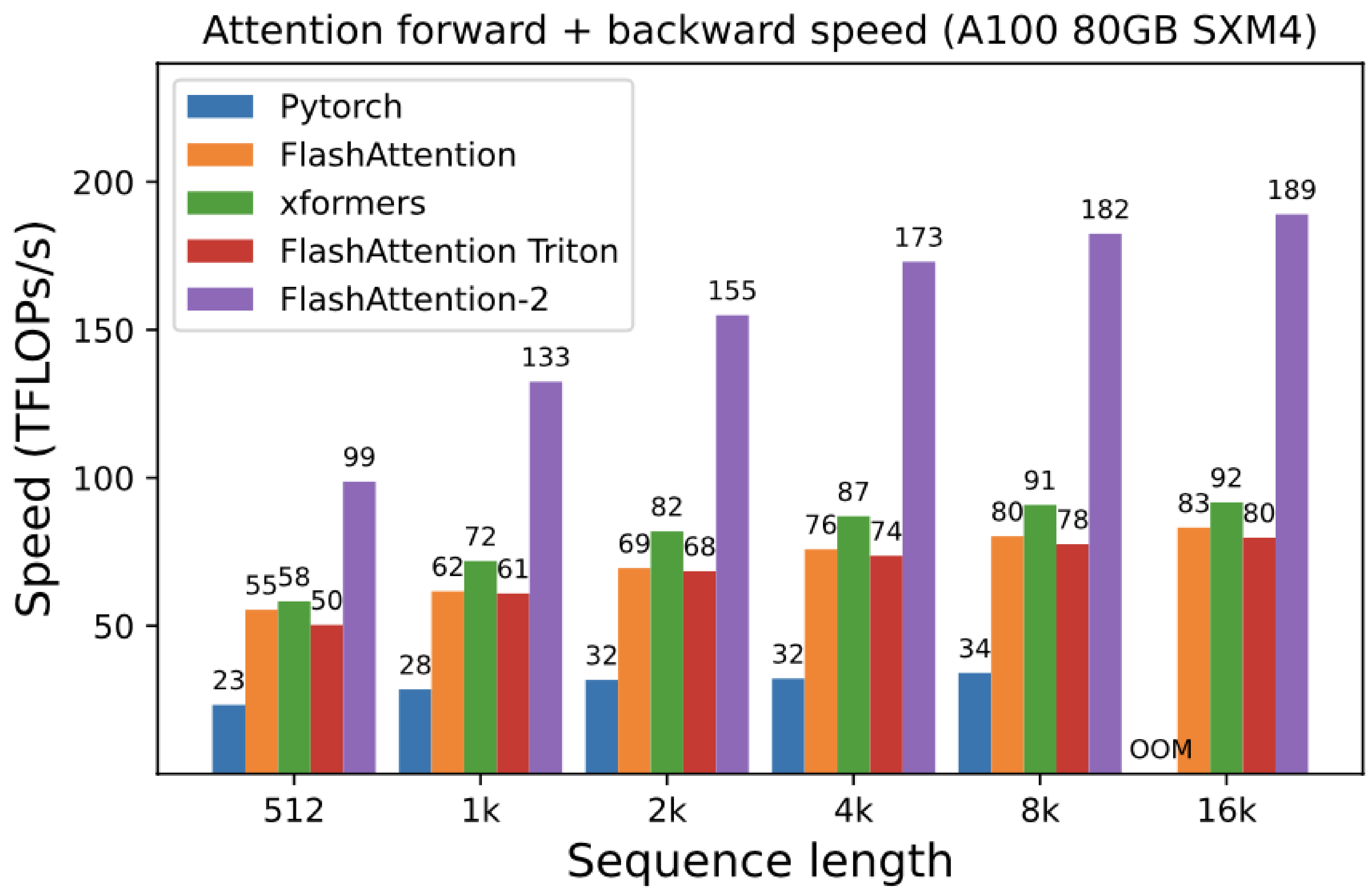
(An A100 has 108 SMMs -> 108 thread blocks)

