InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management

Group: **Ping Gong**, Jiawei Yi and Juncheng Zhang

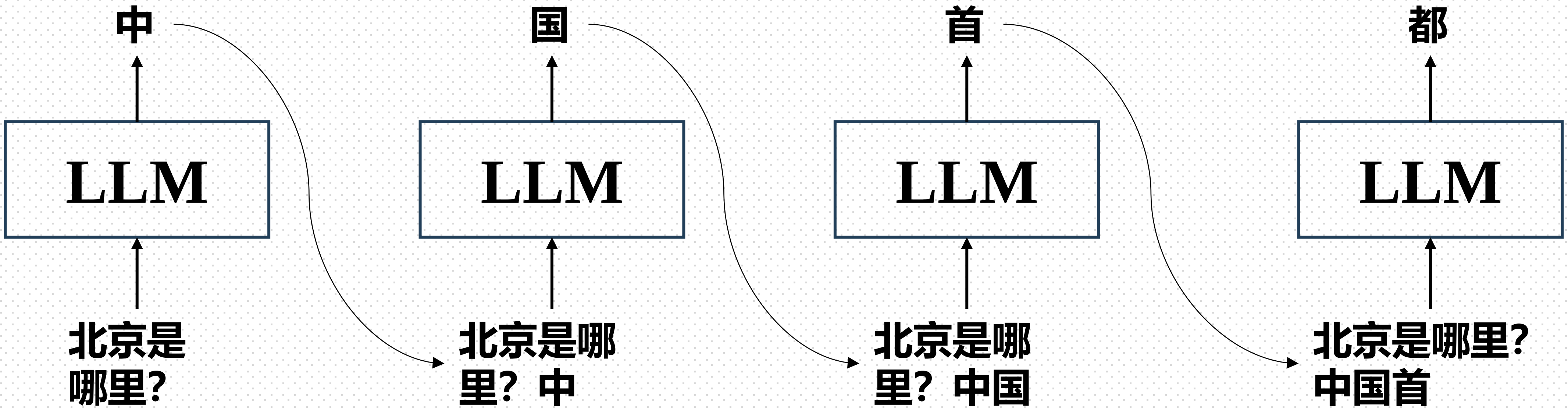
2024-10-15

Contexts

- ❑Background
- ❑Motivation
- ❑InfiniGen
- ❑Evaluations

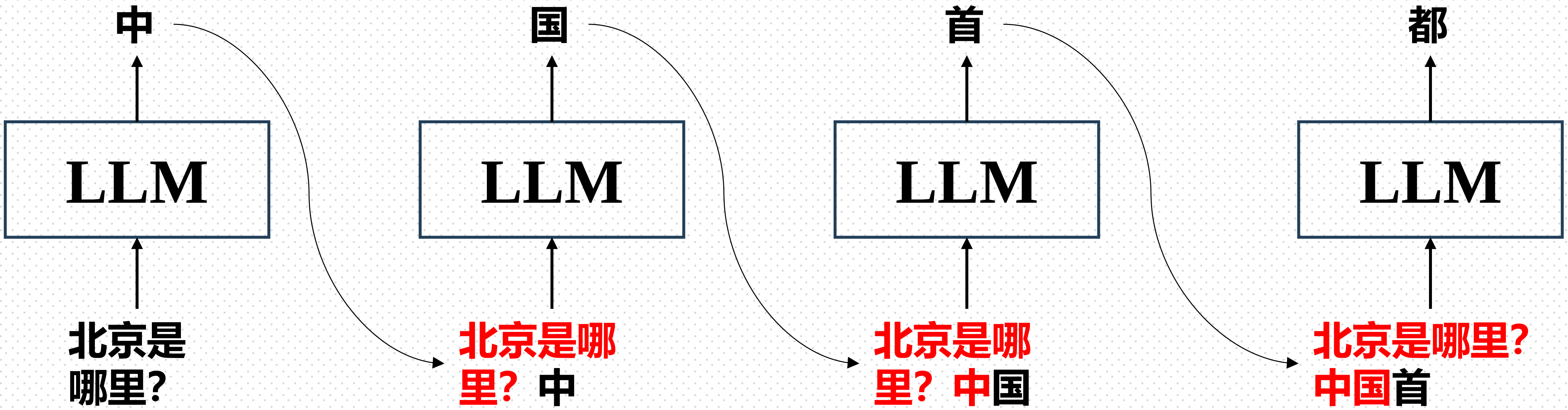
Background – LLM Inference

- ❑LLM is an autoregressive model



Background – LLM Inference

❑LLM is an autoregressive model



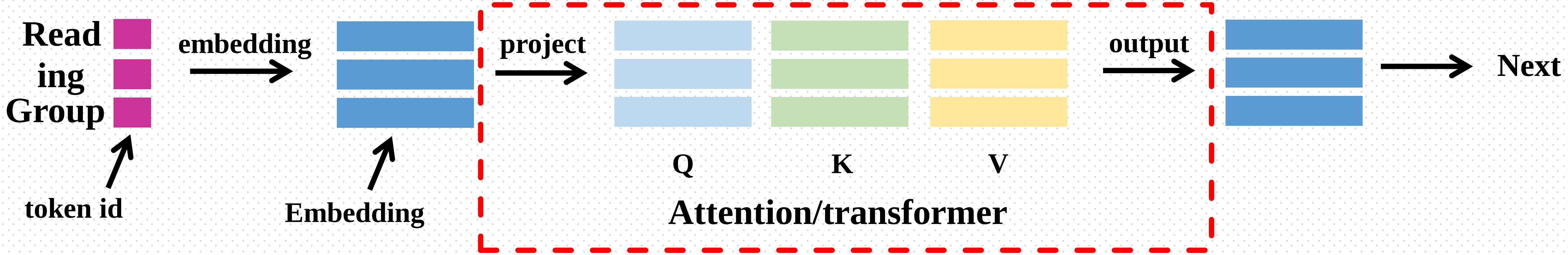
A lot of redundant computing -> KVCache (以存代算)

Background - QKV in attention/transformer

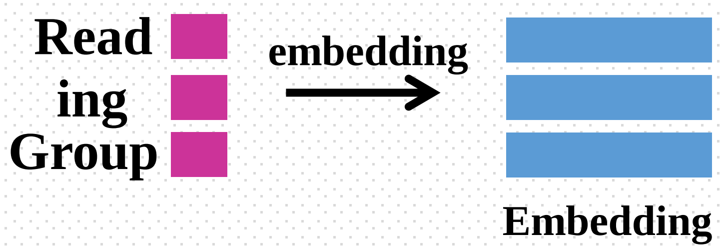
- **What's the meaning of "KV" in "KV Cache"?**
  - **Differs from KV store in storage system**
  - **Intermediate result in LLM inference (Key tensor, Value tensor)**

Background - QKV in attention/transformer

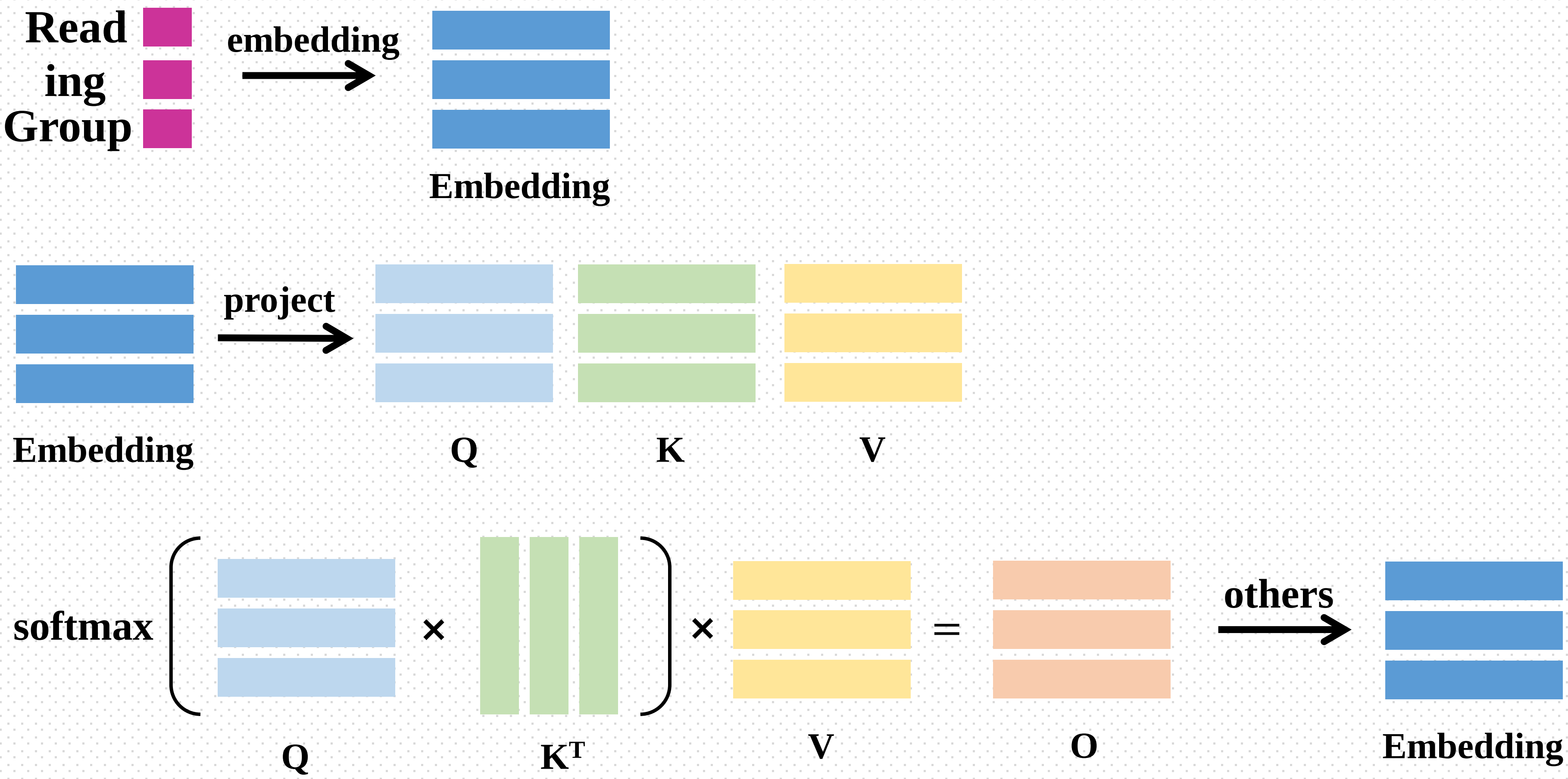
- ❑ What's the meaning of "KV" in "KV Cache"?
  - ❖ Differs from KV store in storage system
  - ❖ Intermediate result in LLM inference (Key tensor, Value tensor)
- ❑ QKV in attention/transformer



