LLM Inference Optimization

Autumn 2026

Lecturer: Yuedong (Steven) Xu

Shenzhen Loop Area Institute

yuedongxu@fudan.edu.cn

Disclaimer

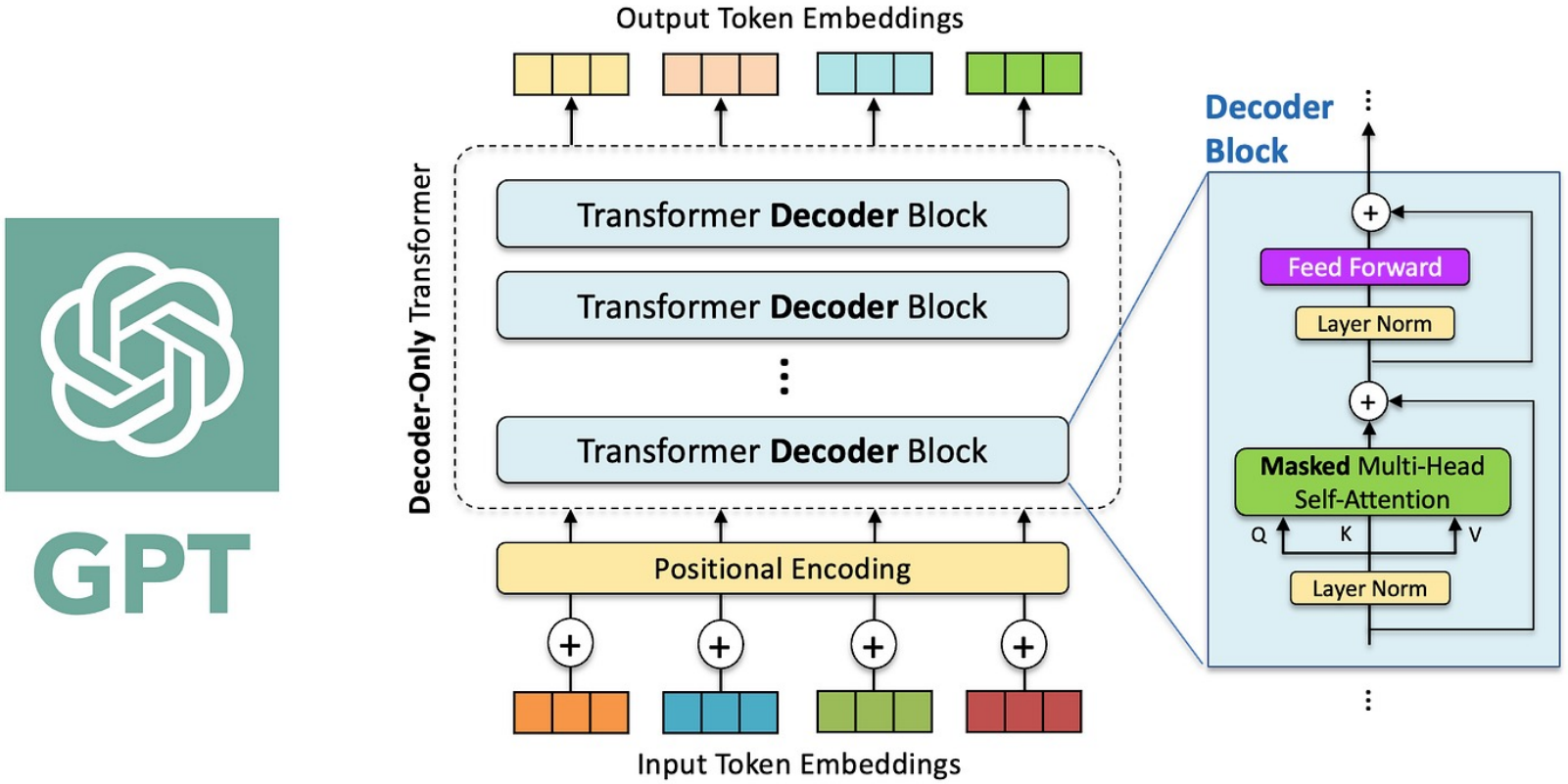
Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blogposts, research talks, tutorial videos, and other materials shared by the research community. Sometimes external animations and exquisite pictures are heavily reused.

Inference Optimization

- **Overview**
- Attention Optimization
- KV Cache Optimization
- Parallel and Distributed Serving

Overview

- Decoder-only Transformer

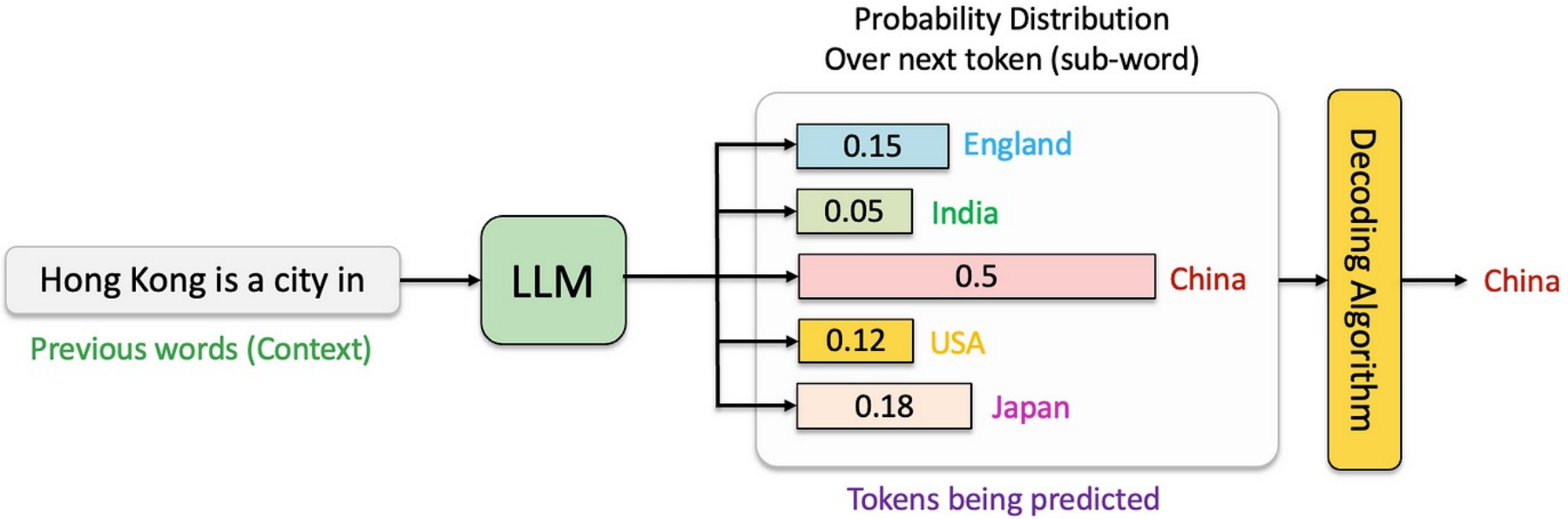


GPT (Generative Pre-trained Transformer) stacking Transformer blocks

Source: The Rise of LLMs: From GPT to Modern Innovations

Overview

- Decoder-only Transformer
  - Generating a probabilistic distribution over possible next token, and a decoding algorithm (greedy/beam search) is employed to select the output token

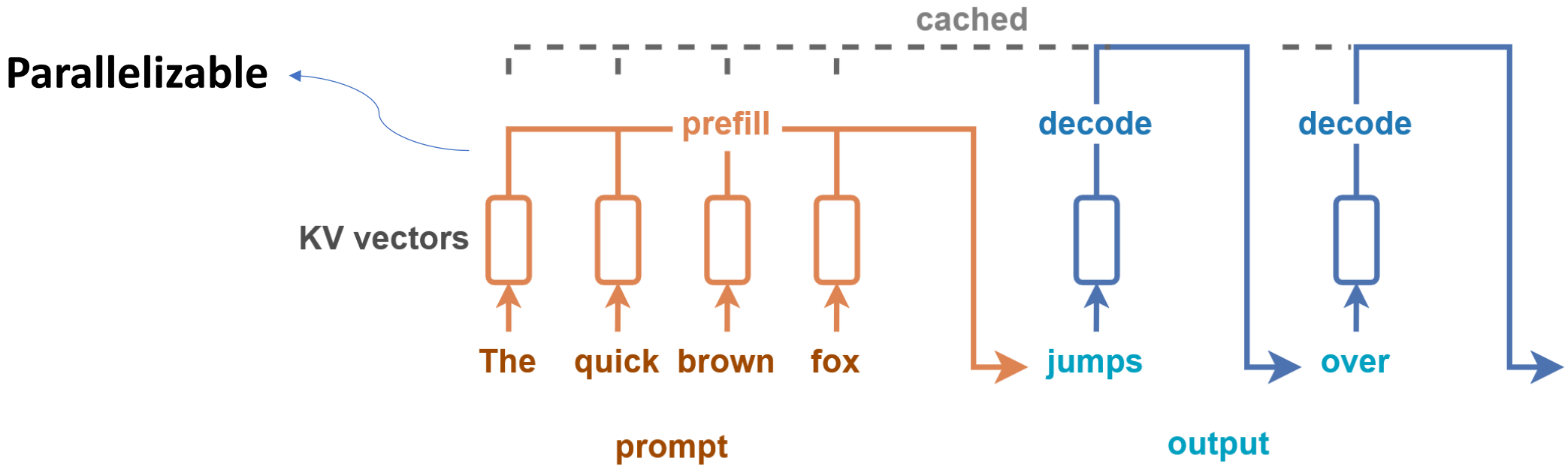


Next-token prediction, i.e. generating output tokens one by one

Source: The Rise of LLMs: From GPT to Modern Innovations

Overview

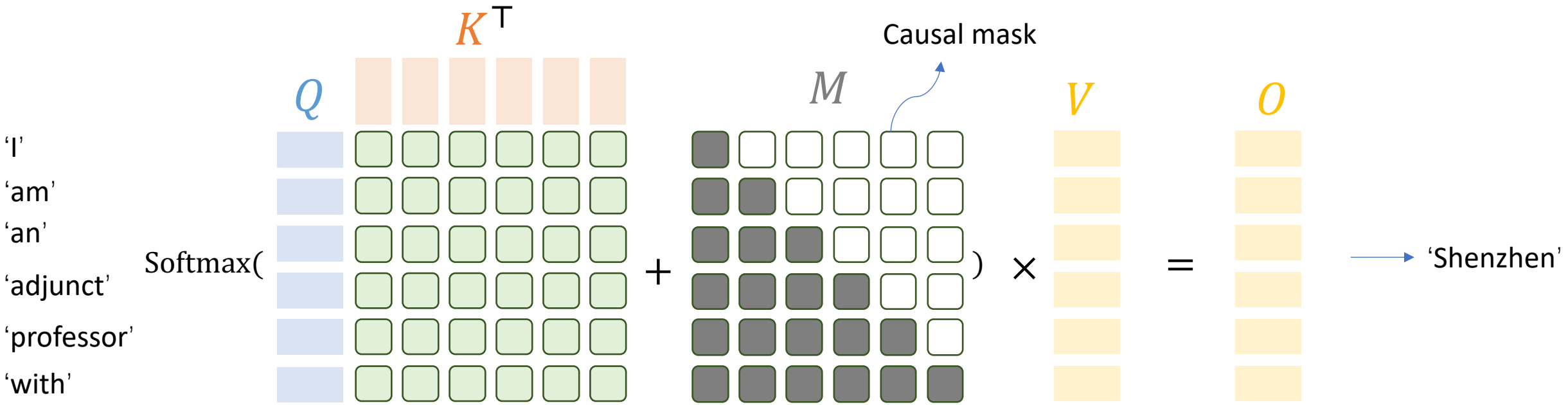
- Autoregressive Decoding
  - **“Prefill”** refers to the **initial parallel computation phase** where the model processes the entire input prompts for subsequent token-by-token generation



Source: [Prefill and Decode for Concurrent Requests - Optimizing LLM Performance]

Overview

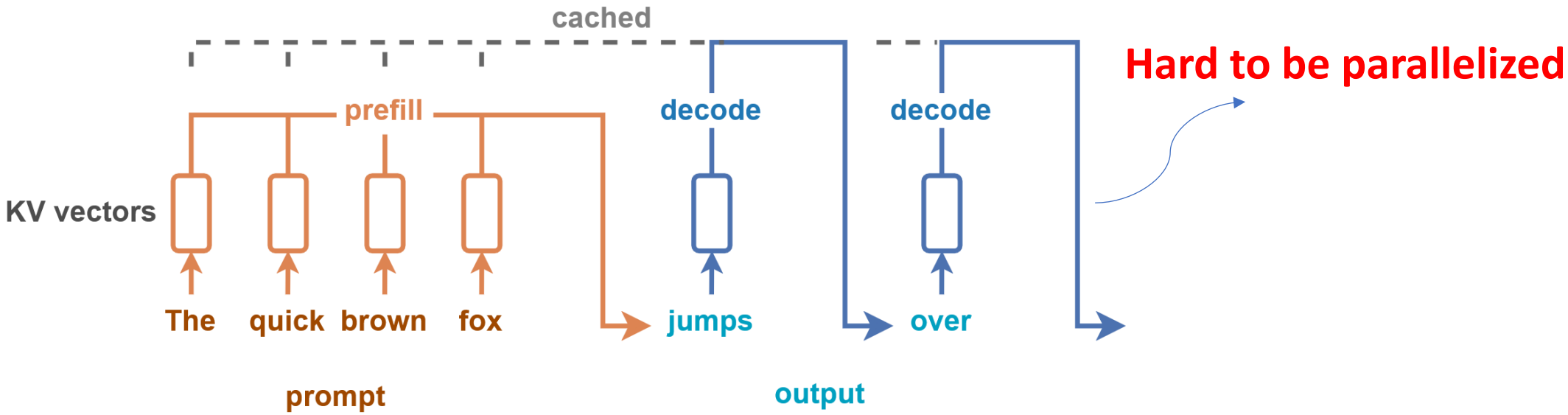
- Prefill: highly parallelizable
  - A small batch size can ”saturate” GPU computation
    - Parallelization over batch size, header size, sequence length and thread-block tiling



Source: Jia-Bin Huang from University of Maryland College Park

Overview

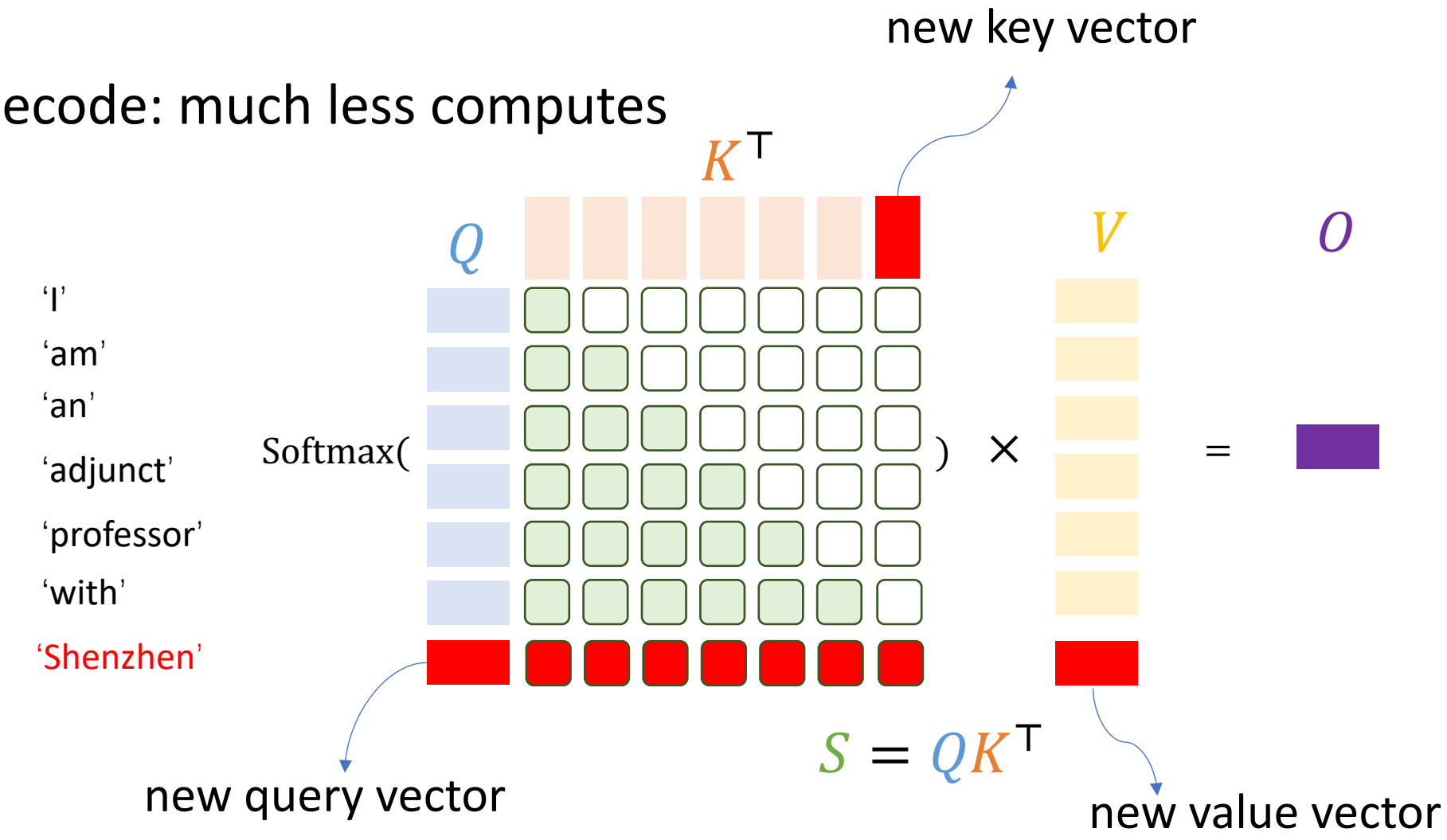
- Autoregressive Decoding
  - **“Decode”** refers to the **iterative process of generating output tokens** one at a time, where each new token is predicted based on the input prompt and all previously generated tokens



Source: [Prefill and Decode for Concurrent Requests - Optimizing LLM Performance]

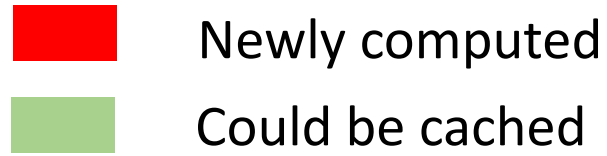
Overview

- Decode: much less computes

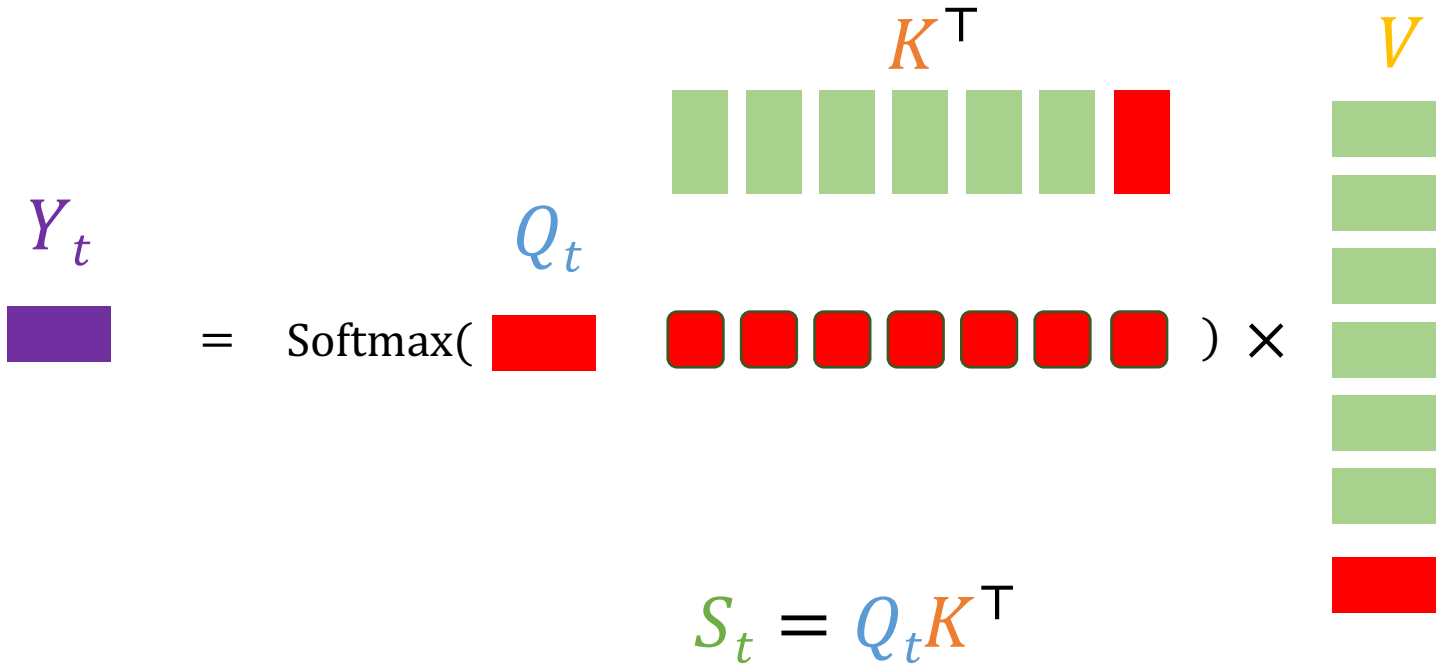


Decode computations

Overview



- Decode: much less computes
  - Why caching Key and Value, other than Query



Decode Compute output token one by one

Overview

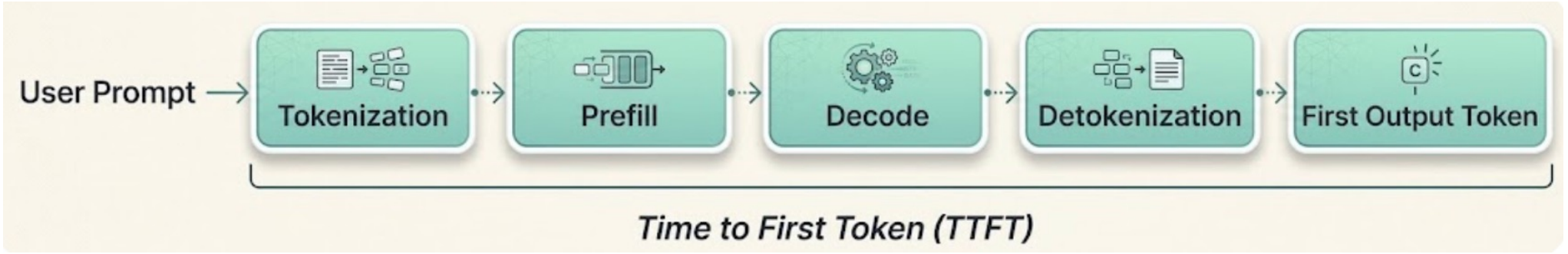
- Why Caching KV?
  - Reducing redundant computations for *Keys* and *Values*
  - Increasing memory consumption

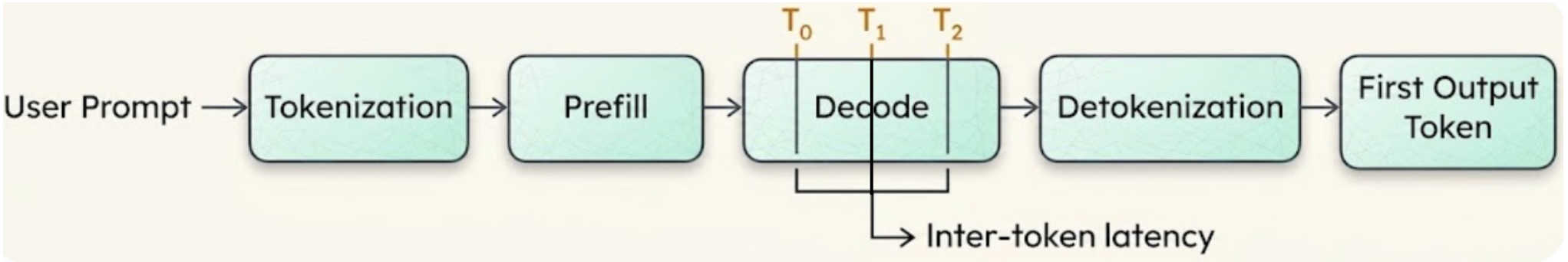
$$\underset{\text{K/V}}{2} \times \underset{\text{Float16}}{2} \times \underset{\text{Sequence length}}{8096} \times \underset{\text{\# of layers}}{80} \times \underset{\text{\# of heads}}{64} \times \underset{\text{dimension}}{64}$$

~10.6 GB!

Overview

- Autoregressive Decoding
  - Token generation core metrics: **TTFT** and **TPOT** (time per output token)

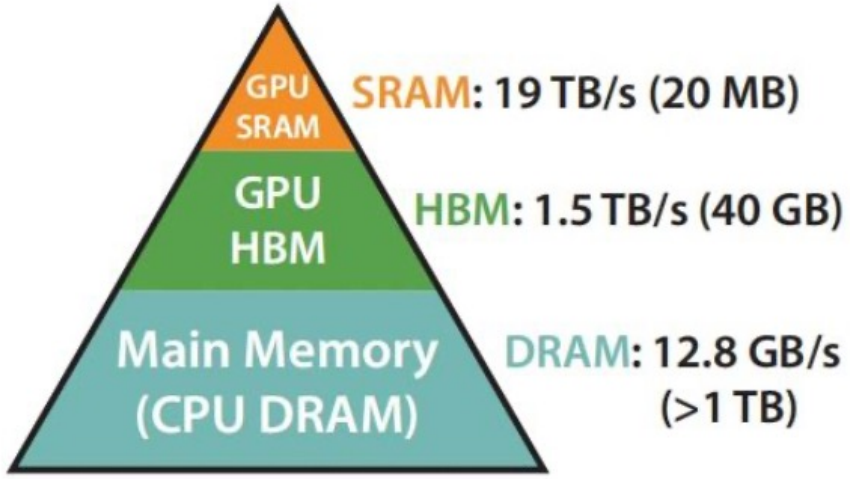




Source: [Prefill and Decode for Concurrent Requests - Optimizing LLM Performance]

Overview

- Hierarchical GPU Memory
  - I/O throughput versus memory size



Memory Hierarchy with Bandwidth & Memory Size

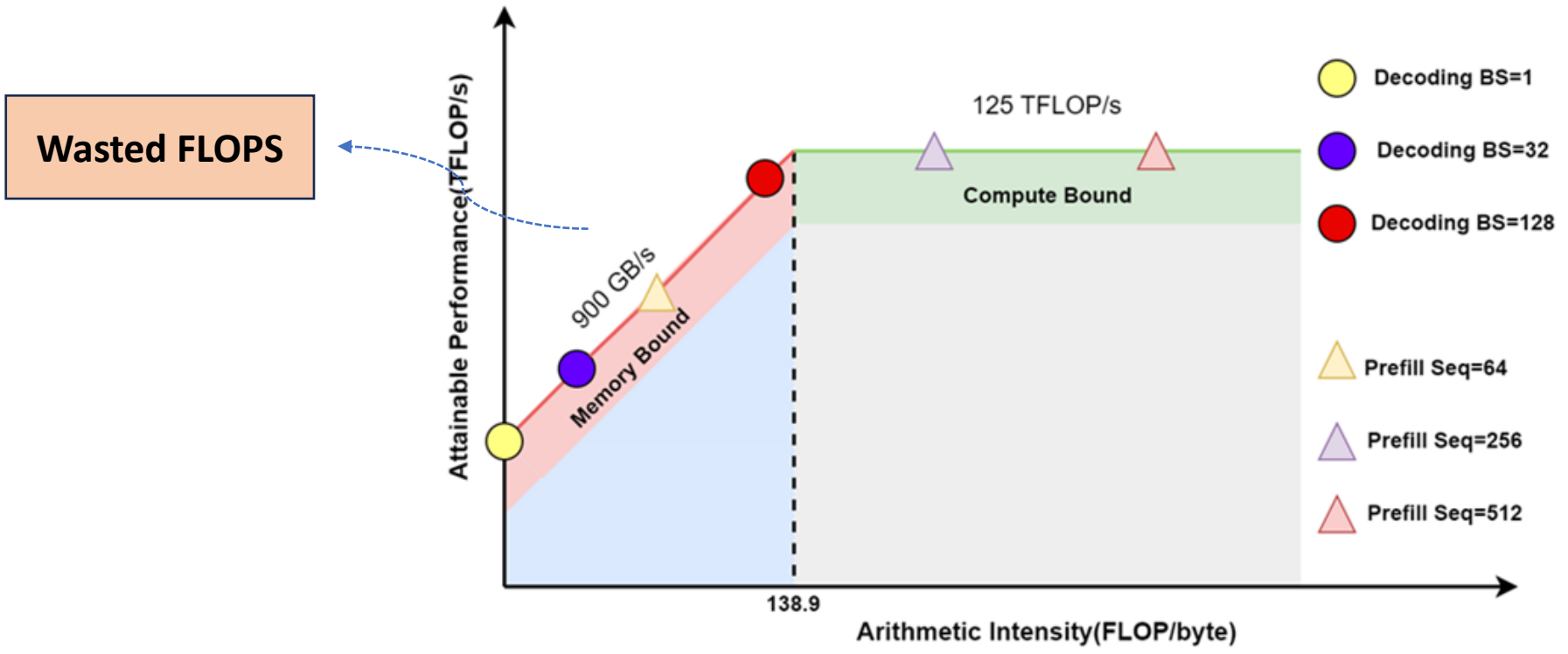
The three memory metrics that are not interchangeable

- capacity (how much fits)
- bandwidth (how fast data moves)
- latency (how long a single access takes).

A macroscope view of I/O tradeoff (V100)

Overview

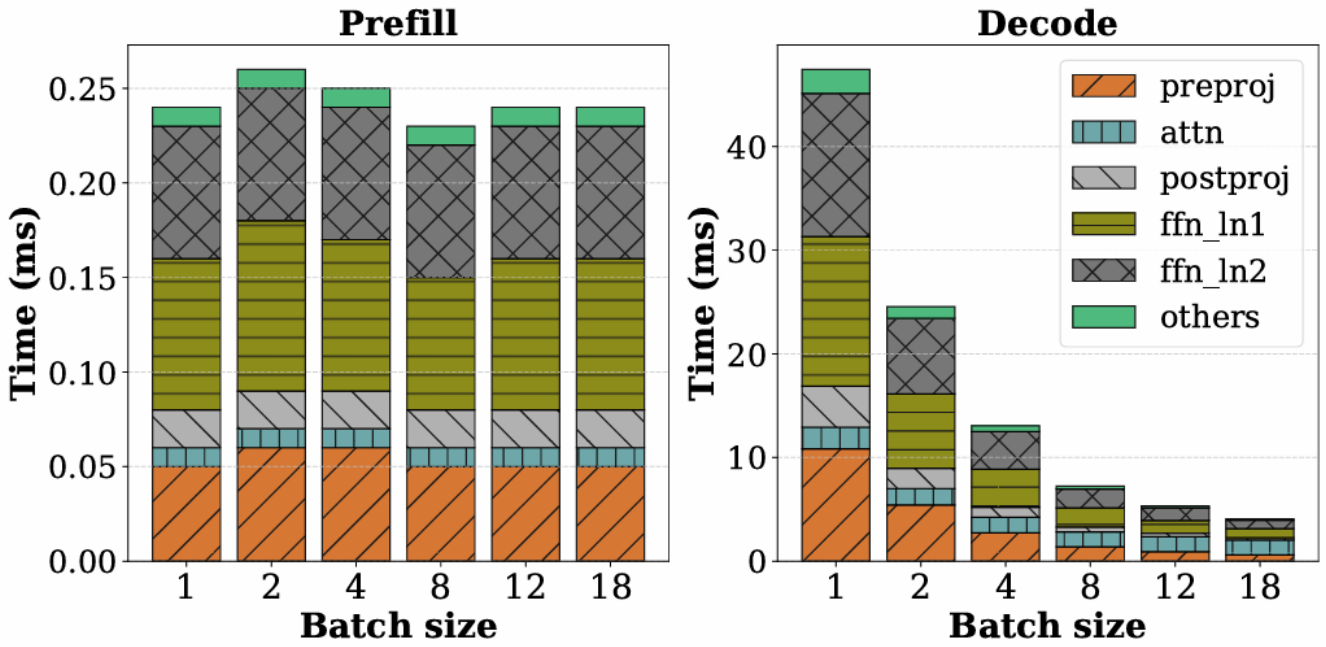
- Prefill and Decode
  - Prefill: Compute intensive, Decode: memory I/O intensive



Roofline model of NVIDIA V100 GPU

Overview

- Prefill and Decode
  - Prefill: Compute intensive, Decode: memory I/O intensive
    - Prefill saturates GPU compute even at batch size of 1
    - Decode under-utilizes GPU compute and costs as much as 200 times prefill for bs=1



Per-token prefill and decode time with different batch sizes (sequence length = **1024**) for LLaMa-13B on A6000 GPU

Source: SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills

Overview

- Challenges of LLM Inference
  - New token generation paradigm
    - **Prefill** and **Decode**
  - Hierarchical memory
    - **Fast** I/O small size versus **slow** I/O large size
  - Heterogeneous bandwidth
    - **High** intra-machine bandwidth versus **low** inter-machine bandwidth
- To emphasize
  - many optimization methods, e.g. **attention optimization** can be employed for model **training** (e.g. sparse attention, linear attention, flash attention)

Source: Jia-Bin Huang from University of Maryland College Park

Inference Optimization

- Overview
- Attention Optimization
  - **Sparse Attention**
  - Flash Attention
  - Linear Attention
- KV Cache Optimization
- Parallel and Distributed Serving

Attention Optimization

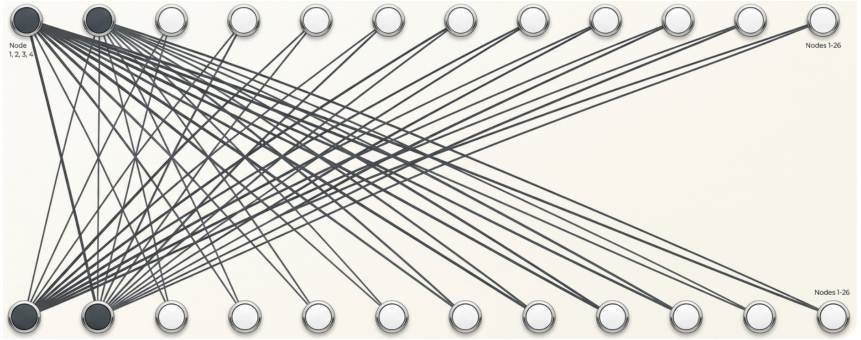
- Real-world operations for $softmax(Q \cdot K^T)$
  - Transferring both matrices from global memory to shared memory for computing, and write the result back to global memory progressively
- Complexity (naïve method)
  - Computation: $O(N^2 d)$
    - Considering multiplications and additions → computing time $T_C = \frac{2N^2 d}{C}$
  - Communication: $O(N^2)$
    - Considering bidirectional I/O read/write → communication time $T_{I/O} = \frac{2*(N^2+2Nd)}{B}$
  - Global storage: $O(N^2)$

(partially) overlapped

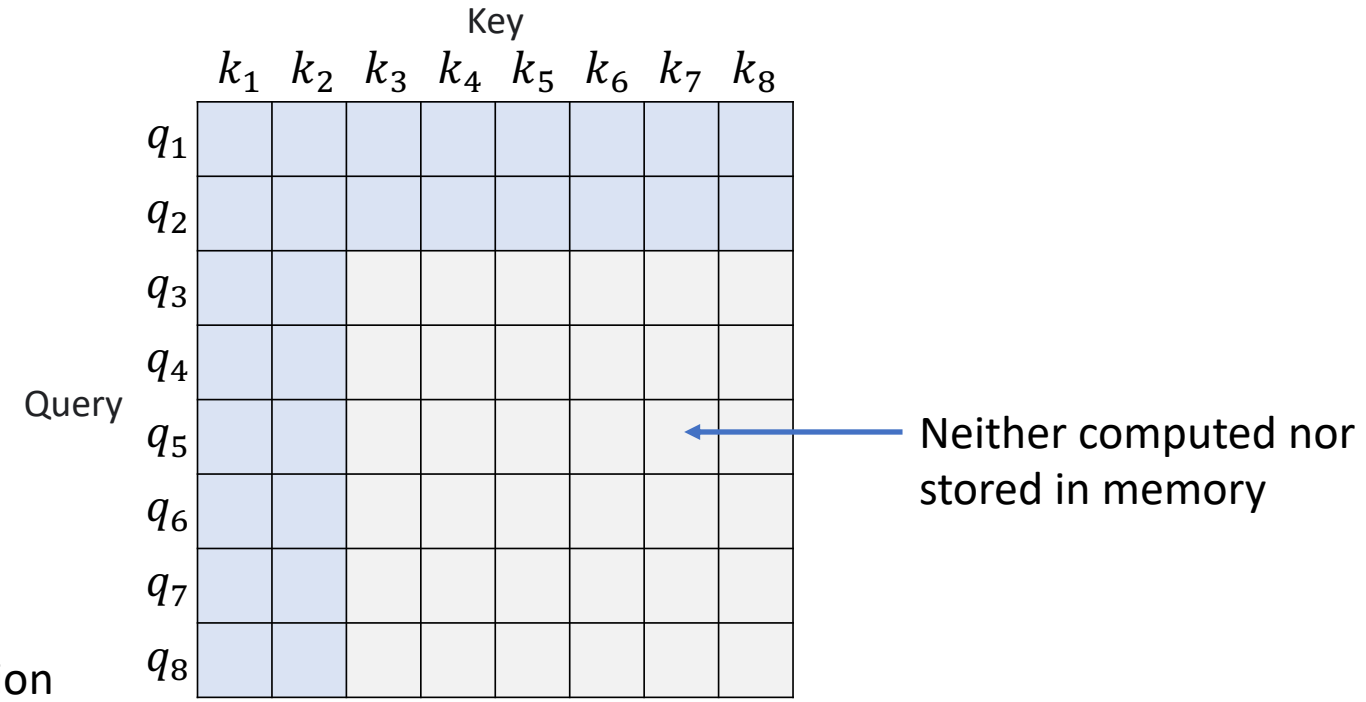
GPT3 with 10K tokens in FP16:
200MB per head per layer

Sparse Attention (Content Agnostic)

- Global attention
  - global nodes act as an information hub, allowing them to attend to every other node in the sequence

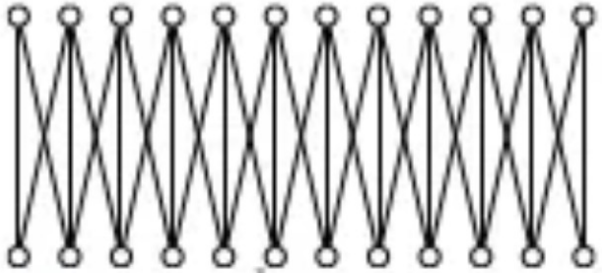


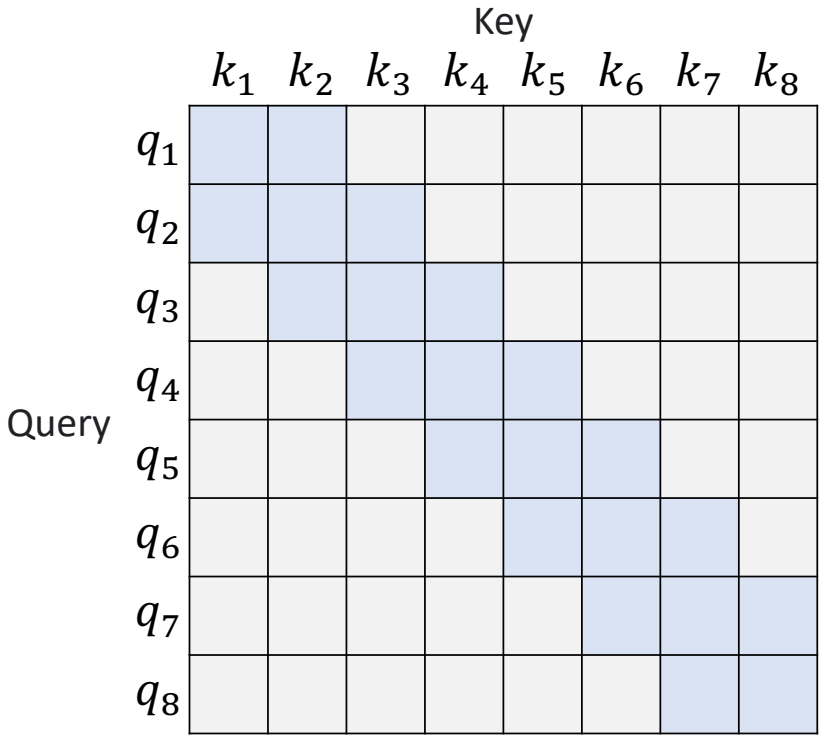
A bipartite graph illustration of QK^T computation



Sparse Attention (Content Agnostic)

- Band attention
  - often termed “local attention” or “sliding window attention”, in which a node’s attentions are confined to neighboring nodes in a local window