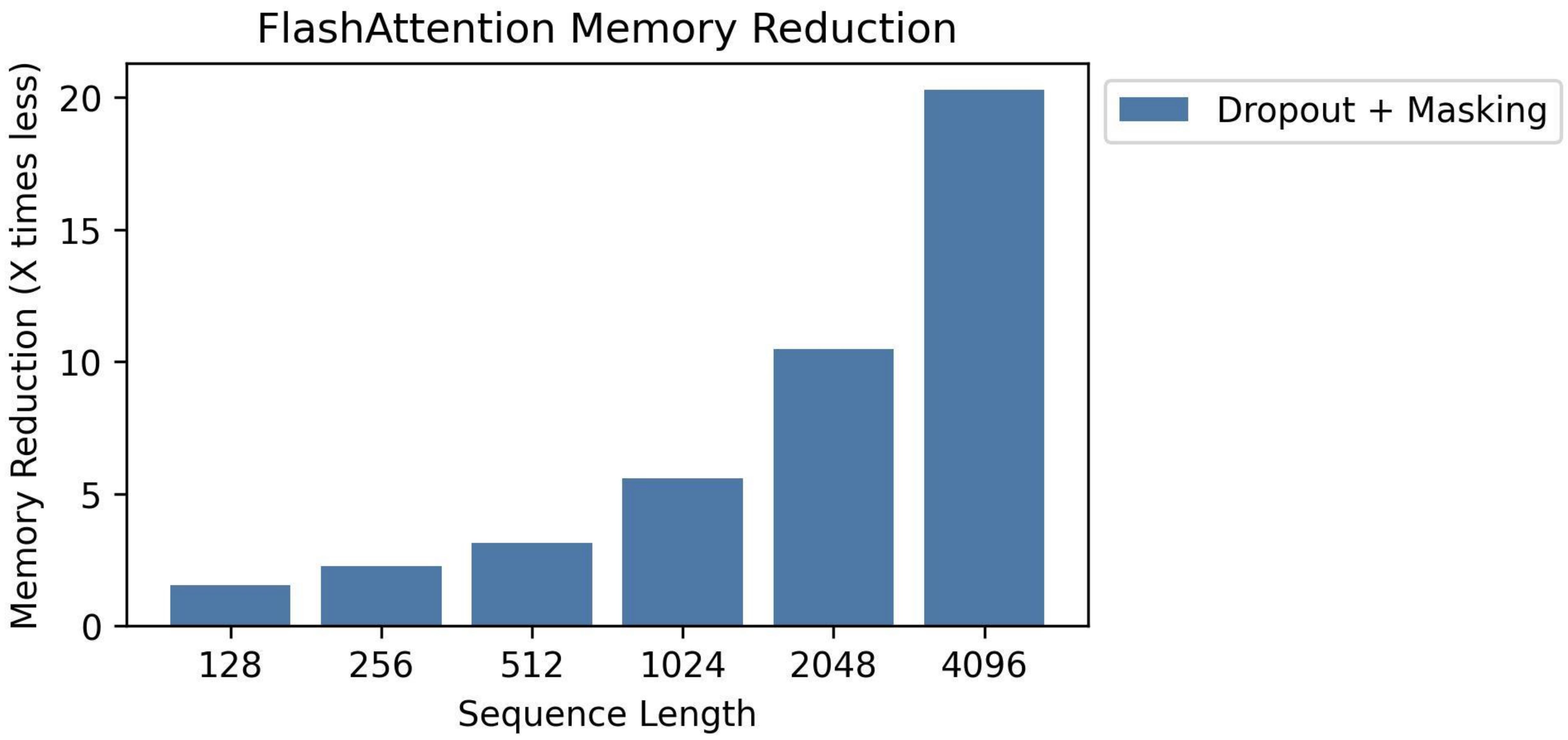
- Step 1: assign different heads to different thread blocks (16-64 heads)
- Step 2: assign different queries to different thread blocks (Why?)