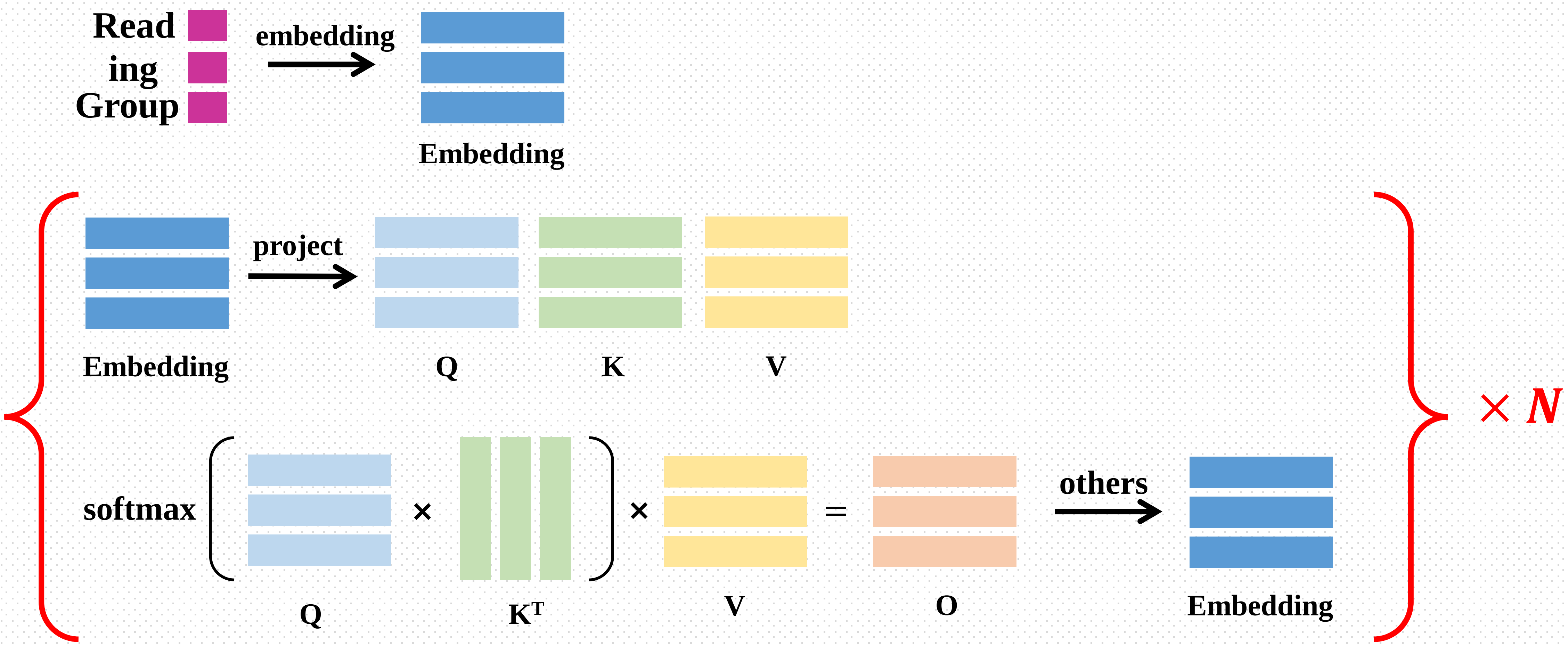
Background - LLM inference



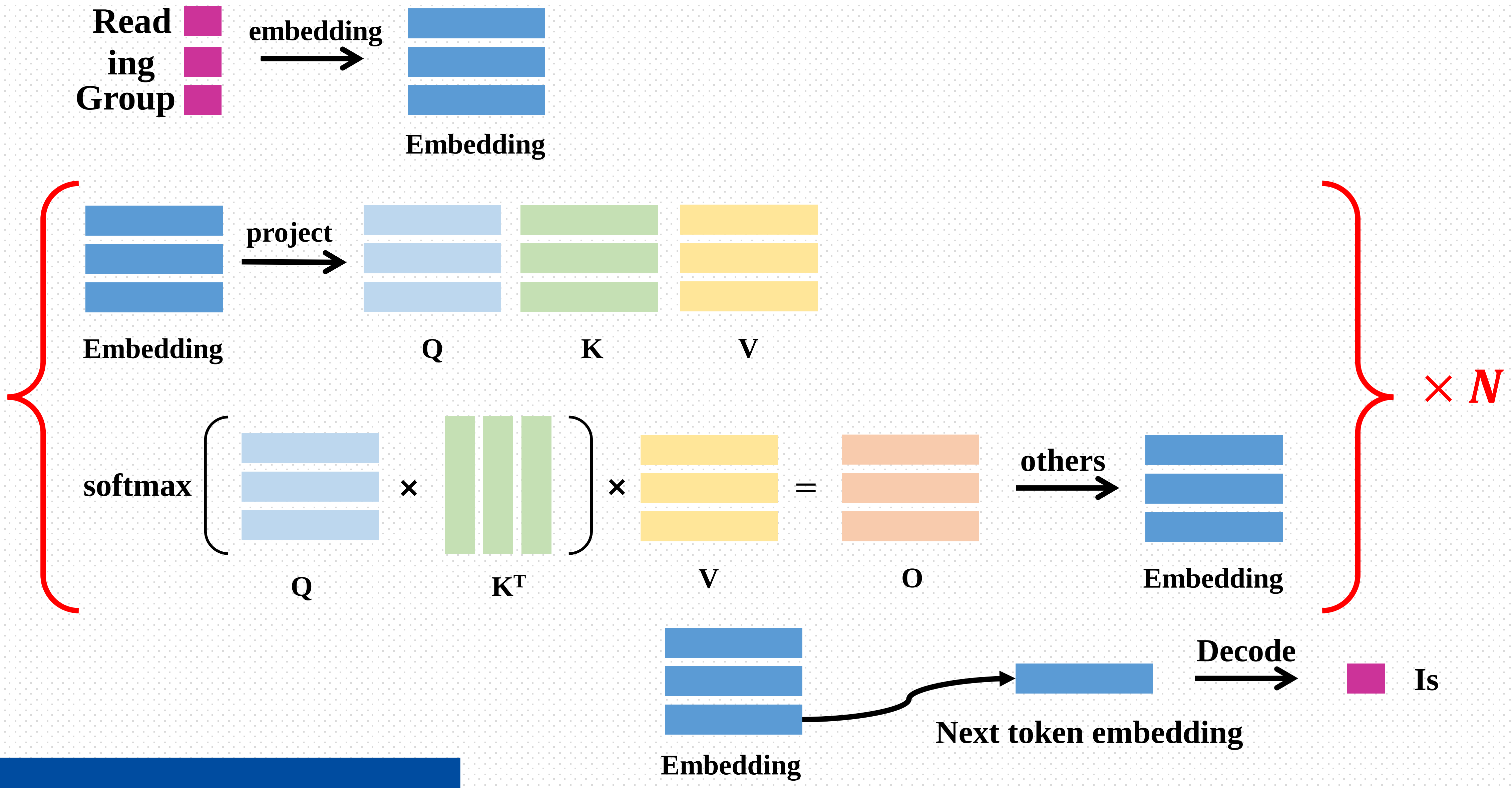
Background - LLM inference



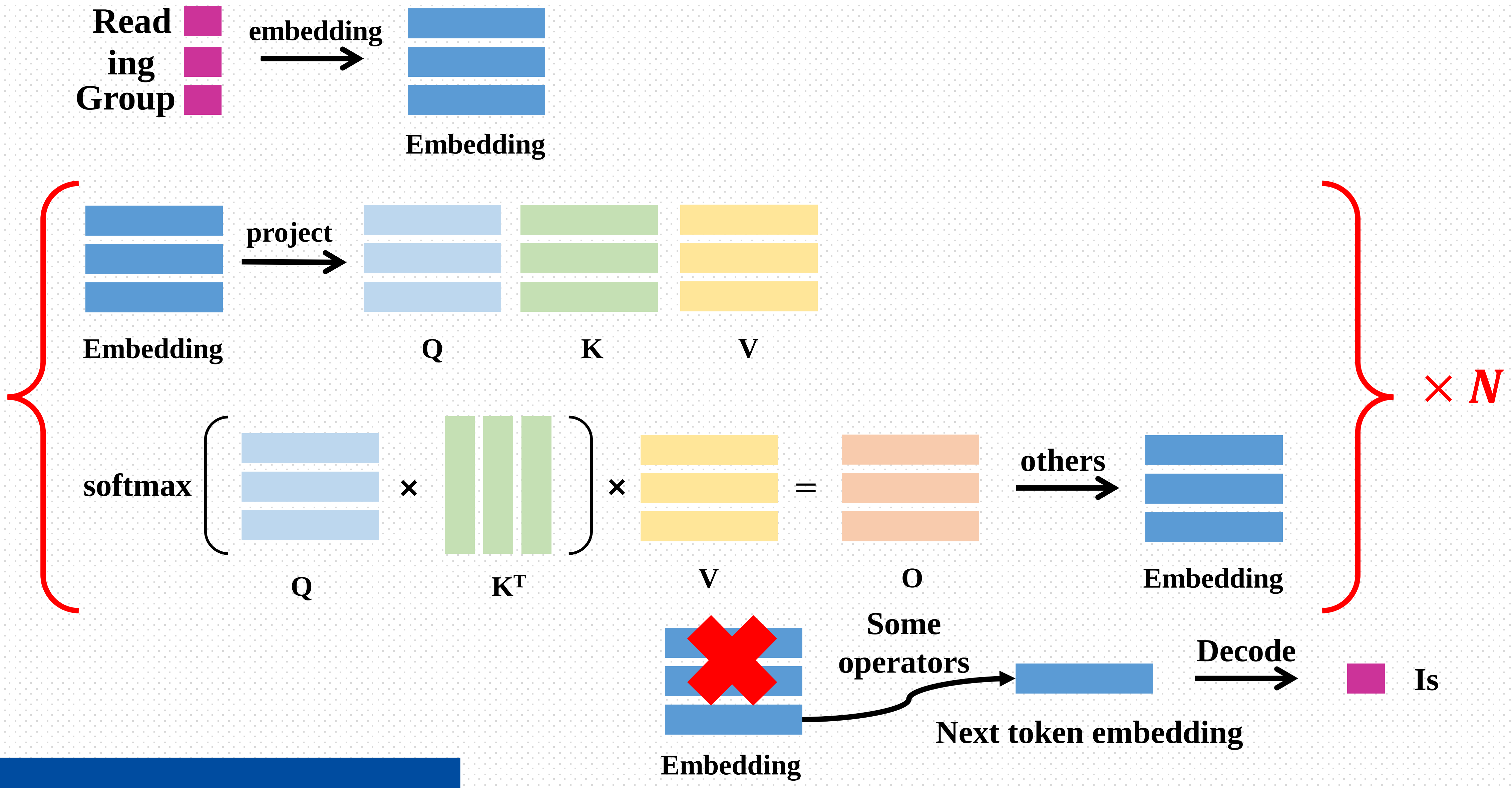
Background - LLM inference



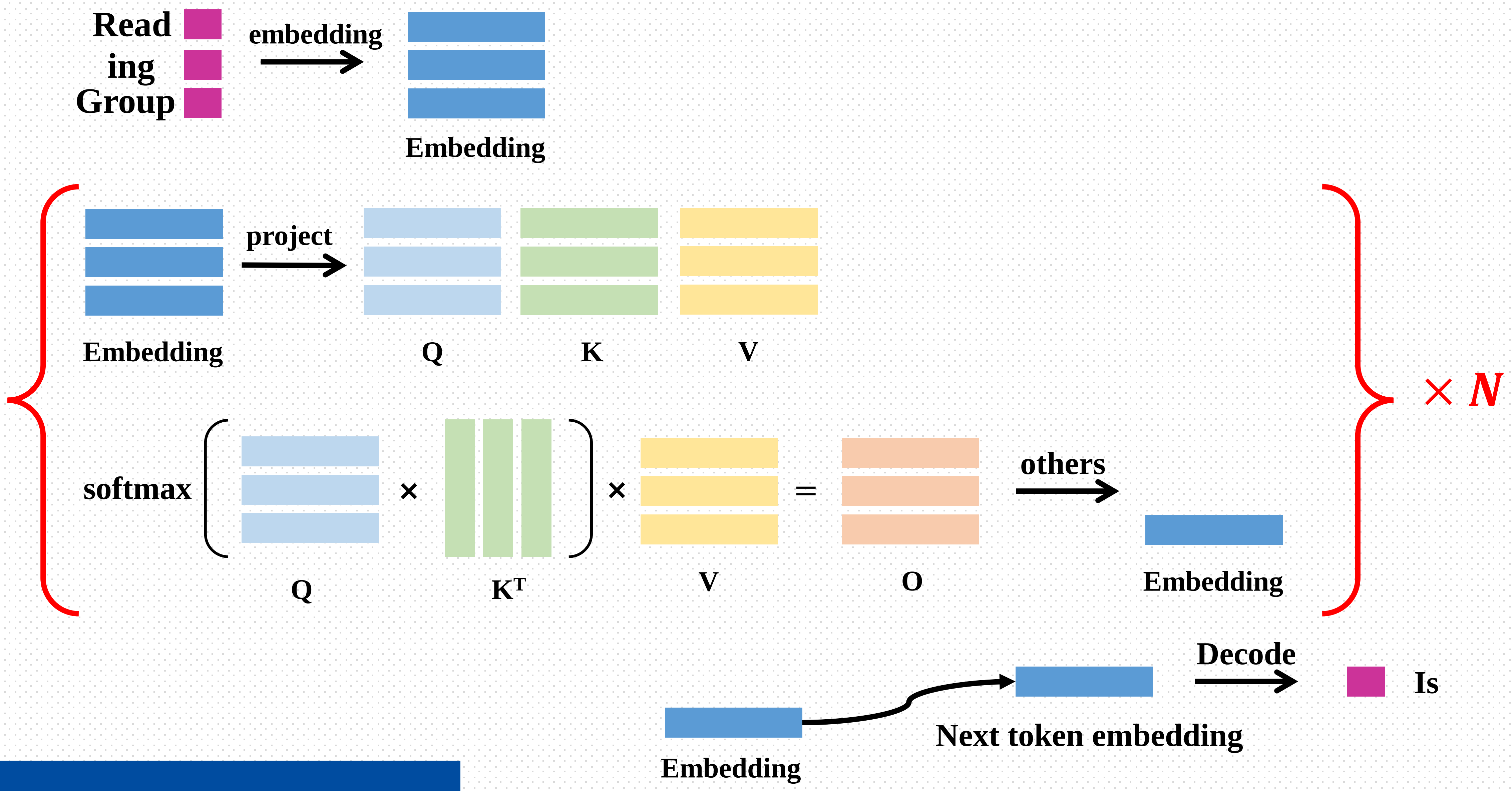
Background - LLM inference



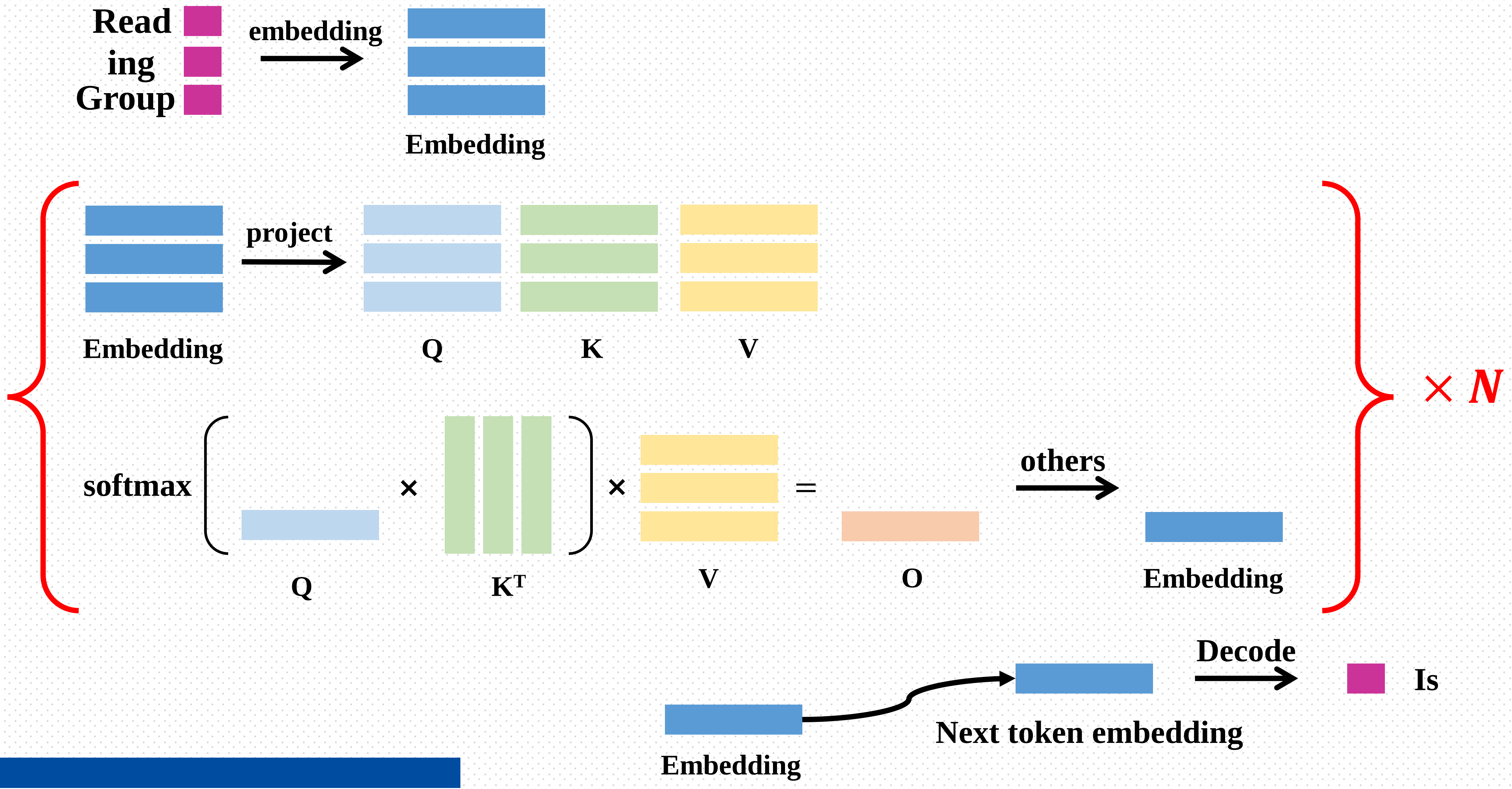
Background - LLM inference



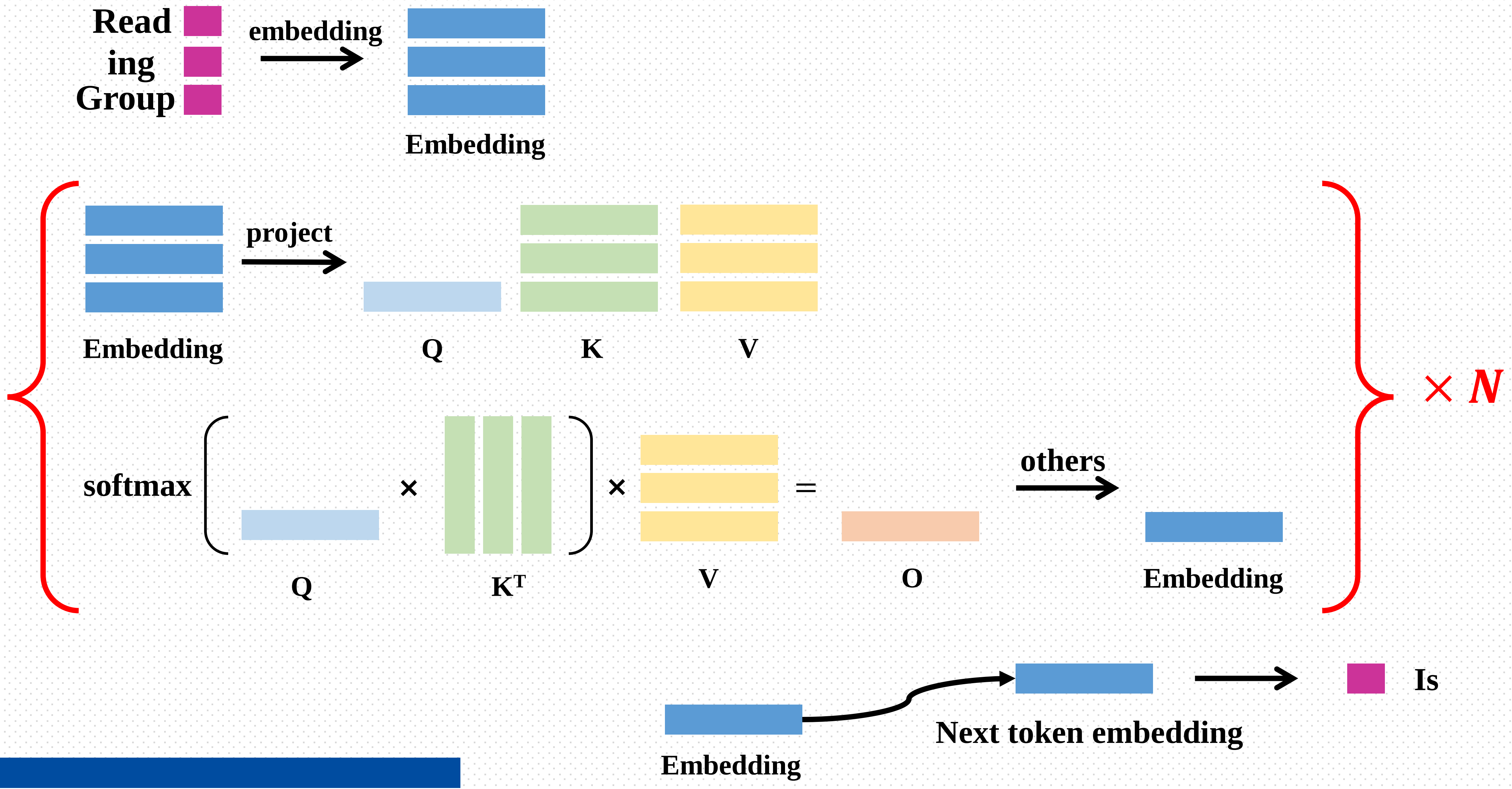
Background - LLM inference