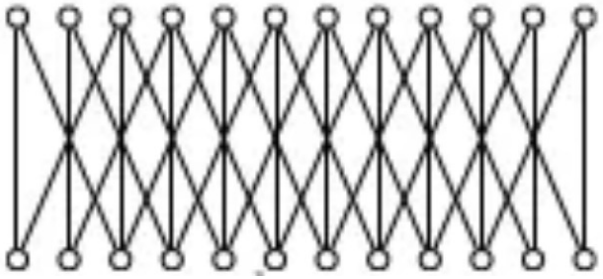


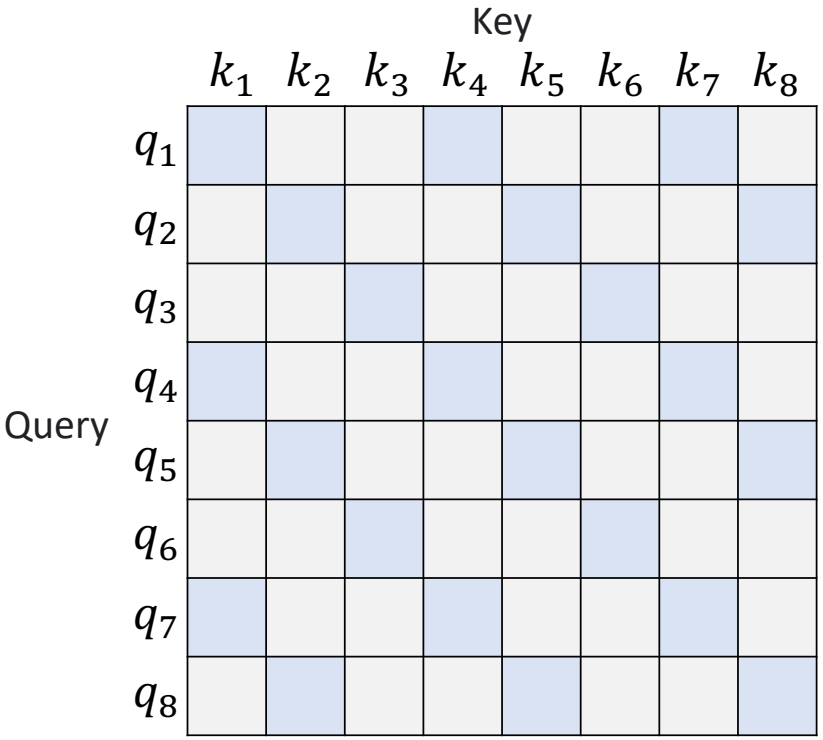


- Reducing complexity from $O(N^2)$ to $O(kN)$
- Limited receptive field (cannot capture long-range dependencies)
- Sacrificing context modeling ability: slow information propagation

Sparse Attention (Content Agnostic)

- Dilated attention
  - using a dilated window with gaps of dilation equal to or greater than 1

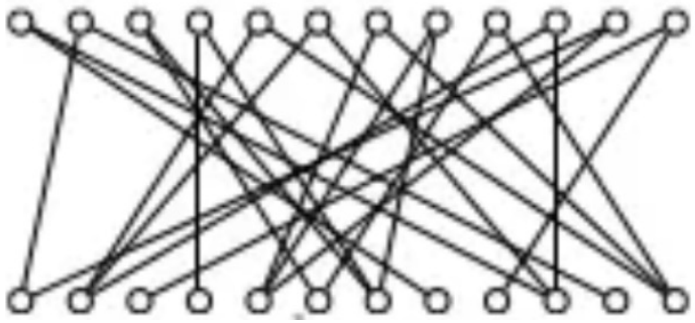


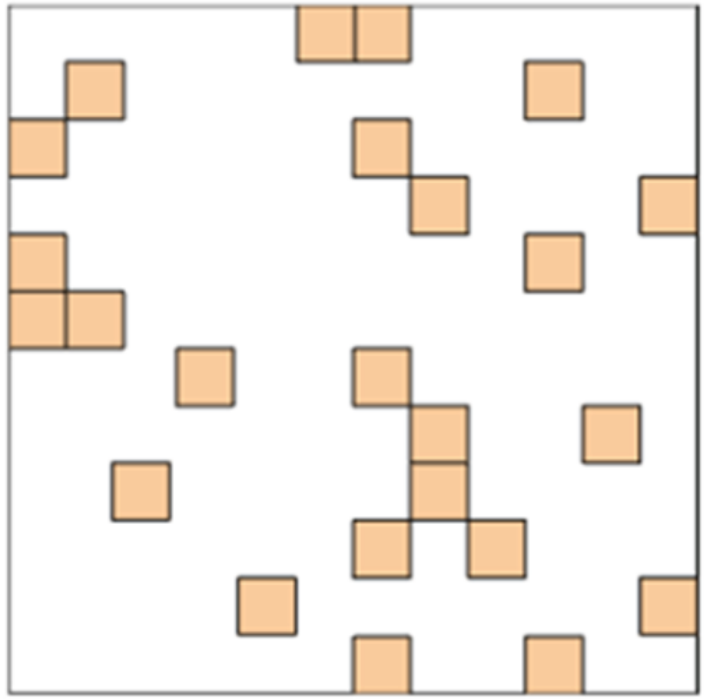


- Reducing complexity from $O(N^2)$ to $O(kN)$
- Missing fine-grained, short-range dependencies (failure to capture important neighbors in those gaps)

Sparse Attention (Content Agnostic)

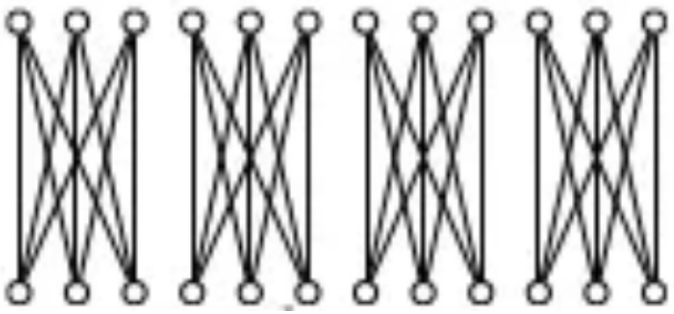
- Random attention
  - each query randomly samples a few keys for better capturing non-local interactions
  - needs less efficient “gather” op

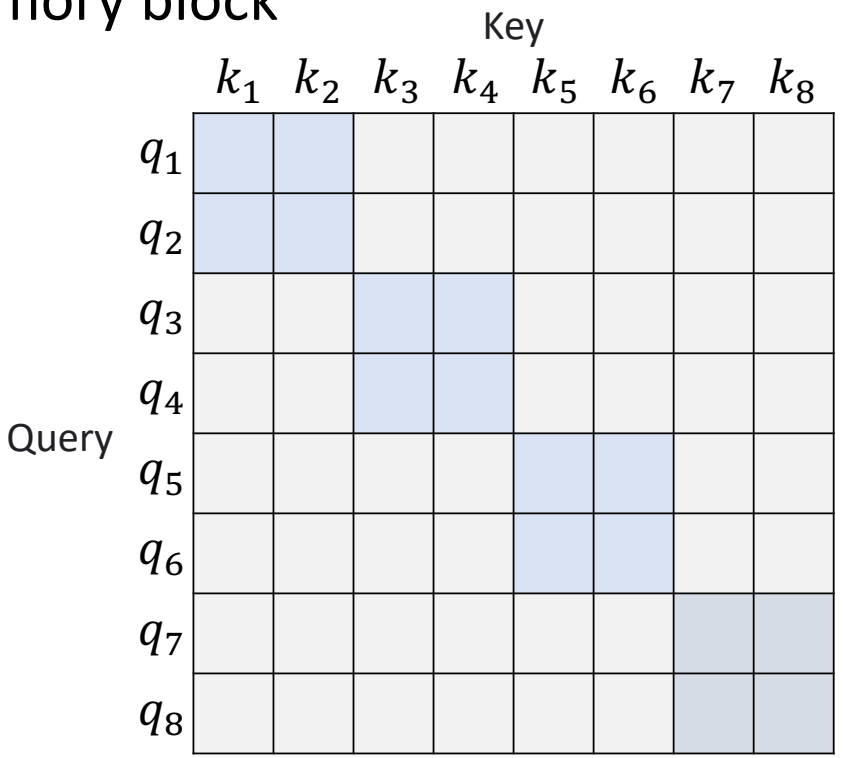




Sparse Attention (Content Agnostic)

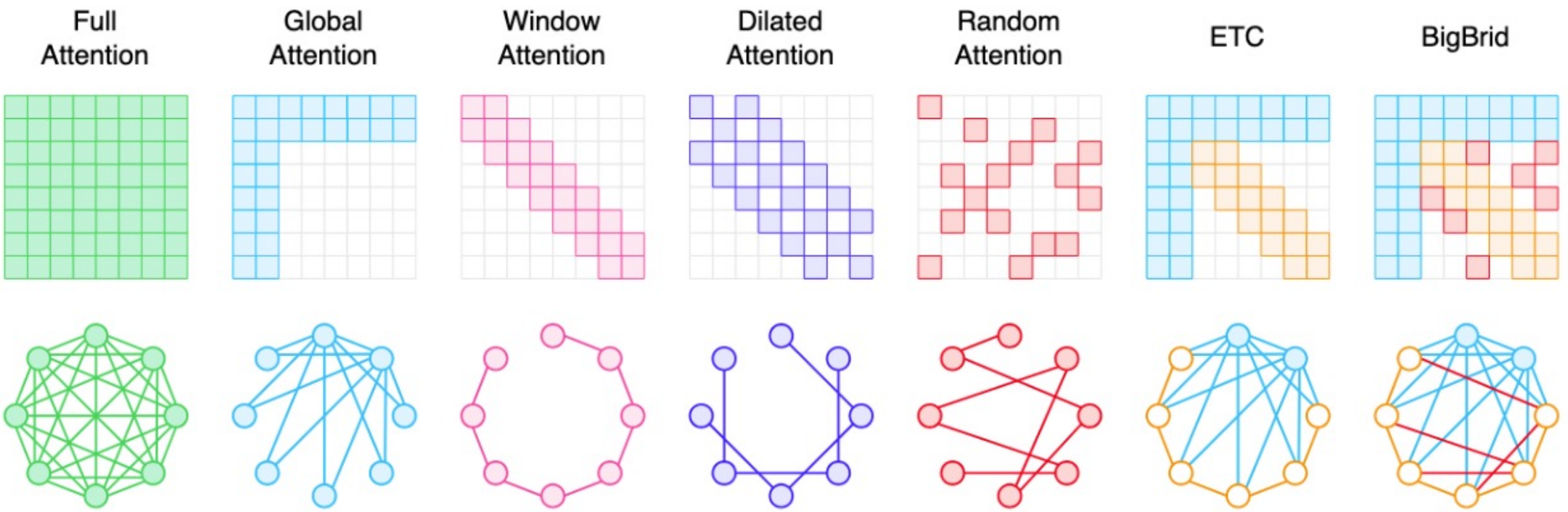
- Block attention
  - input sequence segmented into multiple non-intersection query blocks and each block assigned a local memory block





Sparse Attention (Content Agnostic)

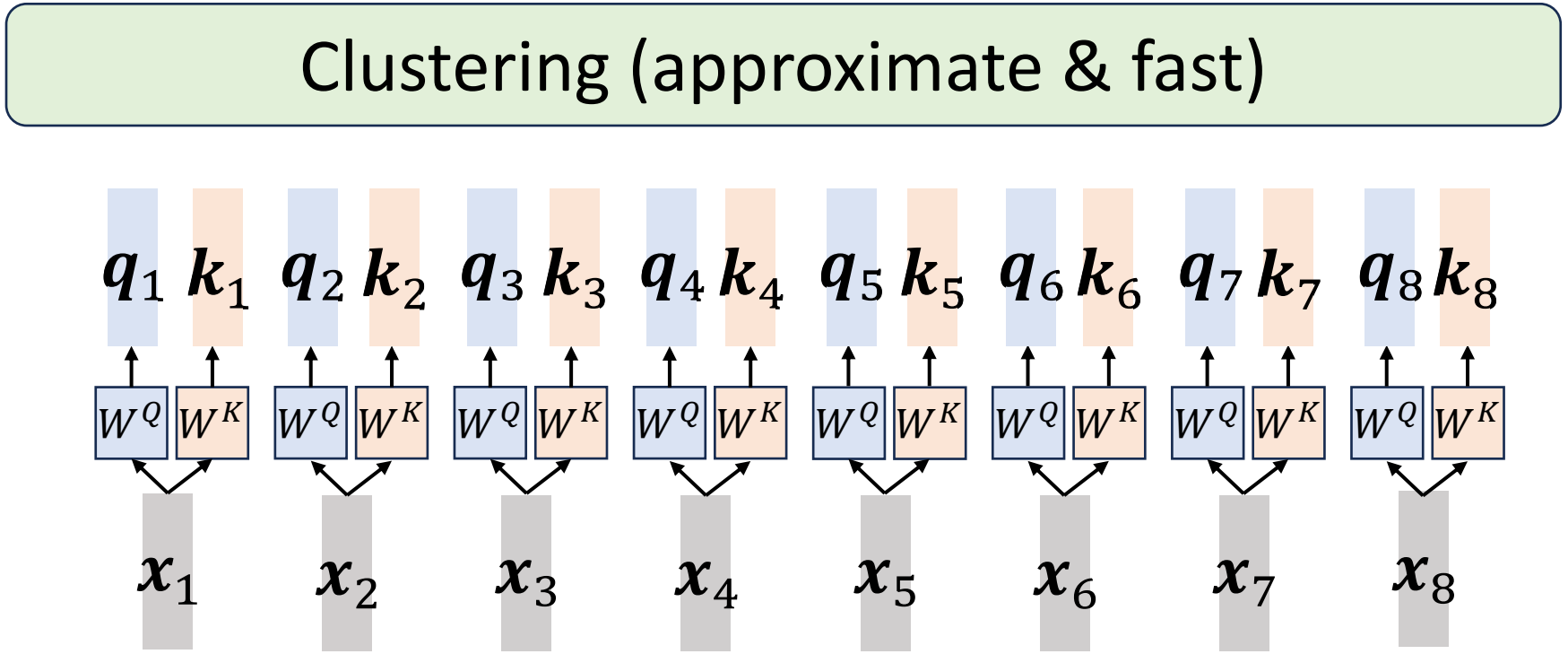
- Basic and compound sparse attention schemes



Source: Speed Always Wins: A Survey on Efficient Architectures for Large Language Models

Sparse Attention (content aware)

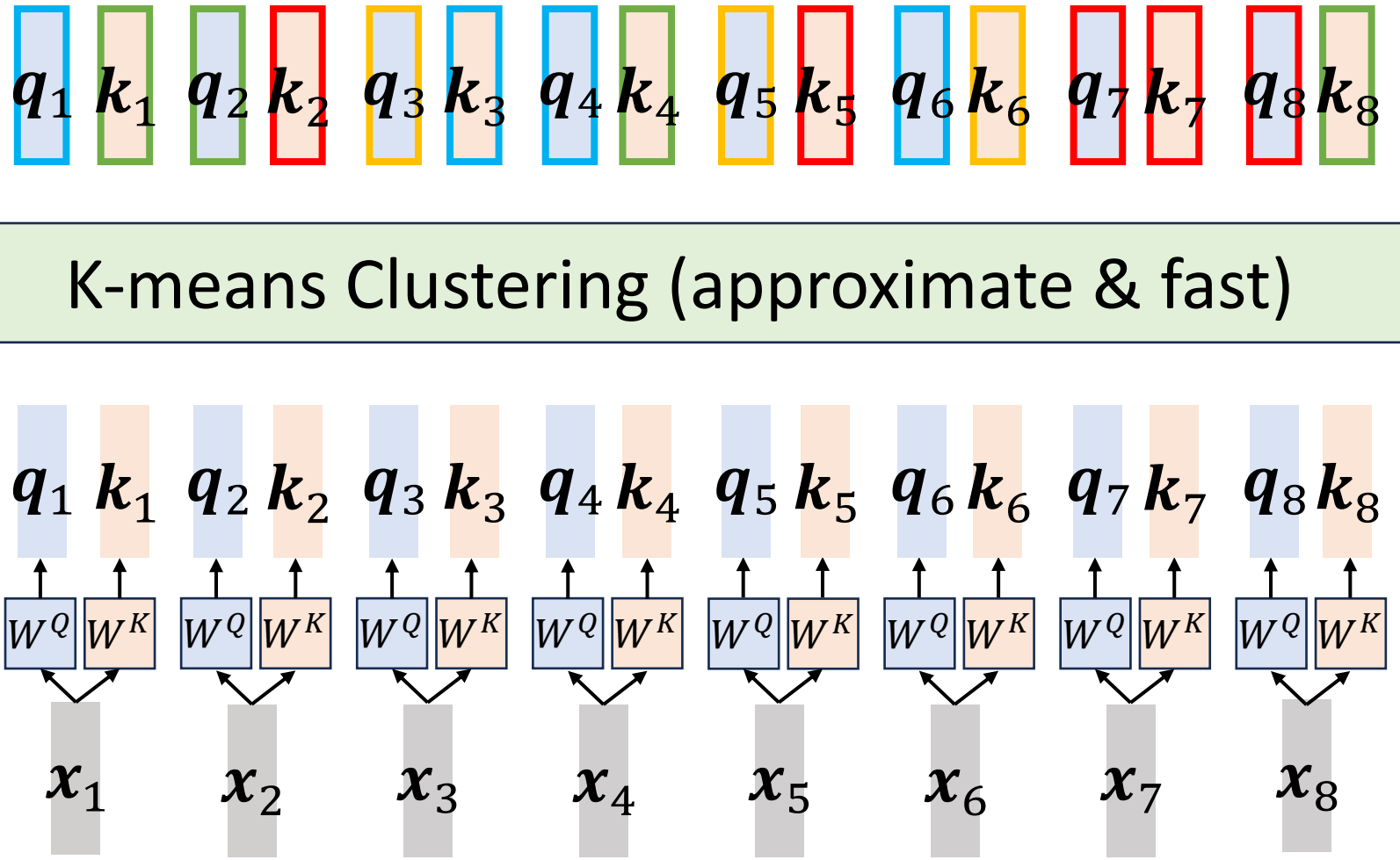
- Routing transformer
  - Key idea: a large inner product means that query and key are “similar”



Source: Jia-Bin Huang from University of Maryland College Park

Sparse Attention (content aware)

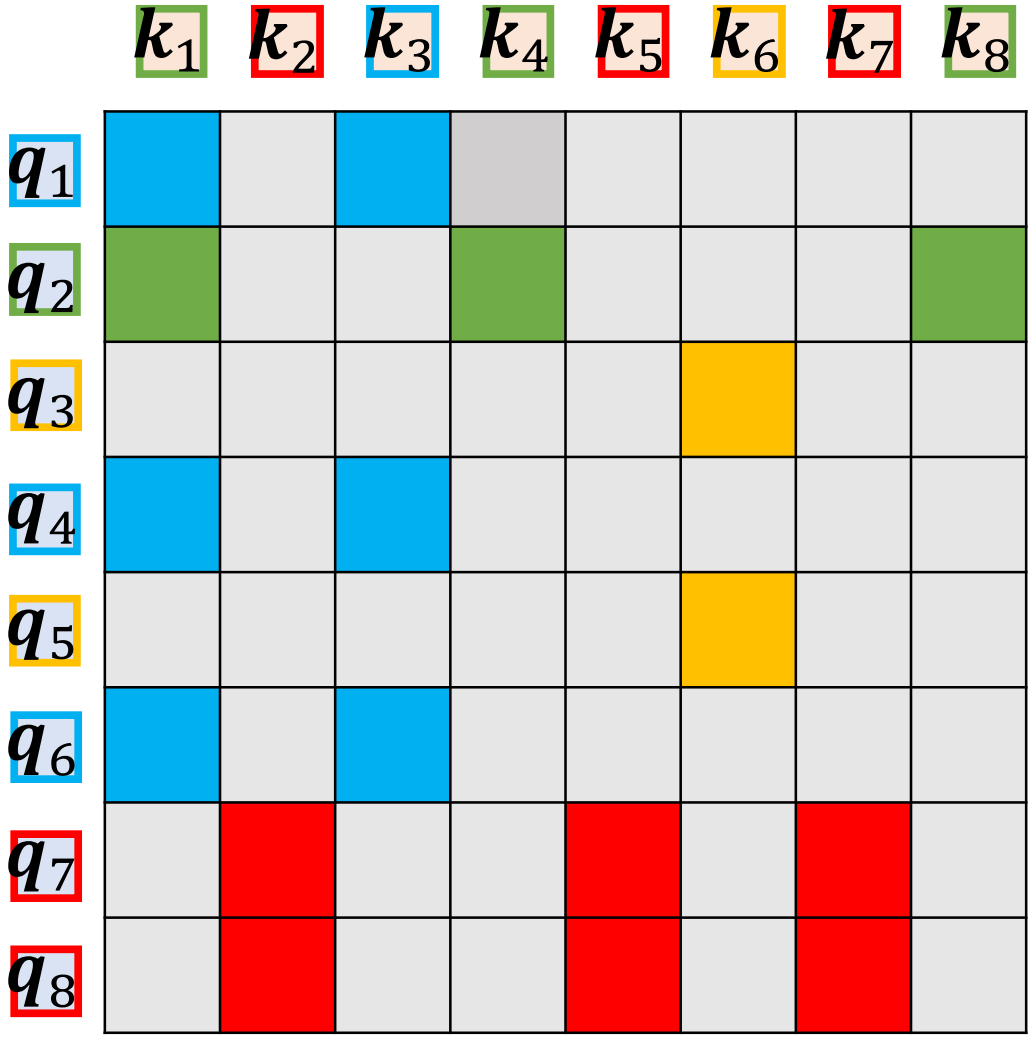
- Routing transformer: online k-means assigns tokens to clusters; attention within each cluster



Source: Jia-Bin Huang from University of Maryland College Park

Sparse Attention (content aware)

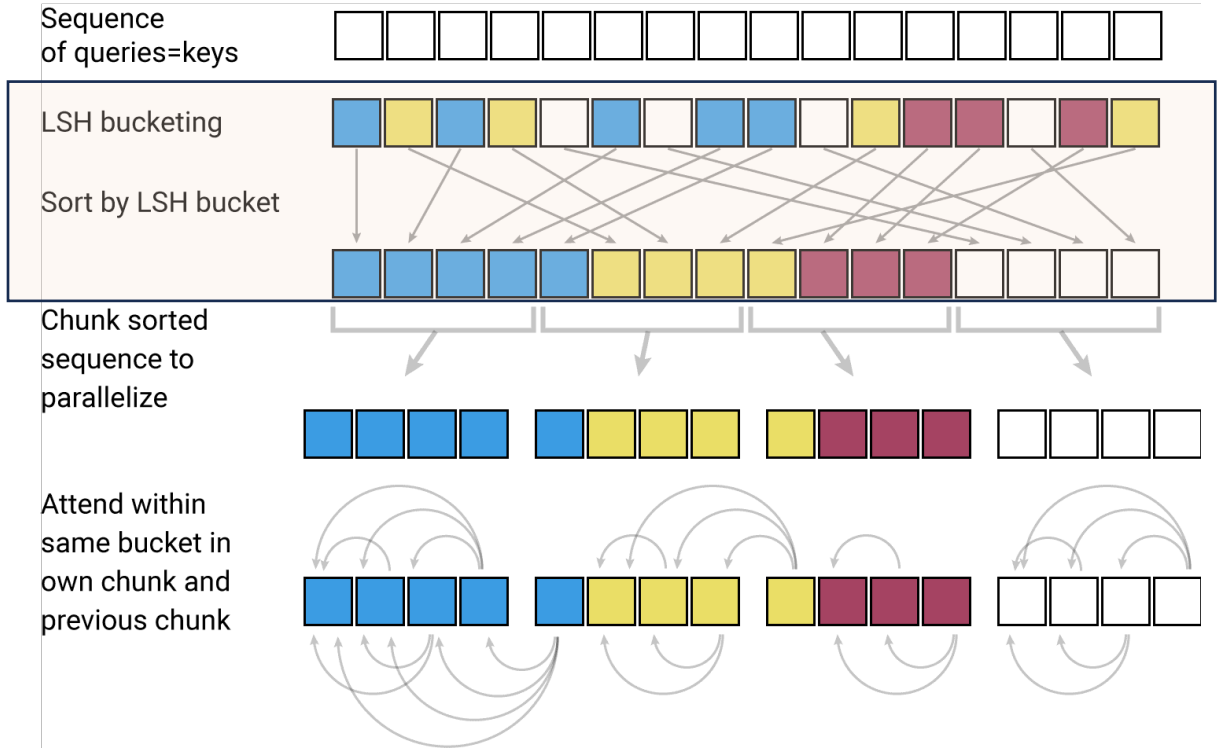
- Clusters
  - $\{q1, q4, q6, k1, k3\}$
  - $\{q2, k1, k4, k8\}$
  - $\{q3, q5, k6\}$
  - $\{q7, q8, k2, k5, k7\}$
- Updating centroid
  - Exponentially weighted moving average (refer to the paper)



Source: Jia-Bin Huang from University of Maryland College Park

Sparse Attention (content aware)

- **Reformer**: bucketing keys and queries (using LSH), and only computing the attention scores for those inside the same bucket



- Using **locality-sensitive hashing** (LSH) to select key-value pairs for each query
- Using Reversible Transformer to reduce memory usage during training
- Similar items (queries and keys) falling in the same bucket with high probability
- Processing sequences of length **64K tokens** or more efficiently

Source: Reformer: The Efficient Transformer

Inference Optimization

- Overview
- Attention Optimization
  - Sparse Attention
  - **Flash Attention**
  - Linear Attention
- KV Cache Optimization
- Parallel and Distributed Serving

Flash Attention

- Prior works
  - Reducing # of scaled dot-product, potentially **sacrificing** attention performance
  - Compute is fast while reading K and V, and writing S back the results are slow
- Challenges
  - I/O cost of loading/storing Q, K and V are non-trivial (because of small SRAM size)

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 $\mathbf{O}$.

Memory Load/Read Memory Write/Store

Flash Attention

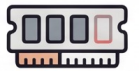
- **From Softmax to Safe Softmax**

Standard Softmax

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Safe Softmax

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^{n} e^{x_j - \max(x)}}$$

- **Online Safe Softmax:** not enough space to store entire x 
  - Alg 1: three-pass algorithm

Definitions & States		
Variable	**Meaning**	**Explanation**
m_i	$\max_{j=1}^{i}\{x_j\}$	Running max, $m_0 = -\infty$
d_i	$\sum_{j=1}^{i} e^{x_j - m_N}$	Accumulated Denominator, uses final m_N, is precise)
a_i	is the final Softmax output	Softmax value for x_i

1 Pass 1: **Find Global Maximum** (m_N)
for $i \leftarrow 1$ **to** N **do**
  $m_i \leftarrow \max(m_{i-1}, x_i)$
end

→ m_N (final)

2 Pass 2: **Calculate Denominator Sum** (d_N)
for $i \leftarrow 1$ **to** N **do**
  $d_i \leftarrow d_{i-1} + \exp(x_i - m_N)$
end

→ d_N (final)

3 Pass 3: **Generate Final Output** (a_i)
for $i \leftarrow 1$ **to** N **do**
  $a_i \leftarrow \exp(x_i - m_N)/d_N$
end

Flash Attention

Online Safe Softmax

- d_i can be computed iteratively

Iterative Denominator Derivation

$$
\begin{aligned}
d'_i &= \sum_{j=1}^{i} e^{x_j - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_i} \right) + e^{x_i - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_{i-1}} \right) e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&= d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}
\end{aligned}
$$

ALGORITHM: 2-PASS ONLINE SOFTMAX

Pass 1: Max & Denominator Sum Update

for $i \leftarrow 1$ **to** N **do**

$m_i \leftarrow \max(m_{i-1}, x_i)$

$d'_i \leftarrow d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$

end for

Pass 2: Compute Final Softmax Output

for $i \leftarrow 1$ **to** N **do**

$a_i \leftarrow \dfrac{e^{x_i - m_N}}{d'_N}$

end for

Flash Attention

One-pass Attention

- Define an intermediate output
- Compute it iteratively in a single streaming pass

Intermediate Output Definition

$$o'_i := \left(\sum_{j=1}^{i} \frac{e^{x_j - m_i}}{l'_i} V[j,:] \right)$$

Flash Attention (1-Pass)

for $i \leftarrow 1$ **to** N **do**

$x_i \leftarrow Q[k,:]K^T[:,i]$

$m_i \leftarrow \max(m_{i-1}, x_i)$

$l'_i \leftarrow l'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$

$o'_i \leftarrow o'_{i-1} \frac{l'_{i-1}}{l'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{l'_i} V[i,:]$

end

$O[k,:] \leftarrow o'_N$

Iterative Derivation

$$o'_i = \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{l'_i} V[j,:]$$

$$= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{l'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{l'_i} V[i,:]$$

$$= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{l'_{i-1}} \cdot \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \cdot \frac{l'_{i-1}}{l'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{l'_i} V[i,:]$$

$$= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{l'_{i-1}} V[j,:] \right) \cdot \frac{l'_{i-1}}{l'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{l'_i} V[i,:]$$

$$= o'_{i-1} \cdot \frac{l'_{i-1}}{l'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{l'_i} V[i,:]$$

Online Rescaling Factor

[Stream Token (K_i, V_i)] → [Update max m_i & sum l'_i] → [Rescale o'_{i-1} & Add New V_i in SRAM] → [Final Output o'_N]

Flash Attention

Tiling: loading and computing at the block granularity

Pseudocode of Flash Attention

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil$, $B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.

2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}$, $\ell = (0)_N \in \mathbb{R}^N$, $m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

5: **for** $1 \le j \le T_r$ **do**

6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

7: **for** $1 \le i \le T_r$ **do**

8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

14: **end for**

15: **end for**

16: Return $\mathbf{O}$.

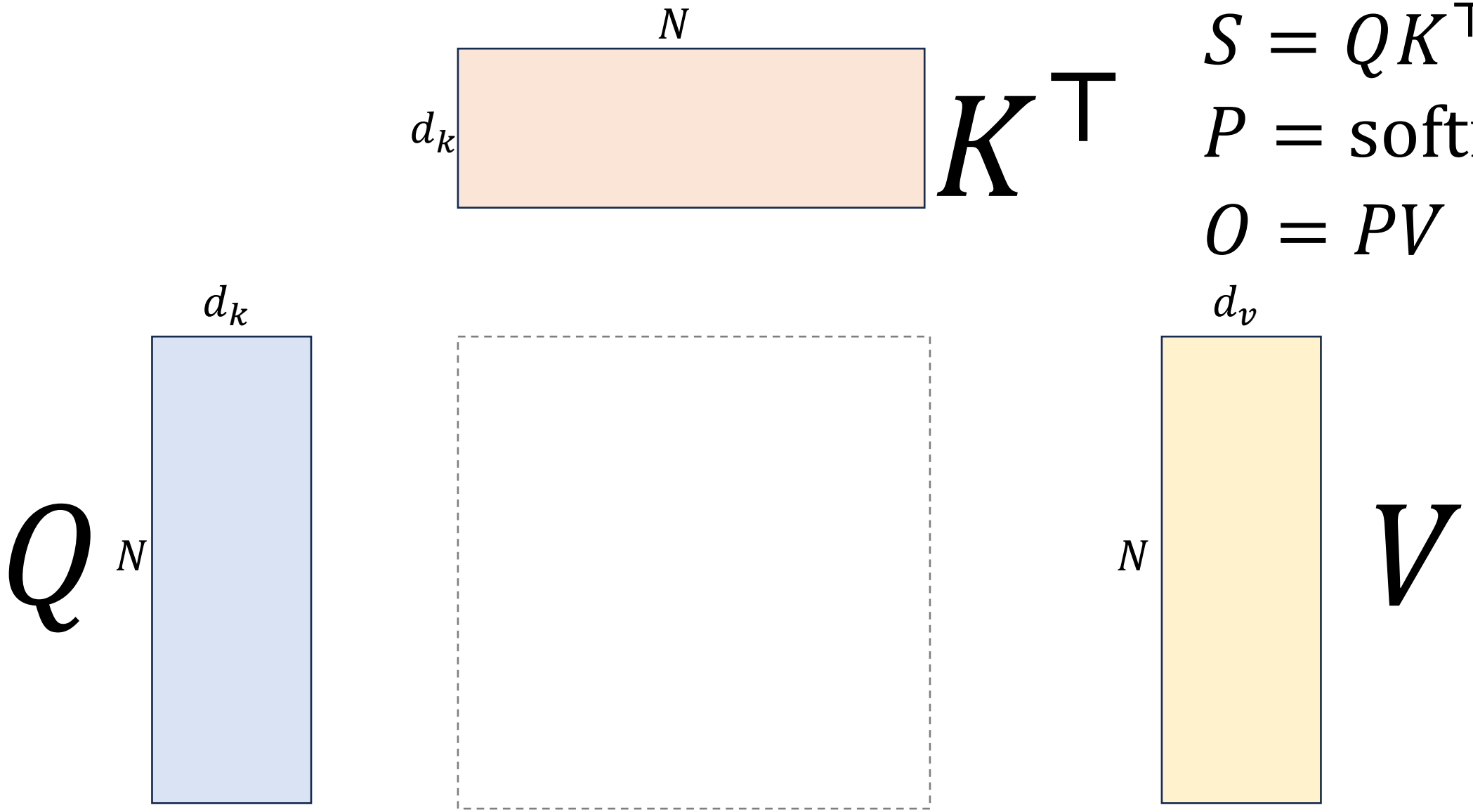
Flash Attention

Outer Loop on Key/Value (over index *j*)

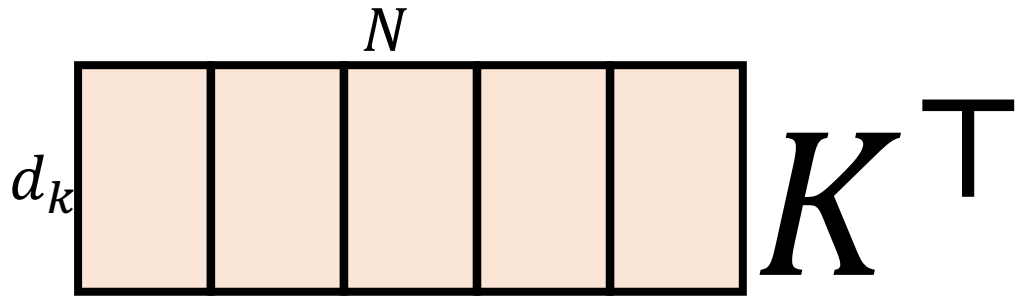
Inner Loop on Query/Output (over index *i*)

Write O_i to HBM
Write l_i to HBM

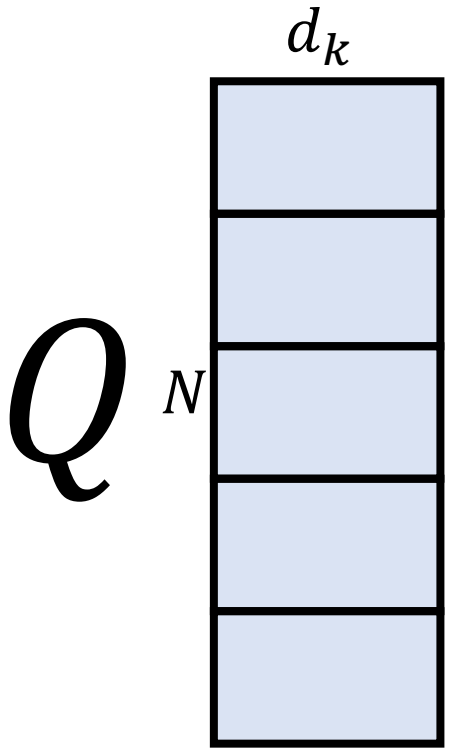
$$S = QK^\top$$
$$P = \text{softmax}(S)$$
$$O = PV$$

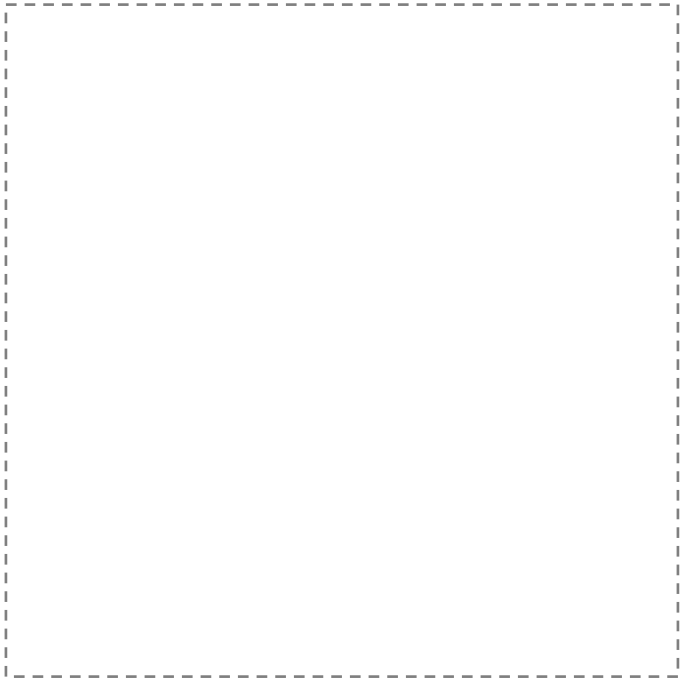


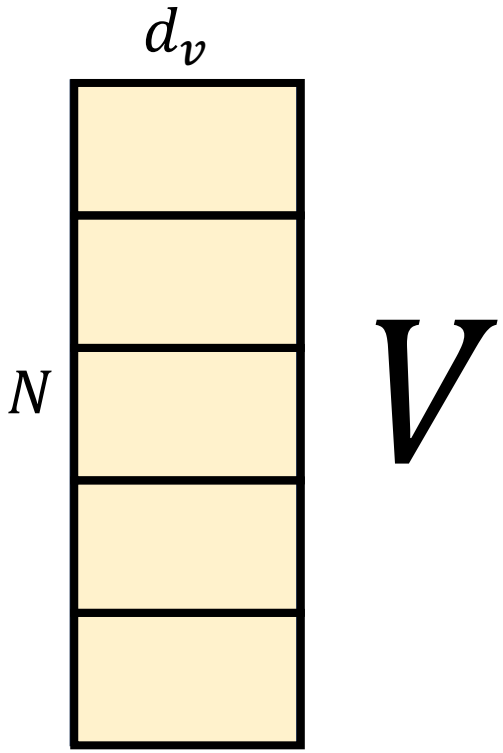
Source: Jia-Bin Huang from University of Maryland College Park



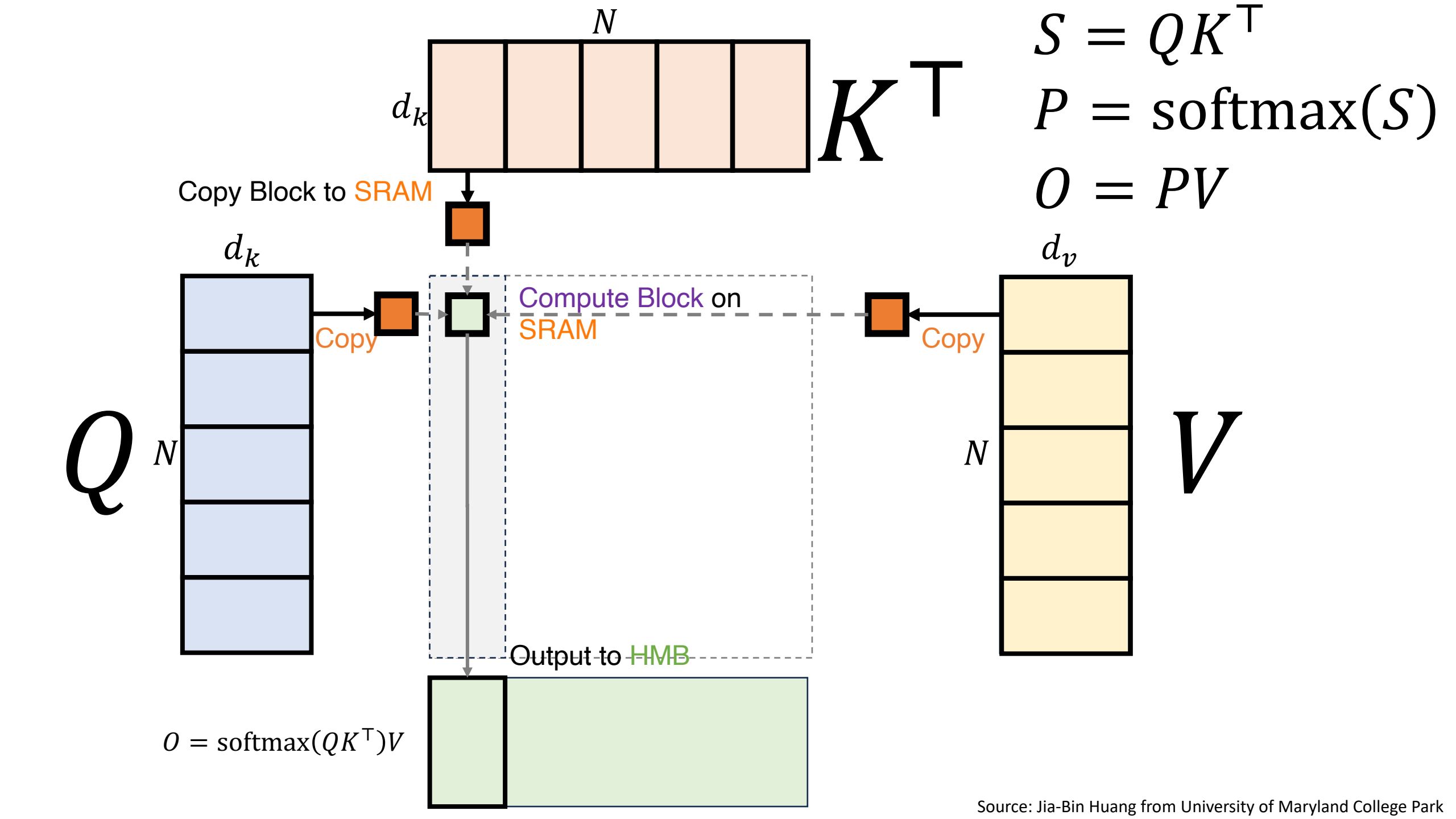
$$S = QK^\top$$
$$P = \text{softmax}(S)$$
$$O = PV$$