Thread blocks cannot communicate; cannot perform softmax when partitioning keys/values

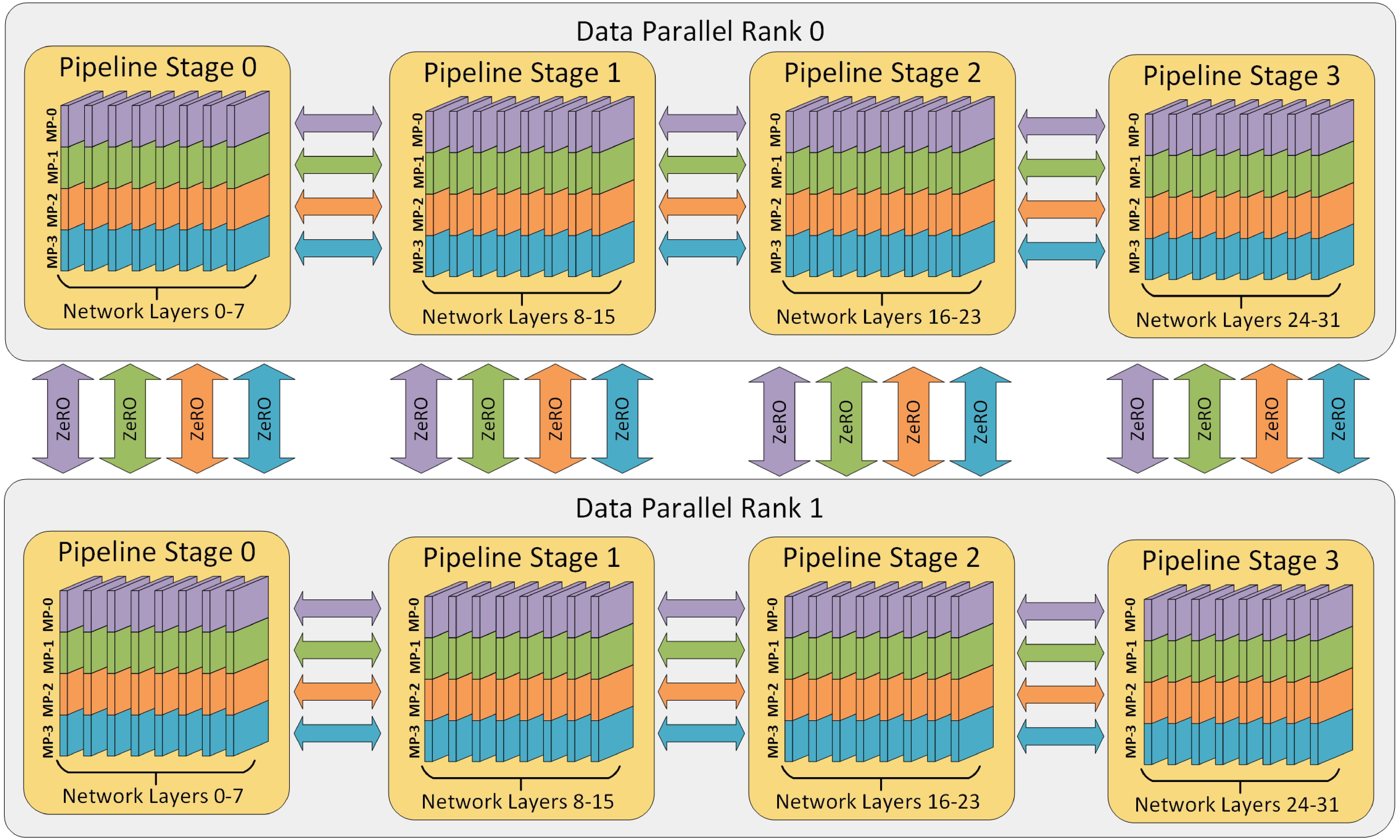
FlashAttention: 2-4x speedup, 10-20x memory reduction





Memory linear in sequence length

How LLMs are trained today



Side effects of Flash Attention

- Because we do not materialize the N x N intermiate matrix, we decrease peak memory

- Because of decreased peak memory, we can use a larger micro batch size (significantly larger, e.g., 1 -> 32)

- Because of large per-device batch size, much higher AI