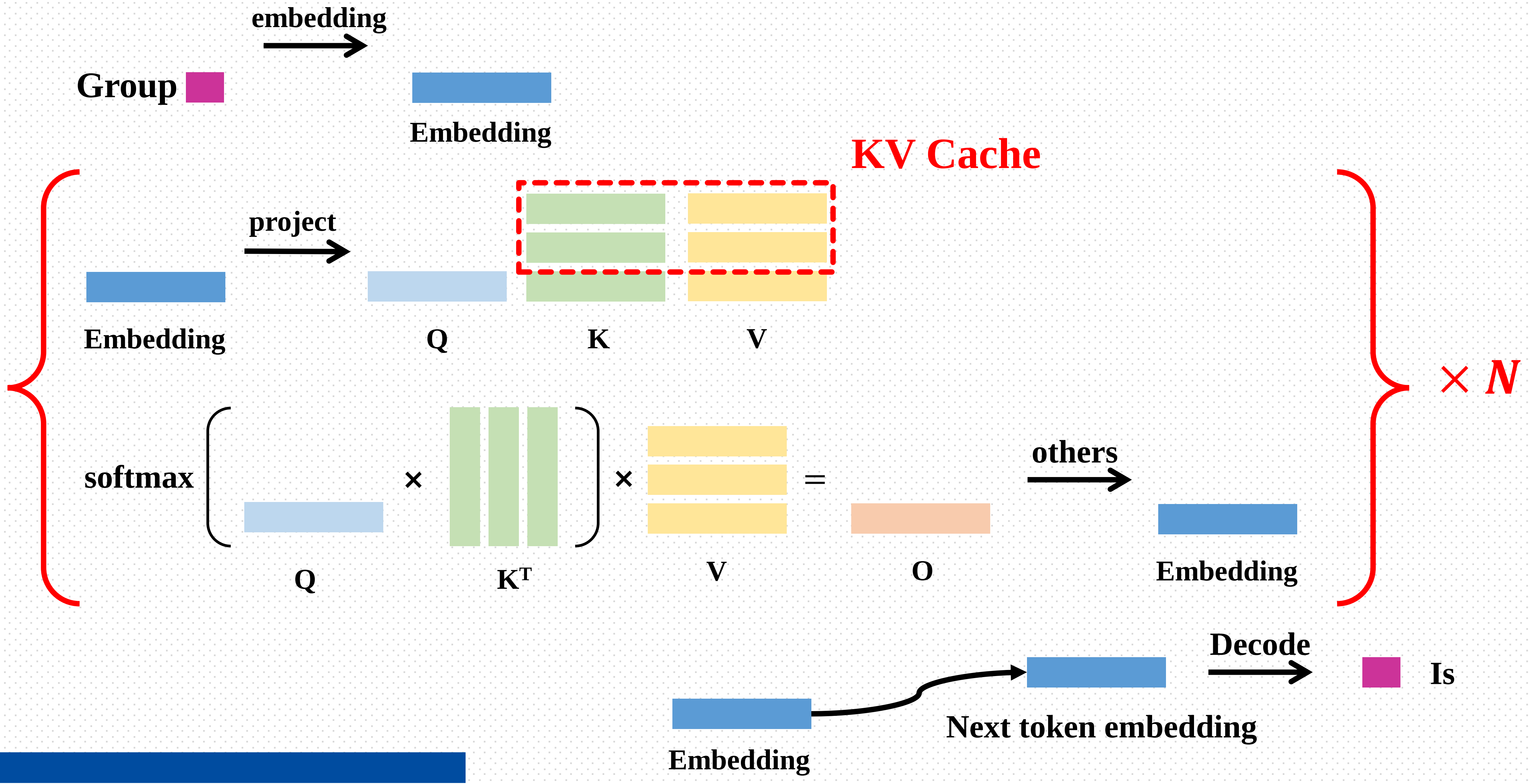
Background - LLM inference



Background - LLM inference

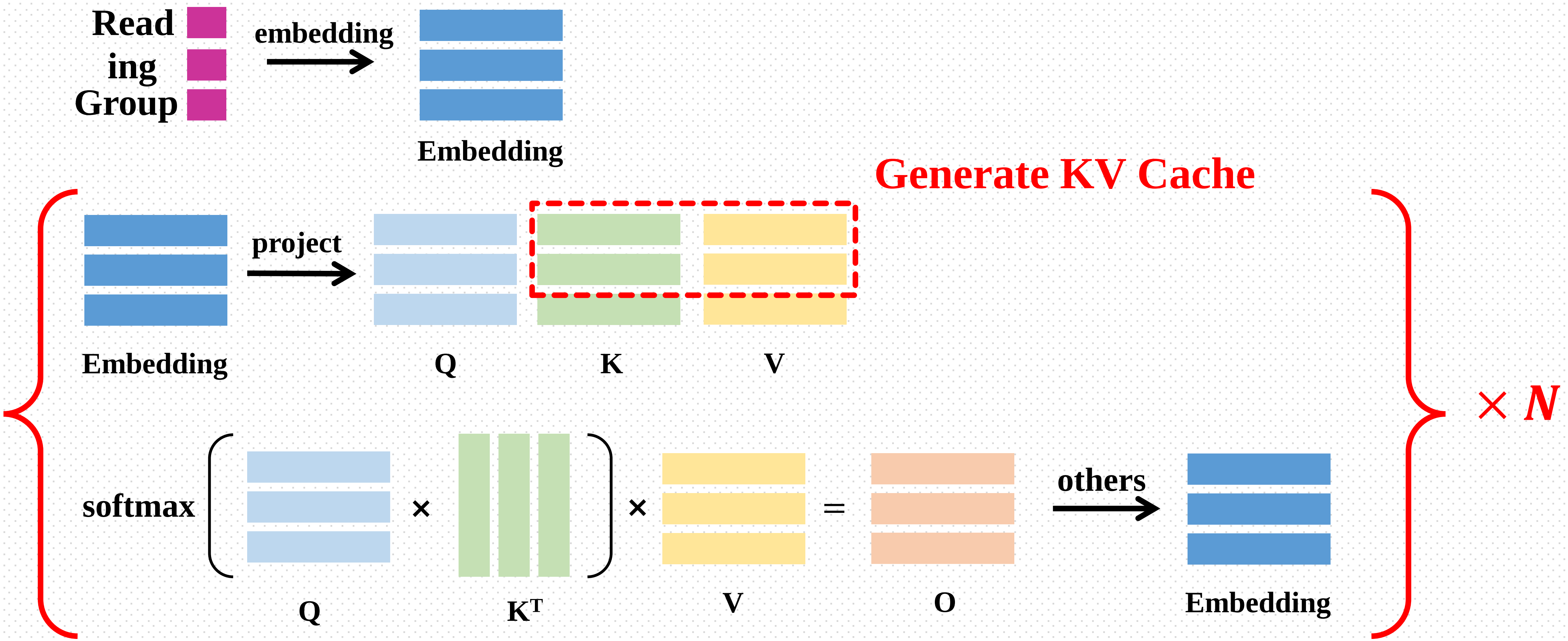


Background - LLM inference



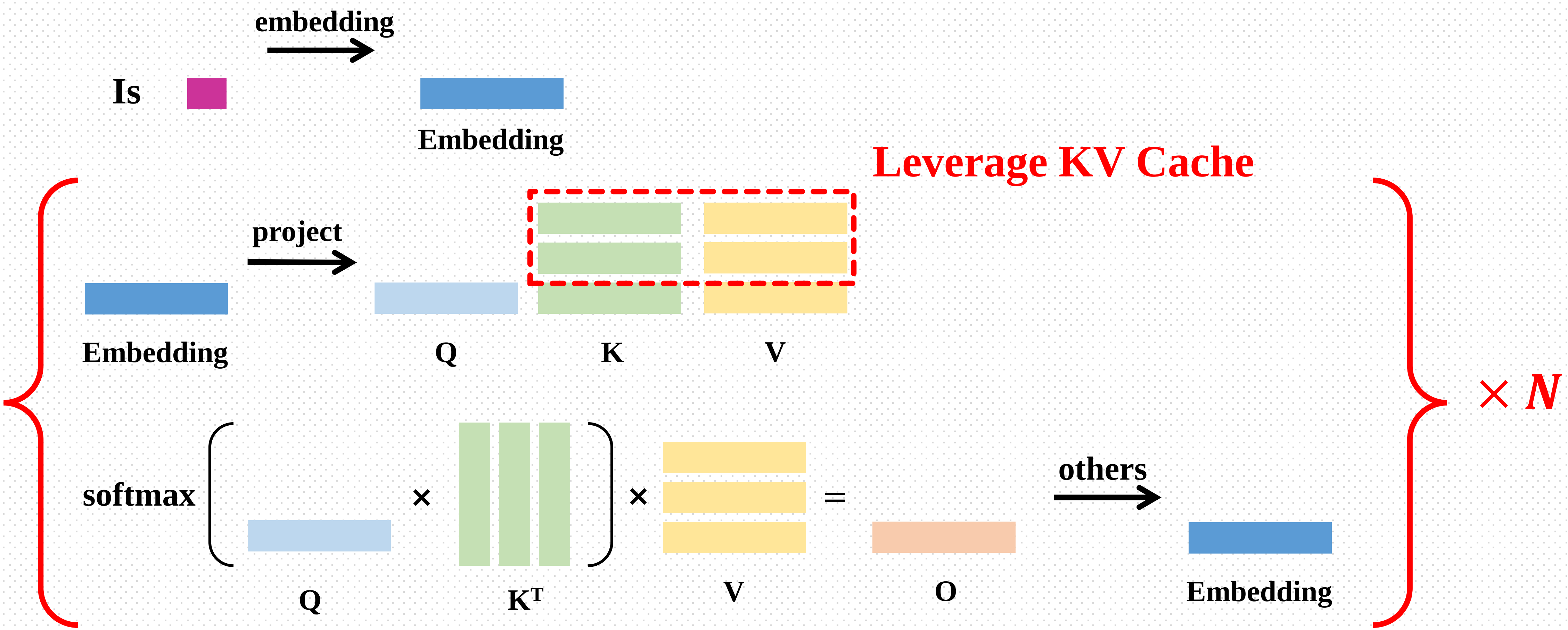
Background - LLM inference

❑Prefill: generate KV cache



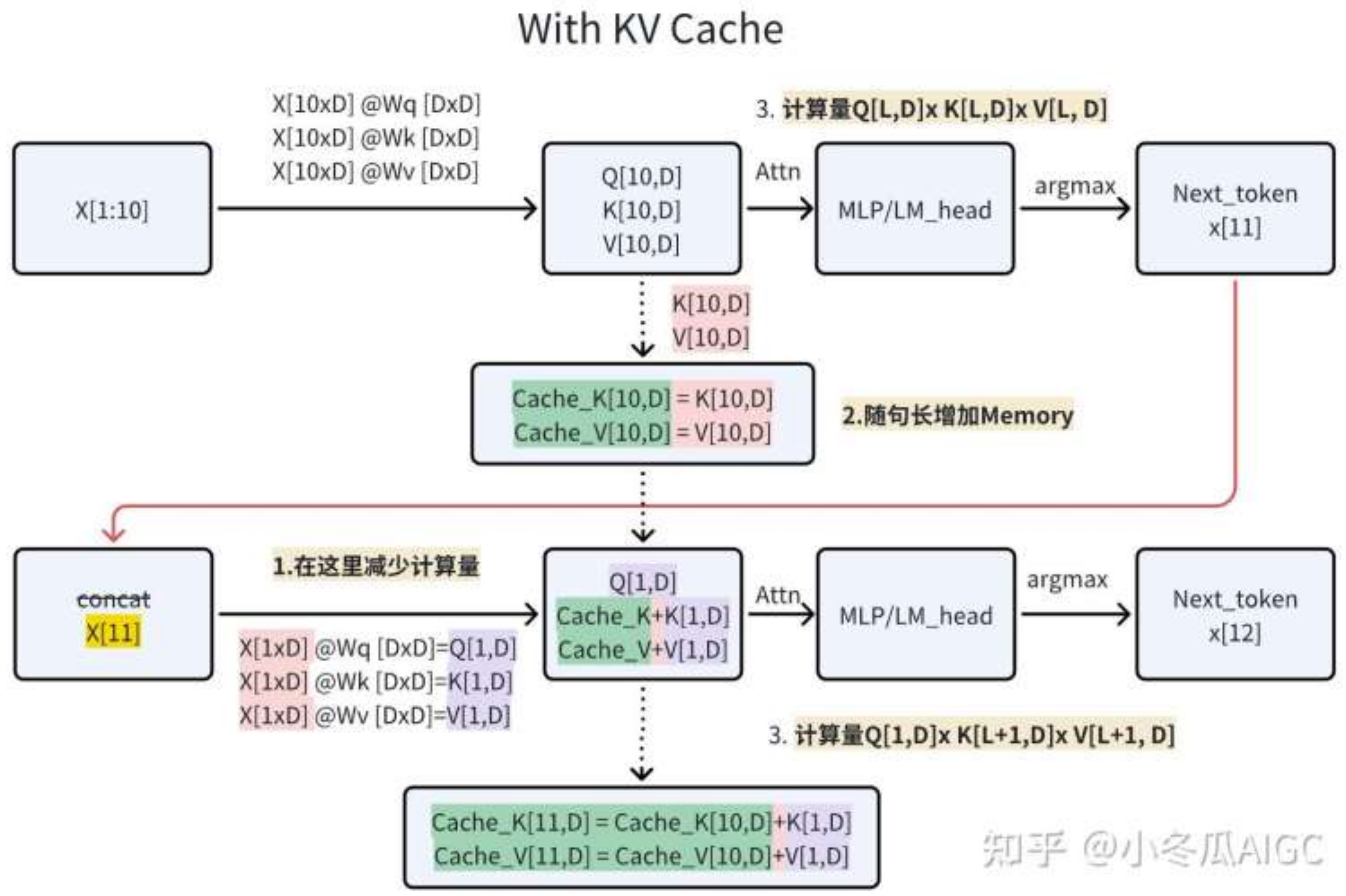
Background - LLM inference

❑Decode: generate next token



Background – LLM inference