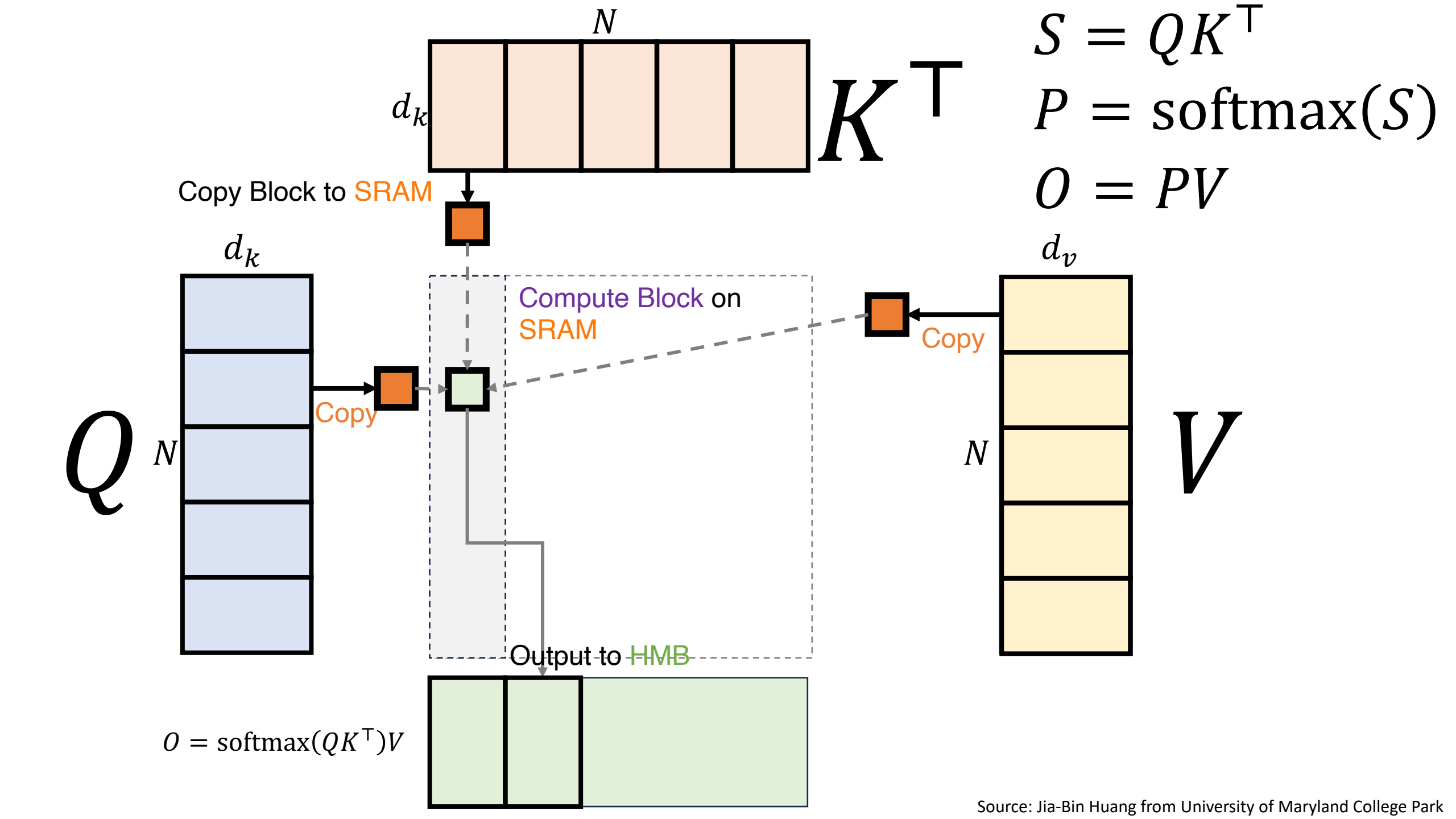




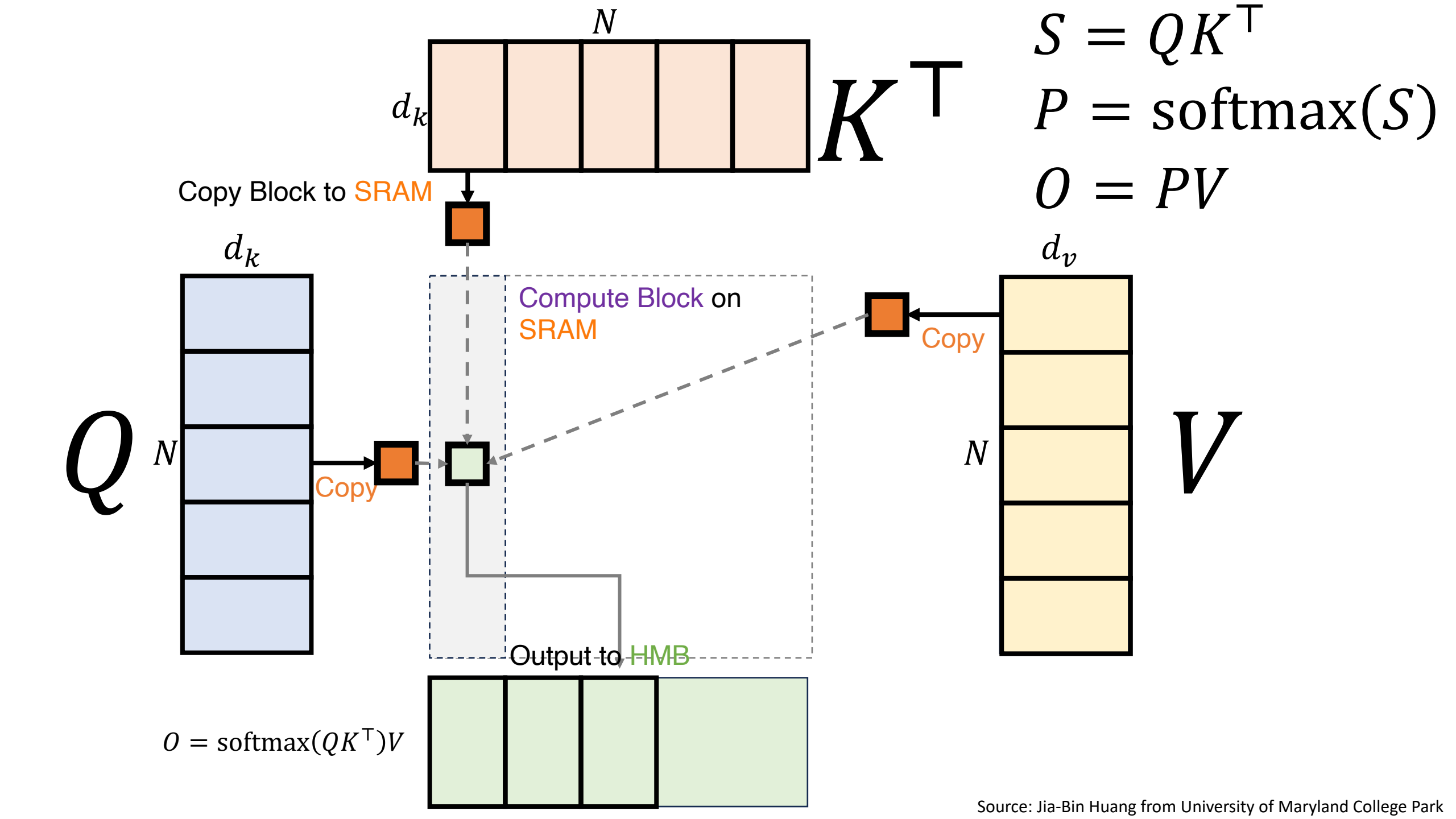
Source: Jia-Bin Huang from University of Maryland College Park



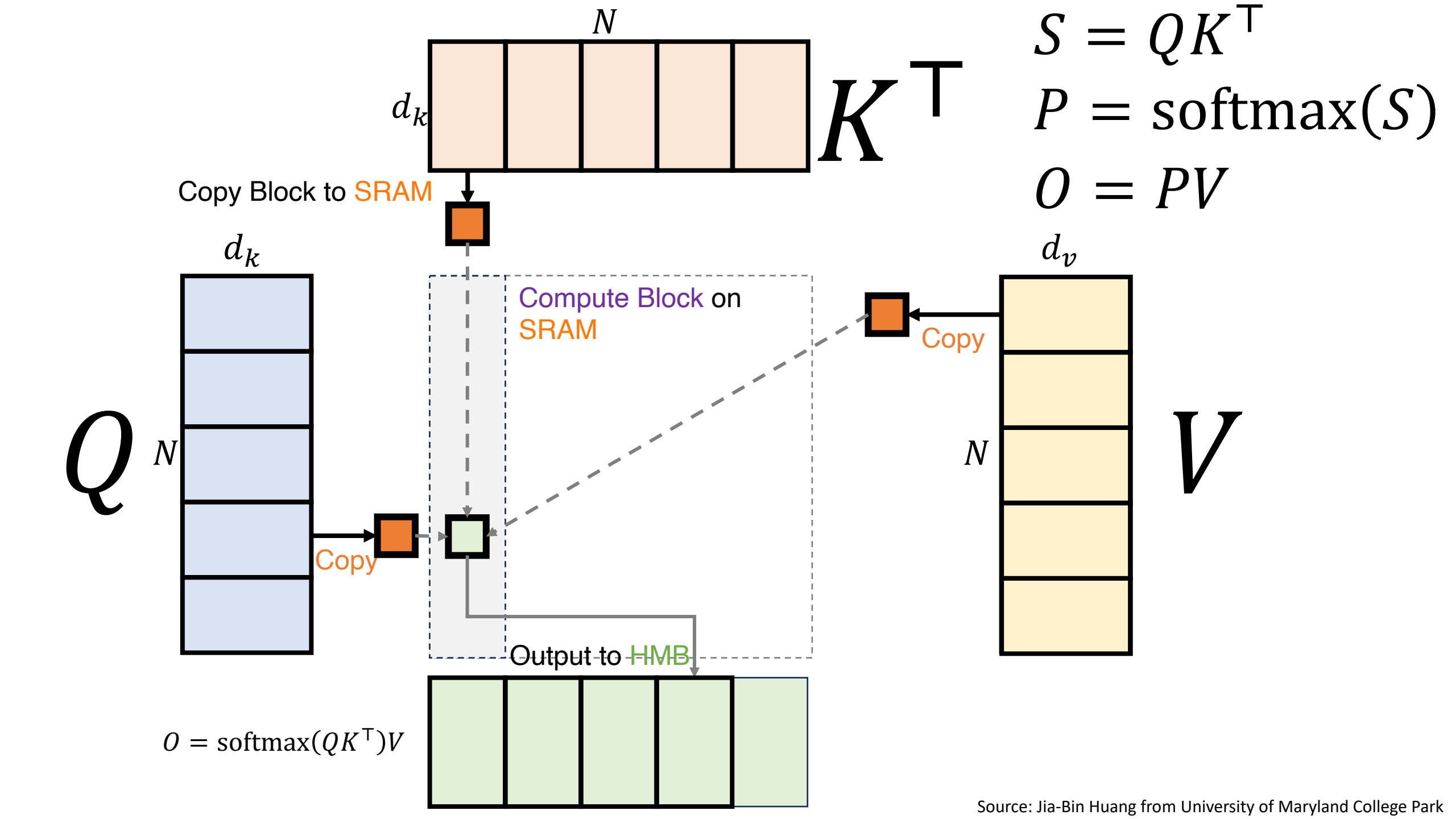
Source: Jia-Bin Huang from University of Maryland College Park



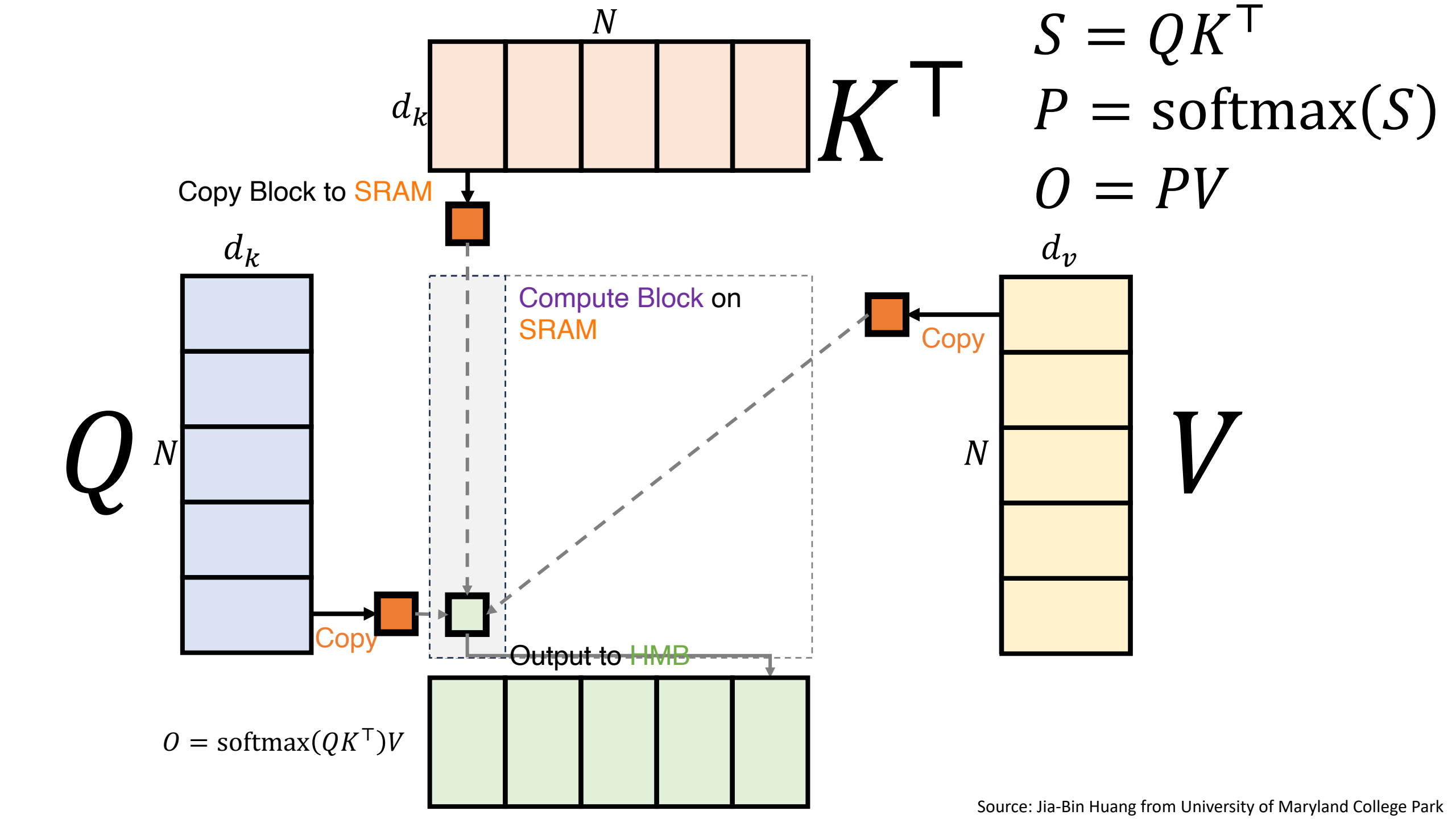
Source: Jia-Bin Huang from University of Maryland College Park



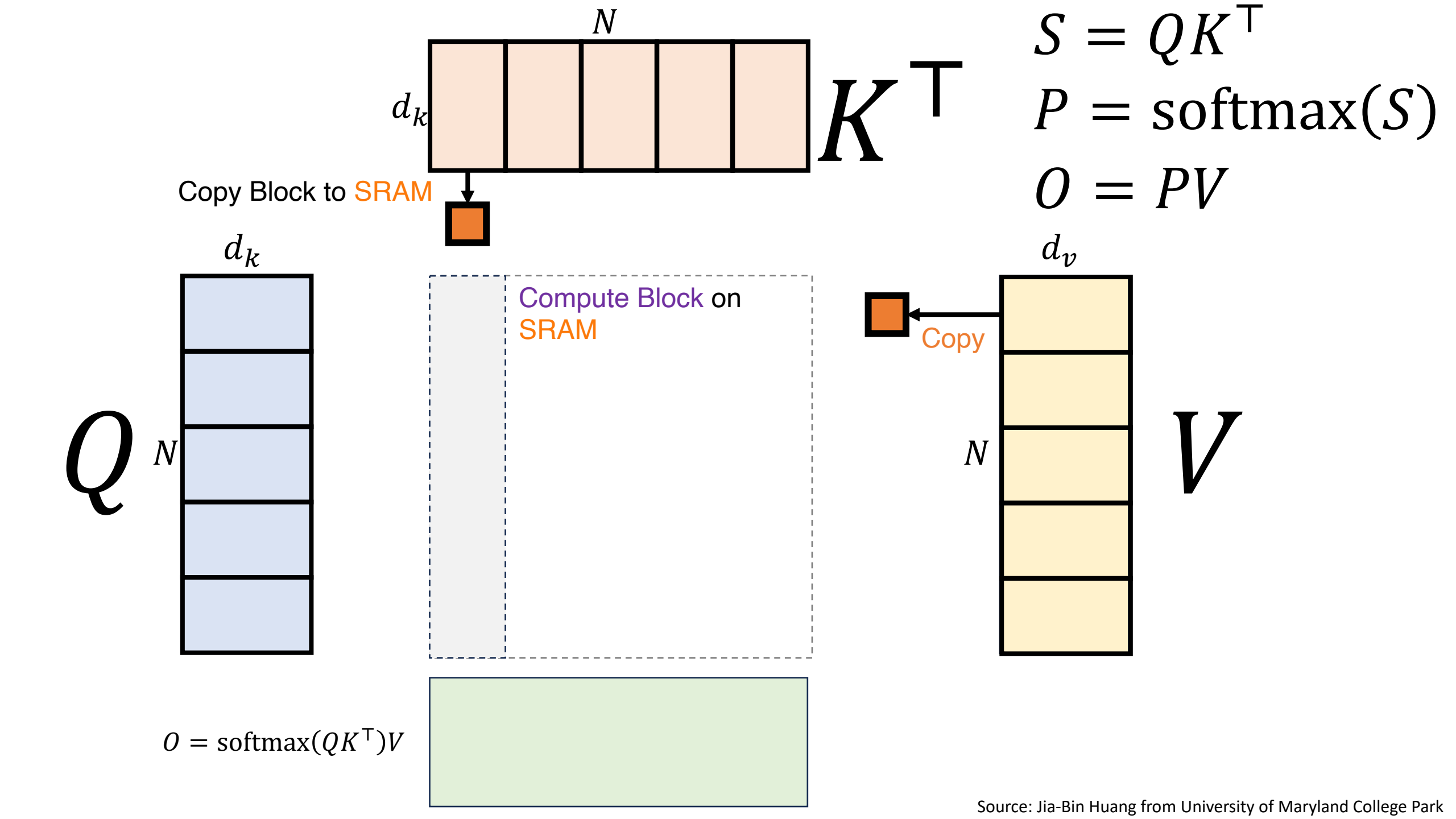
Source: Jia-Bin Huang from University of Maryland College Park



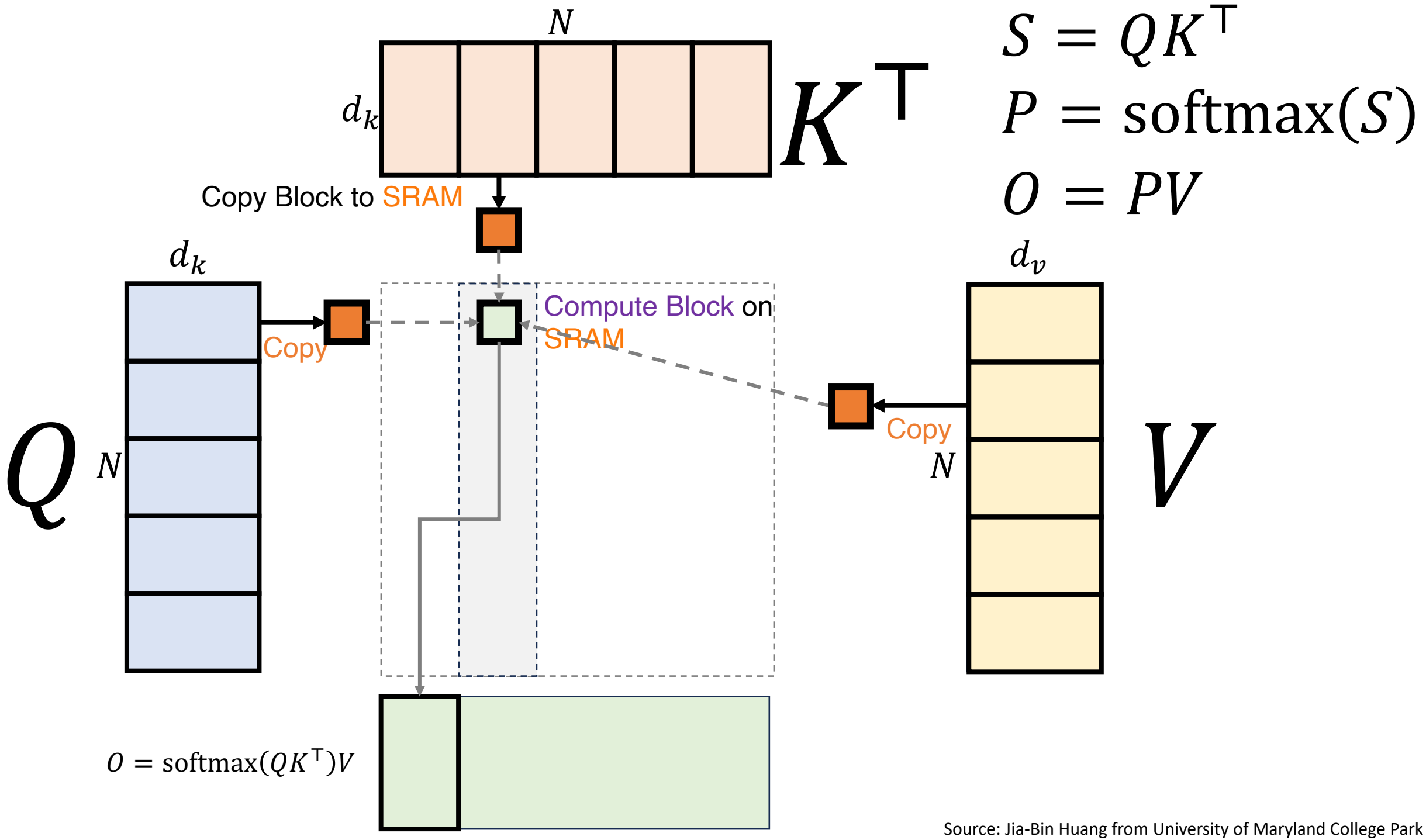
Source: Jia-Bin Huang from University of Maryland College Park



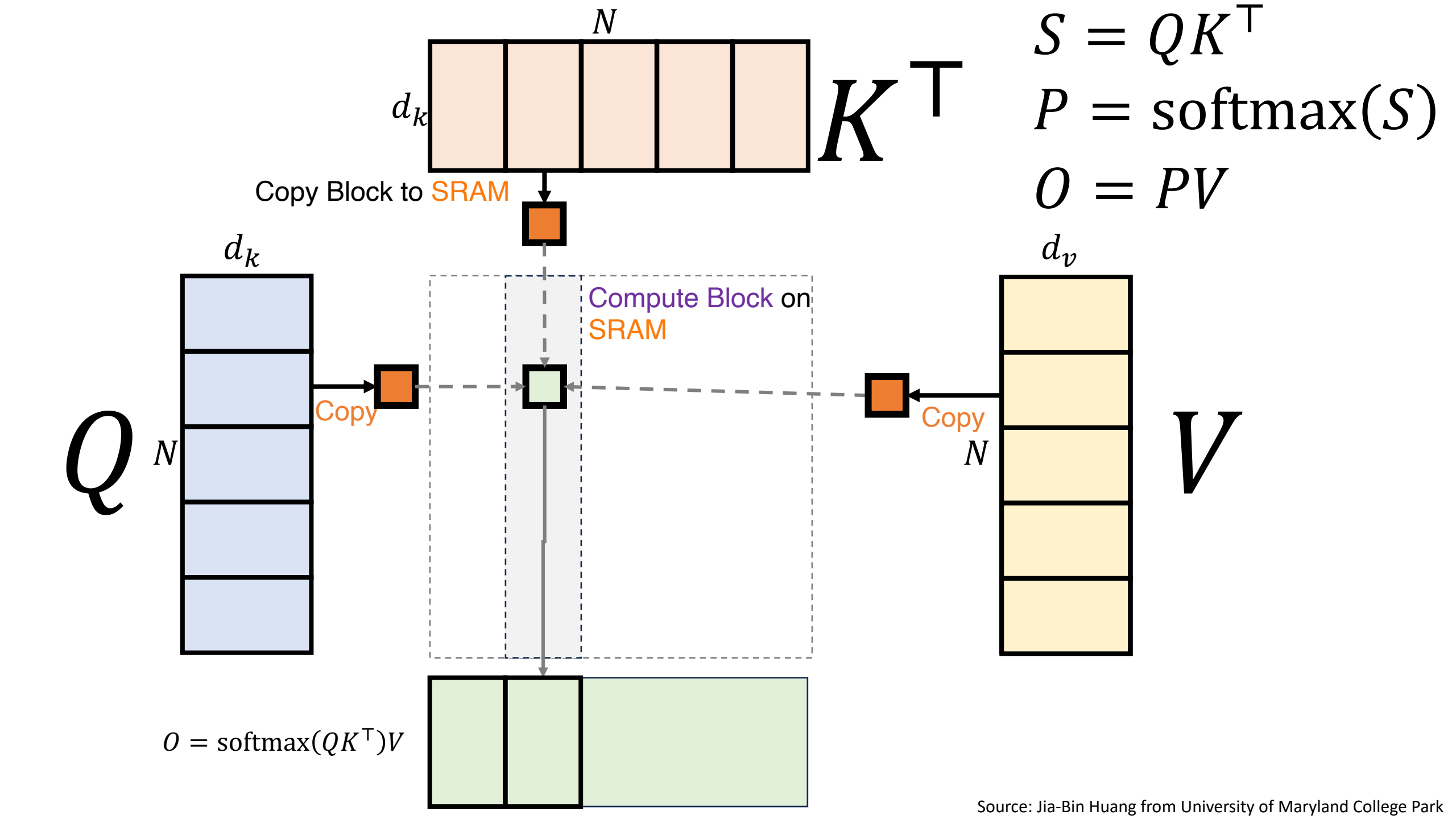
Source: Jia-Bin Huang from University of Maryland College Park



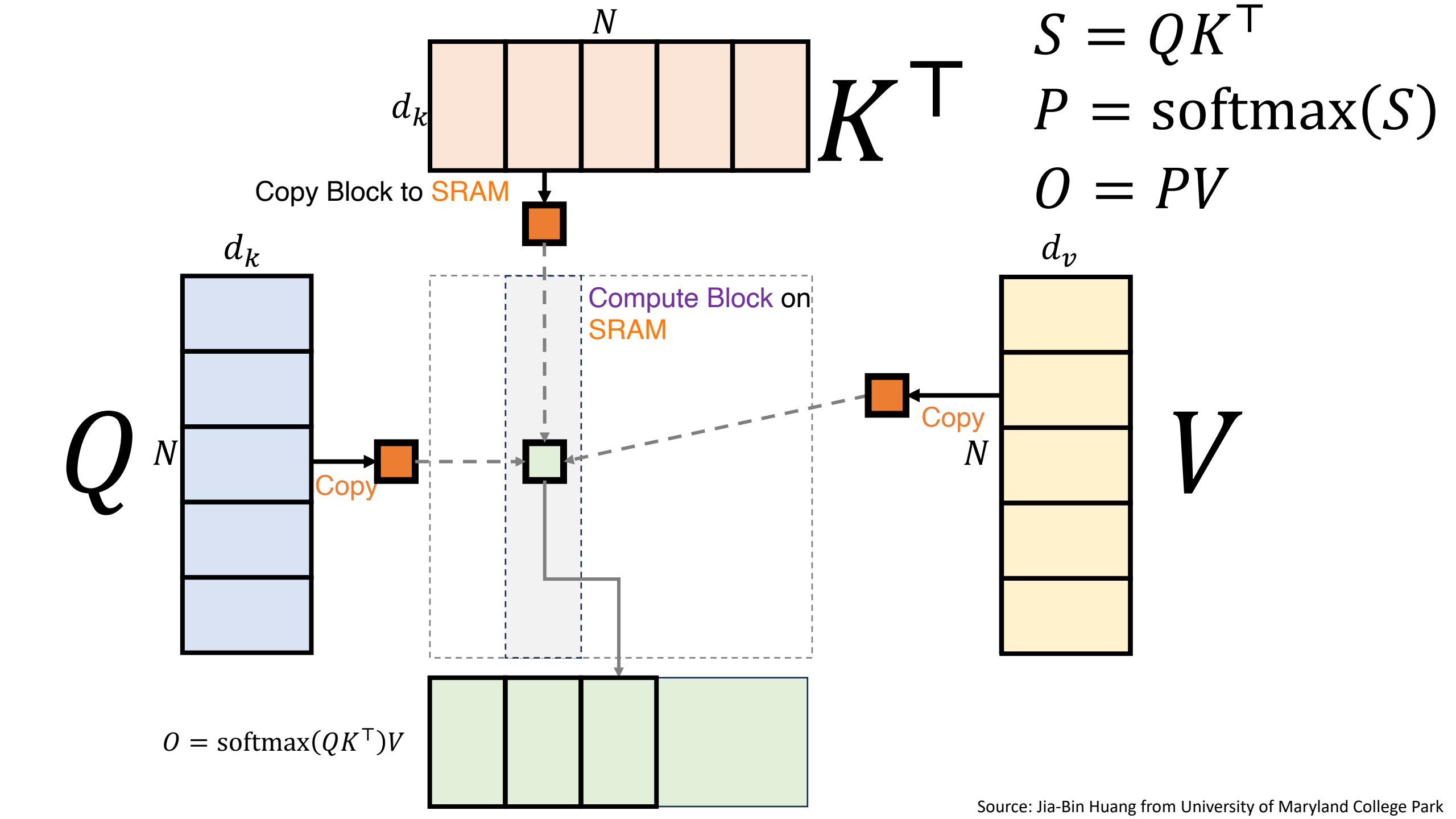
Source: Jia-Bin Huang from University of Maryland College Park



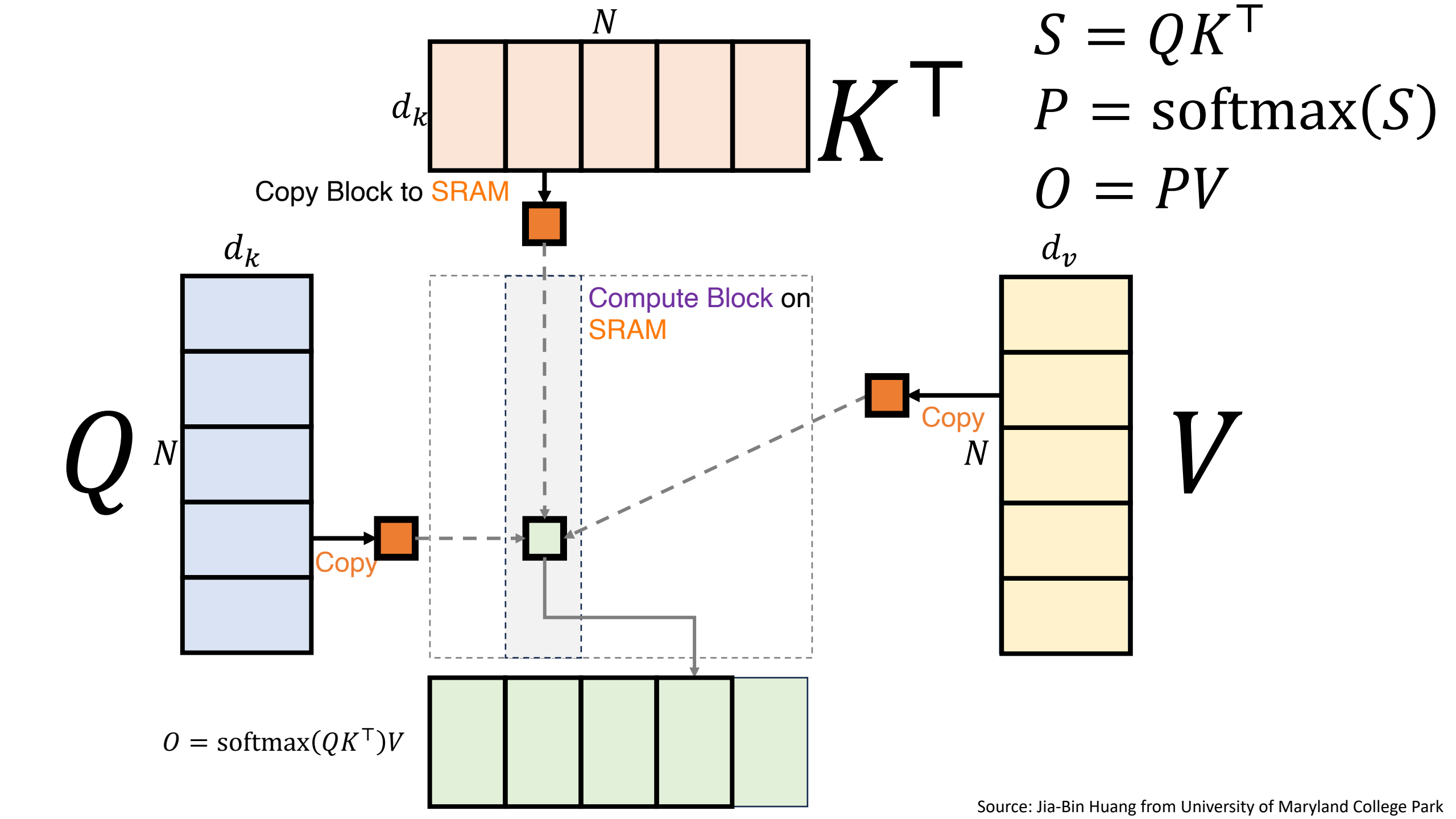
Source: Jia-Bin Huang from University of Maryland College Park



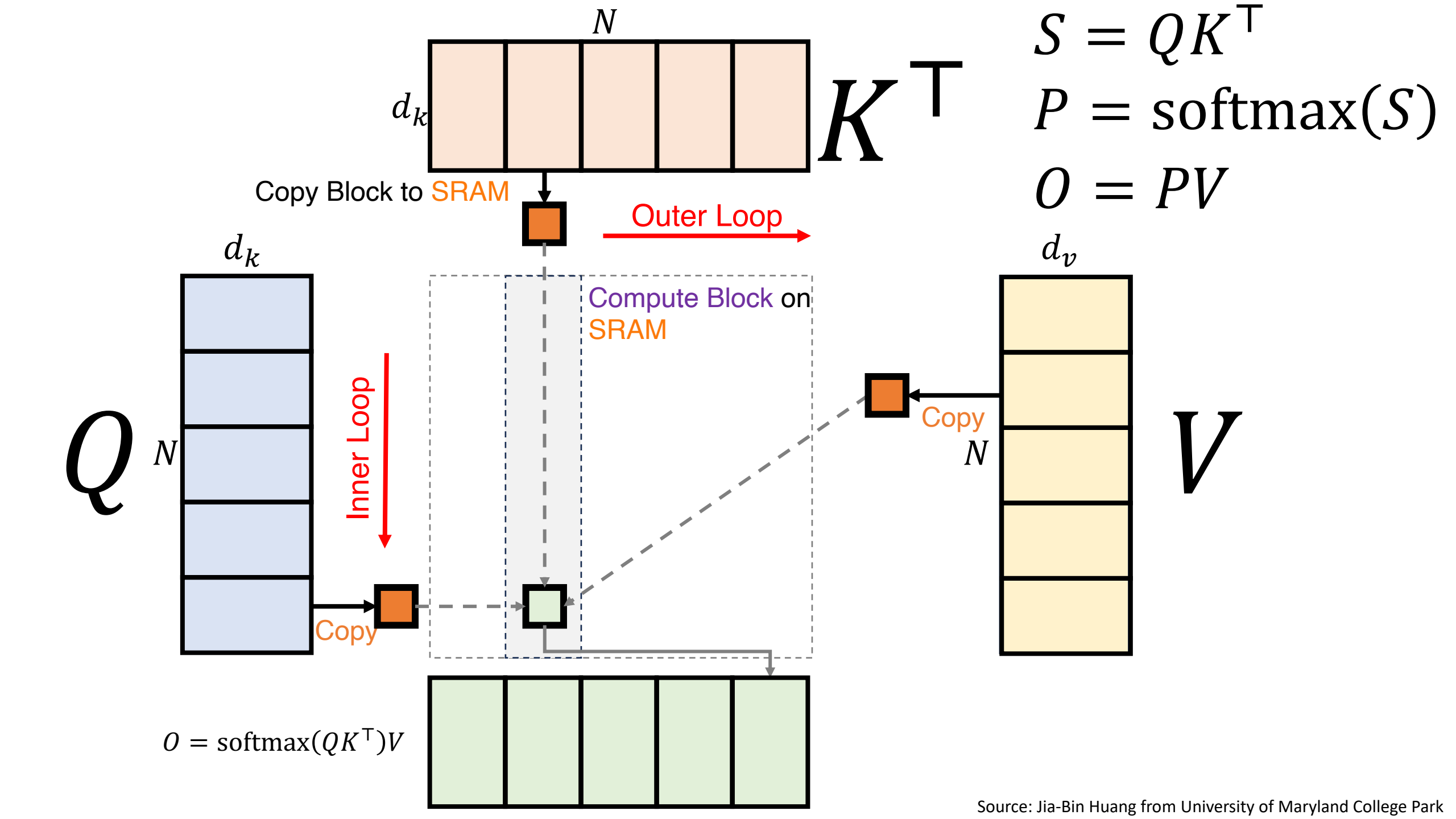
Source: Jia-Bin Huang from University of Maryland College Park



Source: Jia-Bin Huang from University of Maryland College Park



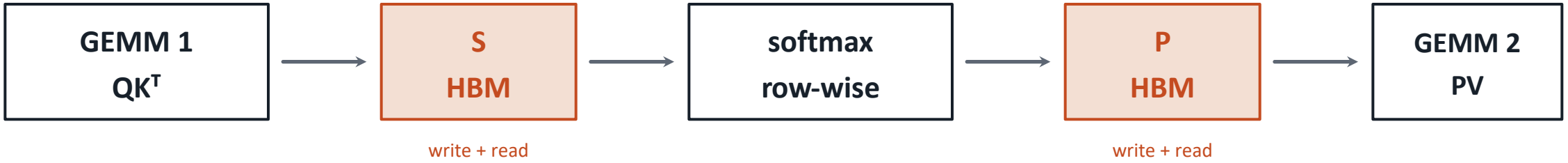
Source: Jia-Bin Huang from University of Maryland College Park



Source: Jia-Bin Huang from University of Maryland College Park

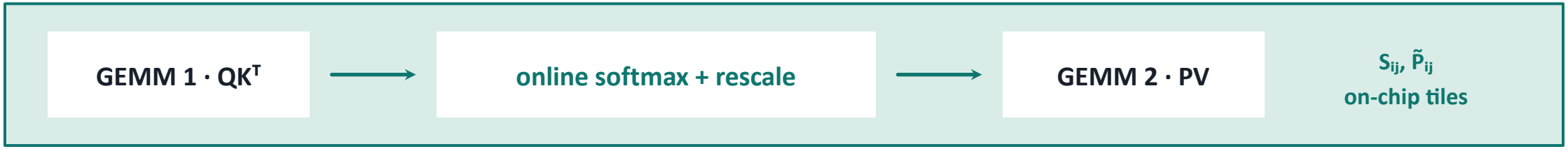
Kernel fusion removes HBM round trips between attention operators

SEPARATE OPERATORS · S AND P MATERIALIZE IN HBM



Each operator boundary can force a round trip: S is written then read; P is written then read.

FLASHATTENTION · ONE FUSED KERNEL FOR THE FORWARD PASS

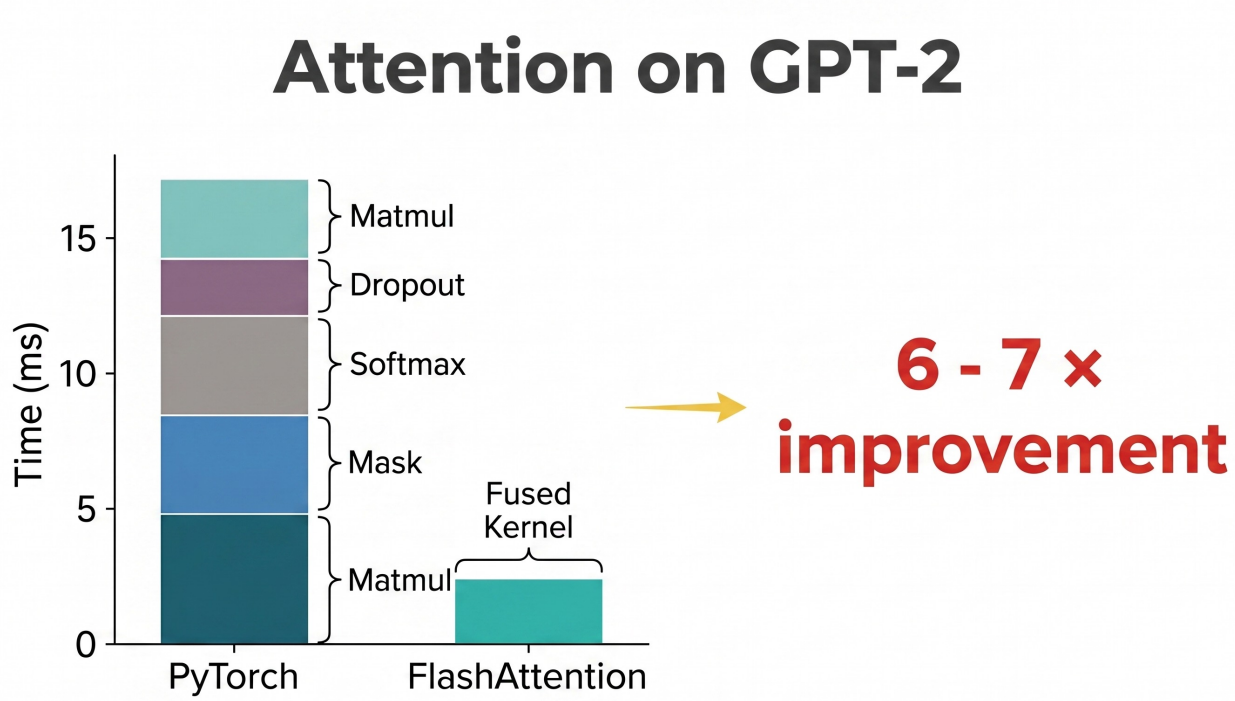


Fusion eliminates inter-operator S/P traffic; tiling and online softmax make the fused working set feasible.

Source: Dao et al., 2022,

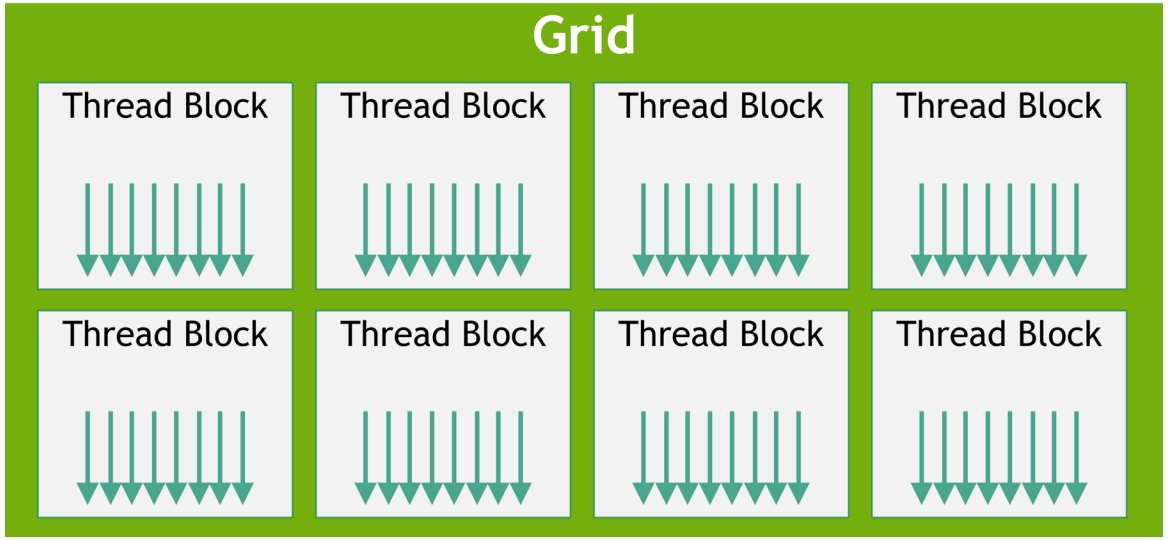
Flash Attention

- Kernel Fusion
  - FlashAttention kernel fusion combining multiple (block-wise) steps of the attention computation—such as scaling, masking, softmax normalization, and matrix multiplications—into custom fused CUDA kernels



Flash Attention

- Imperfection of Flash Attention
  - Underutilized streaming multi-processors for **small batch size** but **long sequences**

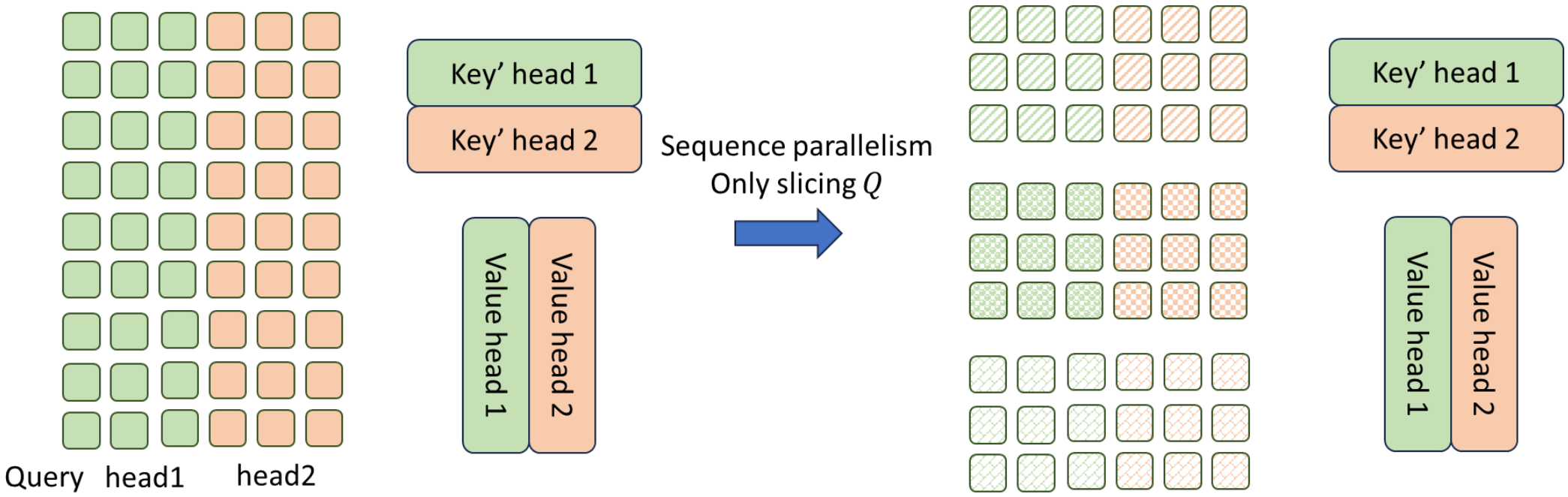


Grid of Thread Blocks
(#batch_size * #num_heads)

- GPU grid
  - Flash Attention enables Parallel computing on **batches and heads**
  - The grid consists of *batch_size* × *head_number* thread blocks
    - Batch_size = 2 and *head_number = 16*, meaning that only 32 out of 108 SMs are used on A100

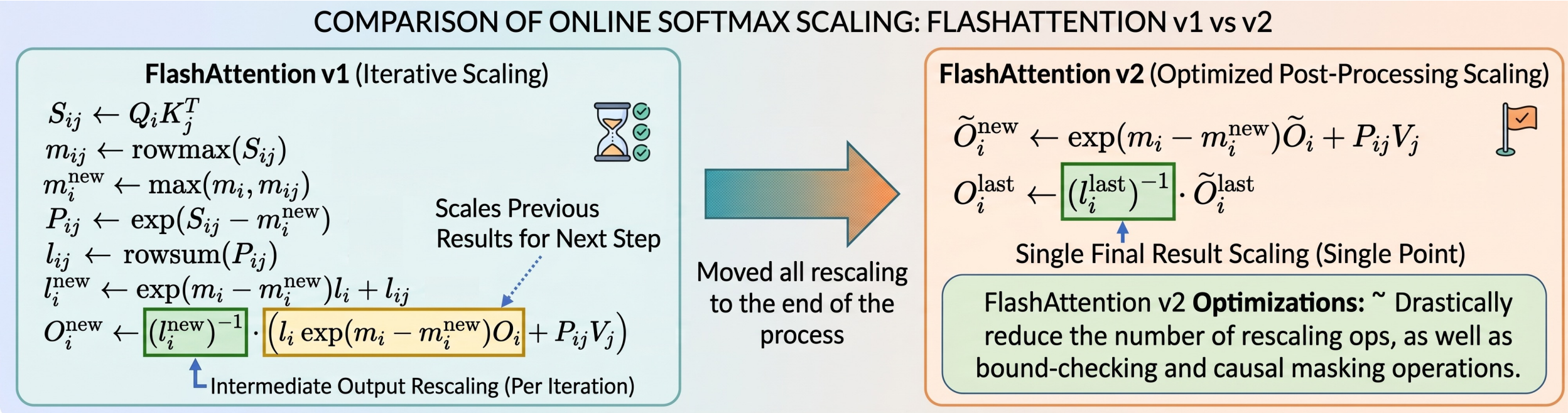
Flash Attention

- Flash Attention v2
  - Making FlashAttention sequence parallel
    - The shape of a grid: grid(num_m_block, batch_size, num_heads)
    - Introducing the idea of *sequence parallelism*



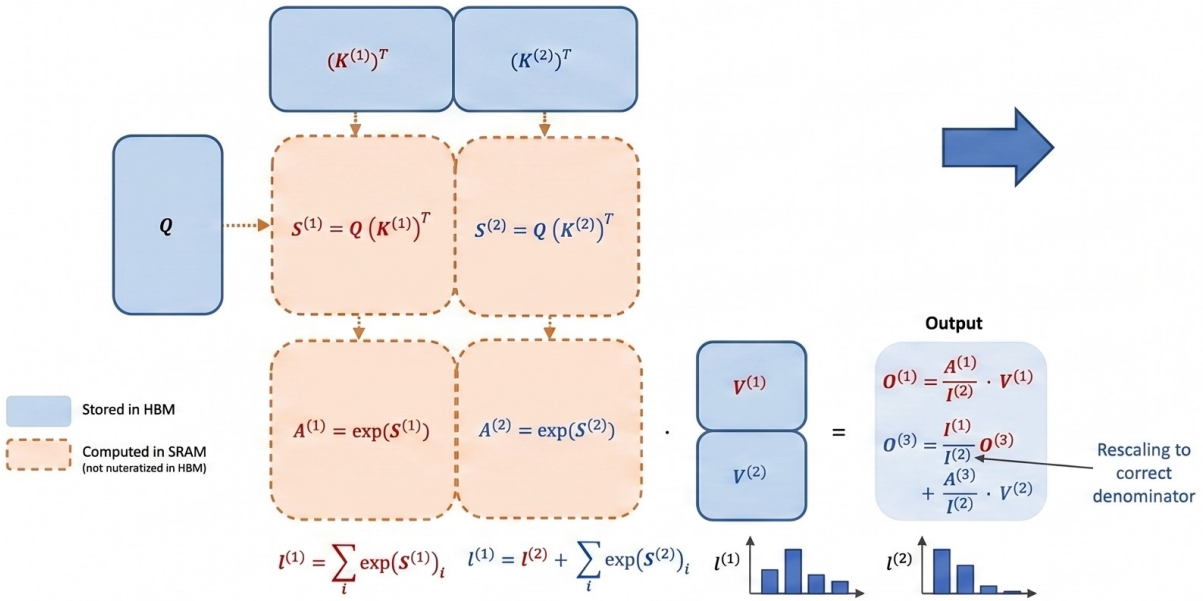
Flash Attention

- FlashAttention v2
  - Algorithm: fewer *non-**matmul*** FLOPs
    - GPU is highly optimized on *matmul* (e.g. Tensor Cores)
      - NVIDIA A100 GPU: 312 TFLOPs/s of FP16/BF16 *matmul*, only 19.5 TFLOPs/s of *non-matmul* FP32



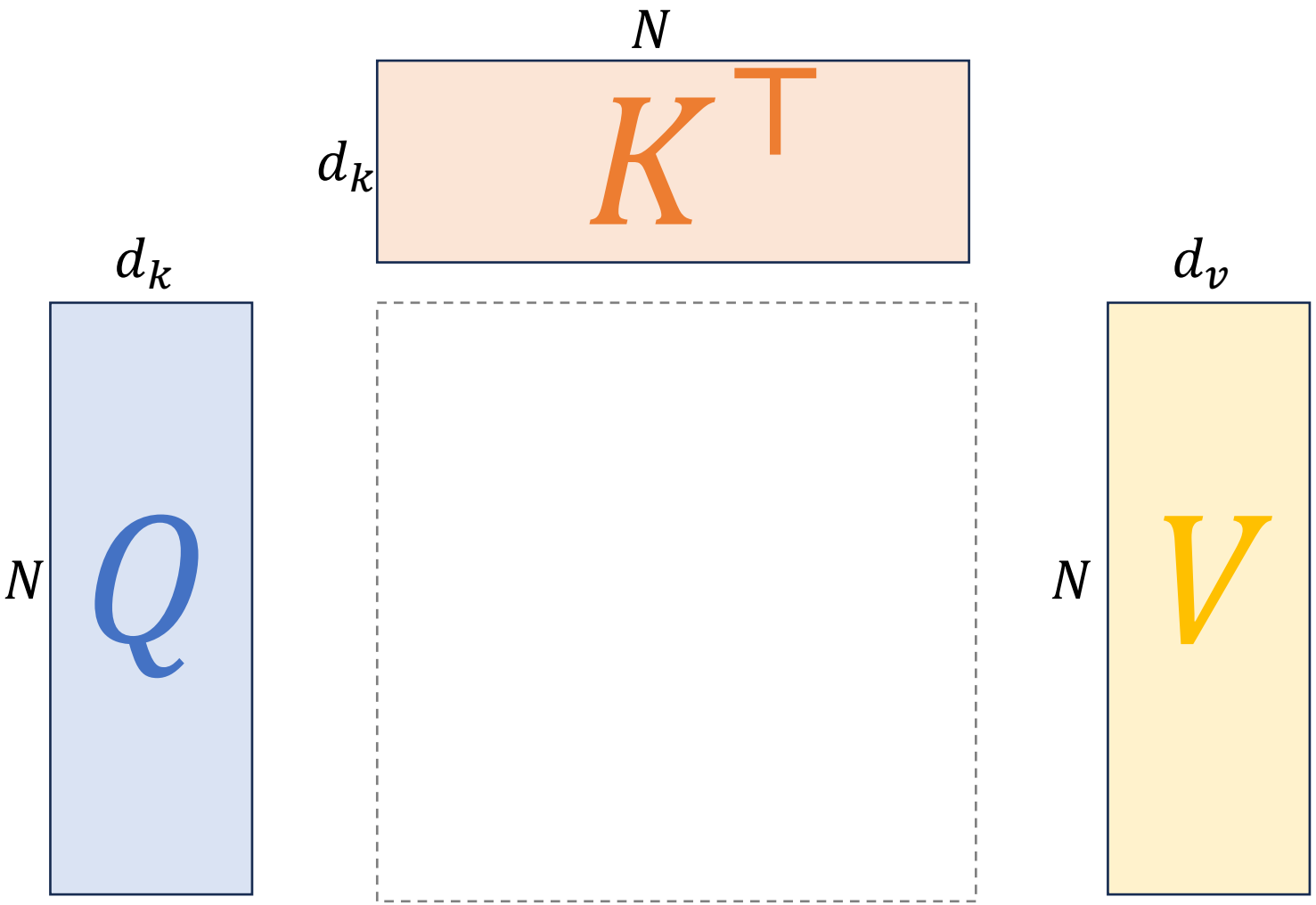
Flash Attention

- Flash Attention v2: Loop Reordering

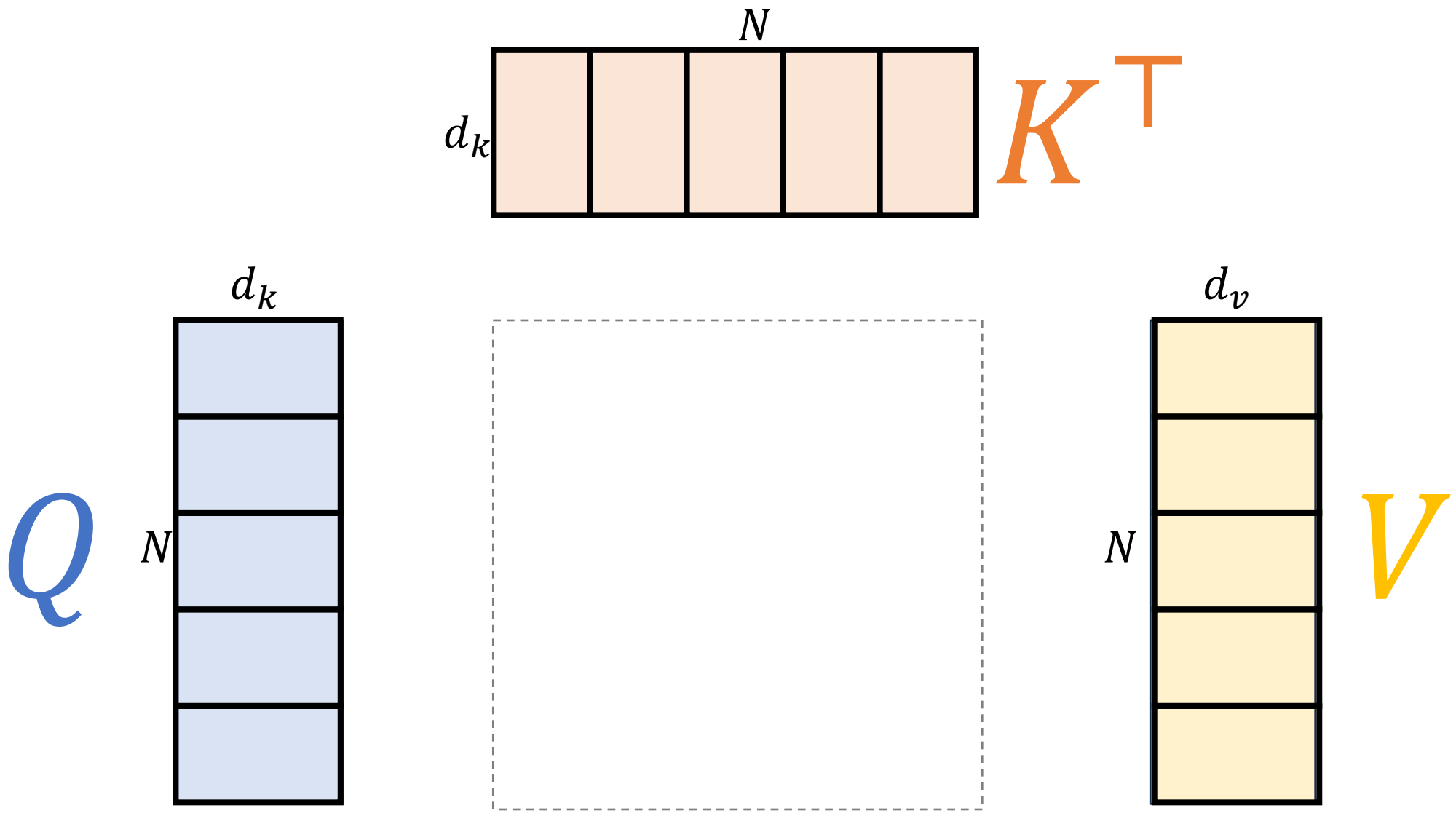


$$
\begin{aligned}
m^{(1)} &= \text{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \text{rowsum}(e^{\mathbf{S}^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r} \\
\tilde{\mathbf{O}}^{(1)} &= e^{\mathbf{S}^{(1)} - m^{(1)}} \mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \max(m^{(1)}, \text{rowmax}(\mathbf{S}^{(2)})) = m \\
\ell^{(2)} &= e^{m^{(1)} - m^{(2)}} \ell^{(1)} + \text{rowsum}(e^{\mathbf{S}^{(2)} - m^{(2)}}) = \text{rowsum}(e^{\mathbf{S}^{(1)} - m}) + \text{rowsum}(e^{\mathbf{S}^{(2)} - m}) = \ell \\
\tilde{\mathbf{P}}^{(2)} &= \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - m^{(2)}} \\
\tilde{\mathbf{O}}^{(2)} &= \text{diag}(e^{m^{(1)} - m^{(2)}})^{-1} \tilde{\mathbf{O}}^{(1)} + e^{\mathbf{S}^{(2)} - m^{(2)}} \mathbf{V}^{(2)} = e^{\mathbf{S}^{(1)} - m} \mathbf{V}^{(1)} + e^{\mathbf{S}^{(2)} - m} \mathbf{V}^{(2)} \\
\mathbf{O}^{(2)} &= \text{diag}(\ell^{(2)})^{-1} \tilde{\mathbf{O}}^{(2)} = \mathbf{O}.
\end{aligned}
$$

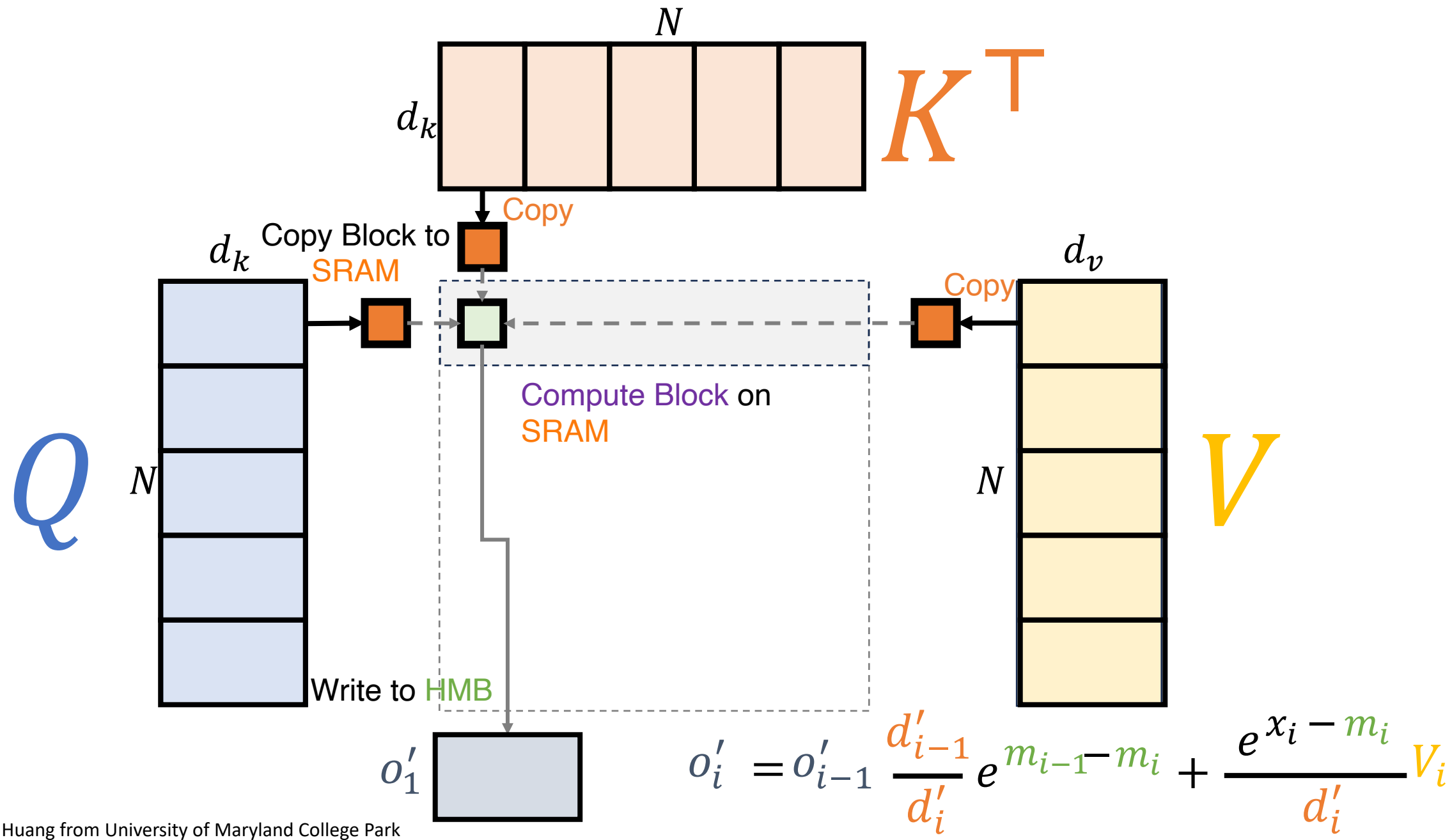
FlashAttention forward pass: key K is partitioned into two blocks and value V is also partitioned into two blocks



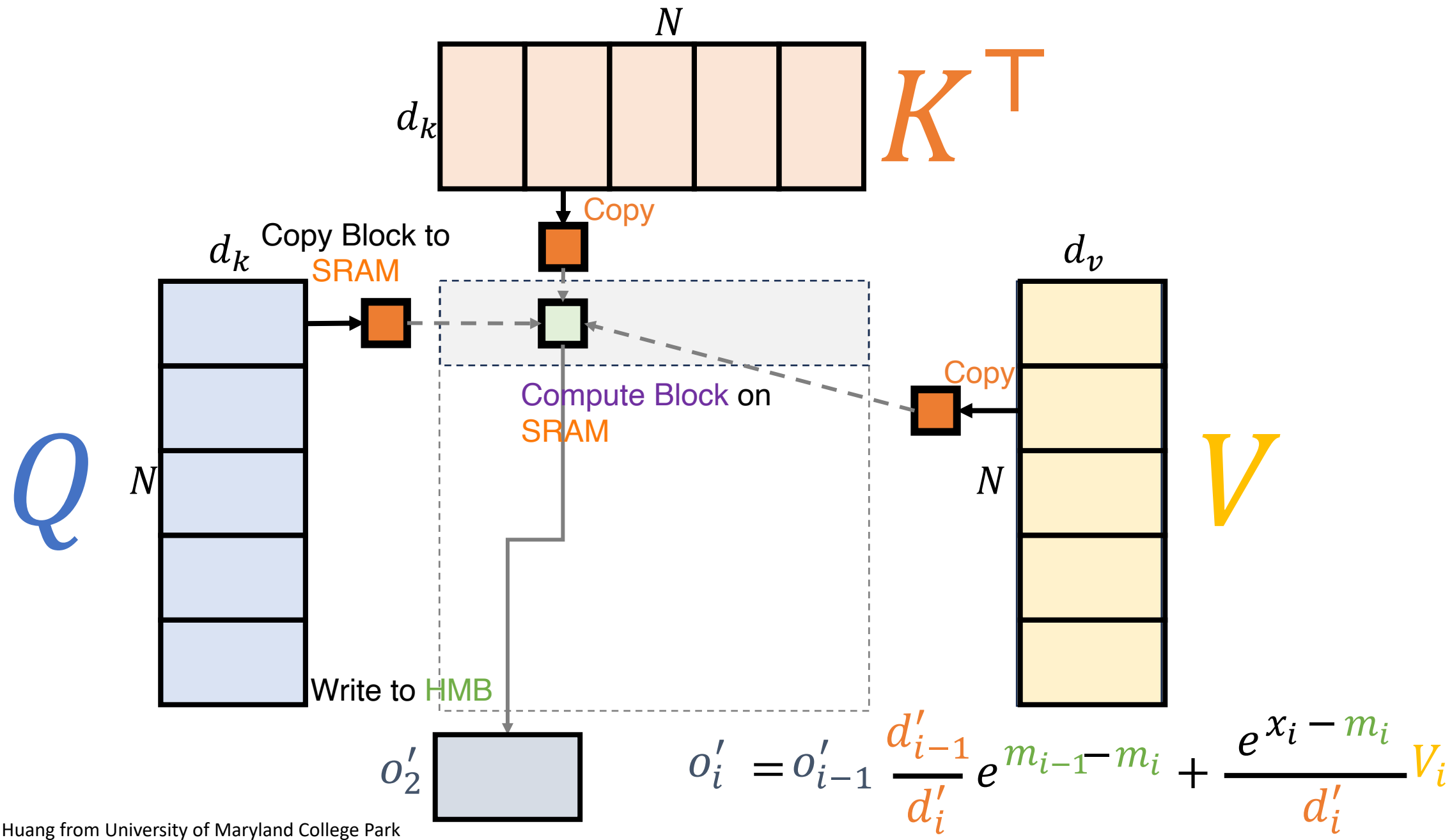
Source: Jia-Bin Huang from University of Maryland College Park



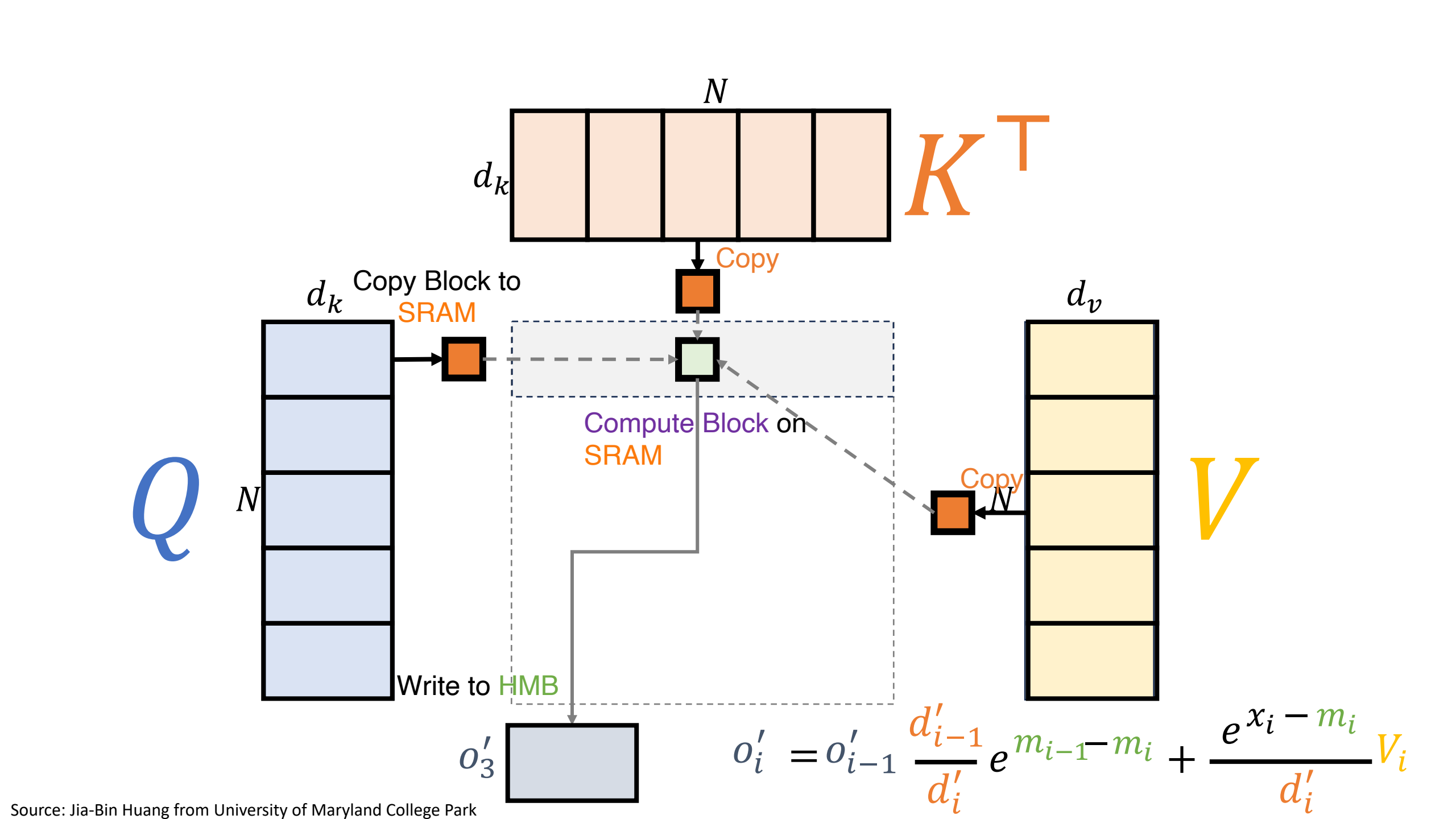
Source: Jia-Bin Huang from University of Maryland College Park



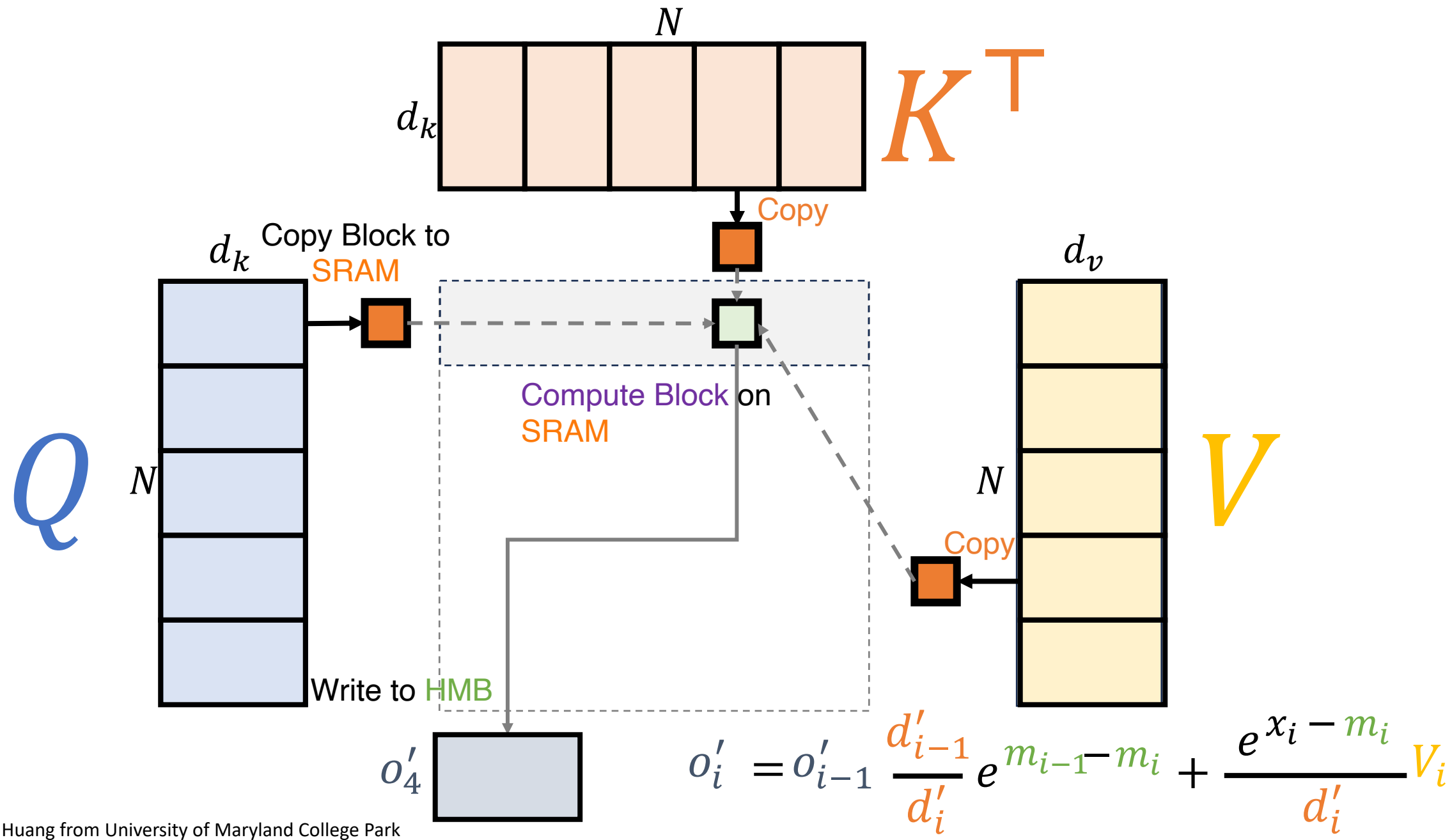
Source: Jia-Bin Huang from University of Maryland College Park



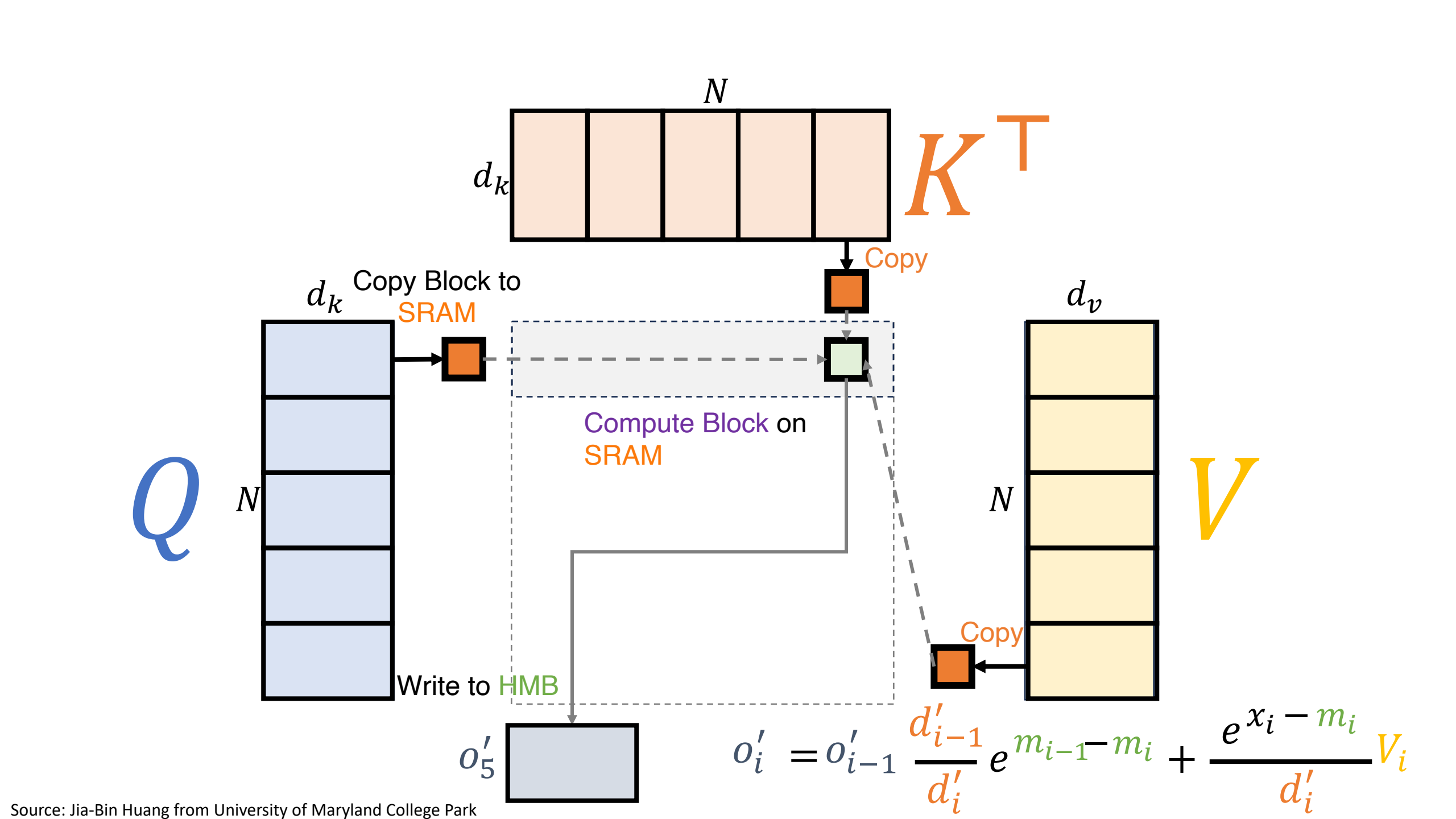
Source: Jia-Bin Huang from University of Maryland College Park