❑LLM Inference: 1 prefill step + N decode step



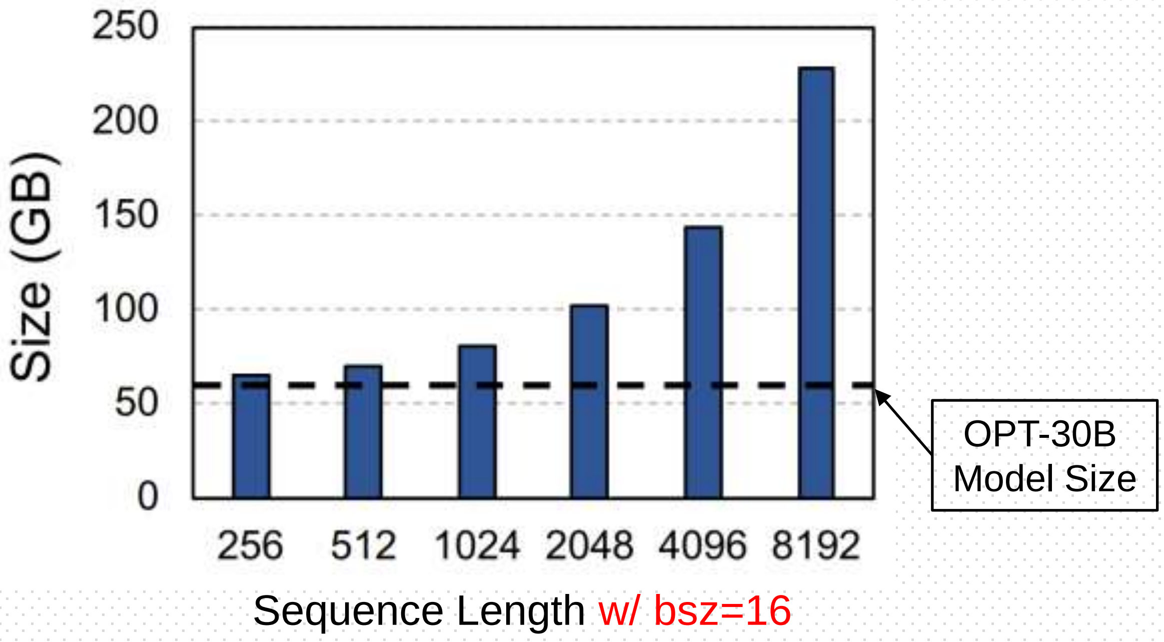
Background – LLM inference

❑But KVCache is the problem!

LLM Model: OPT-30B		
#Layers	**Hidden Dim.**	**Data Type**
48	7168	Float16

KVCache Size:

- **A single token**: $2 * 48 * 7198 * 2B = 1.3MB$
- **32K tokens:** $1.3MB * 32K = 40.6GB$

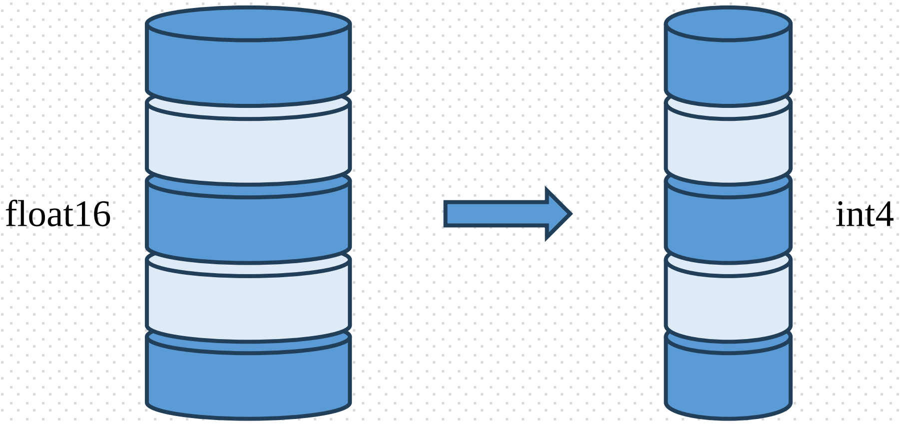


KVCache size can easily exceed GPU memory capacity!

Background – LLM inference

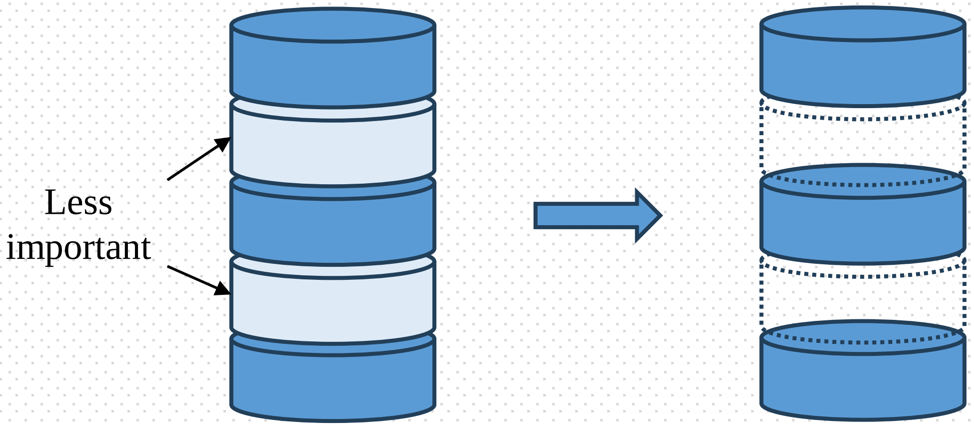
- ❑To solve the problem of KVCache being too large
- ❑Lossy compression
  - ❖Quantization
  - ❖Low-importance tokens eviction
- ❑ Lossless
  - ❖ Offload KVCache to host memory

Background – KVCache compression



Quantization

- **Data precision loss;**
- **The maximum compression rate is fixed**



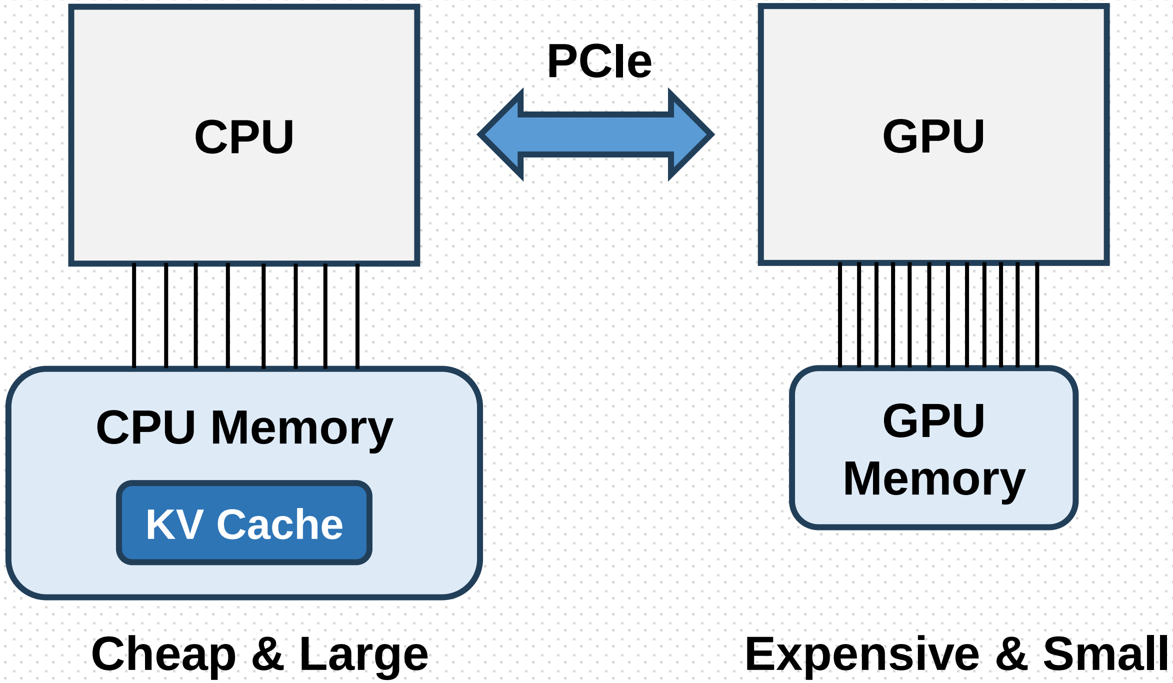
Unimportant tokens eviction (during prefill process)

- **Token information permanent loss;**
- **The importance of tokens varies throughout the decoding process.**

Compression is at a cost of reducing the quality of generated results!

Background – KVCache offloading

- KVCache offloading
  - No information lost!
  - Good model accuracy!
  - But the problem still persists!
    - PCIe bandwidth



Background – KVCache offloading

❑ **High PCIe bandwidth is required for offloading to match GPU inference speed**

Longchat-7B, float16, batch size=1					
Seq. length	**Prefill step time [ms]**	**1 Decoding step time [ms]**	**KVCache size [GB]**	**Prefill required bandwidth [GB/s]**	**Decode required bandwidth [GB/s]**
1K	172.59	27.99	0.50	2.92	**17.99**
2K	327.27	30.73	0.99	3.03	**32.27**
4K	670.07	36.05	1.97	2.94	**54.60**
8K	1343.27	46.91	3.92	2.92	**83.60**
16K	3026.89	68.35	7.83	2.59	**114.52**
32K	7640.90	112.46	15.64	2.05	**139.08**

The PCIe bandwidth is only up to ~28 GB/s

Background – KVCache offloading

- **KVCache + TopK attention**
- **TopK attention**
  - **Attention is sparse[1]. In most transformer layers, n% top KV can carry enough information to maintain model accuracy.**
  - **Only use top n% KVCache with largest attention weights (during decode stage)**
    - $weights = softmax(Q @ K^T)$
    - $indices = TopK(weights, k)$
    - $output = attention(Q, K[indices], V[Indices])$

k	AI2 elem.
64	82.9
128	87.8
256	91.1
512	91.1
1024	**91.9**
2048	91.9
4096	91.9
65536 (vanilla)	91.9

[1] Memory-efficient Transformers via Top-k Attention

Background – KVCache offloading

- ❑KVCache + TopK attention
- ❑Now required PCIe bandwidth is great smaller! (Topk = 10%)

Longchat-7B, float16, batch size=1					
Seq. length	**Prefill step time [ms]**	**1 Decoding step time [ms]**	**KVCache size [GB]**	**Prefill required bandwidth [GB/s]**	**Decode required bandwidth [GB/s]**
1K	172.59	27.99	0.50	2.92	**17.99**
2K	327.27	30.73	0.99	3.03	**32.27**
4K	670.07	36.05	1.97	2.94	**54.60**
8K	1343.27	46.91	3.92	2.92	**83.60**
16K	3026.89	68.35	7.83	2.59	**114.52**
32K	7640.90	112.46	15.64	2.05	**139.08**

× 10%

The PCIe bandwidth is only up to ~28 GB/s

Contexts

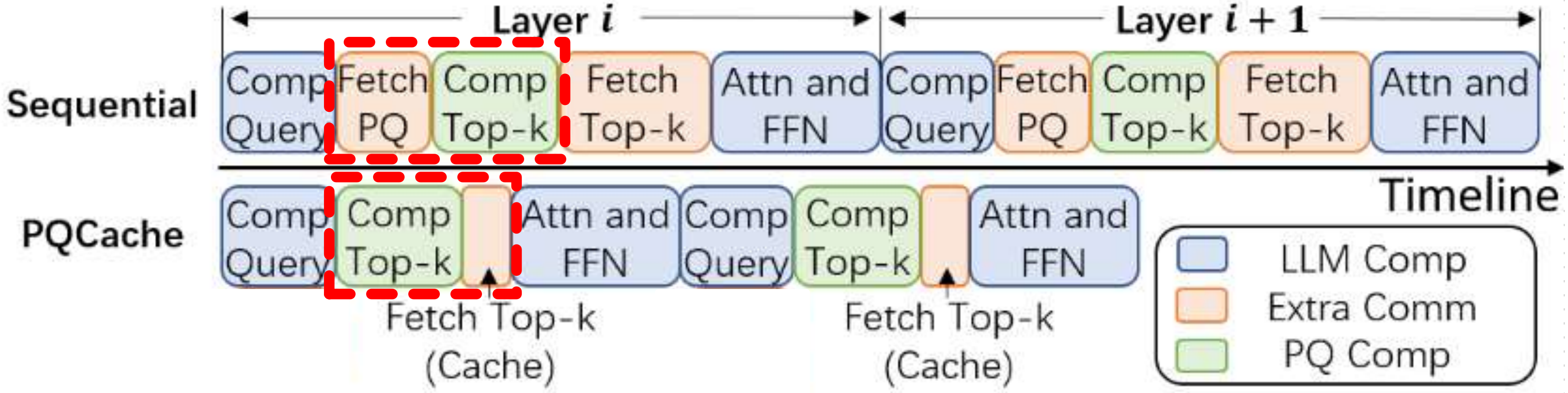
Offloading + TopK attention

- **Overhead of top-k attention and KVCache selecting**
  - **Some new operators (TopK operators + Fetch KVCache) are added to the critical path!**

$$\begin{aligned} weights &= softmax(Q \mathbin{@} K^T) \\ indices &= TopK(weights, k) \\ output &= attention(Q, K[indices], V[Indices]) \end{aligned}$$

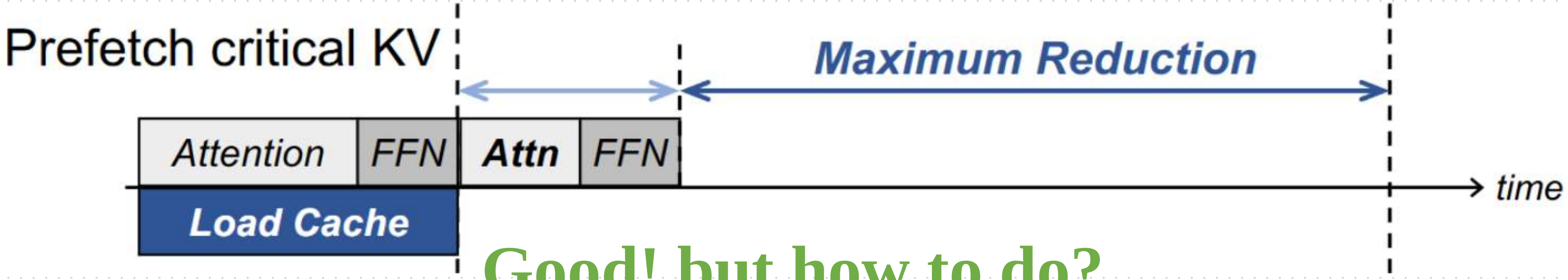
Offloading + TopK attention

- ❑ **Overhead of top-k attention and KVCache selecting**
  - ❖ **Some new operators (TopK operators + Fetch KVCache) are added to the critical path!**
  - ❖ **Cannot overlap the data transfer and computation!**



Offloading + TopK attention + Prefetch

- ❑ Target: completely hide the KVCache loading overhead through overlapping
- ❑ But how?



Good! but how to do?

Contexts

- ❑Background
- ❑Motivation
- ❑InfiniGen
- ❑Evaluations

InfiniGen

- **InfiniGen:**
  - **Algorithm adjustments create opportunities for system optimization.**
- **Technical Contributions:**
  - **Why is prefetching possible?**
  - **How is prefetching implemented?**
  - **Others**

InfiniGen - Why is prefetching possible?

- The inputs for each transformer block are very similar!
- $i-1^{th}$ Transformer:

$$Attn\_out_{i-1} = Attn(LN(Tblock\_in_{i-1}))$$

$$FFN\_out_{\mathrm{i}-1} = FFN\left(LN(Tblock\_in_{i-1} + Attn\_out_{i-1})\right)$$

$$Tblock\_in_{i} = Tblock\_in_{i-1} + Attn\_out_{i-1} + FFN\_out_{\mathrm{i}-1}$$

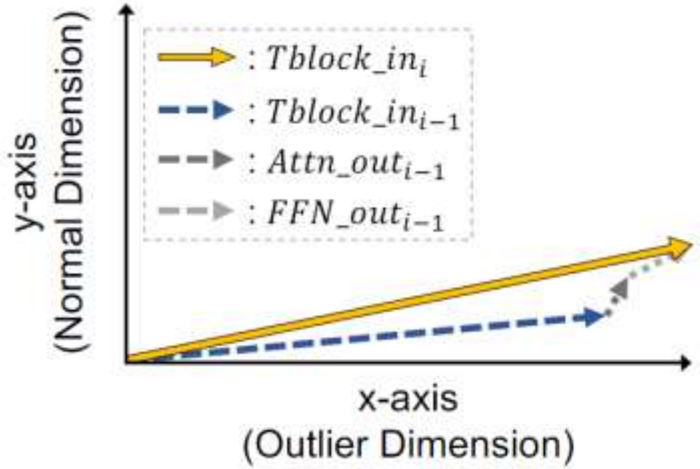
InfiniGen - Why is prefetching possible?