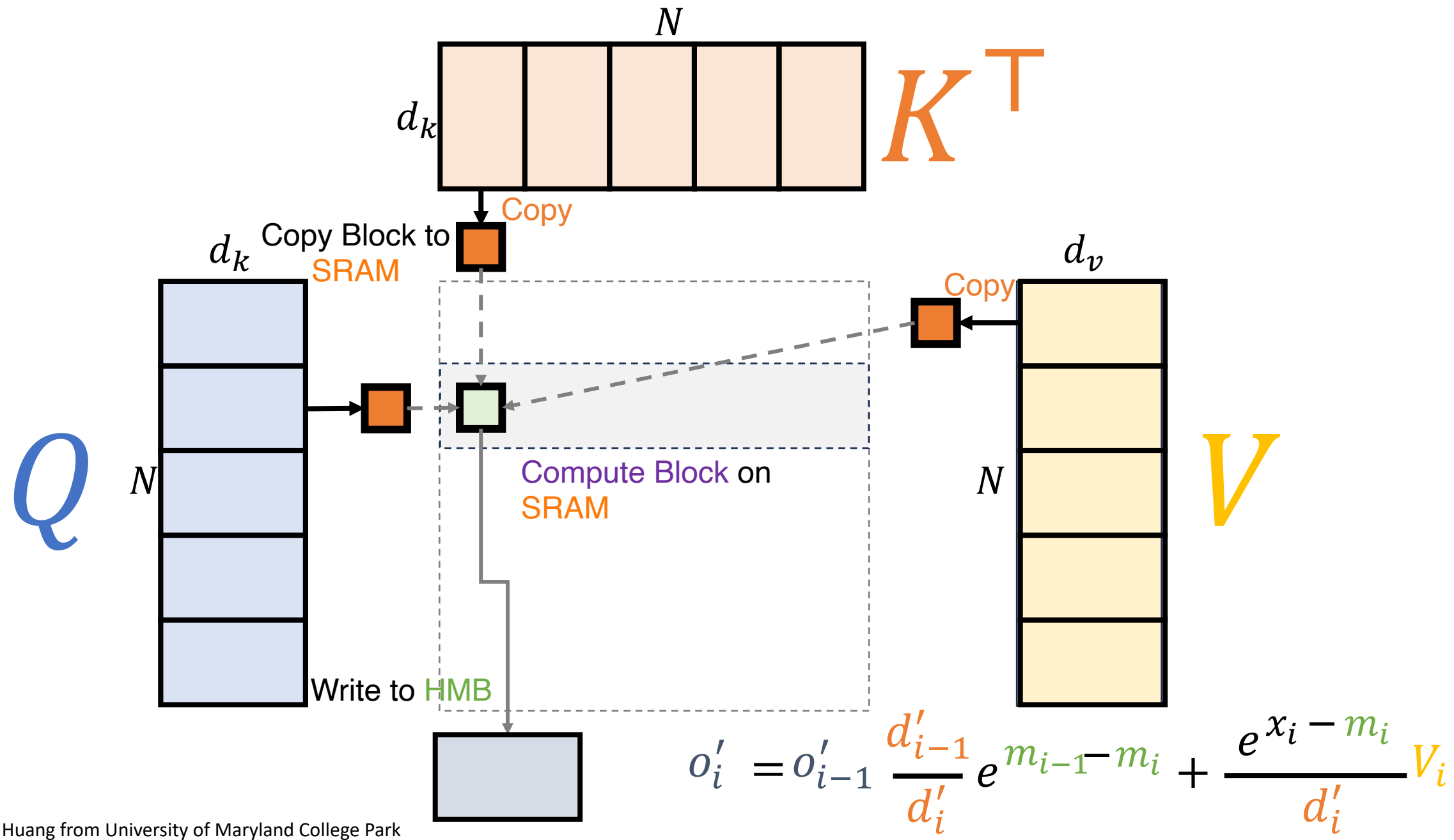
Source: Jia-Bin Huang from University of Maryland College Park



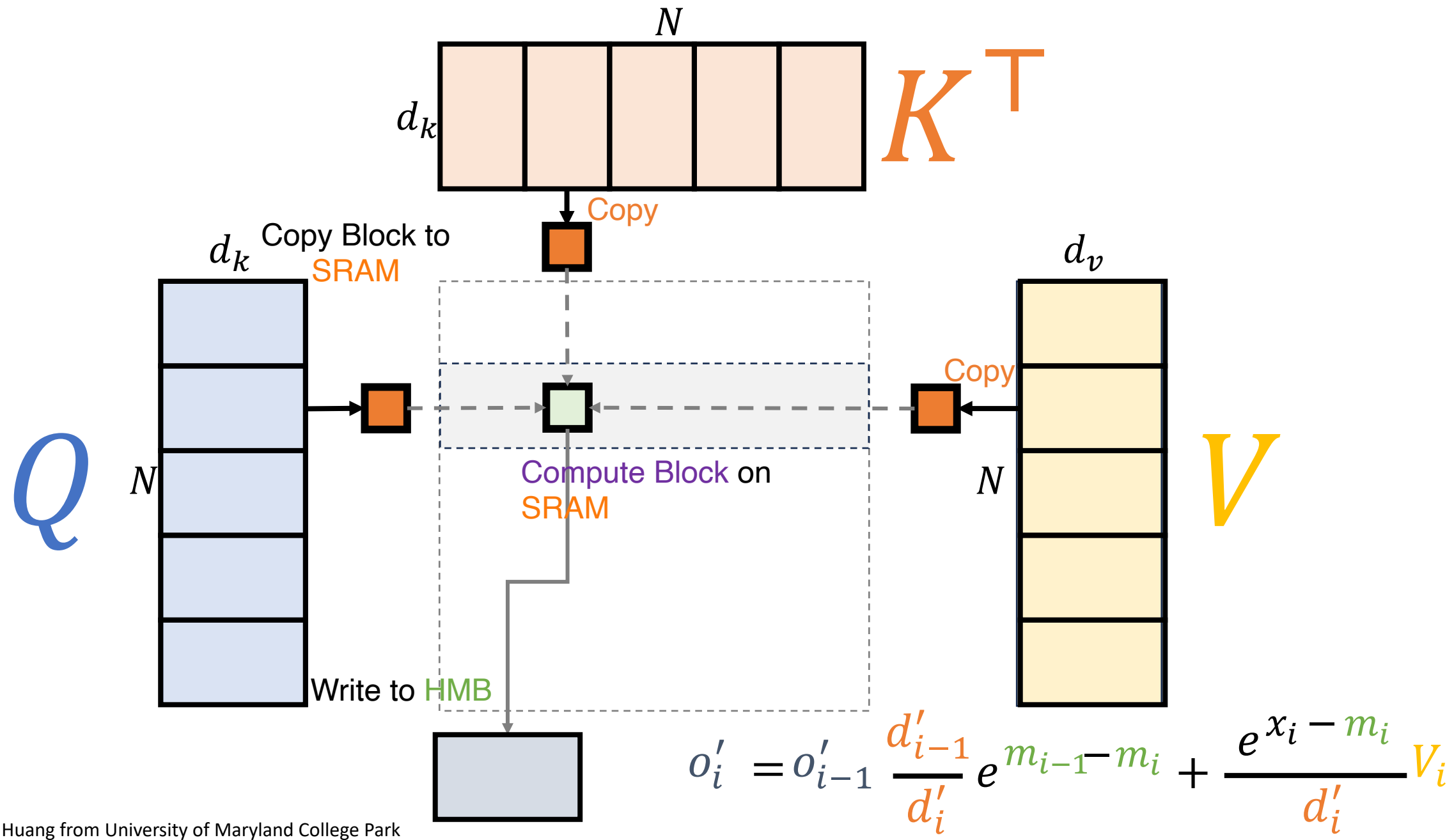
Source: Jia-Bin Huang from University of Maryland College Park



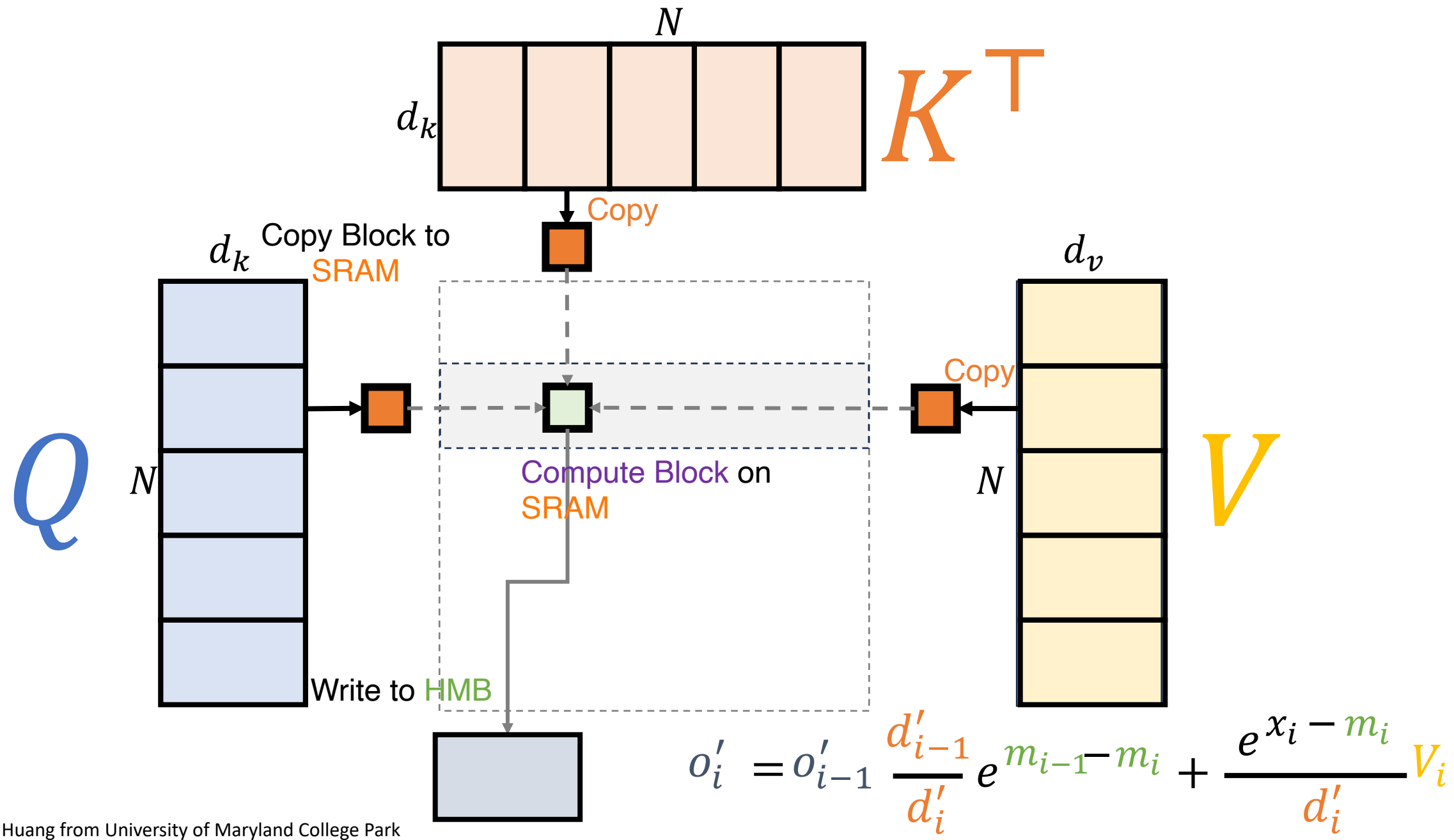
Source: Jia-Bin Huang from University of Maryland College Park



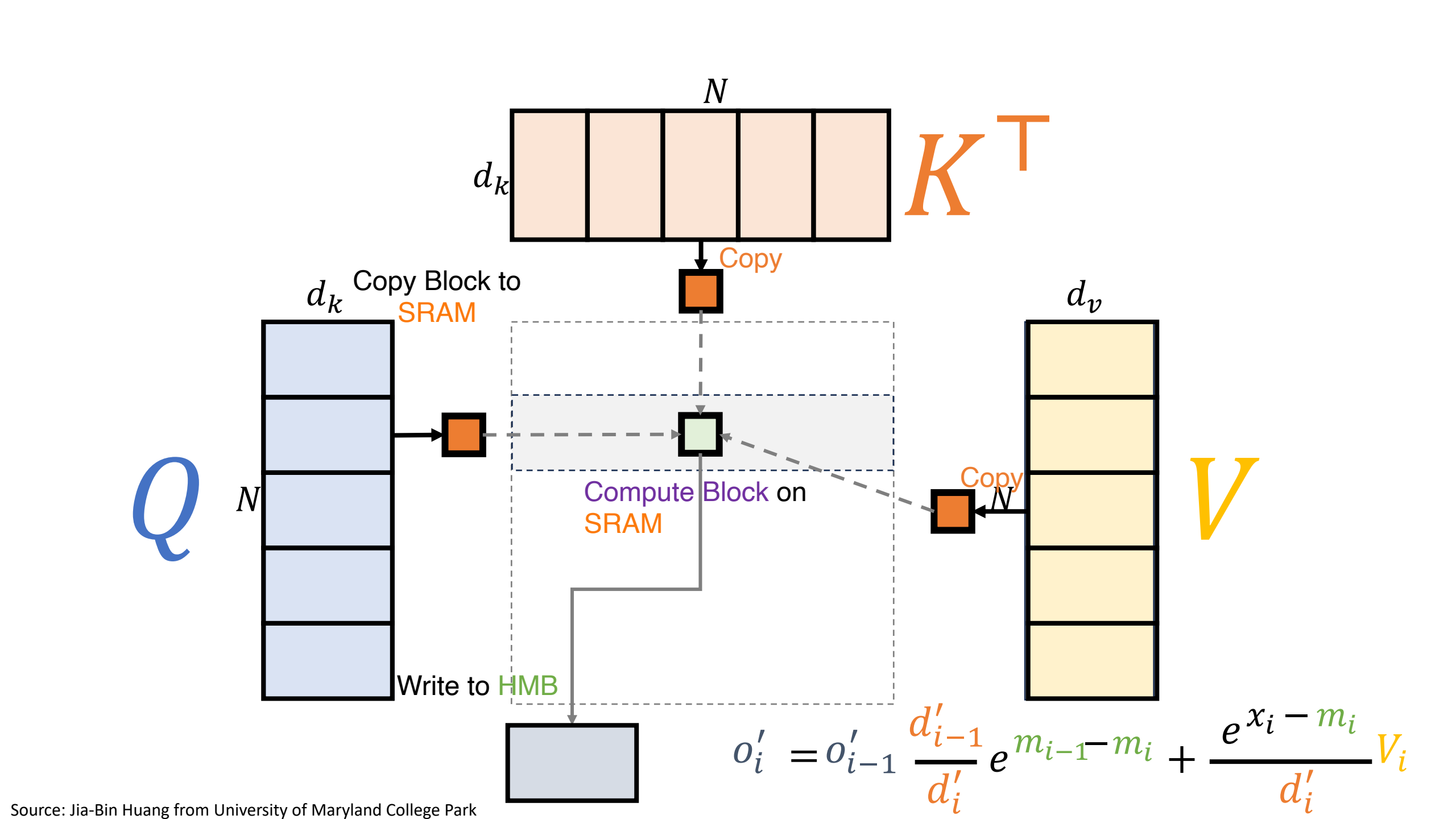
Source: Jia-Bin Huang from University of Maryland College Park



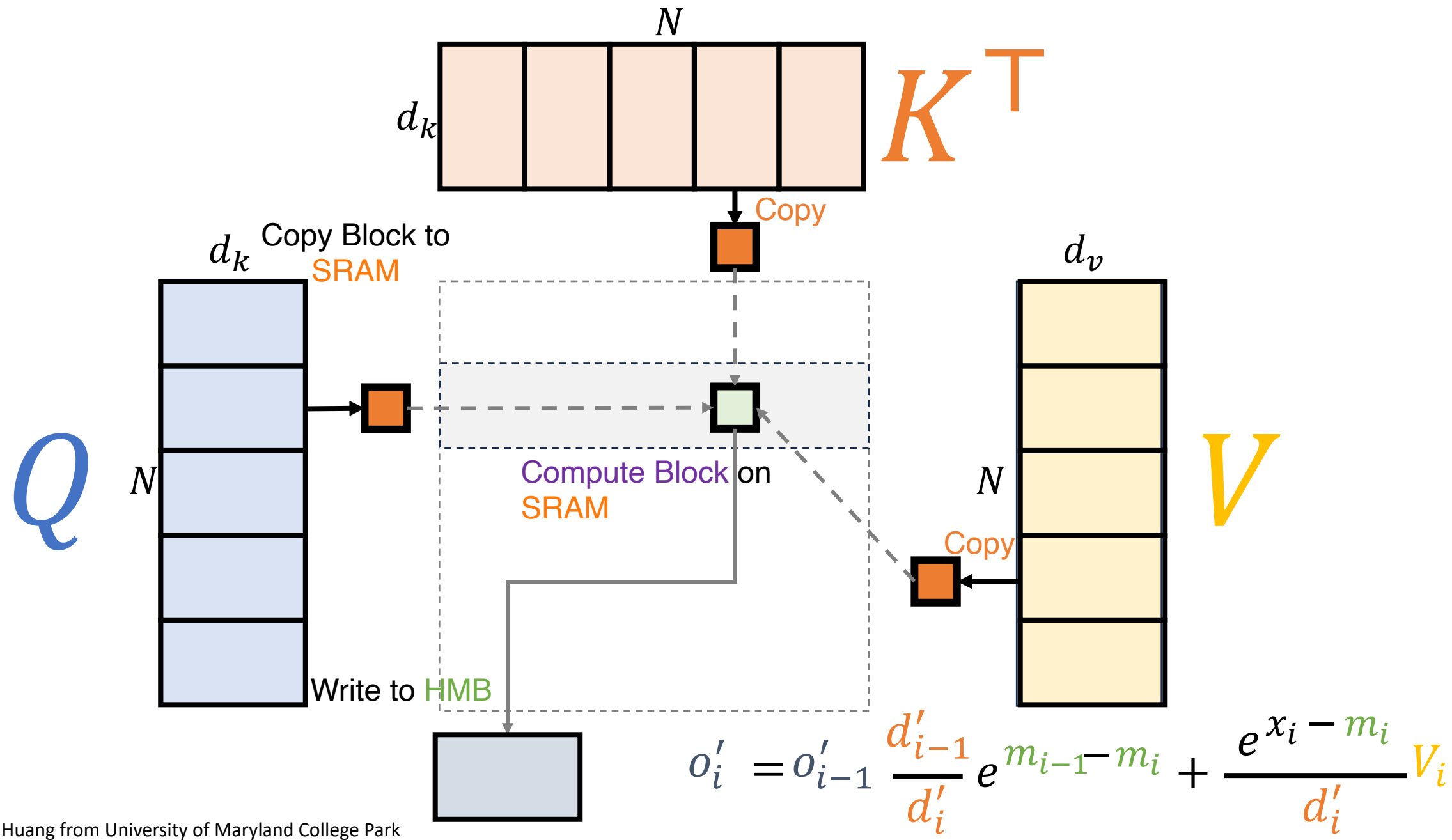
Source: Jia-Bin Huang from University of Maryland College Park



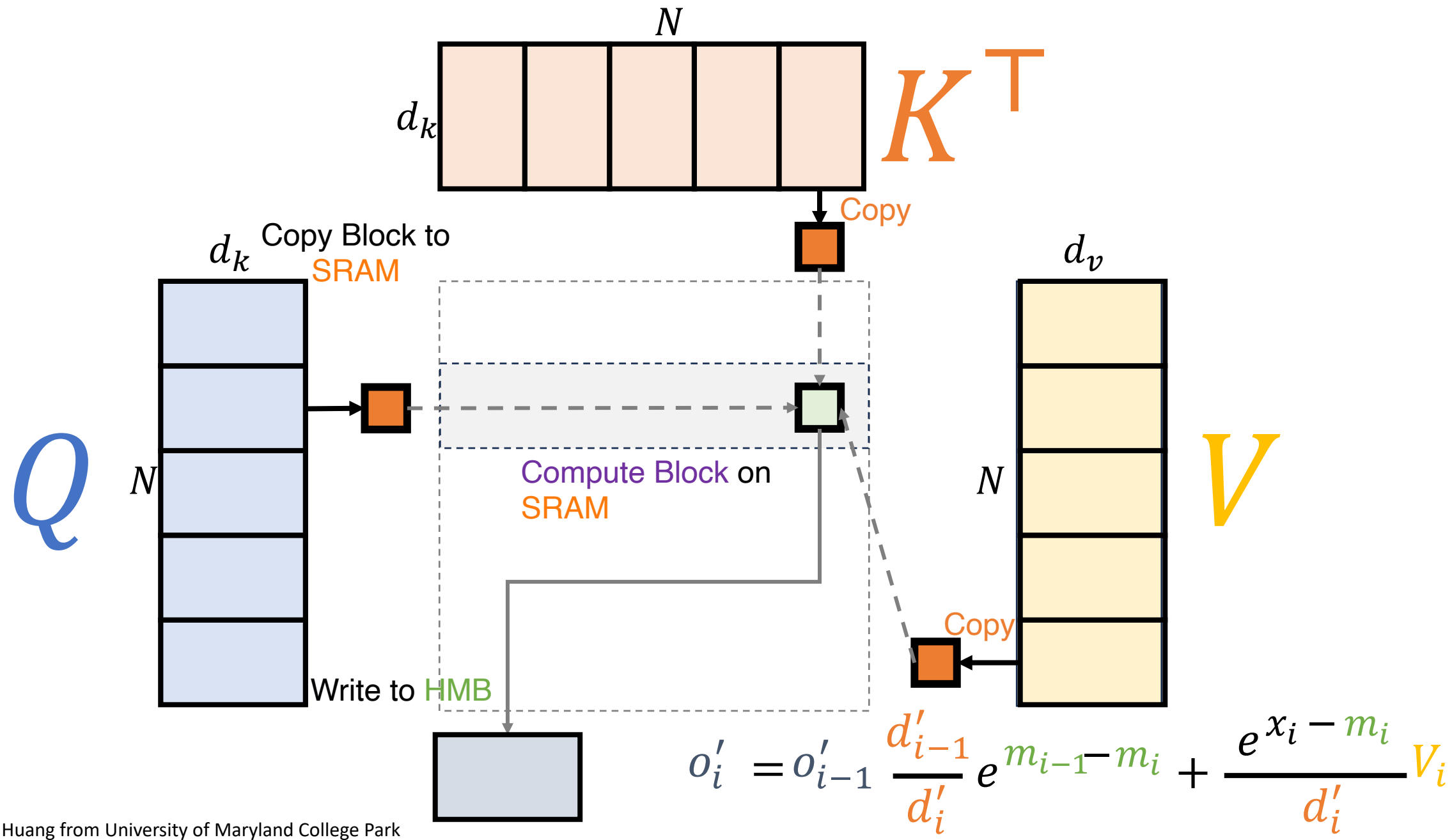
Source: Jia-Bin Huang from University of Maryland College Park



Source: Jia-Bin Huang from University of Maryland College Park



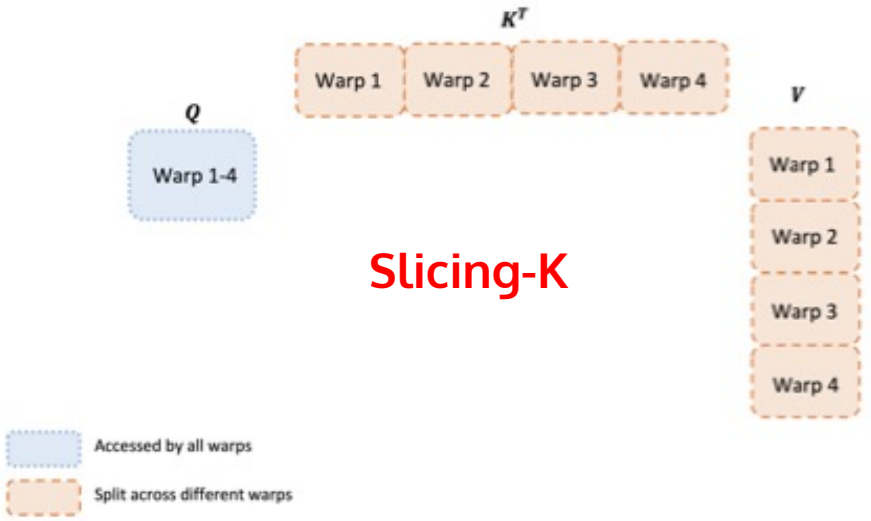
Source: Jia-Bin Huang from University of Maryland College Park



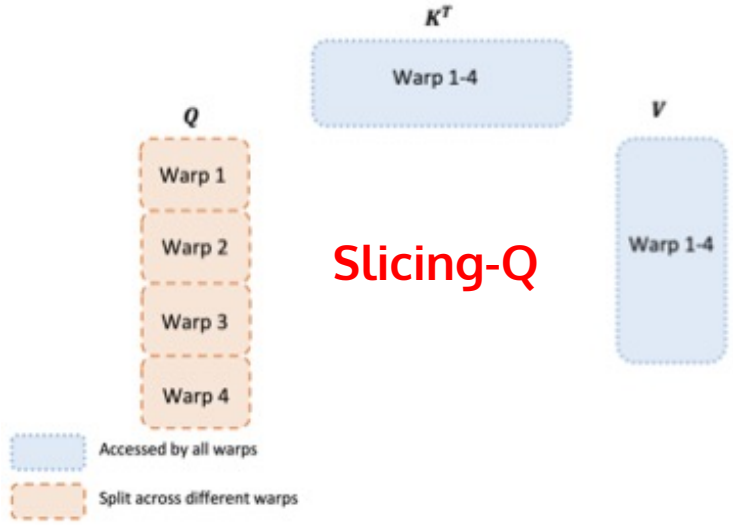
Source: Jia-Bin Huang from University of Maryland College Park

Flash Attention

- FlashAttention v2
  - Better Work Partitioning
    - A thread block is partitioned into multiple warps (32 threads per warp, 4~8 warps per block)



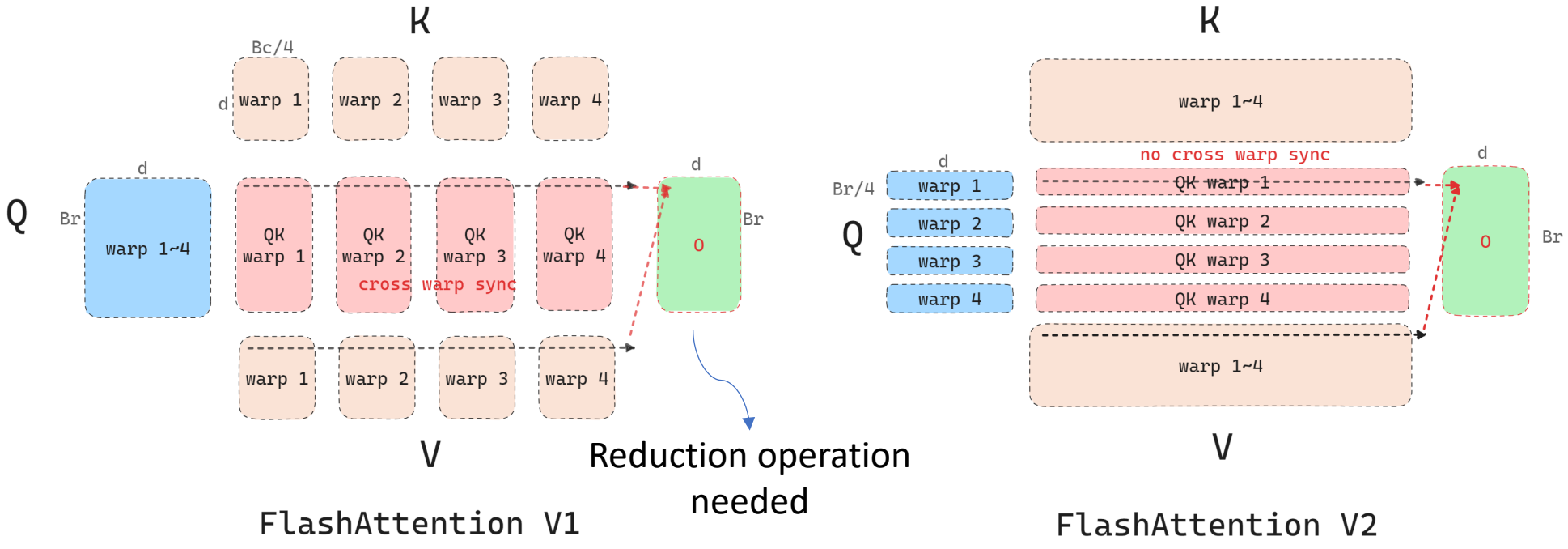
- Slicing K/V at each Warp while maining complete Q
- QK^T has four partitions, and $(QK^T)V$ needs REDUCTION (write to shared memory, synchronize, and add-up intermediate results)



- Slicing Q while keeping K/V accessible by all Warps
- Each warp performs matrix multiply to obtain a slice of QK^T, and no intra-block communication

Flash Attention

- FlashAttention v2
  - Better Work Partitioning
    - A thread block is partitioned into multiple warps (32 threads per warp, 4~8 warps per block)



By Zhihu blogger: Antinomi

Inference Optimization

- Overview
- Attention Optimization
  - Sparse Attention
  - Flash Attention
  - **Linear Attention**
- KV Cache Optimization
- Parallel and Distributed Serving

Linear Attention

- Basic idea

$$\text{Softmax}\left(\frac{1}{\sqrt{d_k}}\, Q_{N \times d_k}\, K^{\top}_{d_k \times N}\right) V_{N \times d_v}$$



Significant computation reduction,
yet prevented by *softmax*

$$Q_{N \times d_k}\left(K^{\top}_{d_k \times N}\, V_{N \times d_v}\right)$$

Linear Attention

- Iterative computation to further reduce complexity

$$o_1 \approx \sum_{i=1}^{N} \frac{\phi(q_1)\cdot\phi(k_i)}{\sum_{j=1}^{N}\phi(q_1)\cdot\phi(k_j)} v_i$$

  - where

$$\sum_{i=1}^{N}[\phi(q_1)\cdot\phi(k_i)]v_i = \phi(q_1)^{\top}\left(\sum_{i=1}^{N} k_i v_i^{\top}\right)$$

$$S_t = \sum_{i=1}^{t} k_i v_i^{\top} \in R^{d\times d}$$

$$S_t = S_{t-1} + k_t v_t^{\top}$$

$$o_t = \phi(q_t)^{\top} S_t$$

Source: Jia-Bin Huang from University of Maryland College Park

Linear Attention

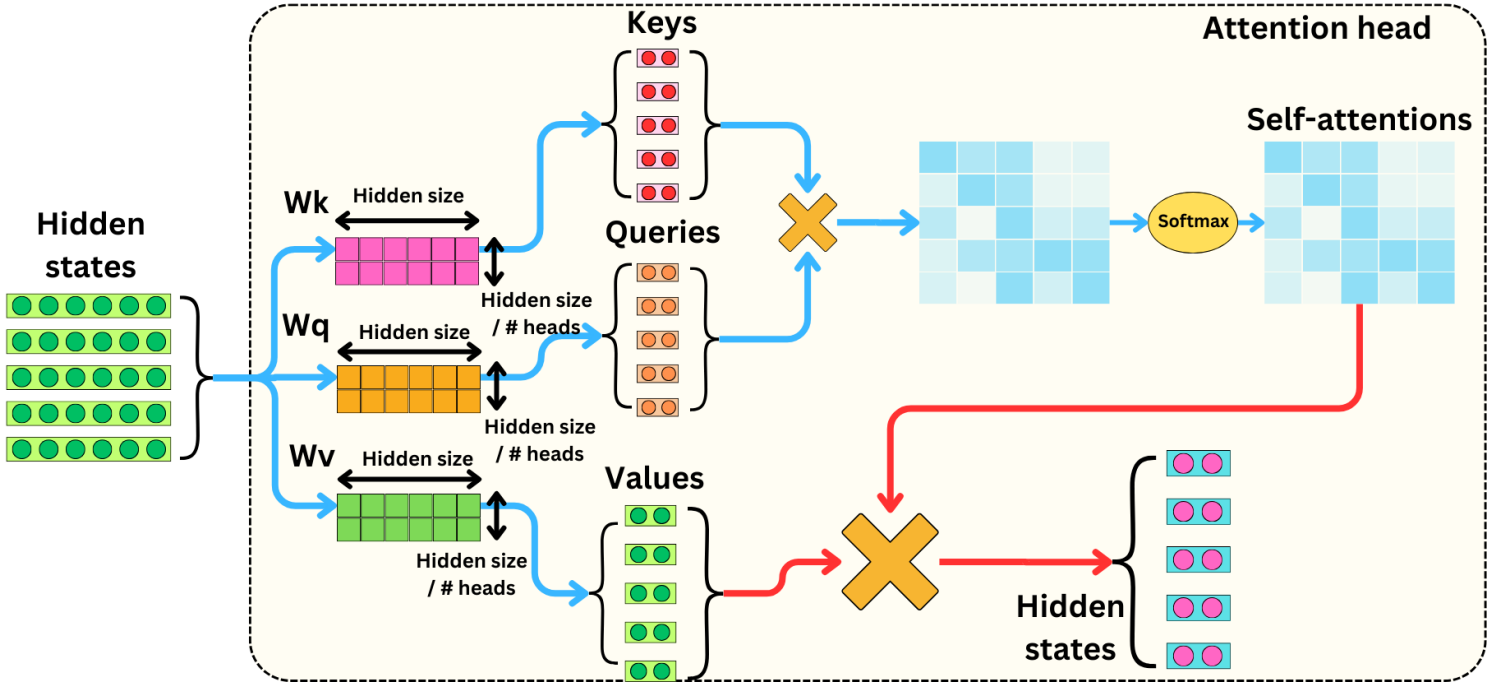
- Today's Linear Attention
  - Baby linear attention: Linear Transformer, SANA, CHELA, LightningAttention, etc.
    - **Efficient** in computation, I/O, storage
    - Performance loss compared with ***Softmax***
  - More advanced linear attention:
    - Delta rule
    - Gamma forget
    - Gated attention
  - Qwen3-Next and Kimi-linear, and many others

Inference Optimization

- Overview
- Attention Computation Optimization
- KV Cache Optimization
  - **Model Architecture Optimization**
  - KV Cache Data Structure Optimization
  - KV Cache Memory Optimization
- Parallel and Distributed Serving

Attention Architecture Optimization

- Multi-Head Attention
  - Each head has its own Query, Key and Value matrices



Source: The Multi-head Attention Mechanism Explained!

Attention Architecture Optimization

- Multi-Query Attention (MQA)
  - All heads in a layer share a single set of W_K and W_V
  - Size of Q unchanged, and sizes of K and V shrink by h times

Component	Weight matrix shape	Number of parameters (exact)
Query projection	$W_Q \in \mathbb{R}^{d \times (h \cdot d_h)}$	$d \cdot (h \cdot d_h) = d^2$
Key projection	$W_K \in \mathbb{R}^{d \times d_h}$	$d \cdot d_h = d \cdot (d/h) = d^2/h$
Value projection	$W_V \in \mathbb{R}^{d \times d_h}$	$d \cdot d_h = d^2/h$
Output projection	$W_O \in \mathbb{R}^{(h \cdot d_h) \times d}$	$(h \cdot d_h) \cdot d = d^2$
Total	—	$2d^2 + 2 \cdot \frac{d^2}{h}$

Shared

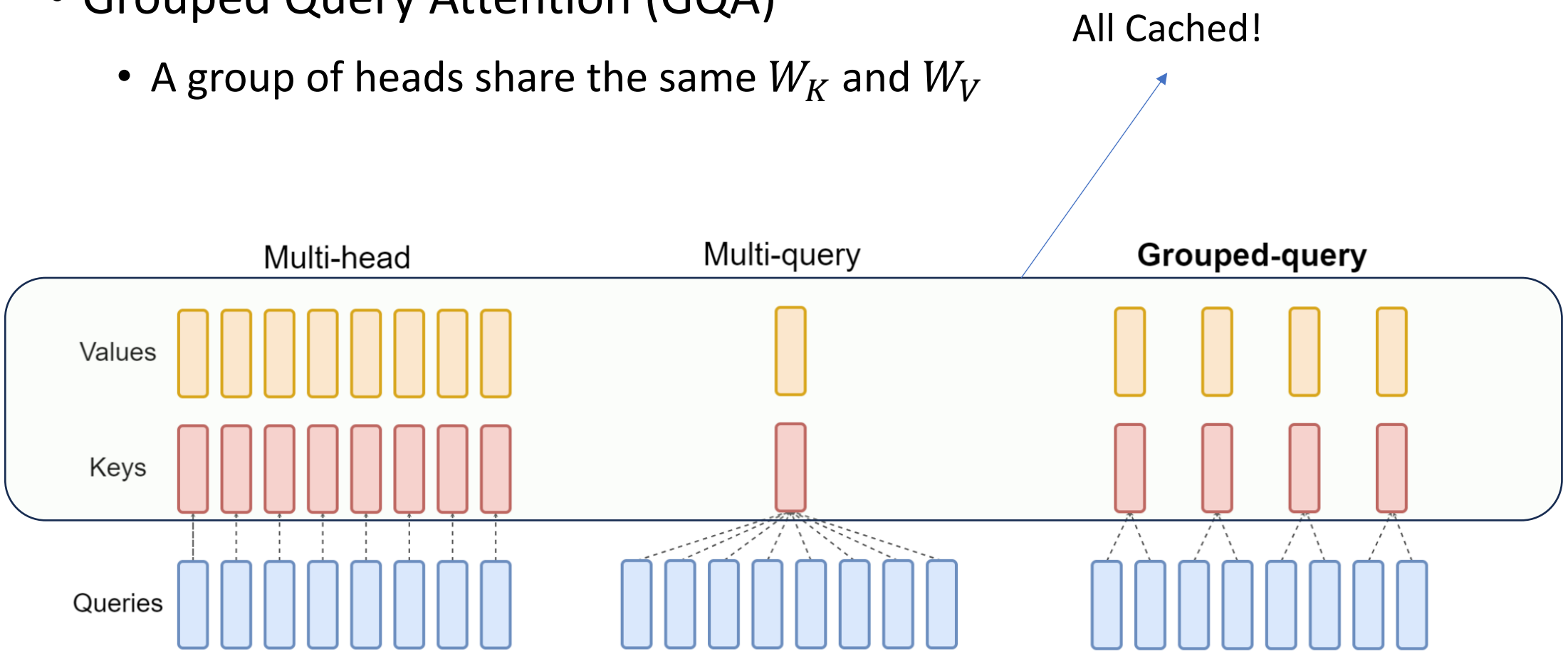
Number of parameters in MQA

$$\frac{MQA}{MHA} = \frac{1 + 1/h}{2}$$

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

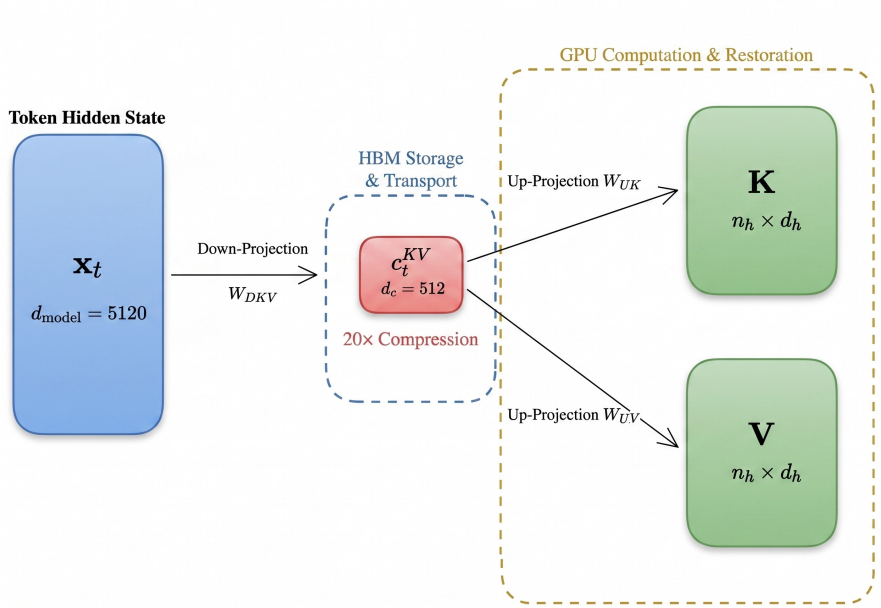
Attention Architecture Optimization

- Grouped Query Attention (GQA)
  - A group of heads share the same W_K and W_V



Attention Architecture Optimization

- Multi-Head Latent Attention (MLA)
  - Compressing K and V into a small Latent Vector $c_t^{KV} \in \mathbb{R}^{d_c}$
  - Decompressing it into full Key and Value during inference
  - Only c_t^{KV} is **cached** where $d_c \ll d_h \cdot n_h$



$$W^K = W^{DKV} \cdot W^{UK}$$

$$W^V = W^{DKV} \cdot W^{UV}$$

Trained parameters, not handcrafted

Source:DeepSeek-v3

Attention Architecture Optimization

- KV Cache Per-Token

MHA	GQA	MQA	MLA
$2d$	$2n_{kv}d_h$	$2d_h$	d_c+d_h^{rope}
32768 $(n_h = 128, d_h = 128)$	256~32768	256	576 $(d_c = 512)$

- Other important approaches
  - CSA/HCA, KDA (Kimi Delta Attention)

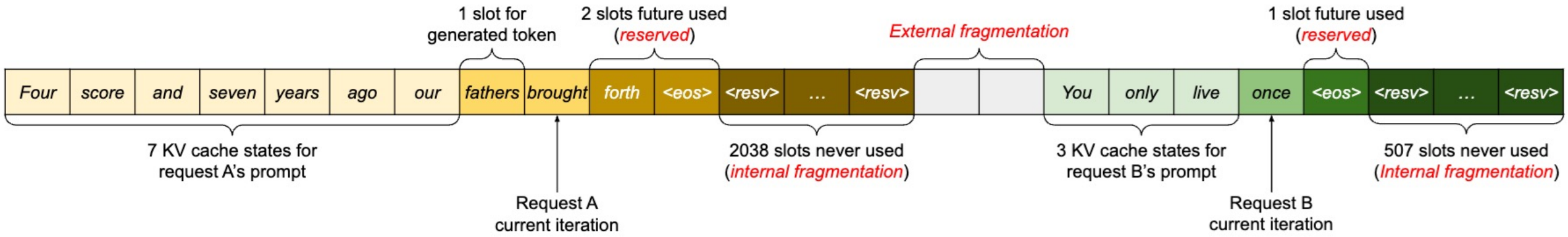
Source:DeepSeek-v3

Inference Optimization

- Overview
- Attention Computation Optimization
- KV Cache Optimization
  - Model Architecture Optimization
  - **KV Cache Data Structure Optimization**
  - KV Cache Memory Optimization
- Parallel and Distributed Serving

Memory Management

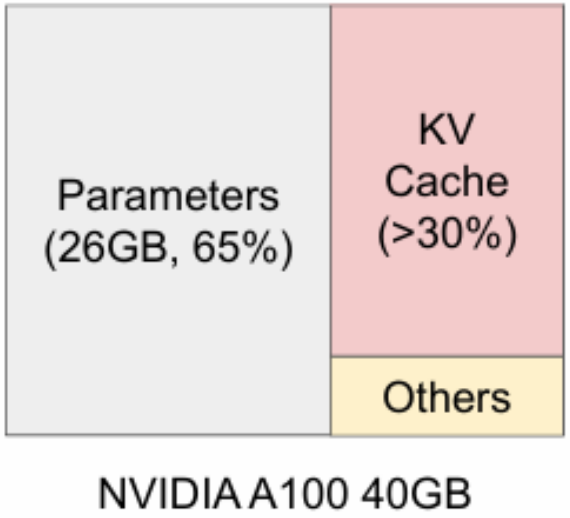
- Paged Attention
  - Uncertainty in ***Decode***: no one know how many tokens will be generated, and how much space needs to be allocated beforehand

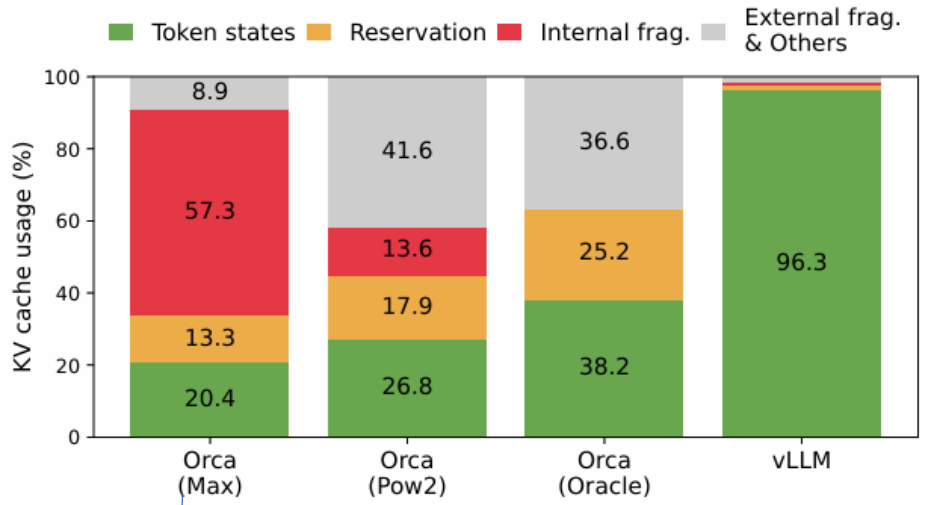


  - Memory fragmentations
    - Reserved fragmentation : reserved memory space not filled with output tokens temporarily
    - Internal fragmentation: reserved but never used even after the token generation completes
    - External fragmentation: memory space insufficient for inserting new KV cache

Memory Management

- Paged Attention
  - An example of memory fragmentation





Grey: parameters persisting in GPU memory throughout serving
Red: KV cache (de)allocated per serving request
Yellow: used ephemerally for activation.