- The inputs for each transformer block are very similar!
- $i-1^{th}$ Transformer:

$$Attn\_out_{i-1} = Attn(LN(Tblock\_in_{i-1}))$$

$$FFN\_out_{i-1} = FFN\big(LN(Tblock\_in_{i-1} + Attn\_out_{i-1})\big)$$

$$Tblock\_in_i = Tblock\_in_{i-1} + Attn\_out_{i-1} + FFN\_out_{i-1}$$



Tensors	OPT-6.7B	OPT-13B	OPT-30B	Llama-2-7B	Llama-2-13B
$Tblock\_in_{i-1}$	**0.95**	**0.96**	**0.97**	**0.89**	**0.91**
$Attn\_out_{i-1}$	0.29	0.28	0.36	0.31	0.27
FFN_out_{i-1}	0.34	0.28	0.35	0.37	0.34

InfiniGen - Why is prefetching possible?

$$hidden\_state_i = LN_i(Tblock\_in_i)$$
$$Q_i = W_i^q \mathbin{@} hidden\_state_i$$
$$K_i = W_i^K \mathbin{@} hidden\_state_i$$
$$weights_i = Softmax\left(Q_i \mathbin{@} K_i^T\right)$$
$$indices = TopK(weights_i, k)$$

The computation of TopK indices

InfiniGen - Why is prefetching possible?

$$hidden\_state_i = LN_i(Tblock\_in_i)$$
$$Q_i = W_i^q \mathbin{@} hidden\_state_i$$
$$K_i = W_i^K \mathbin{@} hidden\_state_i$$
$$weights_i = Softmax\left(Q_i \mathbin{@} K_i^T\right)$$
$$indices = TopK(weights_i, k)$$

The computation of TopK indices

$$hidden\_state_i' = LN_i(Tblock\_in_{i-1})$$
$$Q_i' = W_i^q \mathbin{@} hidden\_state_i'$$
$$K_i' = W_i^K \mathbin{@} hidden\_state_i'$$
$$weights_i' = Softmax\left(Q_i' \mathbin{@} {K_i'}^{\mathrm{T}}\right)$$
$$indices' = TopK(weights_i', k)$$

Calculation of estimated TopK indices

InfiniGen - Why is prefetching possible?

$$hidden\_state_i = LN_i(Tblock\_in_i)$$
$$Q_i = W_i^q \ @\ hidden\_state_i$$
$$K_i = W_i^K \ @\ hidden\_state_i$$
$$weights_i = Softmax(Q_i \ @\ K_i^T)$$
$$indices = TopK(weights_i, k)$$

The computation of TopK indices

$$hidden\_state_i' = LN_i(Tblock\_in_{i-1})$$
$$Q_i' = W_i^q \ @\ hidden\_state_i'$$
$$K_i' = W_i^K \ @hidden\_state_i'$$
$$weights_i' = Softmax\left(Q_i' \ @\ K_i'^{\mathrm{T}}\right)$$
$$indices' = TopK(weights_i', k)$$

Calculation of estimated TopK indices

$Tblock\_in_i$ and $Tblock\_in_{i-1}$ are alike. → Q_i and Q_i' are alike. K_i and K_i' are alike. → $weights_i$ and $weights_i'$ are alike. → $indices$ and $indices'$ are alike.

InfiniGen - Why is prefetching possible?

$$hidden\_state_i = LN_i(Tblock\_in_i)$$
$$Q_i = W_i^q \mathbin{@} hidden\_state_i$$
$$K_i = W_i^K \mathbin{@} hidden\_state_i$$
$$weights_i = Softmax(Q_i \mathbin{@} K_i^T)$$
$$indices = TopK(weights_i, k)$$

The computation of TopK indices

$$hidden\_state_i' = LN_i(Tblock\_in_{i-1})$$
$$Q_i' = W_i^q \mathbin{@} hidden\_state_i'$$
$$K_i' = W_i^K \mathbin{@} hidden\_state_i'$$
$$weights_i' = Softmax\left(Q_i' \mathbin{@} {K_i'}^{\mathrm{T}}\right)$$
$$indices' = TopK(weights_i', k)$$

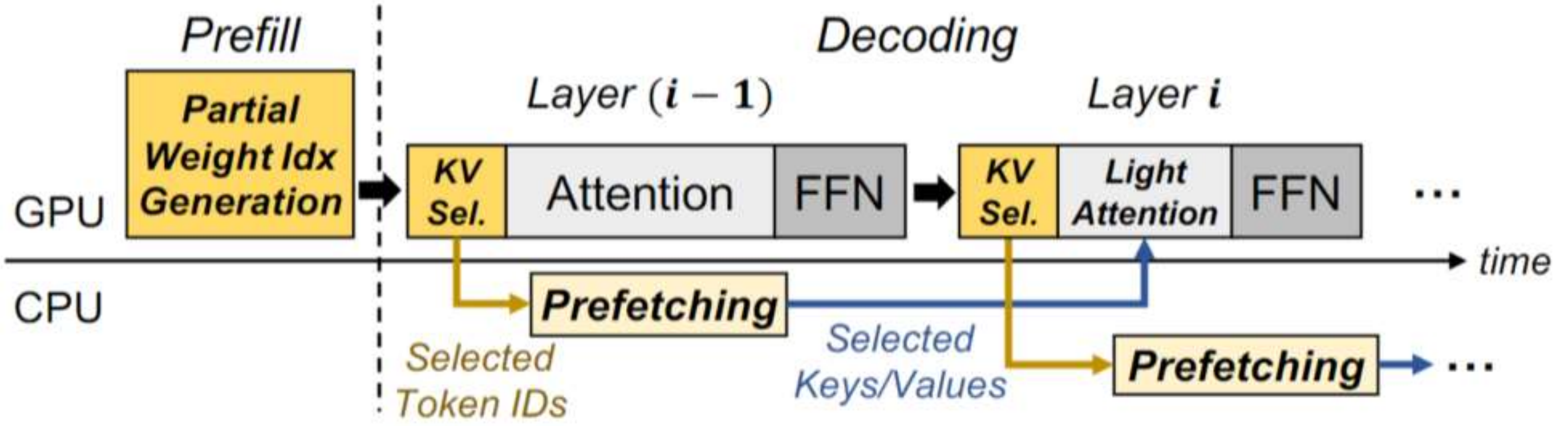
Calculation of estimated TopK indices

$Tblock\_in_i$ and $Tblock\_in_{i-1}$ are alike. → Q_i and Q_i' are alike. K_i and K_i' are alike. → $weights_i$ and $weights_i'$ are alike. → $indices$ and $indices'$ are alike.

The tensor $Tblock\_in_{i-1}$ is obtained during $(i-1)^{th}$ layer. Therefore, it can be utilized for prefetching for i^{th} layer.

InfiniGen - Why is prefetching possible?

- ❑Prefetching opportunities (excluding the initial layer)
  - ❖0/1 layer: full attention
  - ❖Other layers: TopK attention



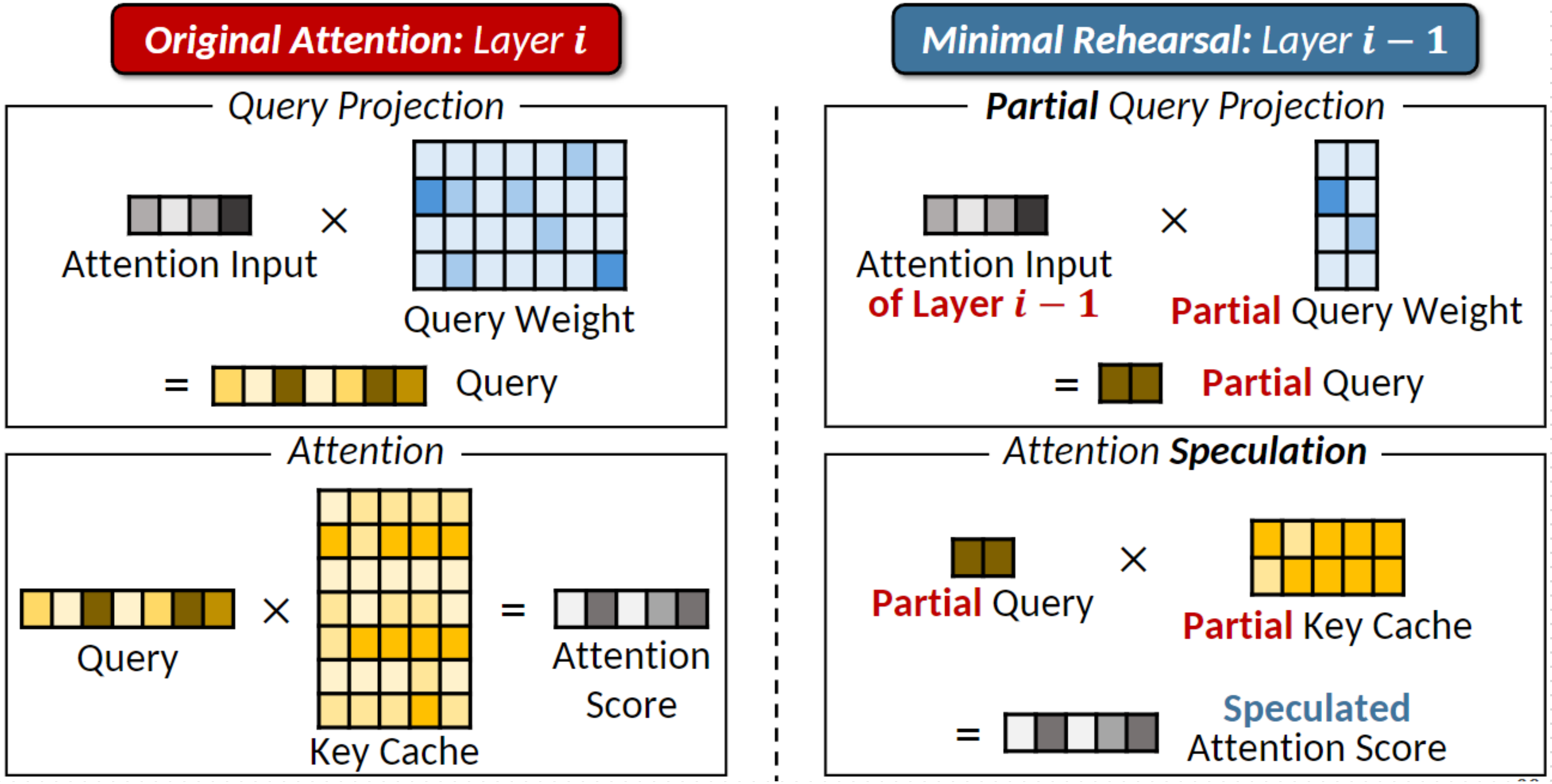
InfiniGen - How is prefetching implemented?

- **The overhead of TopK operator is high!**
  - **Both Memory and Computation.**
- **What's worse, the operator is done on CPU.**

$$hidden\_state'_i = LN_i(Tblock\_in_{i-1})$$
$$Q'_i = W^q_i \,@\, hidden\_state'_i$$
$$K'_i = W^K_i \,@\, hidden\_state'_i$$
$$weights'_i = Softmax\left(Q'_i \,@\, {K'_i}^{\mathrm{T}}\right)$$
$$indices' = TopK(weights'_i, k)$$

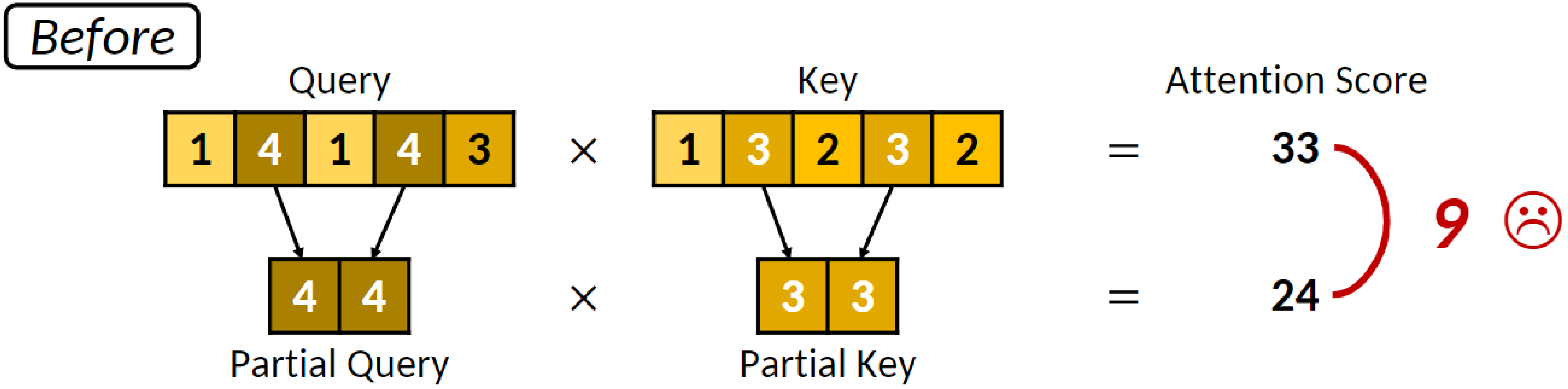
InfiniGen - How is prefetching implemented?

- Reduce computational complexity through dimensionality reduction.



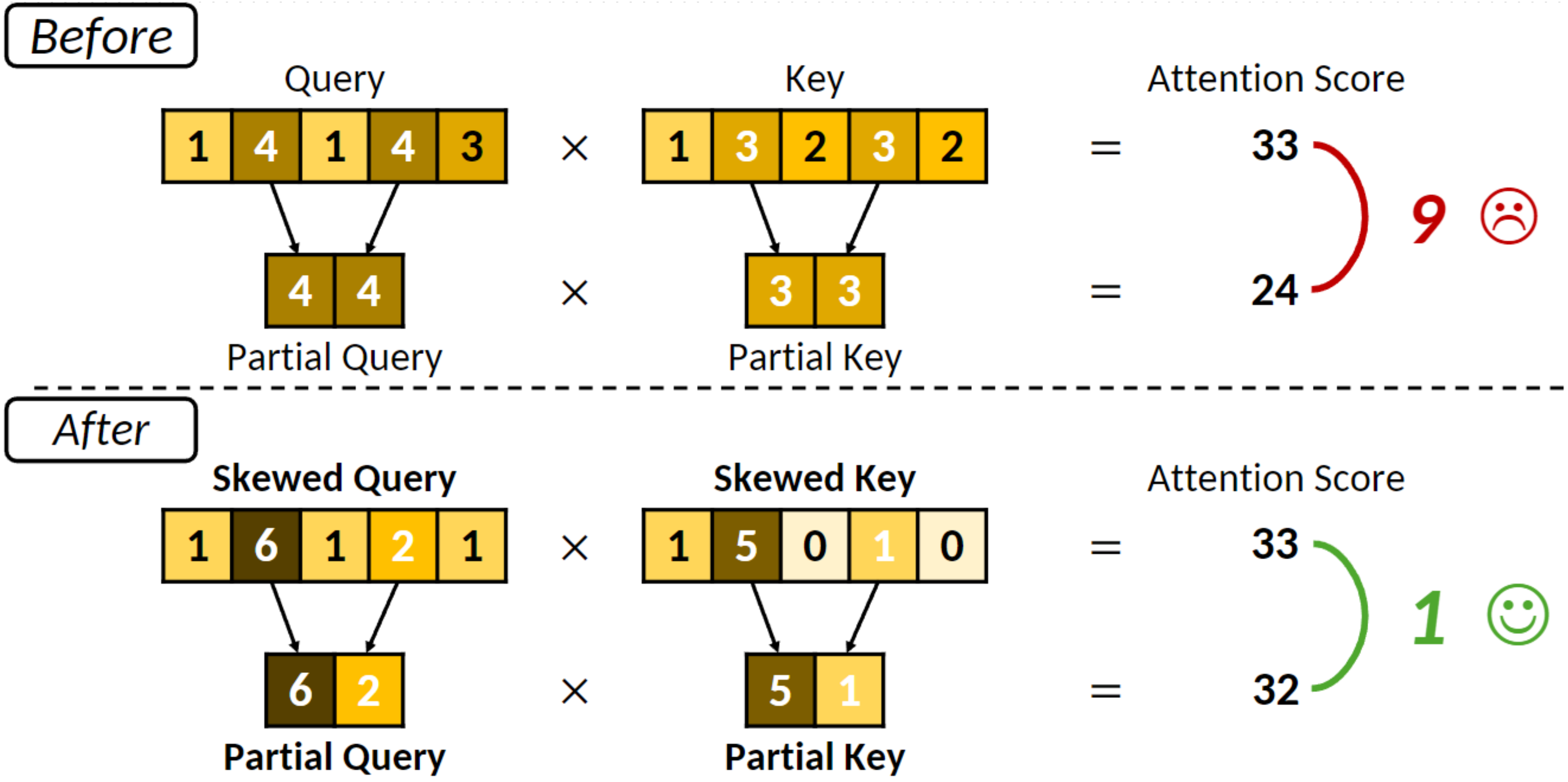
InfiniGen - How is prefetching implemented?

❑Key/Query Skewing



InfiniGen - How is prefetching implemented?

❑Key/Query Skewing

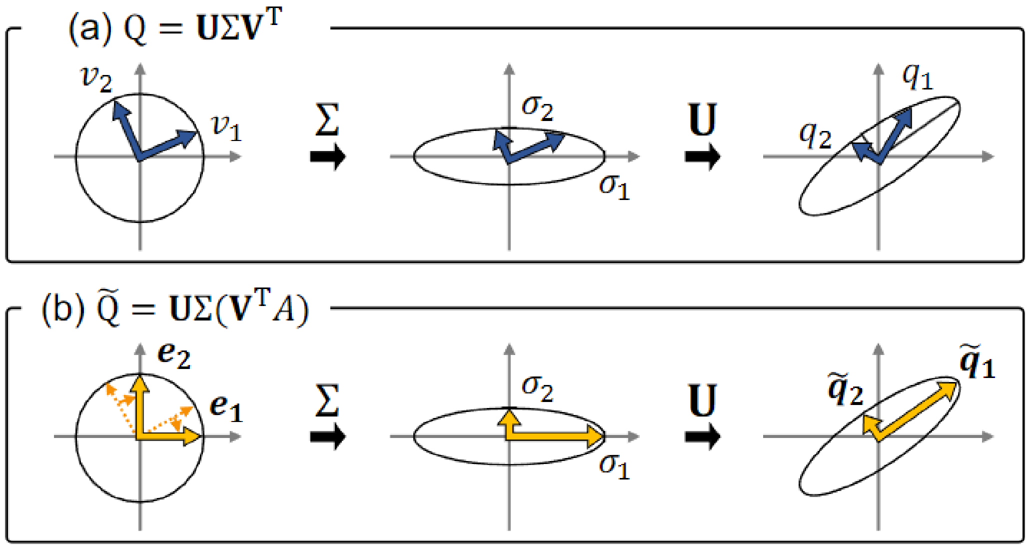


InfiniGen - How is prefetching implemented?

❑Key/Query Skewing

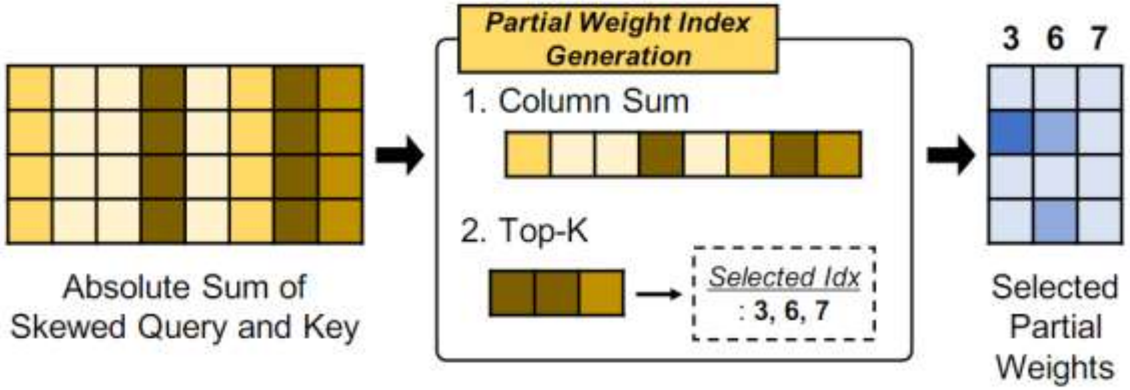
- ❖Offline modification of the query and key weights using singular value decomposition (SVD)
- ❖The identical computation result：

$$(Q \times A) \times (A^T \times K^T) = QK^T \qquad A = V$$



InfiniGen - How is prefetching implemented?

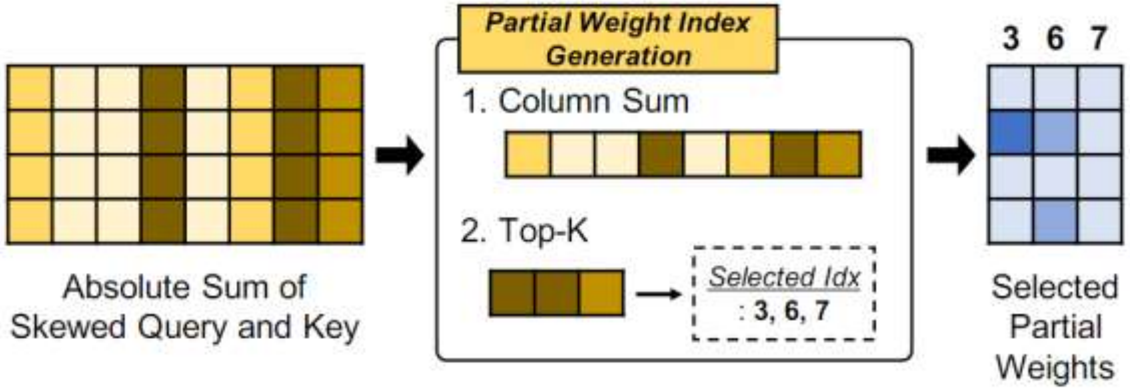
❑Prefill stage: reduce the dimensionality of Key after skewing



$$Indices = TopK(KA + QA).abs().sum(axis = 1), T)$$
$$K' = (KA)[:, Indices]$$

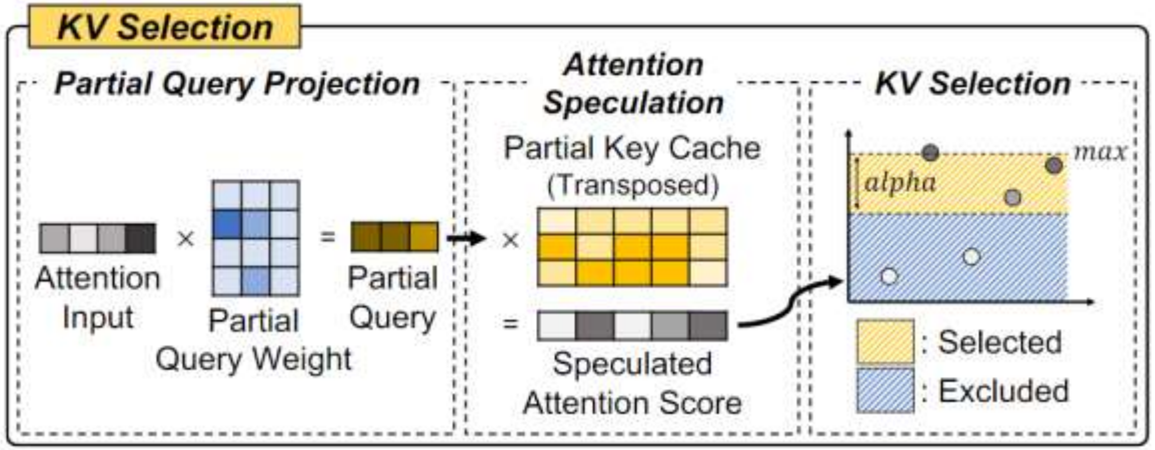
InfiniGen - How is prefetching implemented?

❑Prefill stage: reduce the dimensionality of Key after skewing



❑Decode stage: compute Partial Query and TopK indices

❖The selected weight/score is within the range of $[max - \alpha, max]$.



InfiniGen - Others

❑Technical Contributions:

- ❖Why is prefetching possible?
- ❖How is prefetching implemented?
- ❖When the CPU memory is out of memory (OOM):
  - ➢a counter-based approach can be used to drop entries from the KVCache.

Contexts

- ❑Background
- ❑Motivation
- ❑InfiniGen
- ❑Evaluations

Evaluation — Setup

- **Models**
  - OPT model with 6.7B, 13B and 30B parameters
  - Llama-2 model with 7B and 13B parameter
- **Hardware**
  - 1 × NVIDIA RTX A6000 GPU (48GB memory)
  - Intel Xeon Gold 6136 with 96GB DDR4-2666 memory
  - PCIe 3.0 × 16 (~16GB/s)

Evaluation — Accuracy

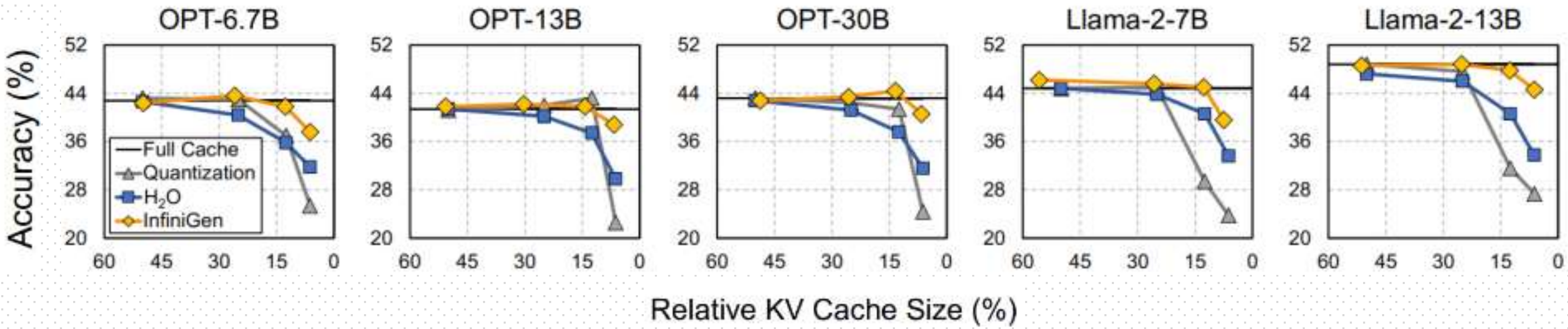
❑Accuracy

❖KVCache management:

- ➢Full: no compression, generation with full KV Cache
- ➢H_2O (NeurIPS 2023): SOTA low-importance token eviction algorithm
- ➢Quantization
- ➢InfiniGen