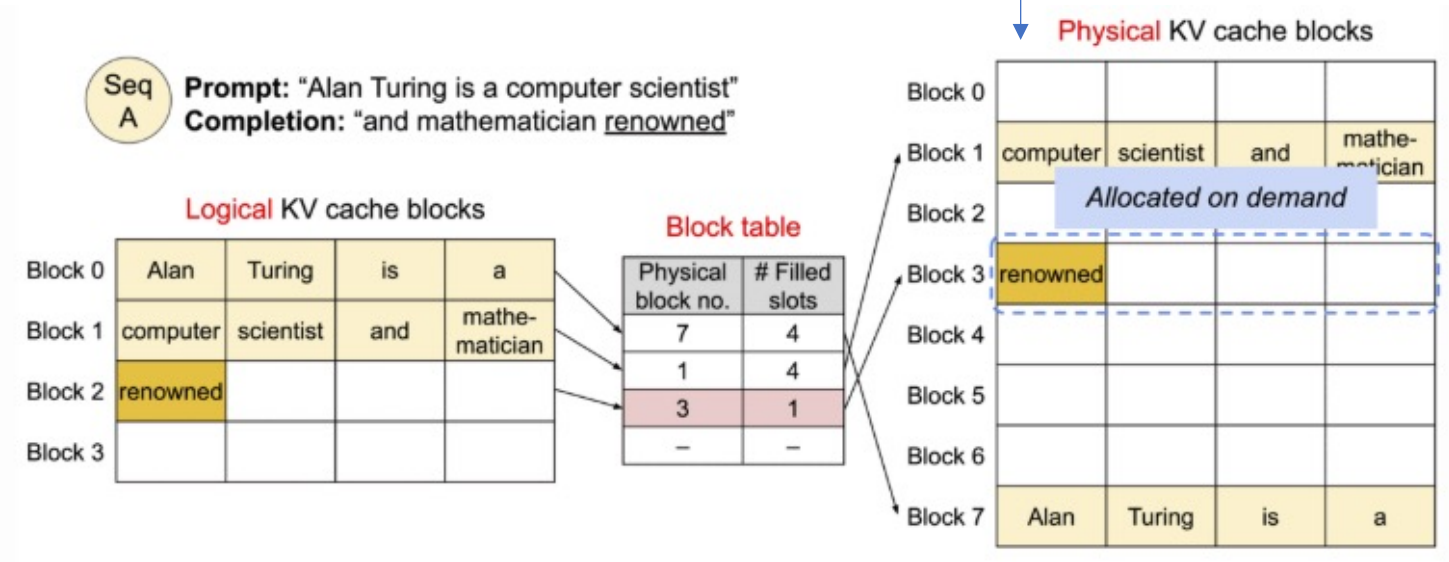
NVIDIA A100 with 40GB HBM

Orca: another serving framework

Memory Management

- Paged Attention
  - Inspired by paged virtual memory
    - KV block table: translation between logical KV blocks (what the AI model sees) and their actual physical locations in GPU memory

What is the size of a physical block?



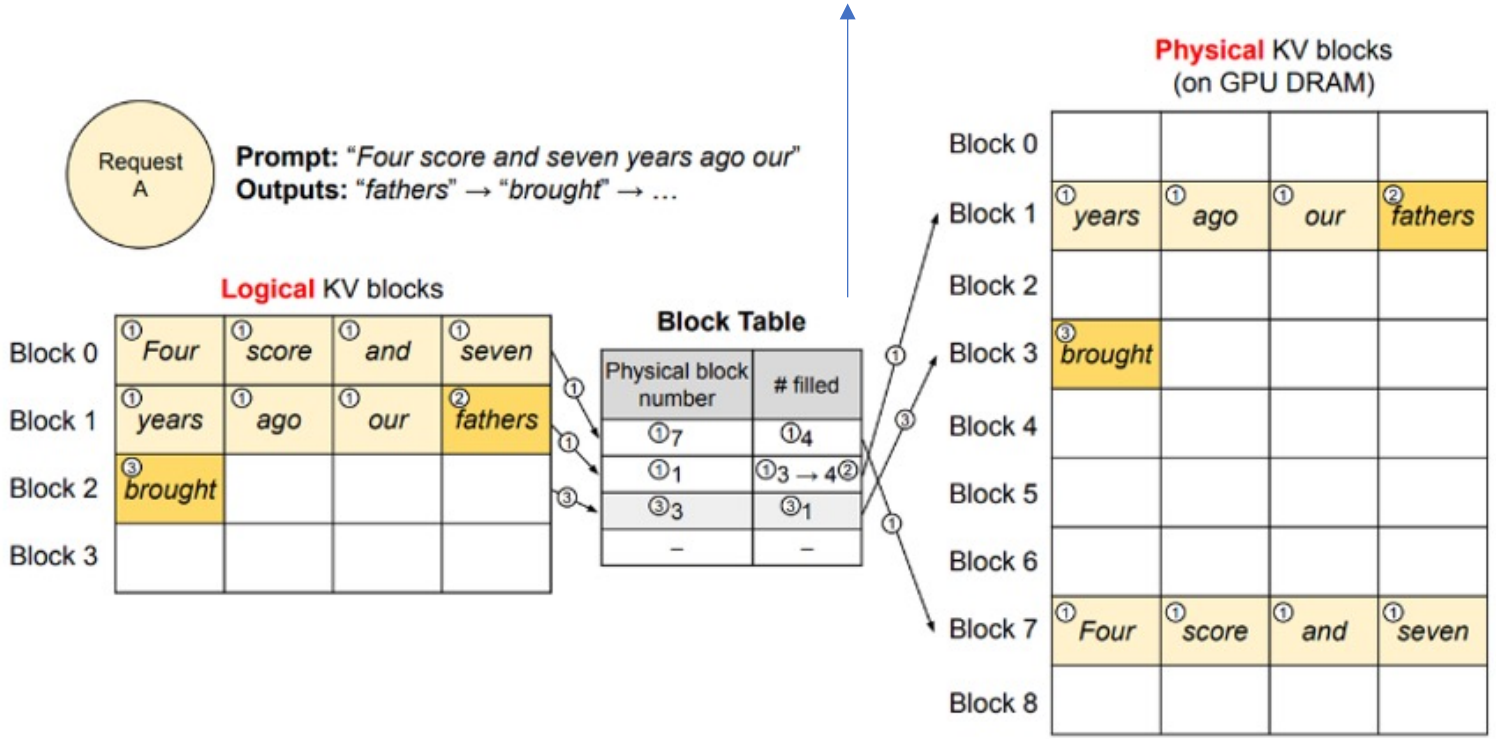
Logica block #0 (#2) is stored physical block #7 (#3) where the KV of 4 (only 1) tokens are filled

Memory Management

- Paged Attention
  - An example of decoding

The last slot of logical block 1 is still available, vLLM stores the newly generated token there, and the block table is updated

A prompt containing 7 tokens: vLLM initially allocates 2 logical blocks to store the KV (block 0 and block 1), then maps them to 2 physical KV blocks (blocks 1 and 7) and records this mapping in the block table.

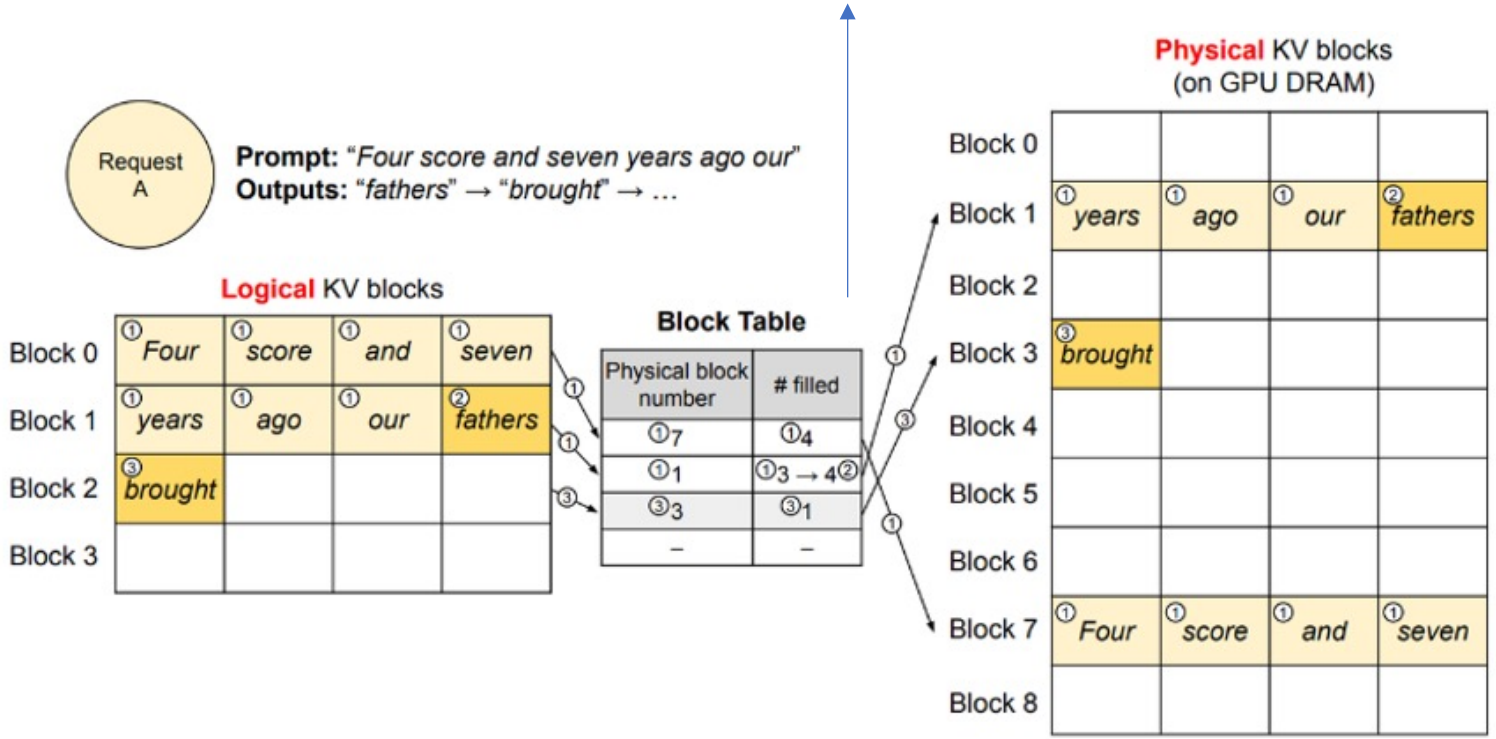


Memory Management

- Paged Attention
  - An example of decoding

The last slot of logical block 1 is still available, vLLM stores the newly generated token there, and the block table is updated

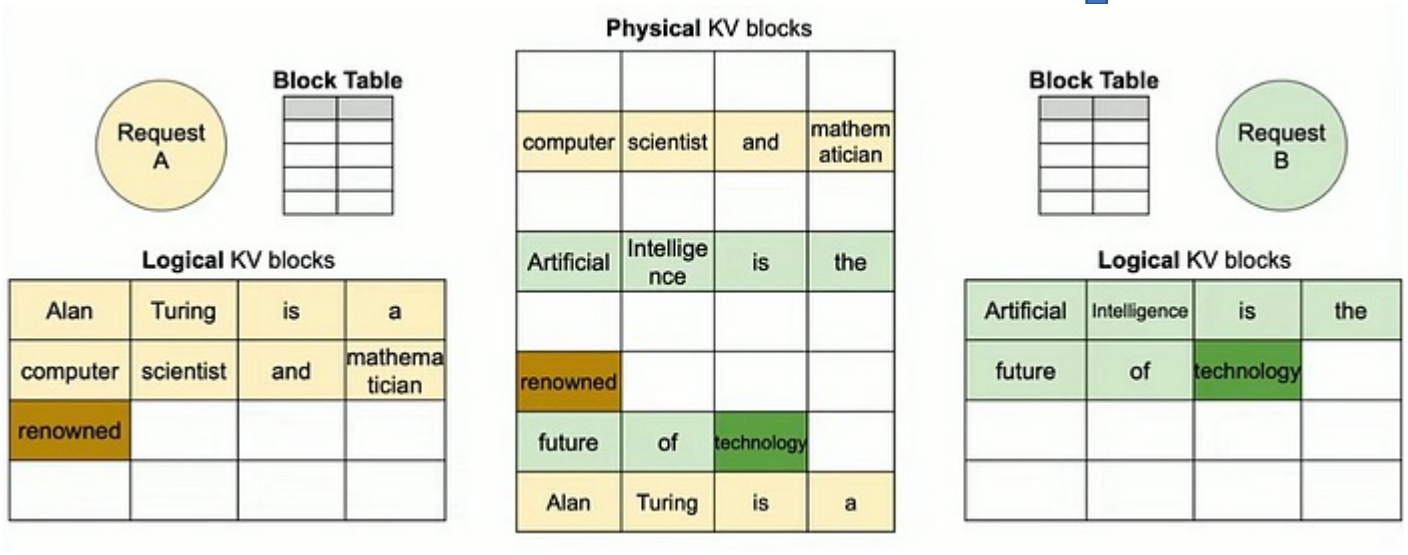
A prompt containing 7 tokens: vLLM initially allocates 2 logical blocks to store the KV (block 0 and block 1), then maps them to 2 physical KV blocks (blocks 1 and 7) and records this mapping in the block table.



Memory Management

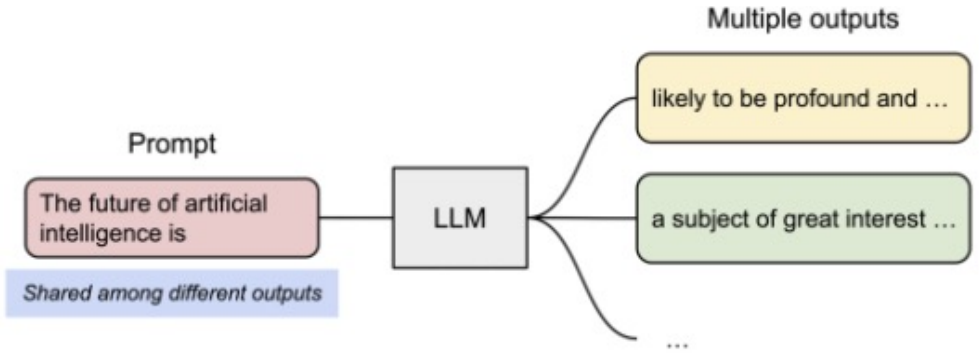
- Paged Attention
  - An example of decoding

Logical blocks of different requests will be mapped to different physical blocks

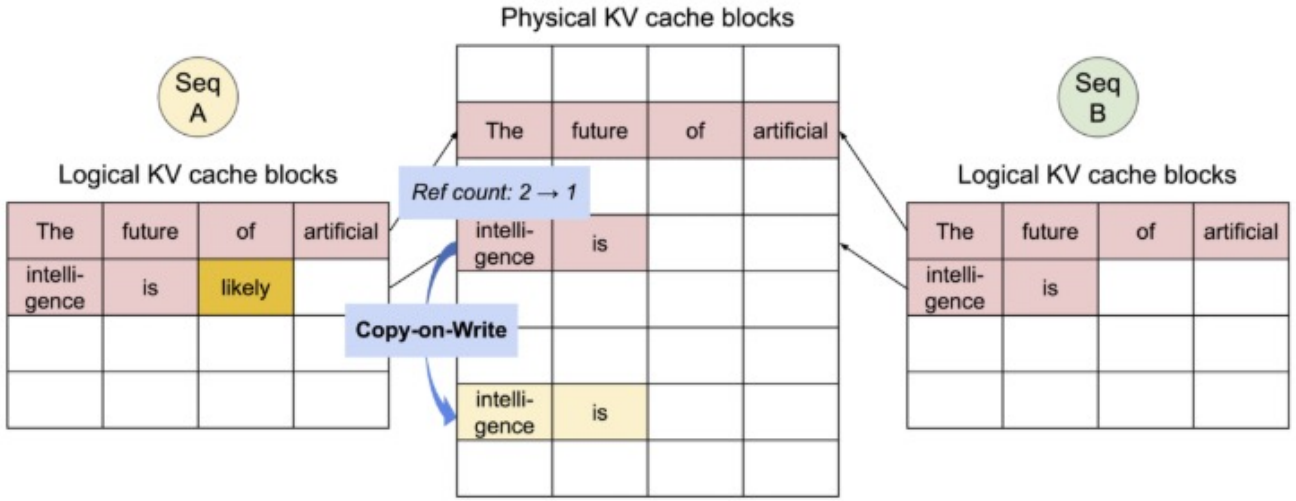


Memory Management

- Paged Attention
  - Parallel sampling or beam search
    - Parallel sampling: multiple generation paths originate from the same prompt
    - Copy-on-Write (CoW): a memory optimization strategy where data isn't immediately duplicated when a copy is requested



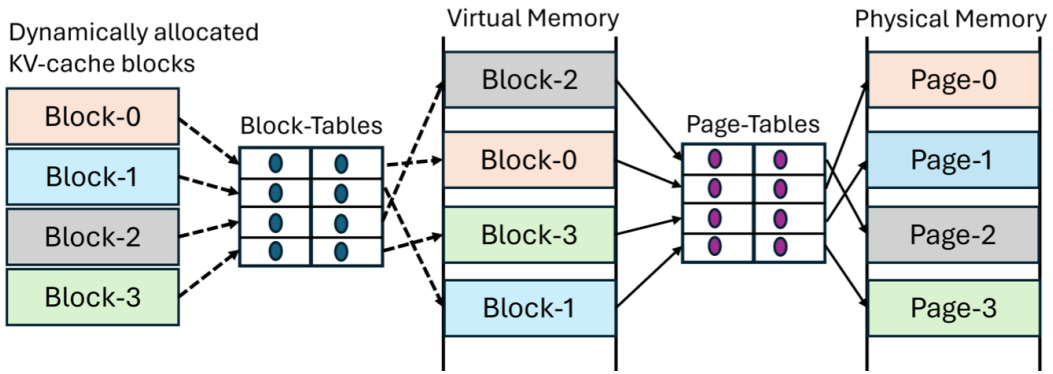
Parallel sampling



logical blocks for the prompts of both sequences are mapped to the same physical blocks

Memory Management

- Revisiting PagedAttention
  - Requires re-writing the attention kernel and adds redundancy in the serving framework



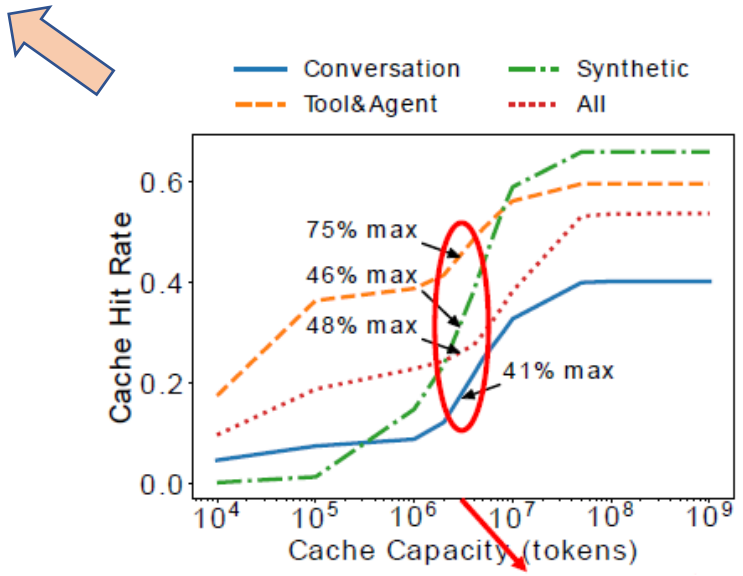
A request allocates four KV cache blocks over time that are usually non-contiguous in virtual memory. The serving system needs to track the virtual memory addresses of KV cache blocks and pass them to the attention kernel at runtime.

- vAttention
  - Mitigates fragmentation in physical memory while retaining the virtual memory contiguity of the KV cache (supplementary information)

Source:vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention

Memory Management

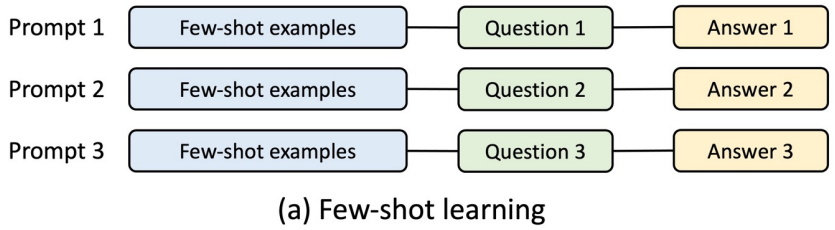
- Prefix Caching
  - **20-80 rule**: a small portion of requests accounts for the majority of usage
  - KV cache reuse across requests
- Application Scenarios
  - Multi-Turn Conversations
  - Retrieval-Augmented Generation
  - Coding Assistants and IDE Integration
  - Few-Shot Prompting / In-Context Learning
  - ……



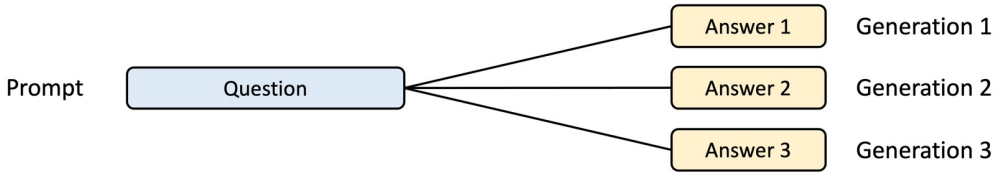
Around 50% of the tokens' KVCache in the real-world workloads can be reused

Memory Management

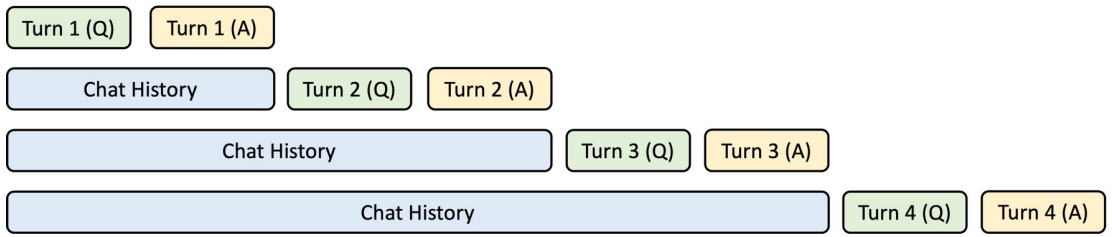
- KV Cache Reuse



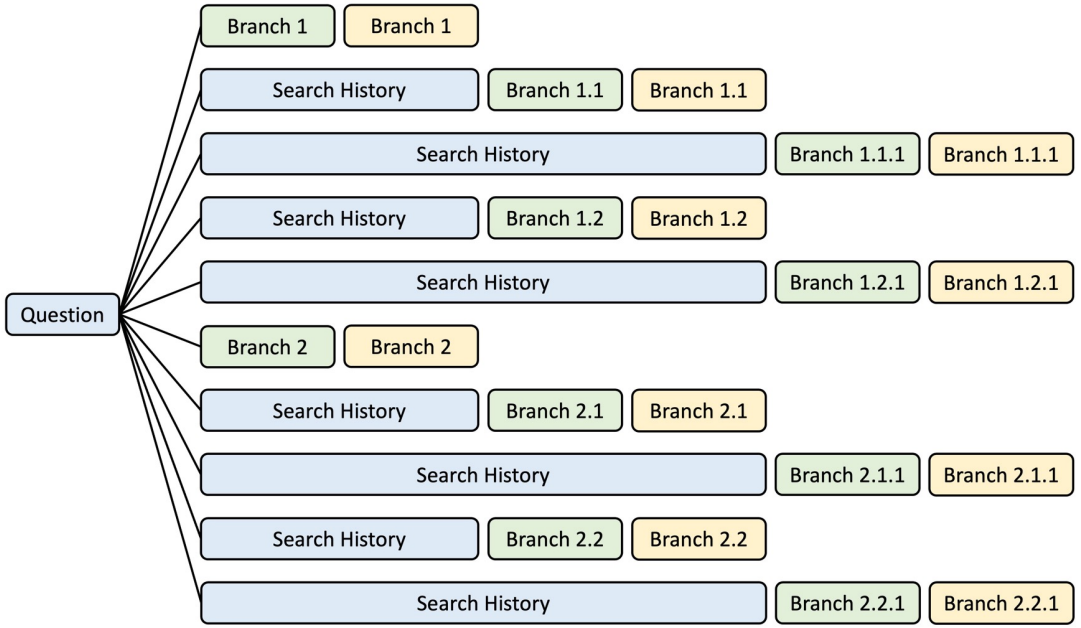
(a) Few-shot learning



(b) Self-consistency



(c) Multi-turn chat

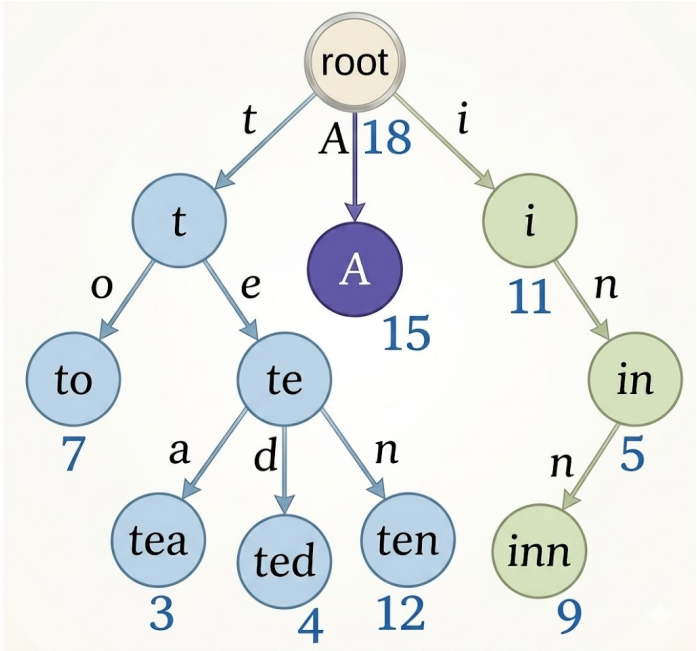


(d) Tree-of-thought

https://medium.com/@marketing_novita.ai/accelerating-llm-workflows-with-sglang-on-novita-ai-051a35292cf1

Memory Management

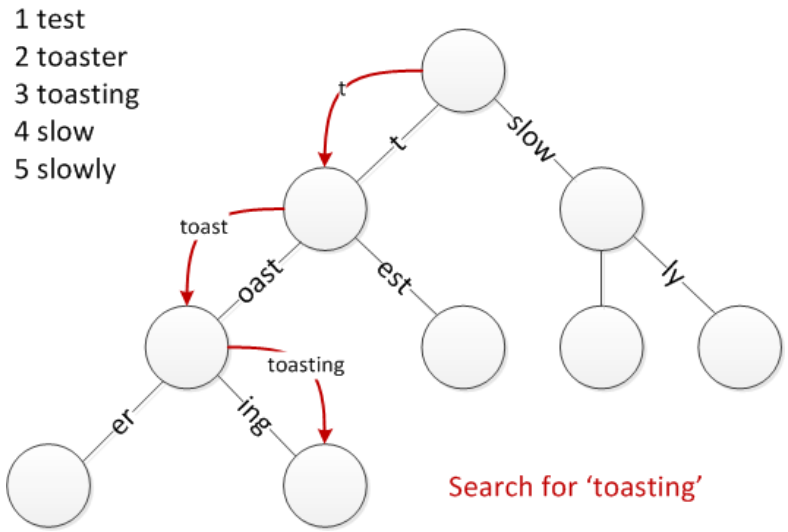
- Radix Tree
  - Trie, a specialized search tree data structure used to store and retrieve strings
  - Root does not store any character, each edge represents a character, and any path from the root to a node corresponds to a prefix of some string



A trie for keys “A”, “to”, “tea”, “ted”, “ten”, “i”, “in”, and “inn”. Each complete English word has an arbitrary integer value associated with it.

Memory Management

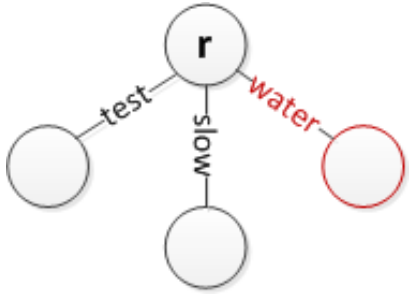
- Radix Tree
  - A space-optimized trie (prefix tree)
    - The number of children of every internal node is at most the radix r of this tree
    - If a node is the only child, it will be merged to its parent; an edge contains one or more characters
    - Efficient for small sets and for sets of strings that share long prefixes



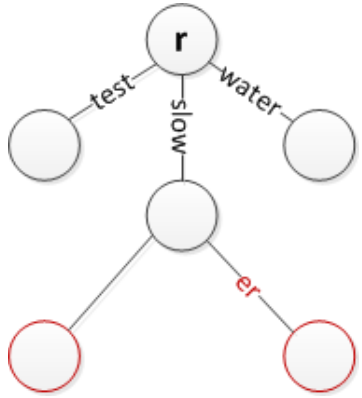
An illustration of storing five words and querying “toasting”

Memory Management

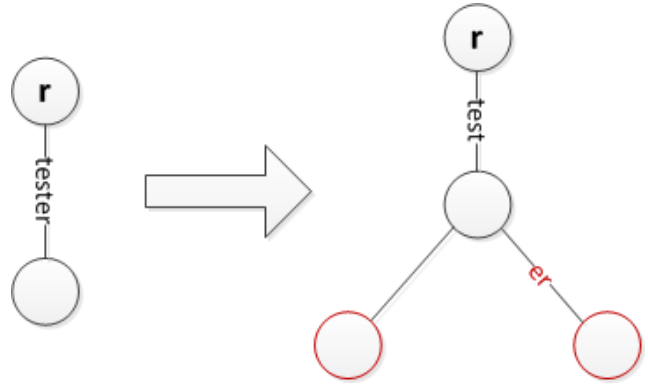
- Radix Tree
  - Insert and delete



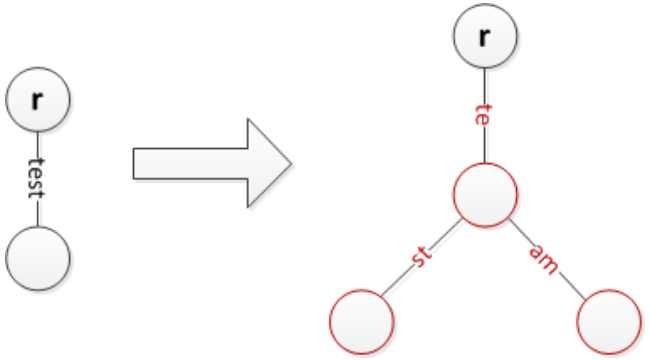
Insert 'water' at the root



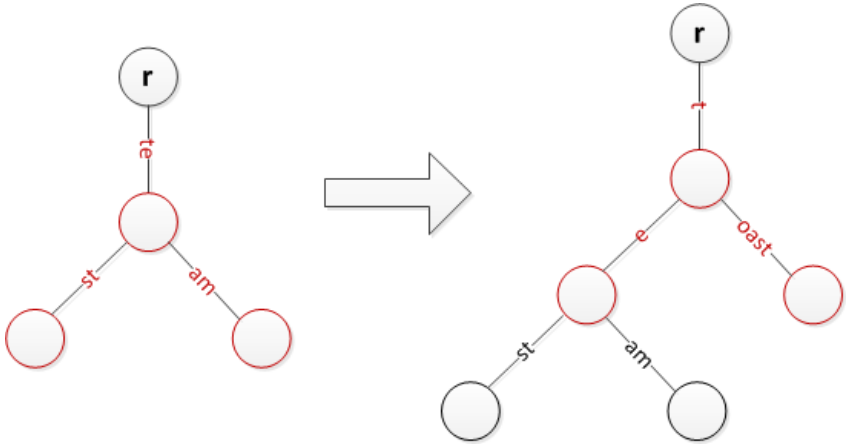
Insert 'slower' while keeping 'slow'



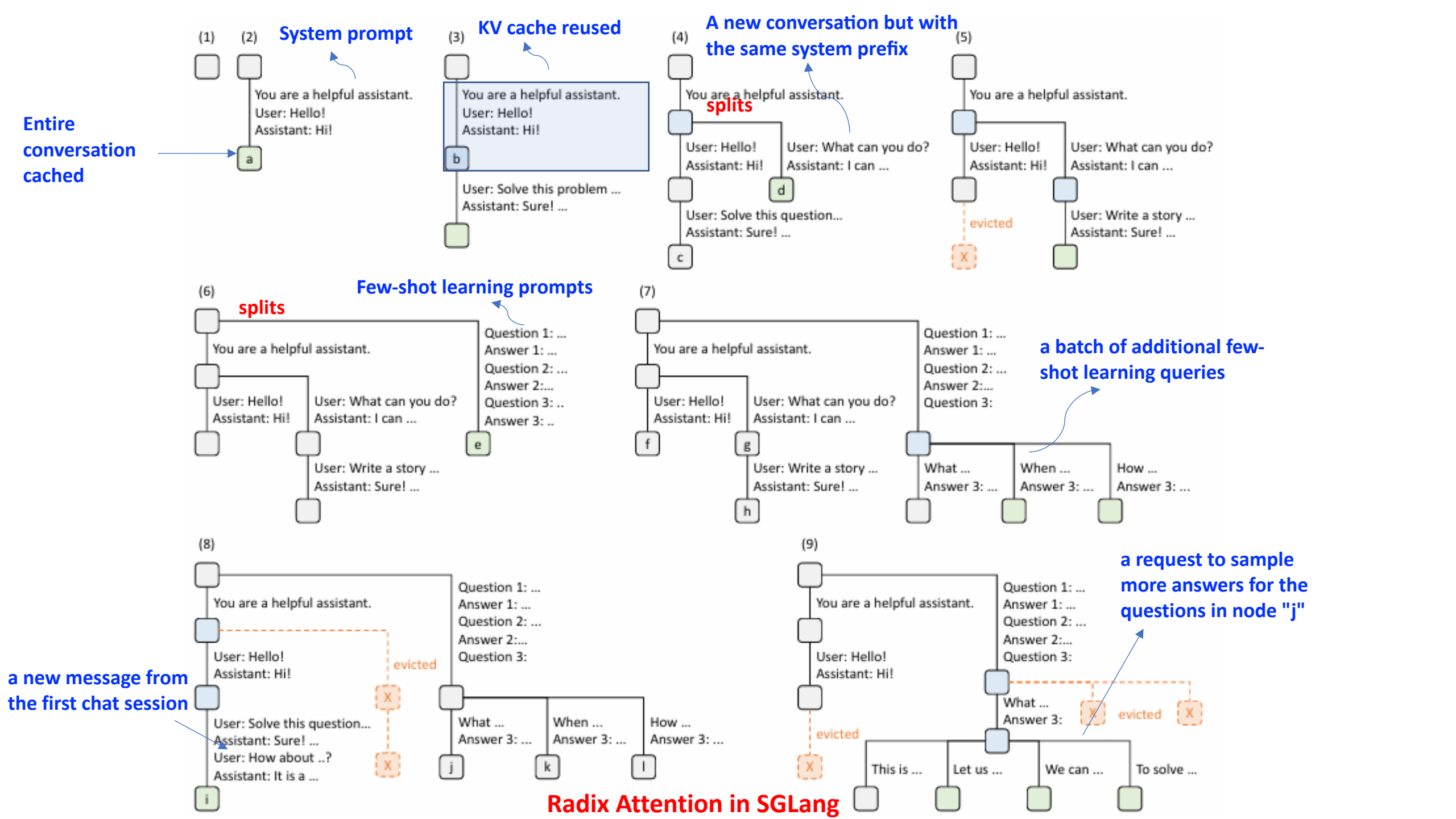
Insert 'test' which is a prefix of 'tester'



Insert 'team' while splitting 'test'
and creating a new edge label 'st'

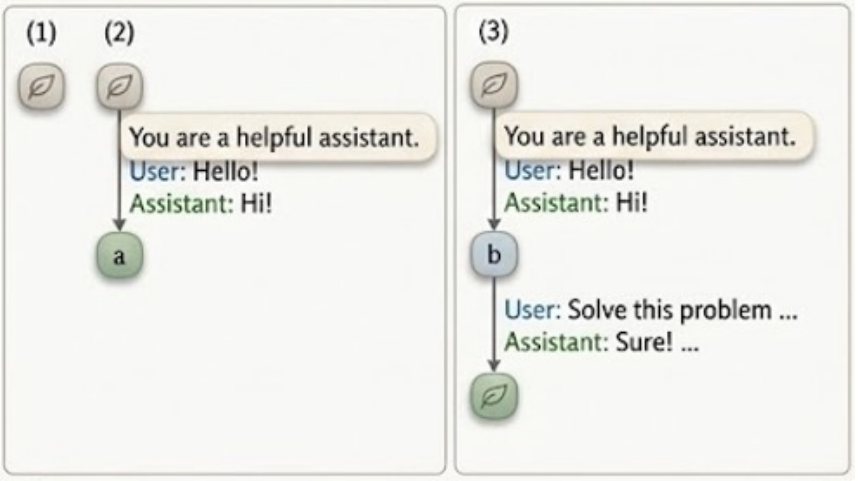


Insert 'toast' while splitting 'te' and
moving previous strings a level lower

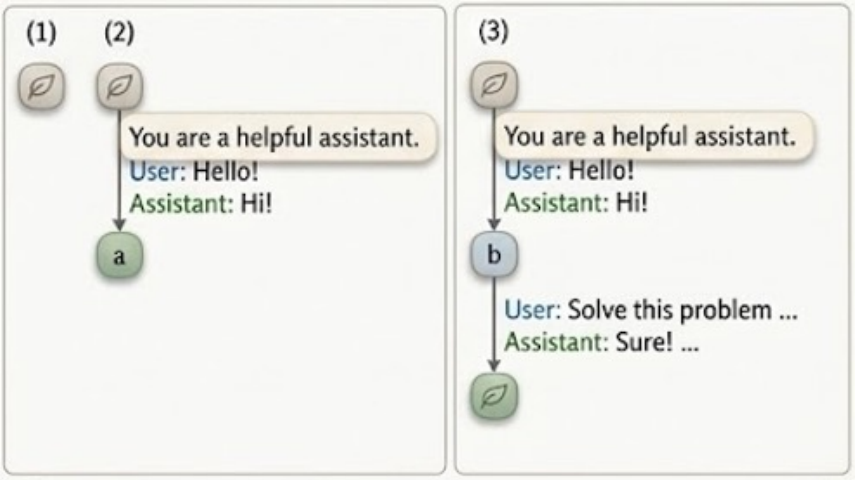

Entire conversation cached
(1)
(2)
System prompt
You are a helpful assistant.
User: Hello!
Assistant: Hi!
a
(3)
KV cache reused
You are a helpful assistant.
User: Hello!
Assistant: Hi!
b
User: Solve this problem ...
Assistant: Sure! ...
(4)
A new conversation but with the same system prefix
You are a helpful assistant.
splits
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
d
User: Solve this question...
Assistant: Sure! ...
c
(5)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
evicted
X
User: Write a story ...
Assistant: Sure! ...
(6)
splits
Few-shot learning prompts
You are a helpful assistant.
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
User: Write a story ...
Assistant: Sure! ...
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3: ..
Answer 3: ..
e
(7)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
User: What can you do?
Assistant: I can ...
f
g
User: Write a story ...
Assistant: Sure! ...
h
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
a batch of additional few-shot learning queries
What ...
Answer 3: ...
When ...
Answer 3: ...
How ...
Answer 3: ...
(8)
a new message from the first chat session
You are a helpful assistant.
User: Hello!
Assistant: Hi!
evicted
X
User: Solve this question...
Assistant: Sure! ...
User: How about ..?
Assistant: It is a ...
X
i
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
What ...
Answer 3: ...
When ...
Answer 3: ...
How ...
Answer 3: ...
j
k
l
(9)
You are a helpful assistant.
User: Hello!
Assistant: Hi!
evicted
X
Question 1: ...
Answer 1: ...
Question 2: ...
Answer 2:...
Question 3:
a request to sample more answers for the questions in node "j"
What ...
Answer 3:
X
evicted
X
This is ...
Let us ...
We can ...
To solve ...
Radix Attention in SGLang



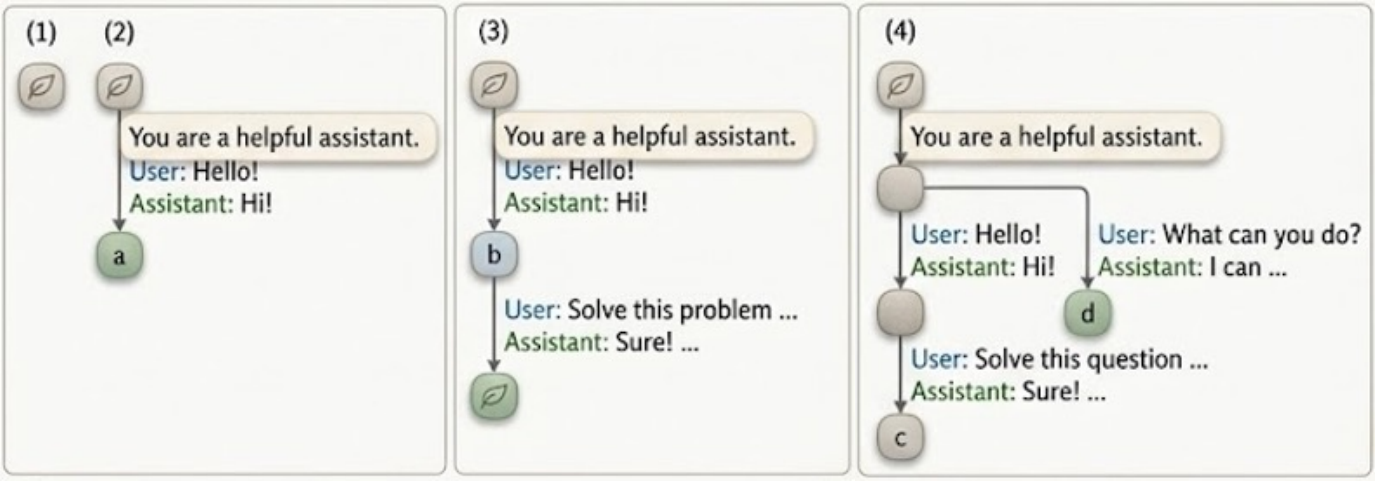
Step 1: the radix tree is initially **empty**; Step 2: the server processes an incoming user message "Hello" and responds with the LLM output "Hi". The **system prompt** "You are a helpful assistant", the **user message** "Hello!", and **the LLM reply** "Hi!" are consolidated into the tree as a single edge linked to a **new node**.



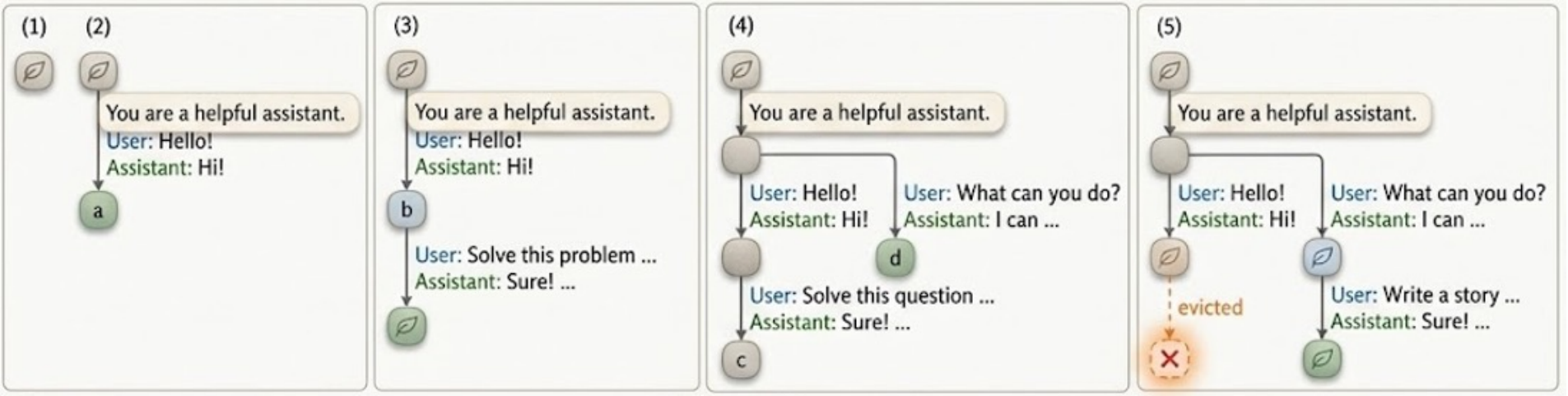
Step 3: a new prompt arrives and the server finds the prefix of the prompt in the radix tree and reuses its KV cache. The new turn is appended to the tree as a new node.



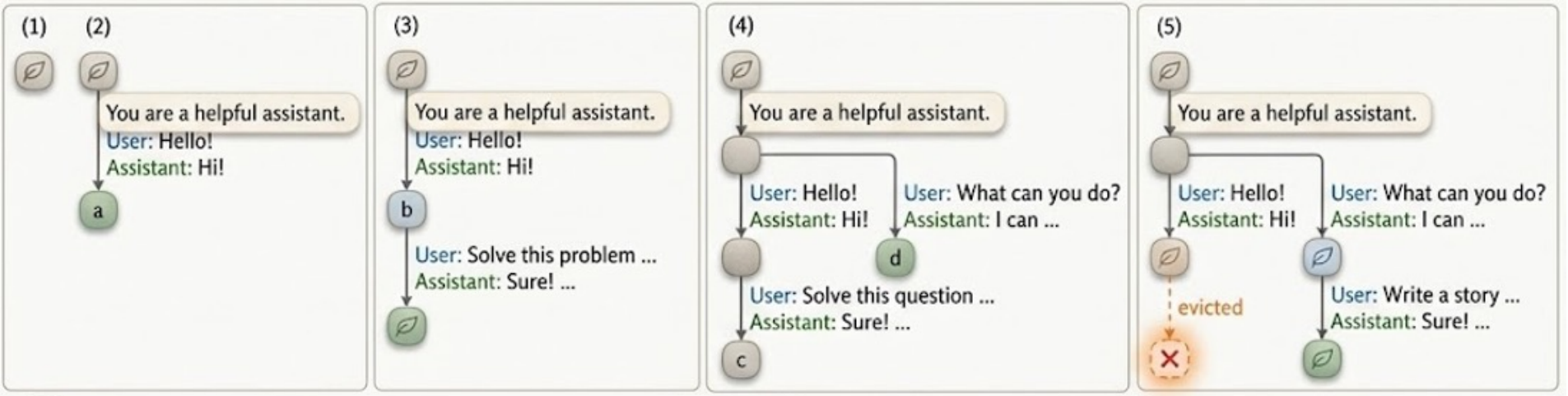
Step 3: a new prompt arrives and the server finds the prefix of the prompt in the radix tree and reuses its KV cache. The new turn is appended to the tree as a new node.

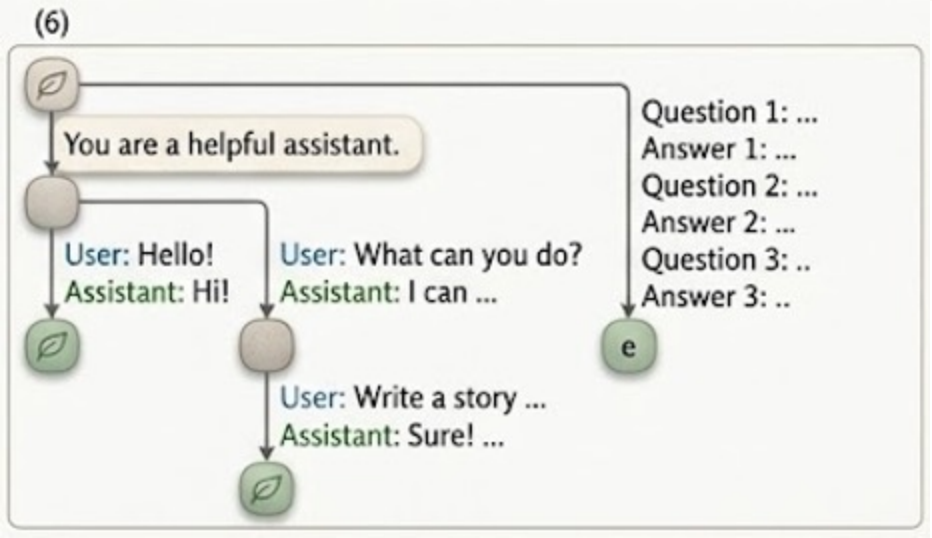


Step 4: a new chat session begins. The node ``b'' from (3) is split into two nodes to allow the two chat sessions to share the system prompt.

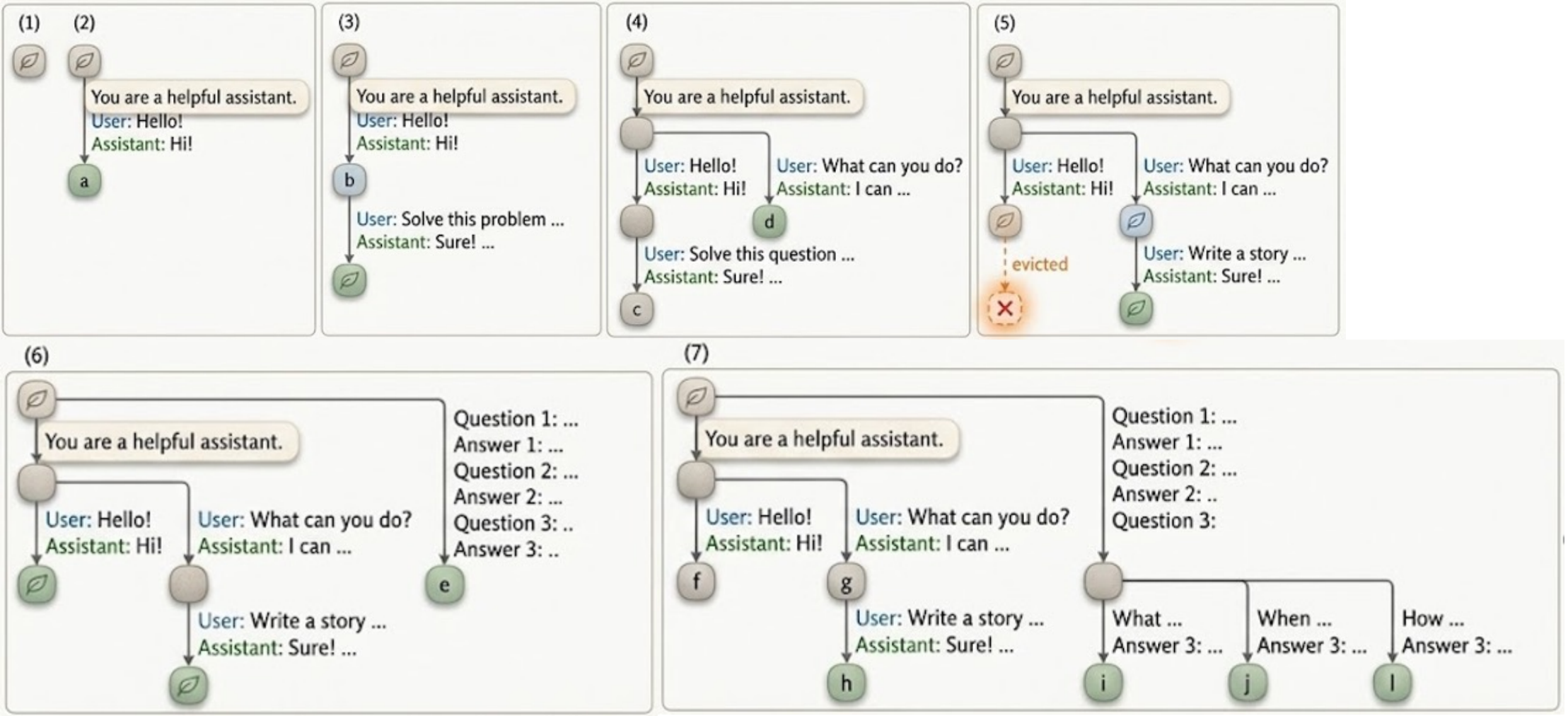


Step 5: the second chat session continues. However, due to the memory limit, node "c" from (4) must be evicted.

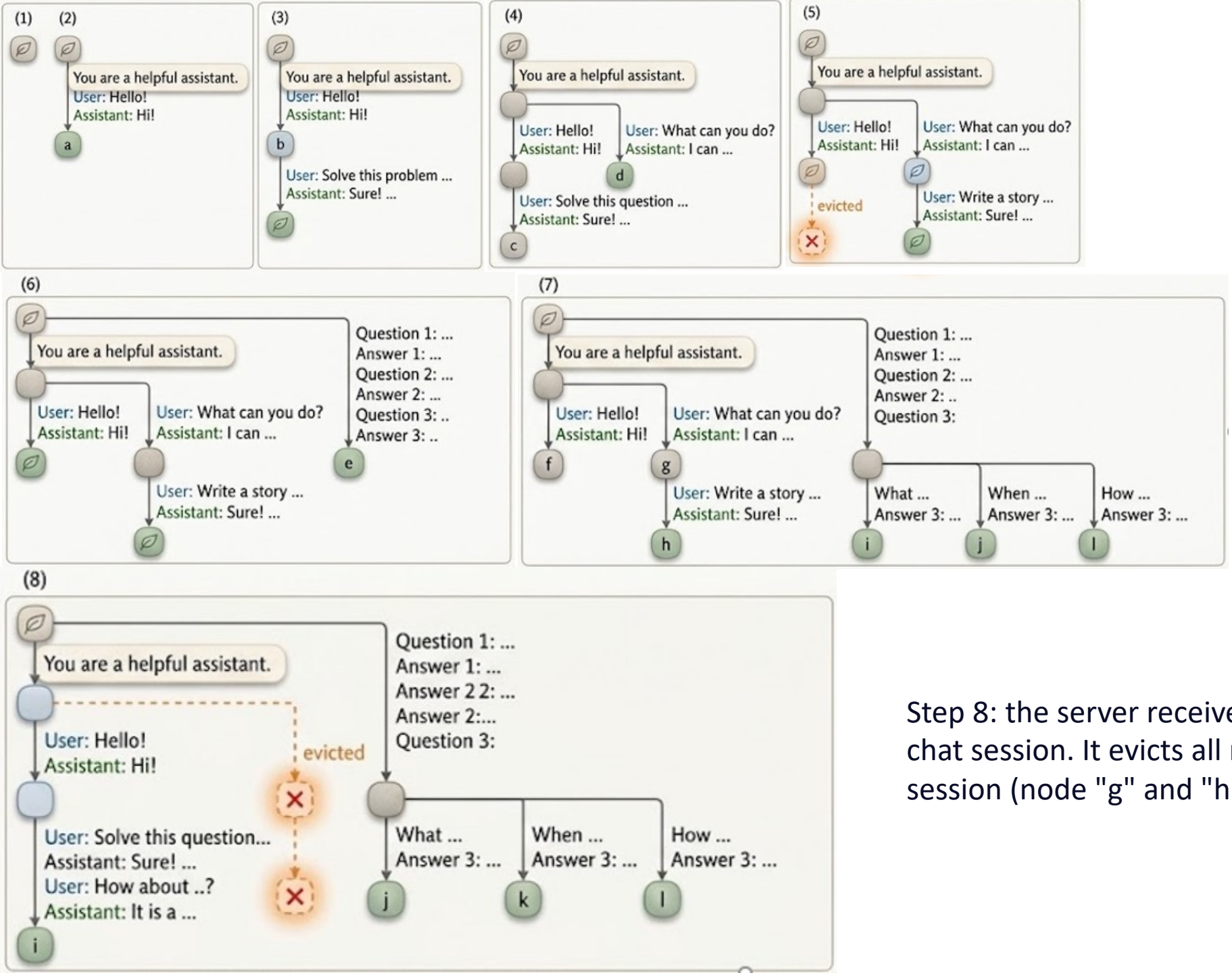




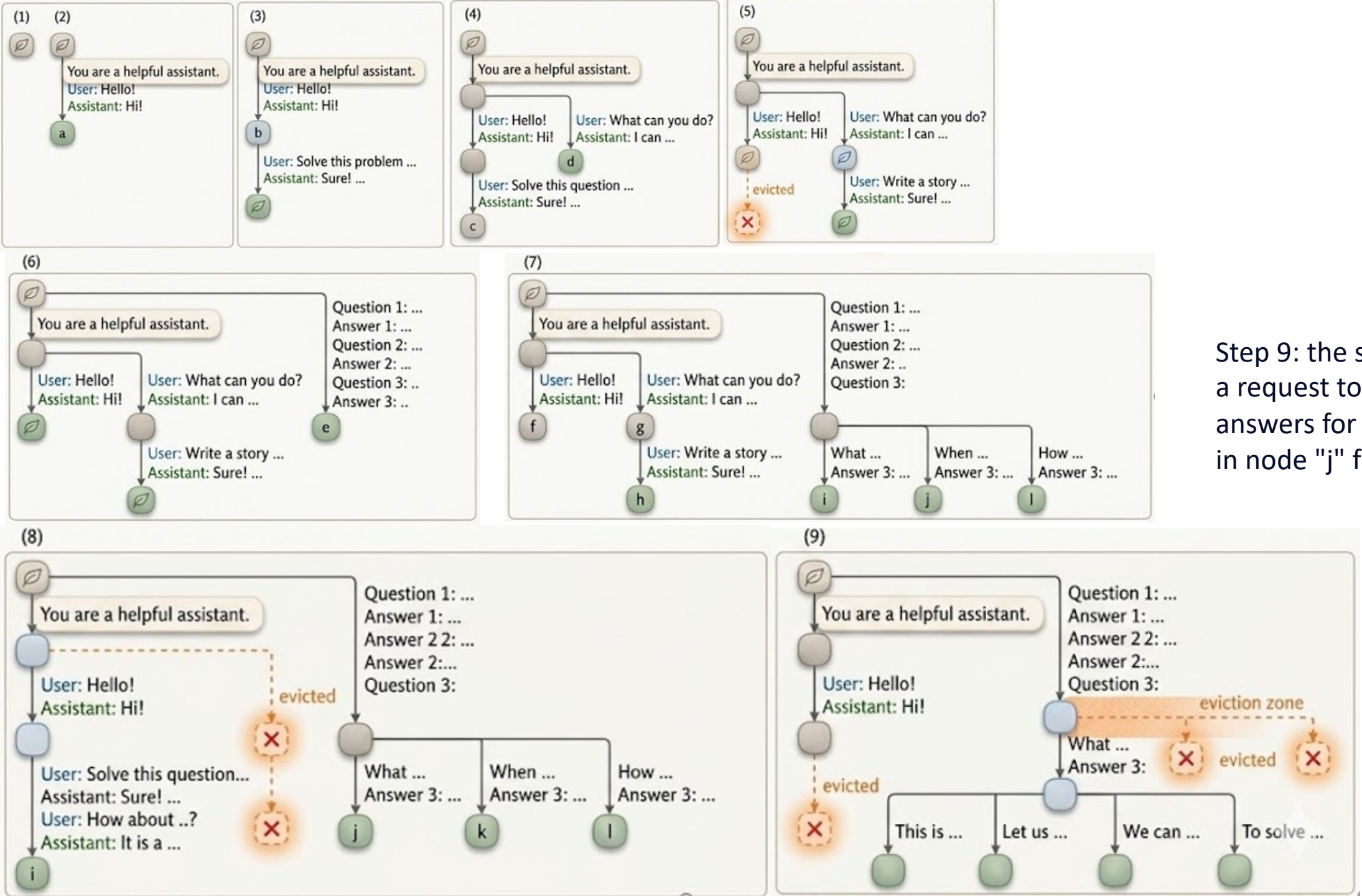
Step 6: the server receives a few-shot learning query, processes it, and inserts it into the tree. The root node is split because the new query does not share any prefix with existing nodes.



Step 7: the server receives a batch of additional few-shot learning queries. These queries share the same set of few-shot examples, so we split node 'e' from (6) to enable sharing.



Step 8: the server receives a new message from the first chat session. It evicts all nodes from the second chat session (node "g" and "h") as they are least recently used.



Step 9: the server receives a request to sample more answers for the questions in node "j" from (8)

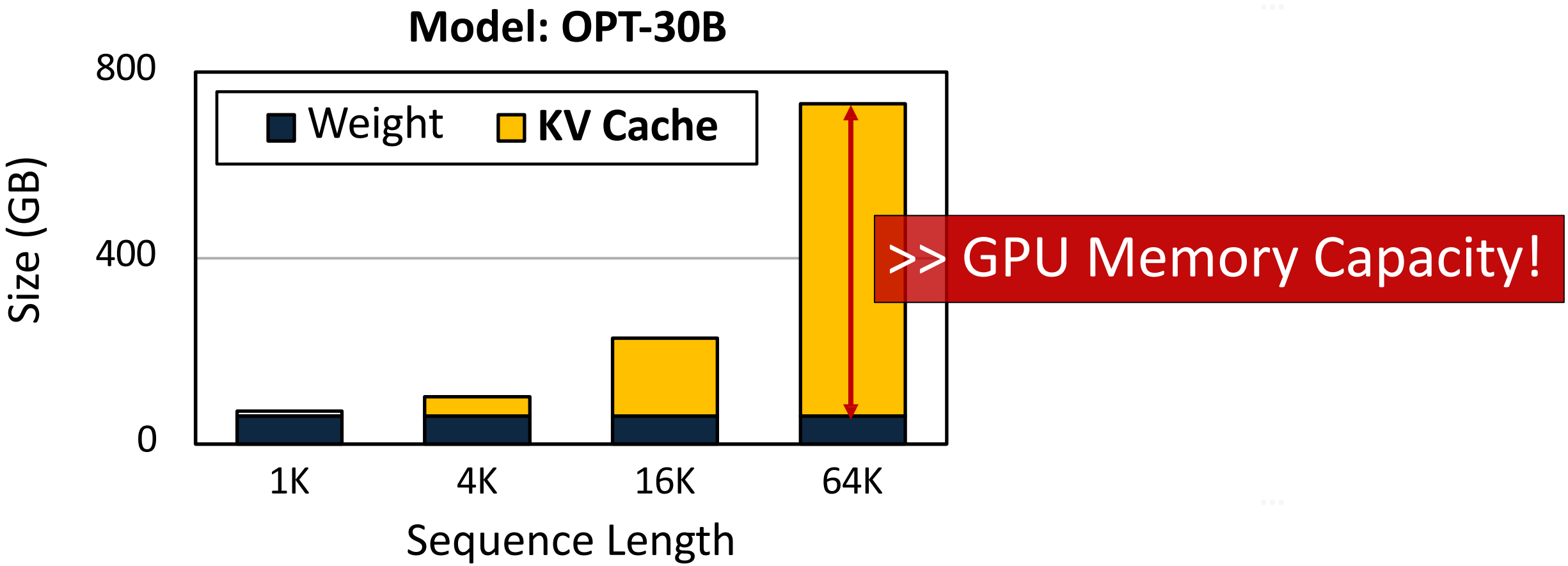
Memory Management

- More to say
  - KV reuse beyond rigid prefix is possible
  - KV reuse across different LLMs (but with the same origin) is possible

Inference Optimization

- Overview
- Attention Computation Optimization
- KV Cache Optimization
  - Model Architecture Optimization
  - KV Cache Data Structure Optimization
  - **KV Cache Memory Optimization**
- Parallel and Distributed Serving

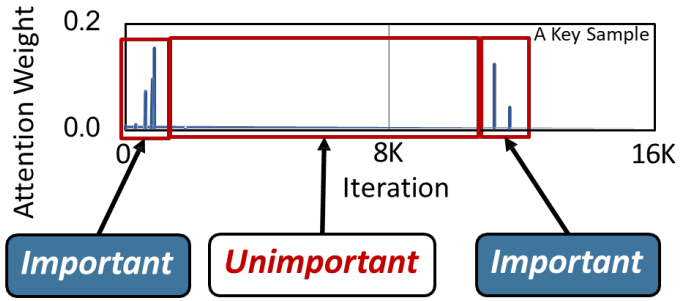
Memory Optimization

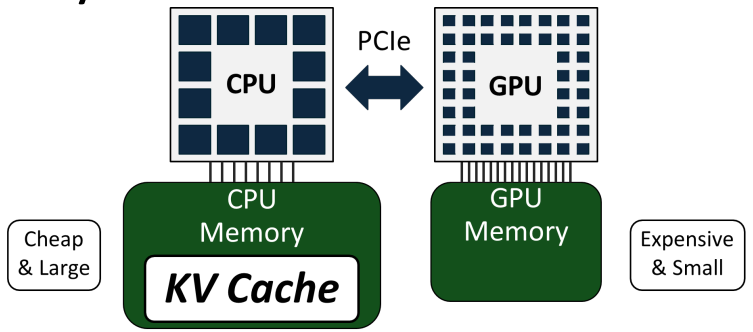


Source: InfiniGen Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management

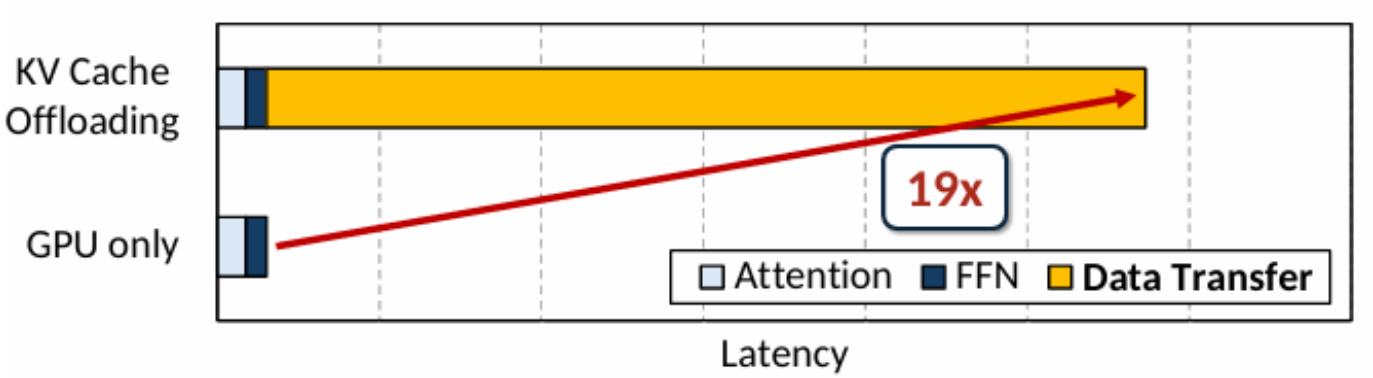
Memory Optimization

- Eviction
  - Permanent eviction: simple, but may cause an accuracy drop
  - Temporary eviction: offloads to CPU memory or SSD with limited bandwidth





  - ***Not*** all KV cache data needs to stay in GPU memory at all times



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

Dynamic sparse at inference: H2O, StreamingLLM, Quest

one KV cache, three regimes — token positions along the sequence



StreamingLLM recipe: cache = sink + window — stable perplexity out to 4M tokens, no fine-tuning (paper-reported)

H2O — Heavy Hitter Oracle

Zhang et al., 2023

- track the attention mass each token accumulates across generation
- evict low-score, non-recent tokens; keep heavy hitters + recent window
- ~20–30% of the cache kept with near-zero quality loss (paper-reported)

StreamingLLM

Xiao et al., 2023

- softmax needs a sink: the first tokens absorb attention no matter what — evicting them breaks the model
- keep sink + sliding window, drop the middle
- the standard baseline for streaming chat and infinite-length inputs

Quest

Tang et al., 2024 · Meta

- page-level selection at decode time — never materialize the full score row
- keys stored in pages of 16 tokens with per-page min/max; query scores pages by upper bound, loads top-K
- ≈2× faster long-context decode (paper-reported)

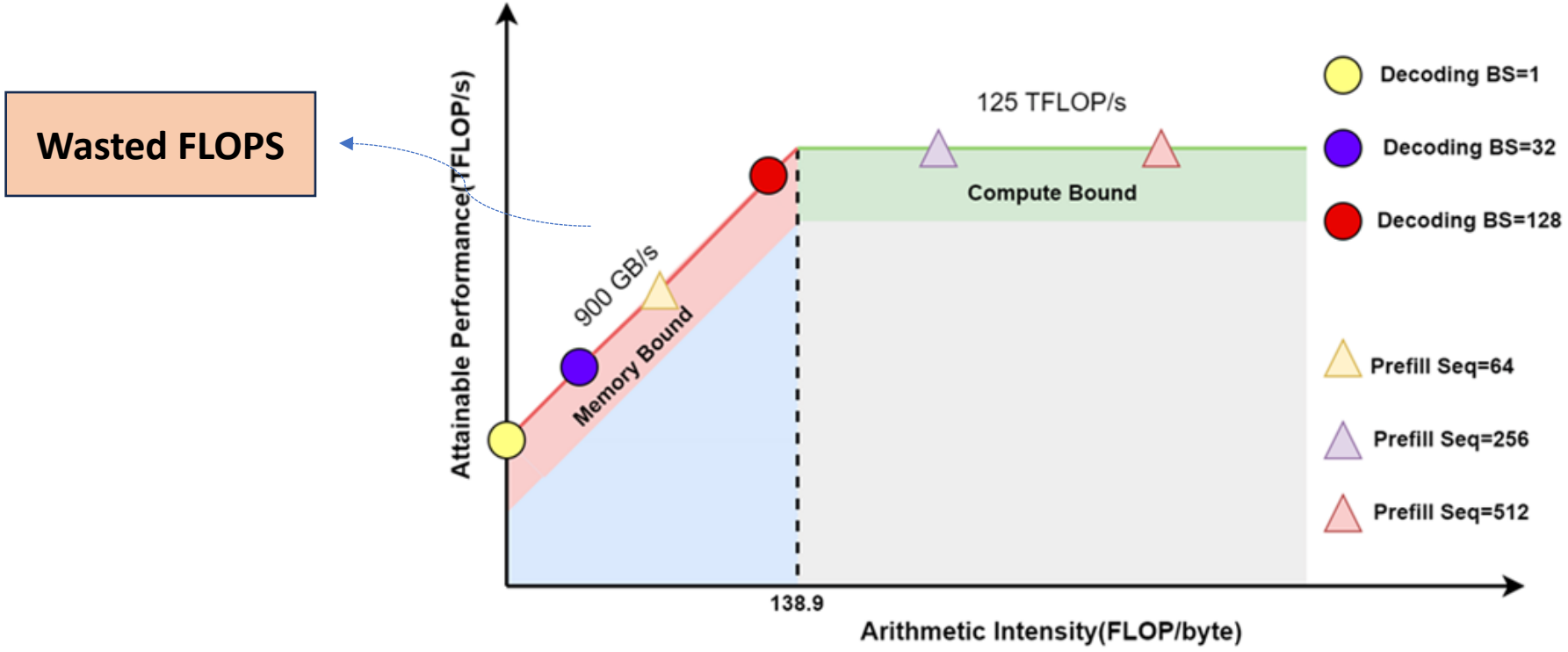
> Inference-time dynamic sparsity = systems techniques for pretrained dense models — no retraining, applied straight to the KV cache.

Inference Optimization

- Overview
- Attention Optimization
- Continuous Batching
- KV Cache Optimization
- Parallel and Distributed Serving
  - **Continuous Batching**
  - Prefill-Decode Disaggregation

Batching to Meet Compute-Bound

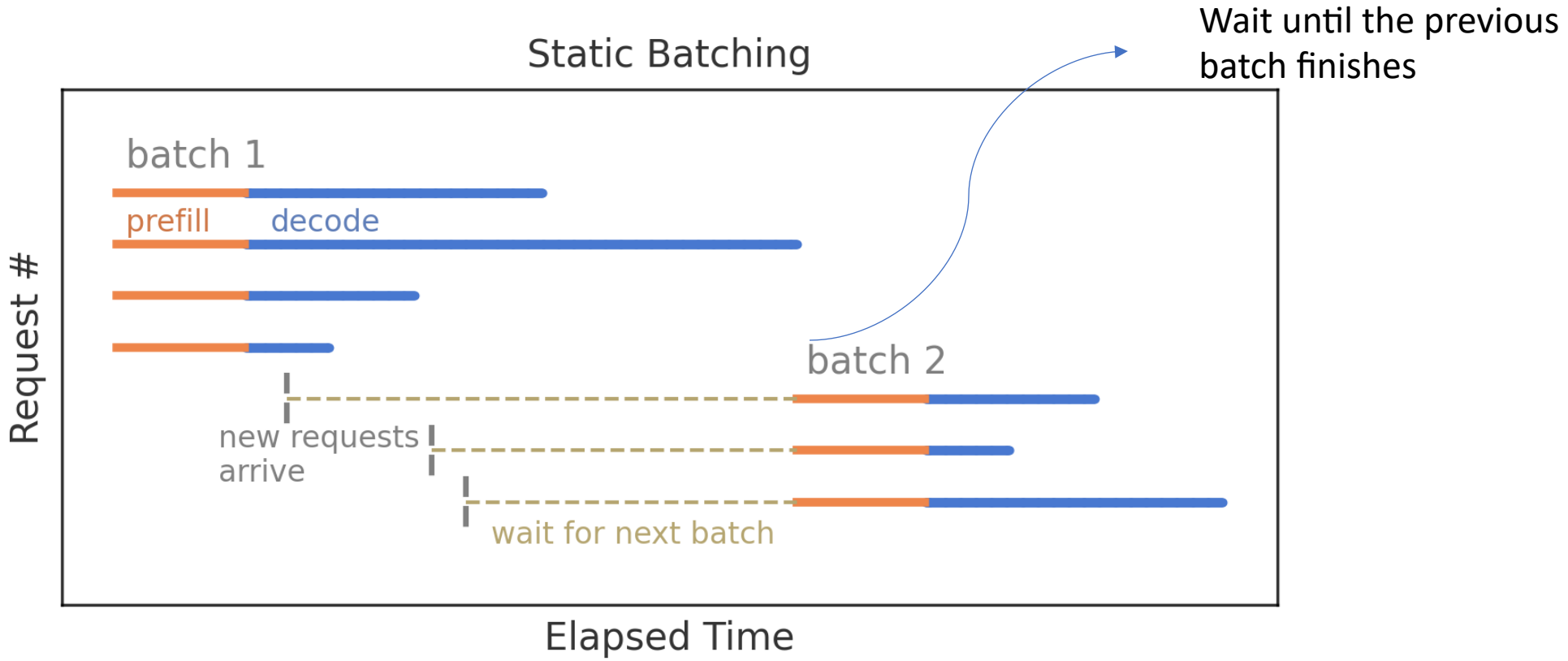
- Why batching multiple requests?
  - Generating tokens for a large number of prompts to match the compute bound



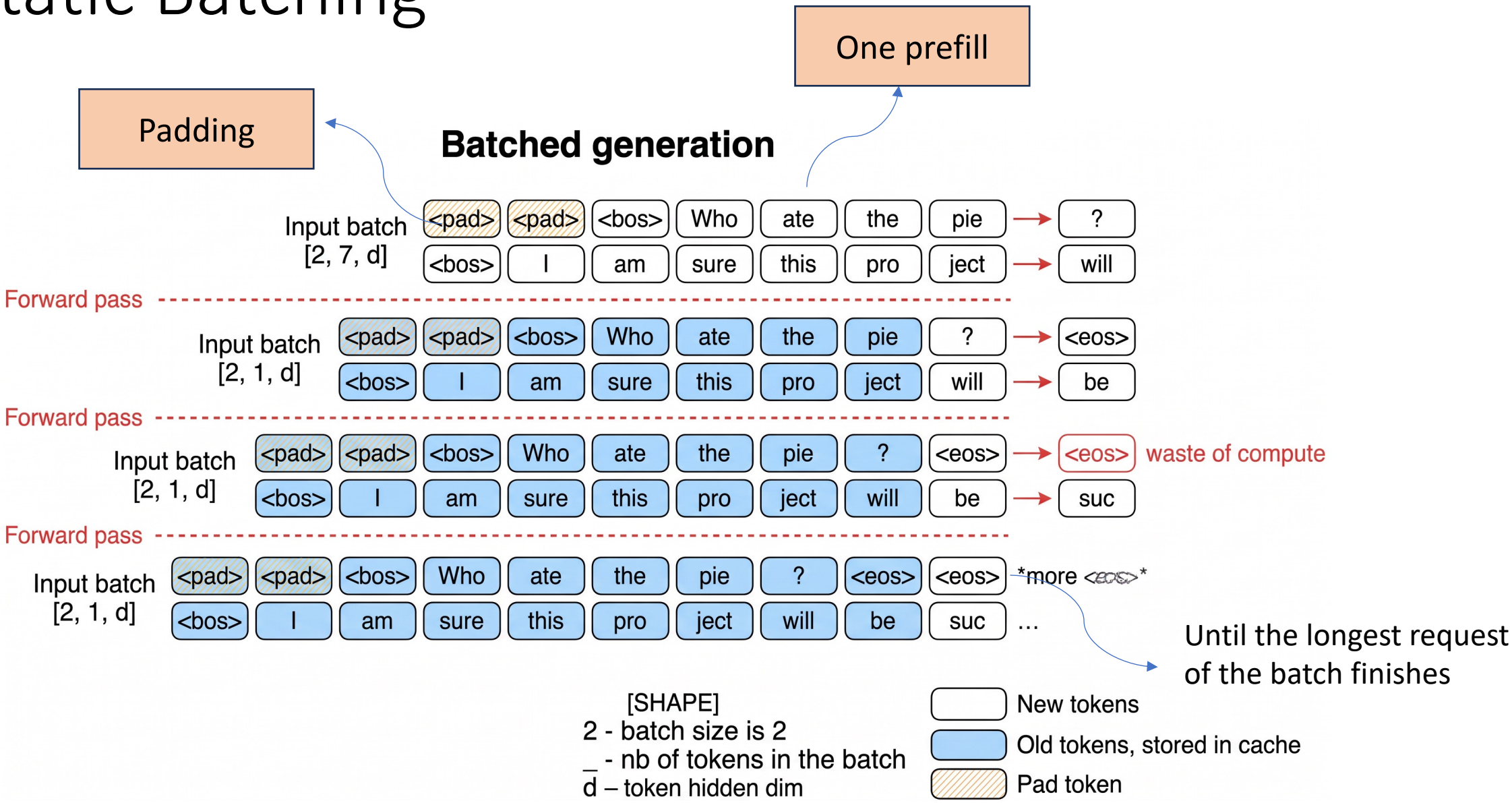
Roofline model of NVIDIA V100 GPU

Static Batching

- Naively serving multiple requests simultaneously
  - Unequal input sequence lengths, unknown output sequence lengths