Evaluation — Accuracy

❑Accuracy



InfiniGen outperforms all the baseline, achieves near lossless accuracy

Evaluation — Speedup and Latency

❑Baseline

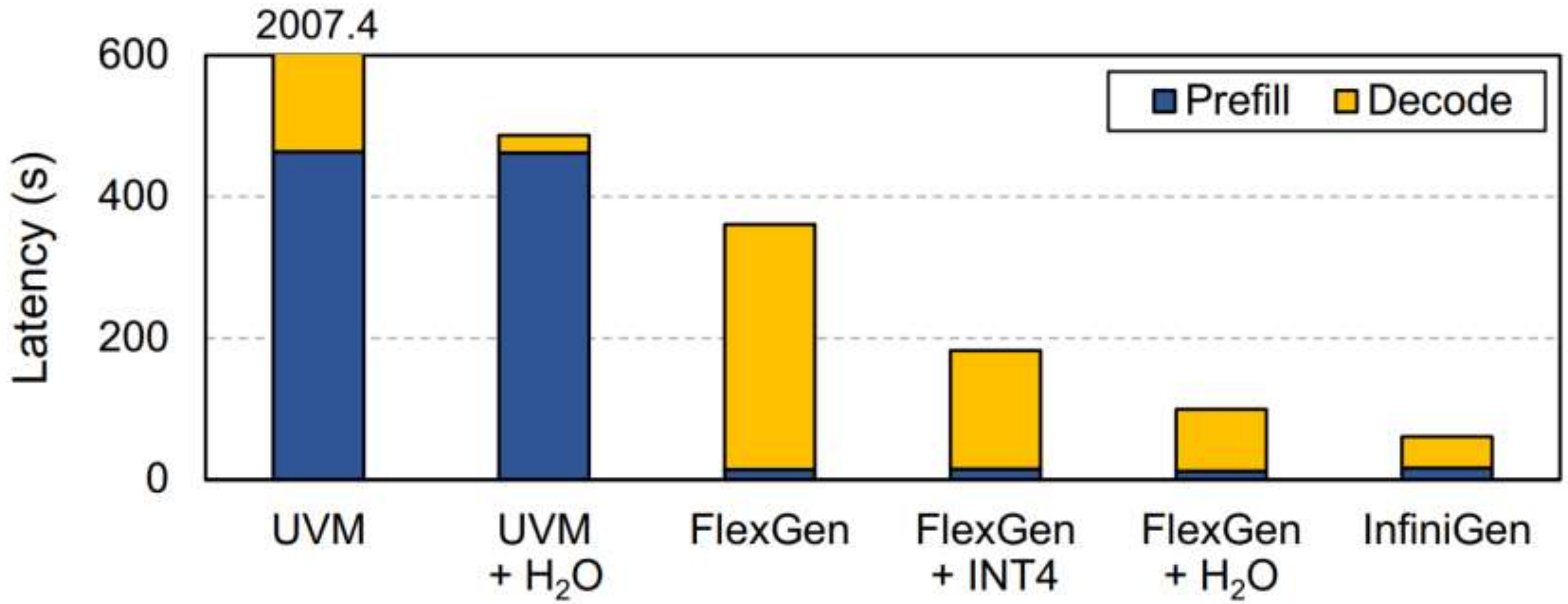
- ❖FlexGen (ICML 2023)
  - ➢ All KVCache on the CPU memory w/ prefetching
  - ➢ All KVCache are stored on the CPU memory and only Model are on the GPU.
- ❖Unified Virtual Memory (UVM):
  - ➢ All KVCache on the CPU memory w/o prefetching
  - ➢ The data movement between CPU and GPU are managed by NVIDIA driver.
- ❖InfiniGen (TopK: up to 10%)

❑KVCache management:

- ❖H_2O: 5 compression ratio
- ❖Quantization (INT4): 4 compression ratio

Evaluation — Speedup and Latency

❑Latency



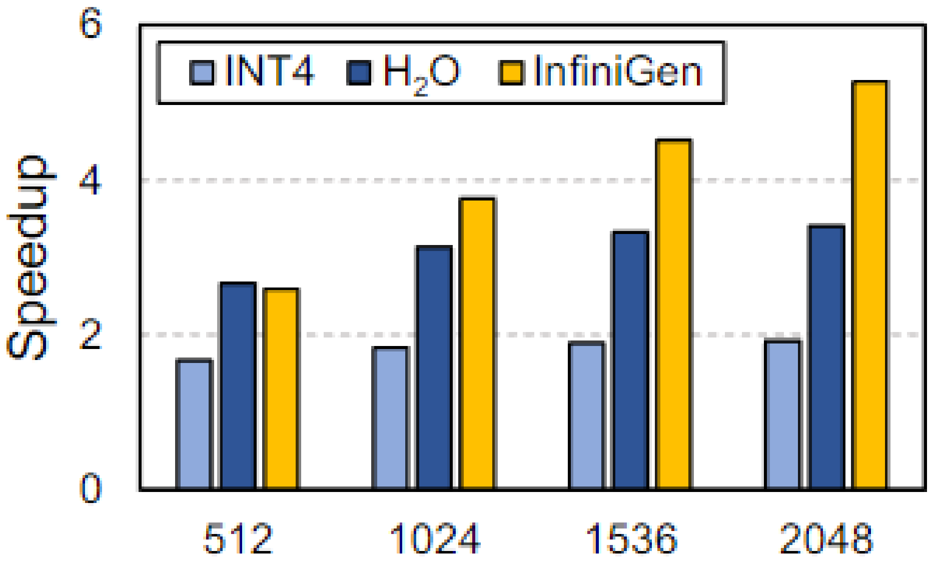
OPT-13 model, with 1920 input tokens, 128 output tokens, bs=20 (1920 for prefill + 128 decode)

Evaluation — Speedup and Latency

❑Speedup

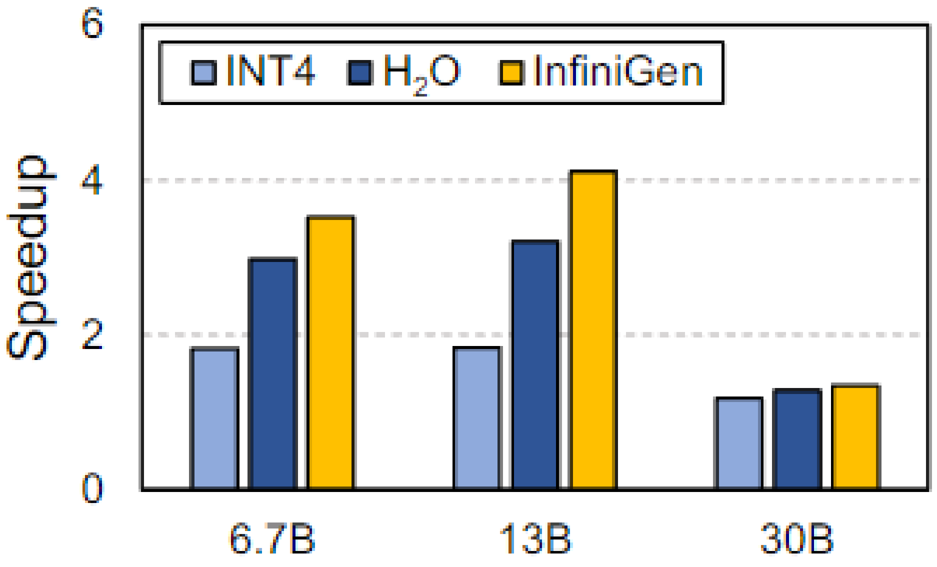
- ❖FlexGen + INT4/H2O
- ❖InfiniGen

1 is the FlexGen



(a) Sequence Length

OPT-13B, 128 output, bs = 8



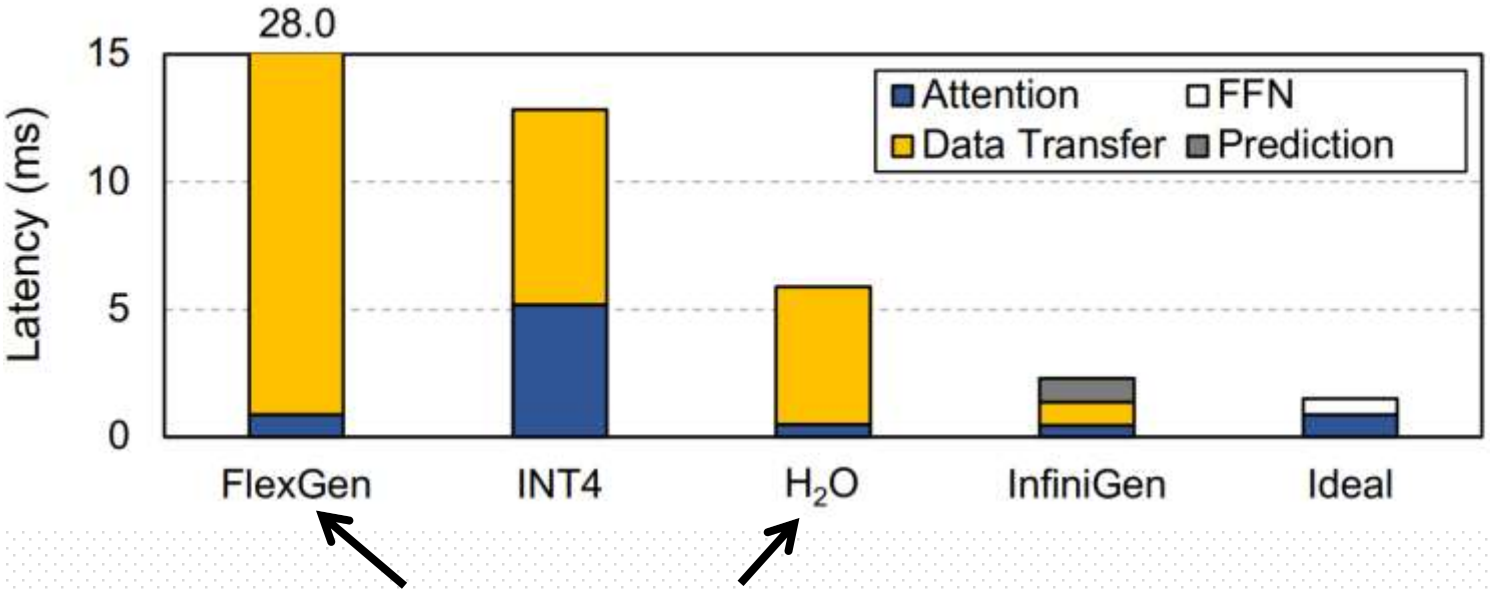
(b) Model Size

1920 input tokens,128 output tokens, bs=4

Evaluation — Breakdown

❑Breakdown and prefetch overhead

❖OPT-13B with 8 *2048 inputs

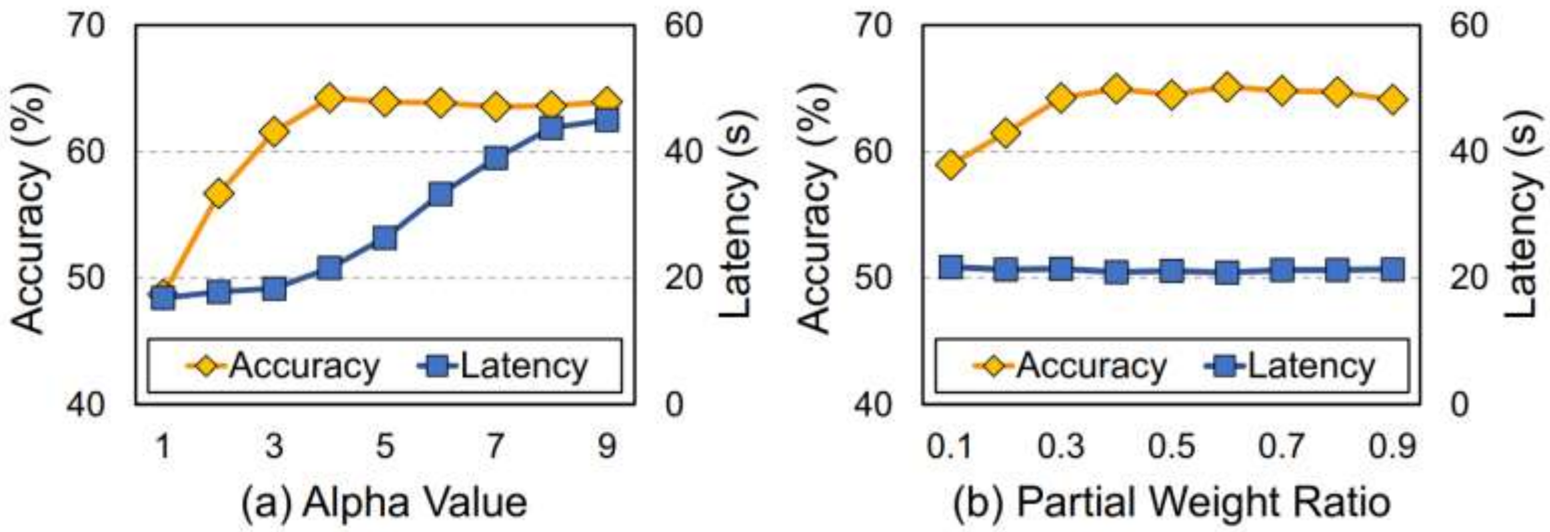


FlexGen and H_2O spend most time in data transfer (96.9% and 91.8% respectively)

Evaluation

❑Sensitivity analysis

- ❖With higher alpha value, accuracy and latency both increase (4 is enough)
- ❖Higher Partial Weight Ratio will cause a higher accuracy and higher memory consumption, and latency remains stable (0.3 is enough)



End

- ❑Thank you!
- ❑Q&A