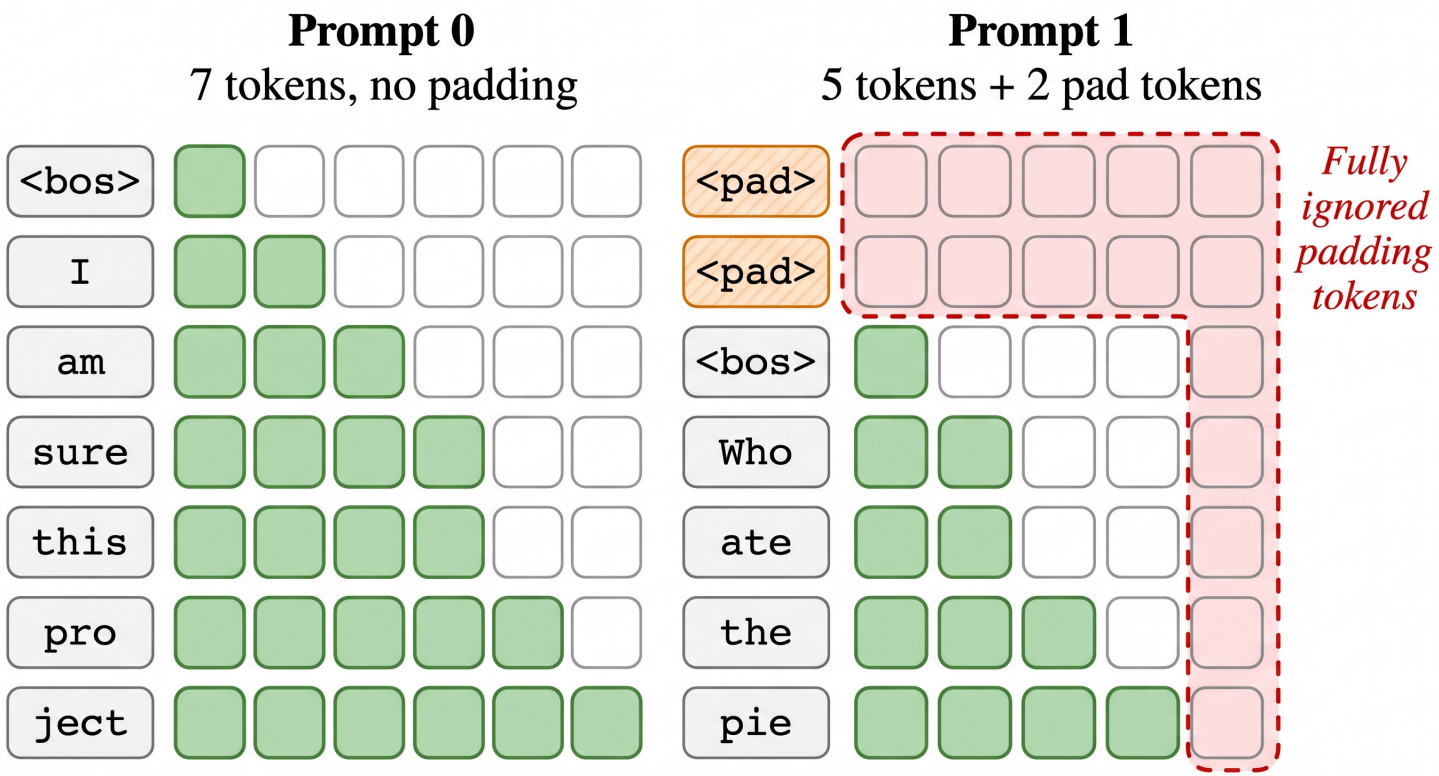
Static Batching



Source: https://huggingface.co/blog/continuous_batching

Static Batching

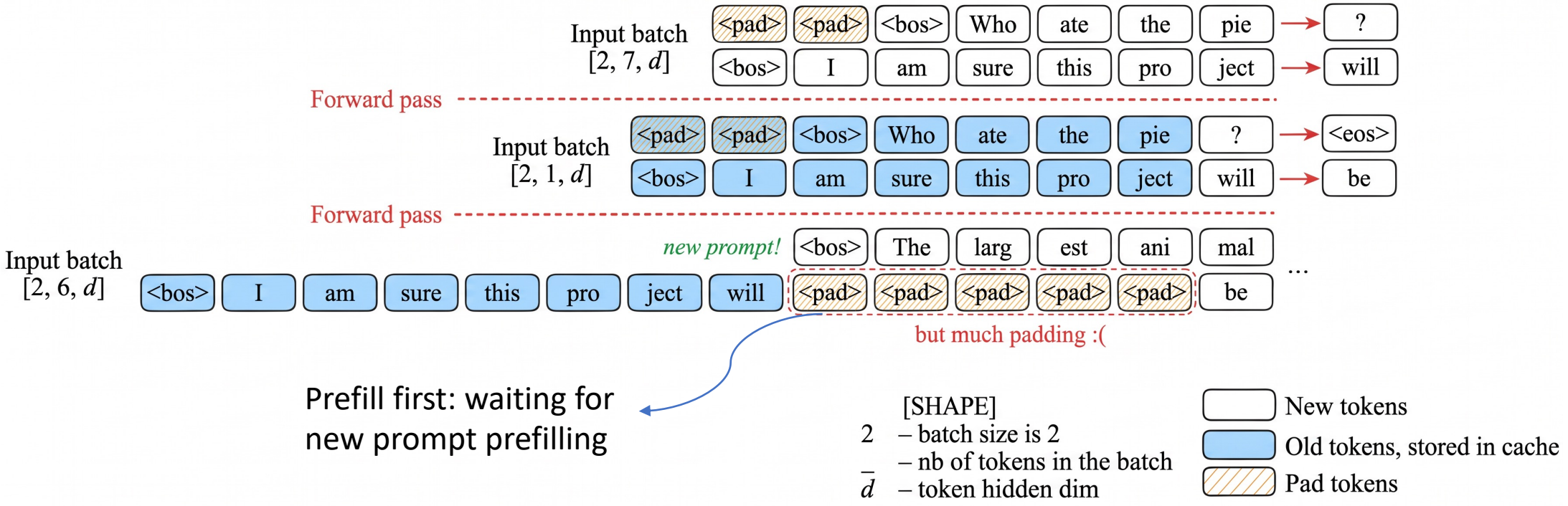
- How to implement it?
  - Avoiding the computations of certain diagonal lines via attention masking



https://huggingface.co/blog/continuous_batching

Continuous Batching

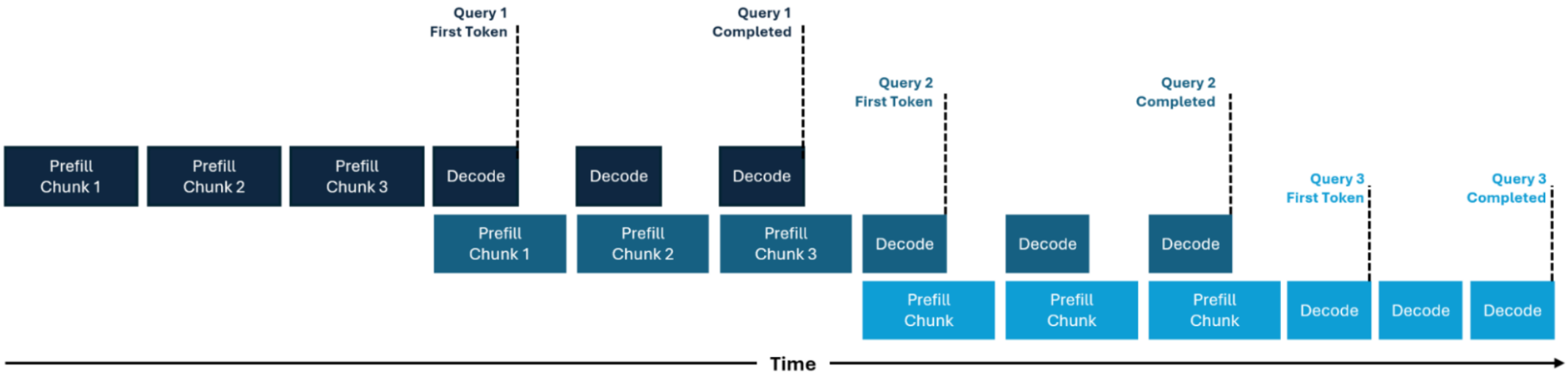
- Insert an inference request in the batch when an old request finishes



https://huggingface.co/blog/continuous_batching

Continuous Batching

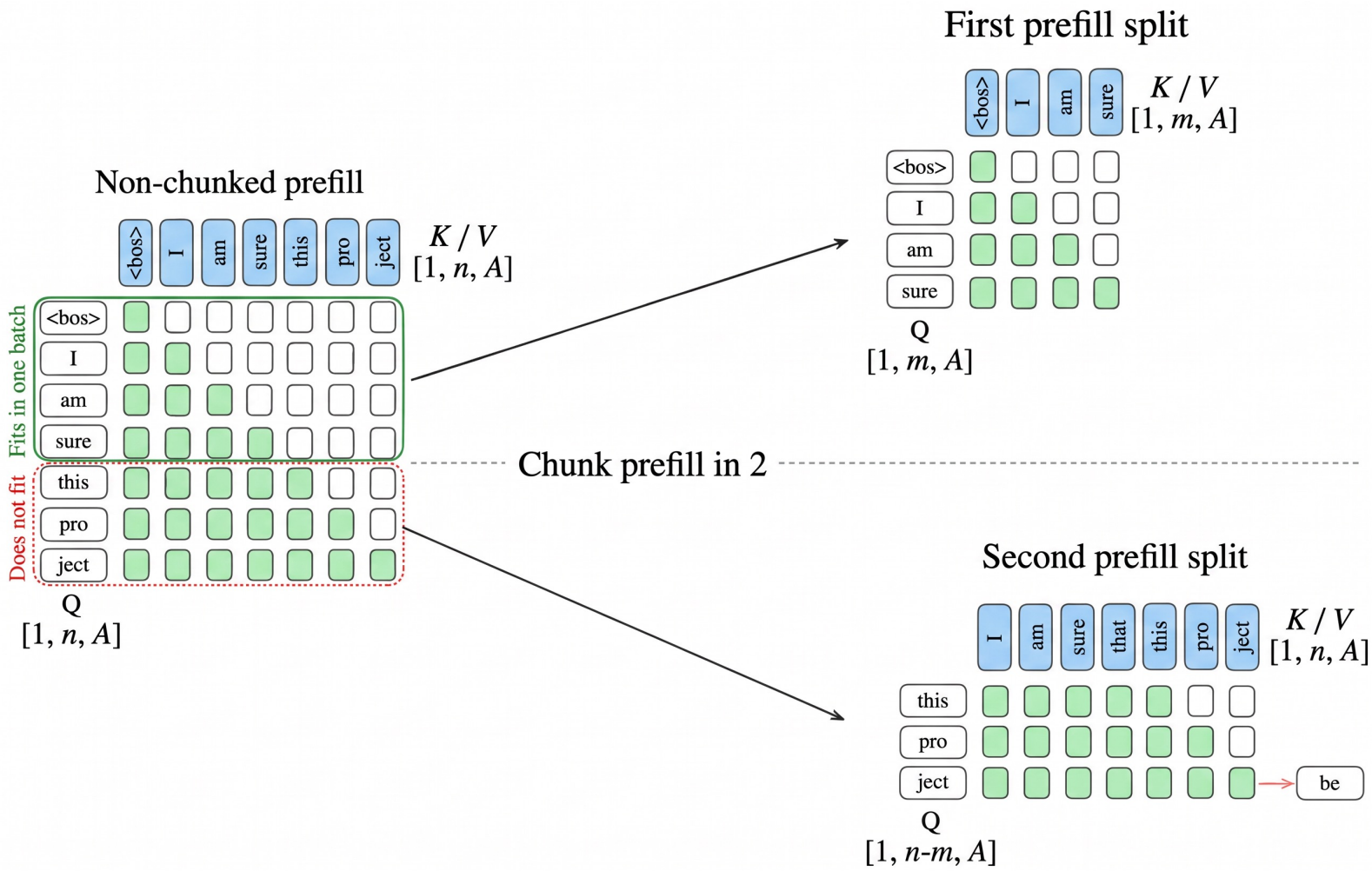
- Chunked Prefill
  - **Elastic Scheduling**: split the entire prompt into smaller chunks: more scheduling flexibility and enables mixing prefill and decode.
  - **Decode-Maximal Batching**: combine a single prefill chunk with multiple decode requests in the same batch, allowing decode operations to "piggyback" on the more compute-intensive prefill operations



Source: Streamlining AI Inference Performance and Deployment with NVIDIA TensorRT-LLM Chunked Prefill

Continuous Batching

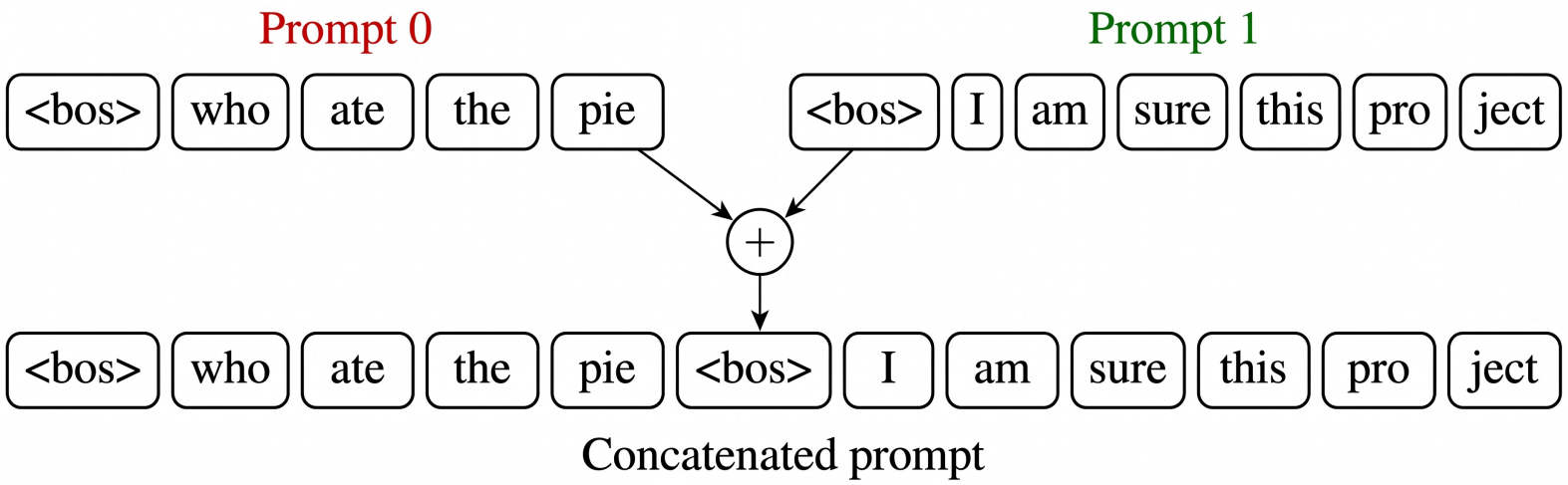
- How to enforce separate prefill?



Source: Streamlining AI Inference Performance and Deployment with NVIDIA TensorRT-LLM Chunked Prefill

Continuous Batching

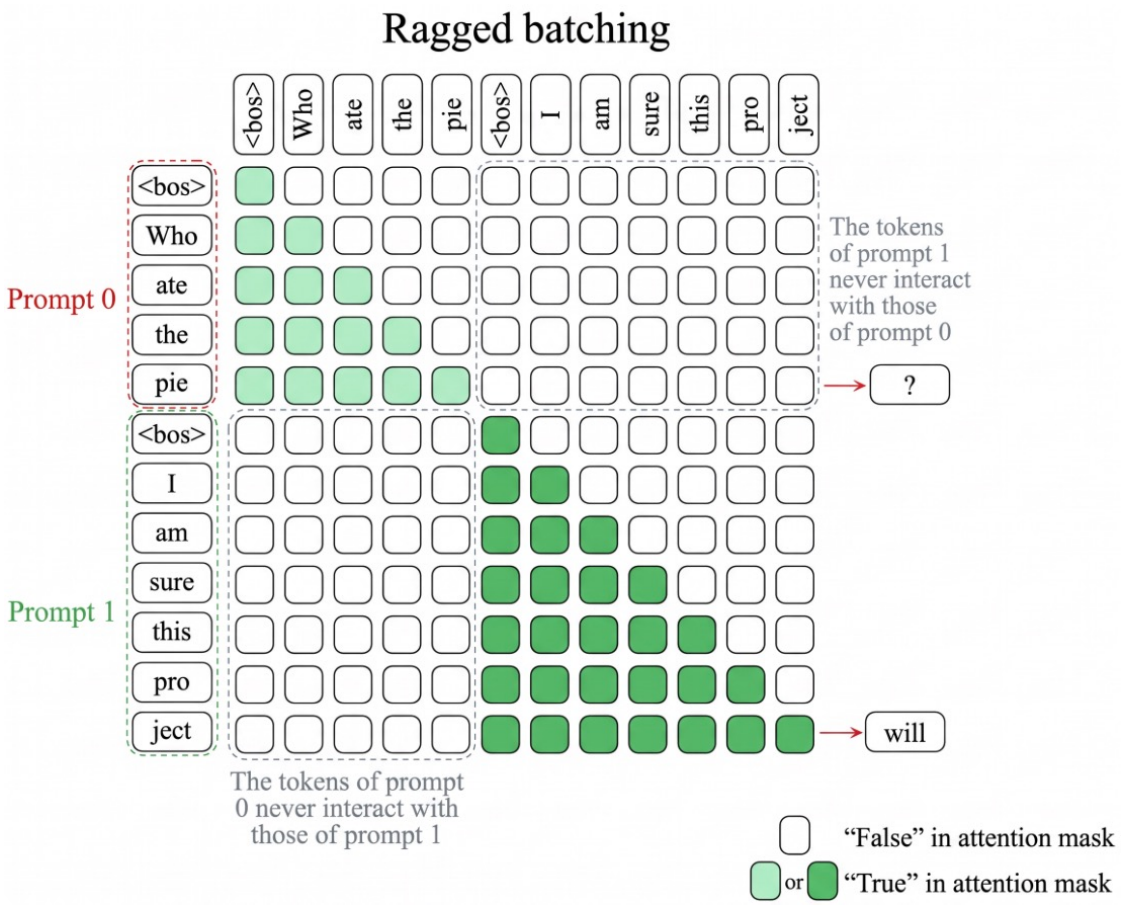
- Concatenating prompts
  - ***Decode*** and ***Prefill*** of different prompts operate simultaneously without bubbles



  - Question: how to concatenate them?

Continuous Batching

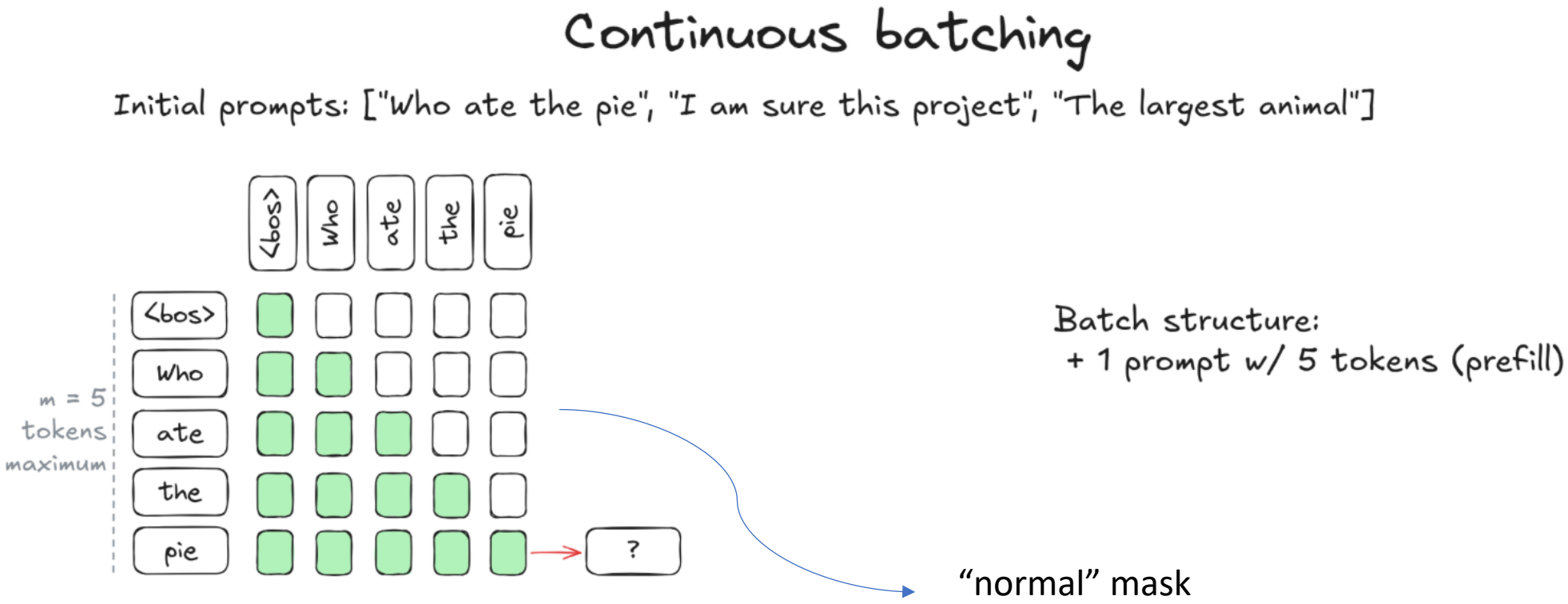
- Continuous batching = ragged batching + dynamic batching
  - Attention scores of tokens belonging to *prompt 0* and *prompt 1* should not be computed together
  - Ragged batching because sequence lengths are 'ragged' or uneven, no need for padding tokens



Source: https://huggingface.co/blog/continuous_batching

Continuous Batching

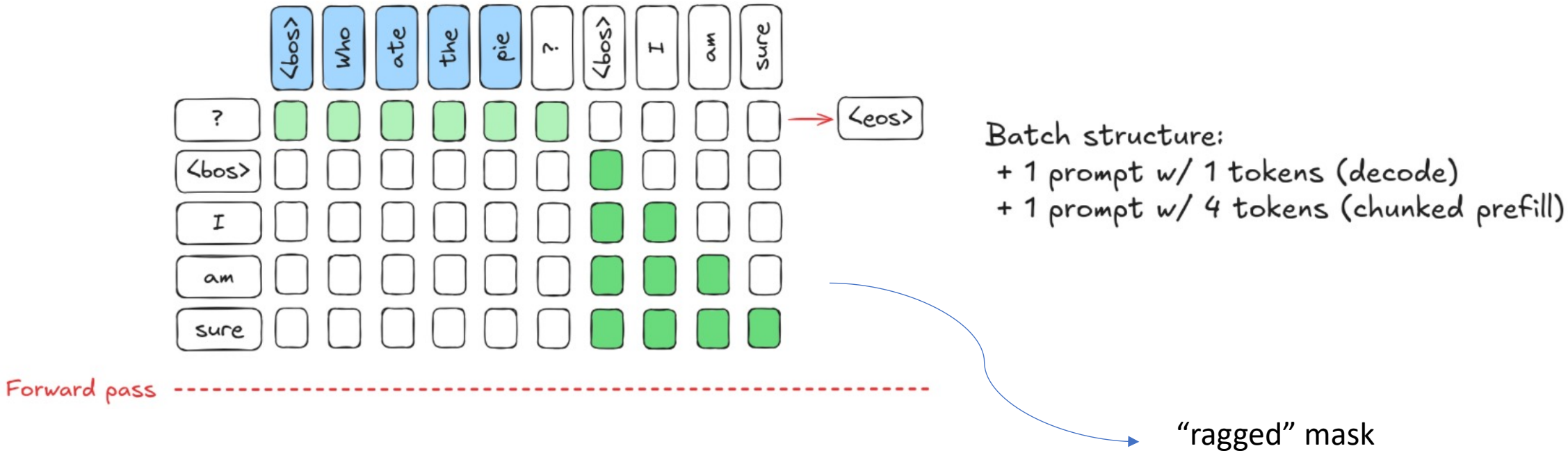
- Continuous batching = ragged batching + dynamic batching



Source: https://huggingface.co/blog/continuous_batching

Continuous Batching

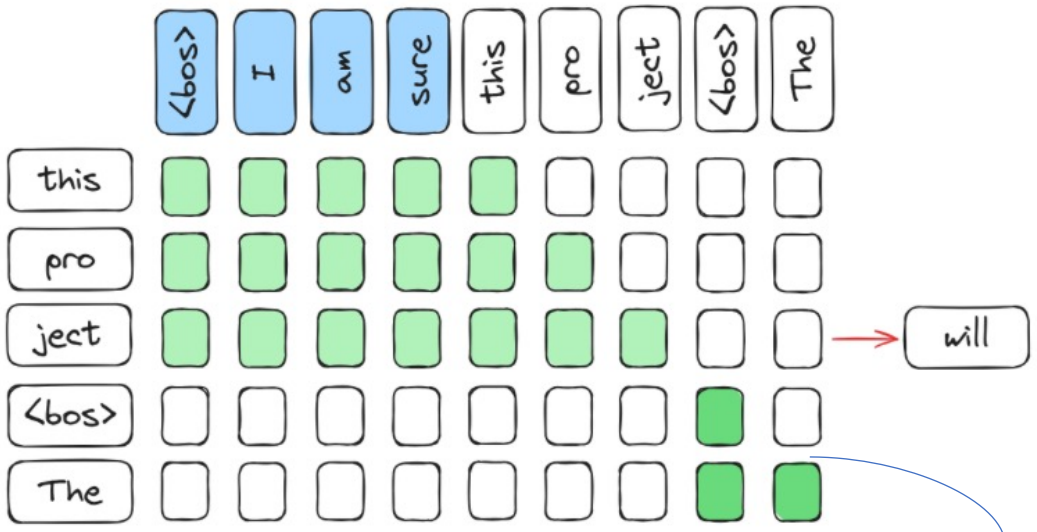
- Continuous batching = ragged batching + dynamic batching
  - After the completion of the first request, the prefill of the second prompt is scheduled
  - Only the **four tokens** in the second prompt is processed concurrently via chunked prefill



Source: https://huggingface.co/blog/continuous_batching

Continuous Batching

- Continuous batching = ragged batching + dynamic batching
  - ’this’, ‘pro’, ‘ject’ are scheduled via chunked prefill and attend to all previous tokens in this prompt
  - Another prompt can also be prefilled



Batch structure:
+ 1 prompt w/ 3 tokens (end of chunked prefill)
+ 1 prompt w/ 2 tokens (chunked prefill)

“ragged” mask

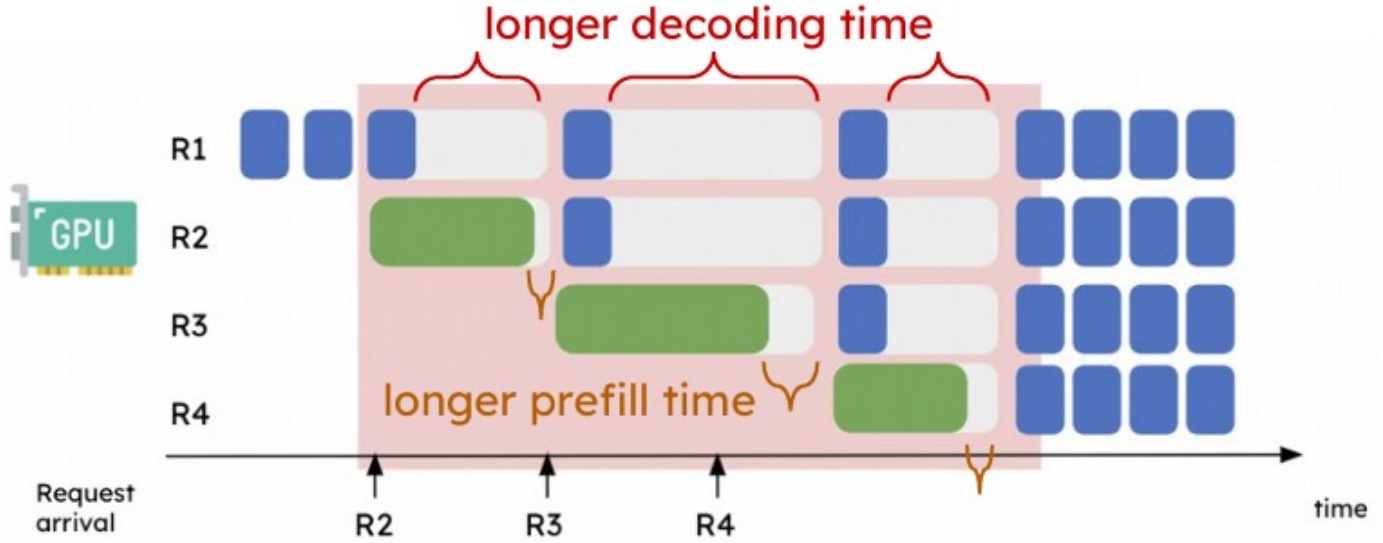
Source: https://huggingface.co/blog/continuous_batching

Inference Optimization

- Overview
- Attention Optimization
- Continuous Batching
- KV Cache Optimization
- Parallel and Distributed Serving
  - Continuous Batching
  - **Prefill-Decode Disaggregation**

Prefill-Decode Disaggregation

- Batching prefill and decoding phases together hurts both TTFT and TPOT
  - Representative methods: ***Orca***, ***Chunked Prefill*** and ***Prefill-Decode disaggregation***

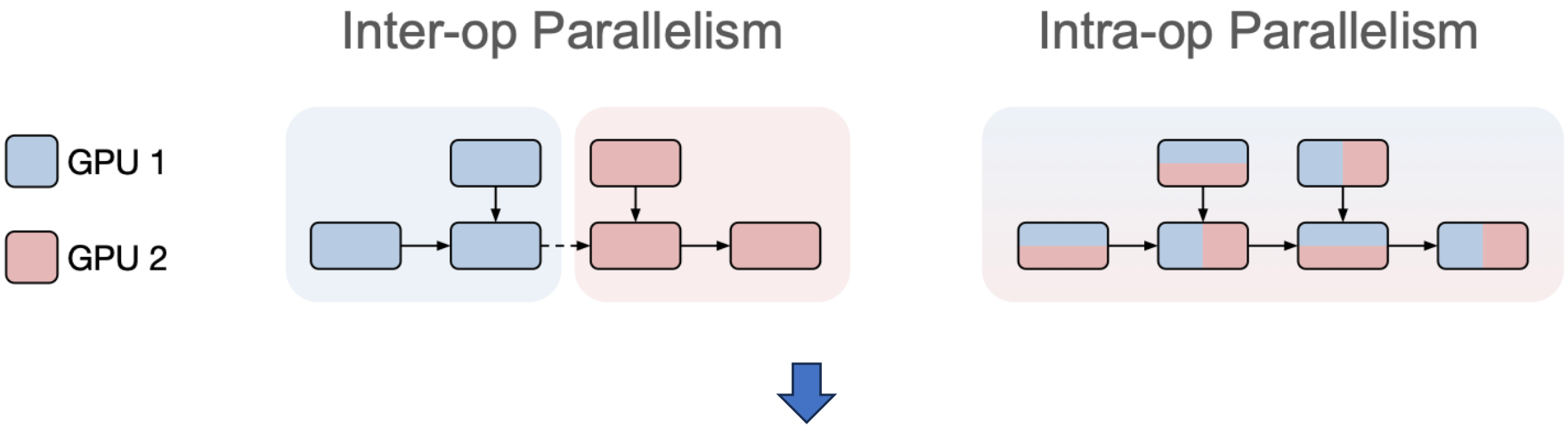


Prefill and decode times are prolonged because of mutual interference

Source: DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Prefill-Decode Disaggregation

- Prefill and decode may need different parallelization schemes



Scale up in the same way?

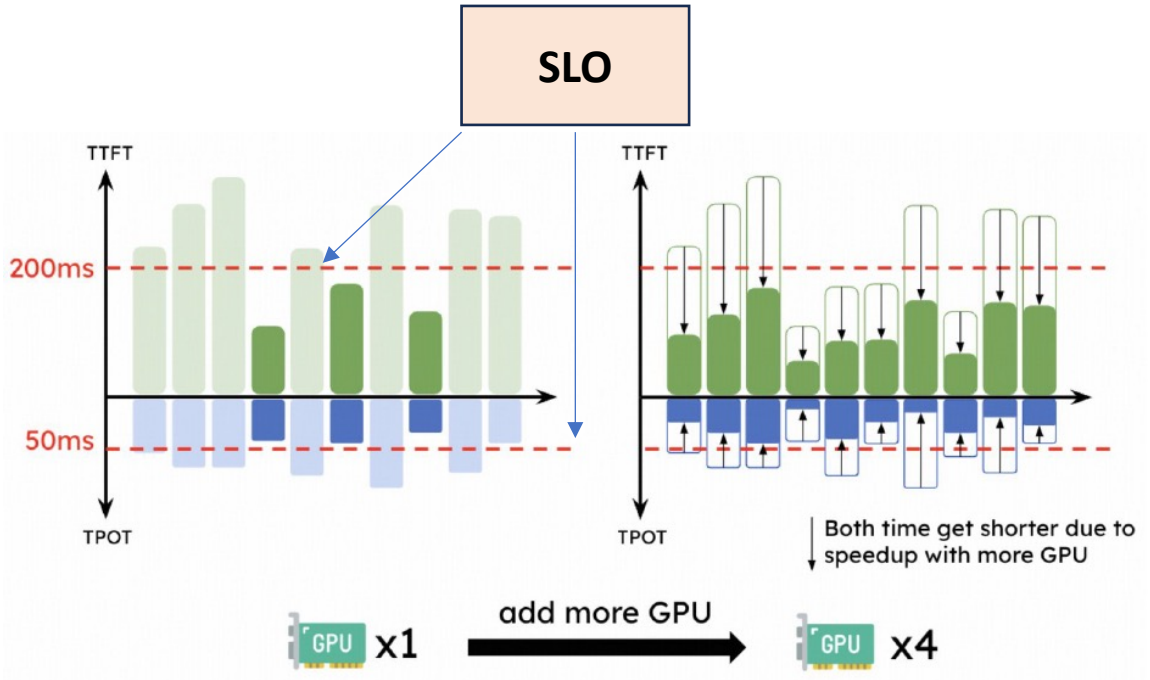
Source: DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Prefill-Decode Disaggregation

- Application-dependent Service Level Objective (SLO)
  - **High throughput != high goodput:** Different apps have various latency requirements

	TTFT	TPOT
Chat	Low (< 1s)	Match read speed (~ 100ms)
Search	Very Low (~ 200ms)	Match read speed (~ 100ms)
Program	Very Low (~ 200ms)	Very Low (~ 50ms)

LLM as a service: diverse SLO

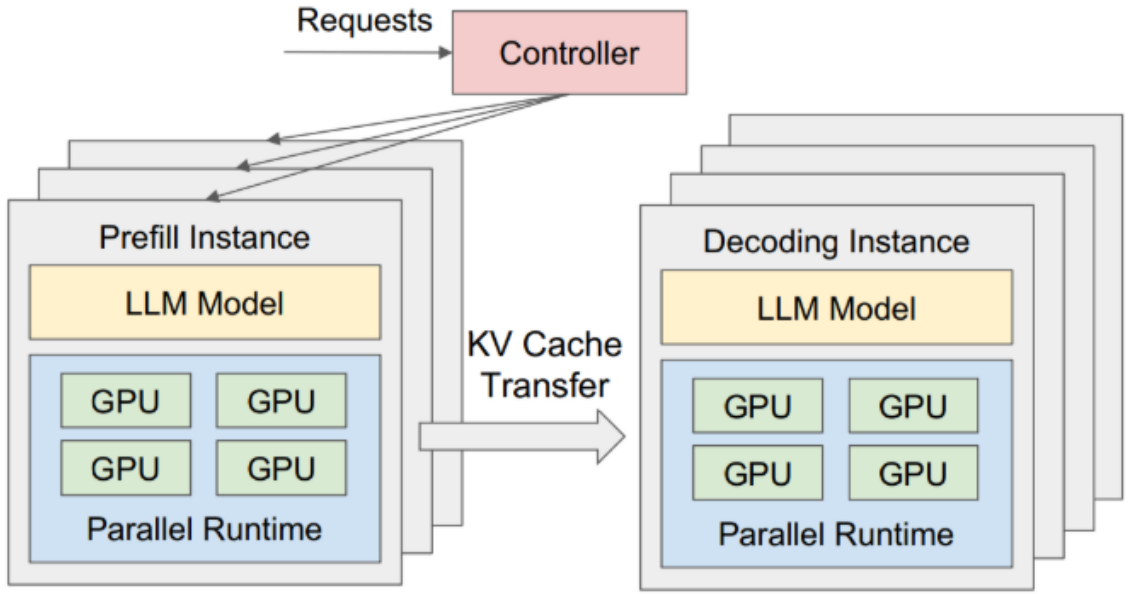


Overprovisioning for LLM serving:
using 4 GPUs increases overall cost

Source: https://hao-ai-lab.github.io/blogs/distserve/

Prefill-Decode Disaggregation

- PD Disaggregation
  - Compute-bound prefill and memory-bound decode on different GPU pools
  - Prefill GPUs ***transfer*** KV cache to Decode GPUs
  - Optimizing prefill and decode separately via the most suitable parallelism

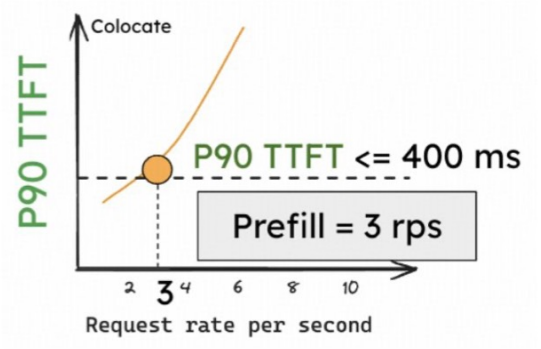


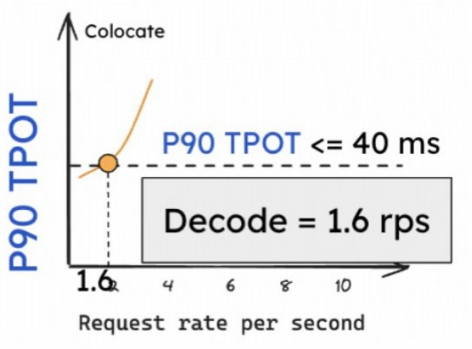
Source: DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving

Prefill-Decode Disaggregation

- Application-dependent SLO

Per-GPU throughput **does not change**

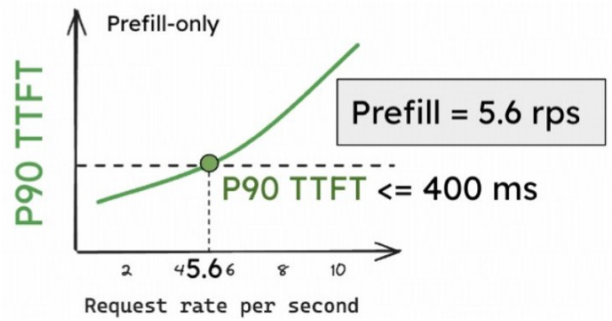


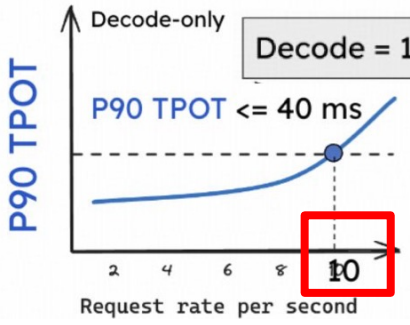


Colocation

GPU

Max System rps
= Min(Prefill, Decode)
= 1.6 rps / GPU





Disaggregation (2P1D)

GPU GPU GPU

Max System rps
= Min (5.6 x **2**, 10) rps / 3 GPU
= 3.3 rps / GPU

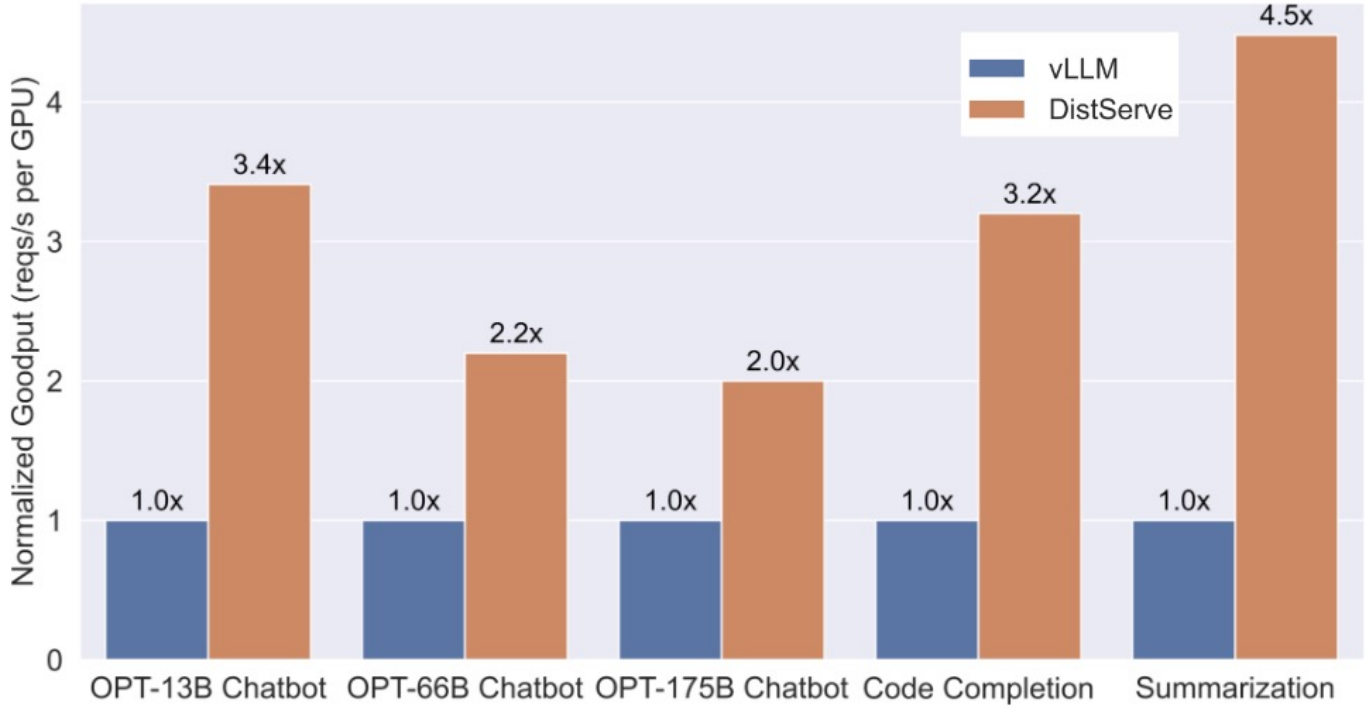
Improved throughput

Doing only one type of job: decoding

Source: DistServe: Disaggregating Prefill and Decoding for Goodput-optimi

Prefill-Decode Disaggregation

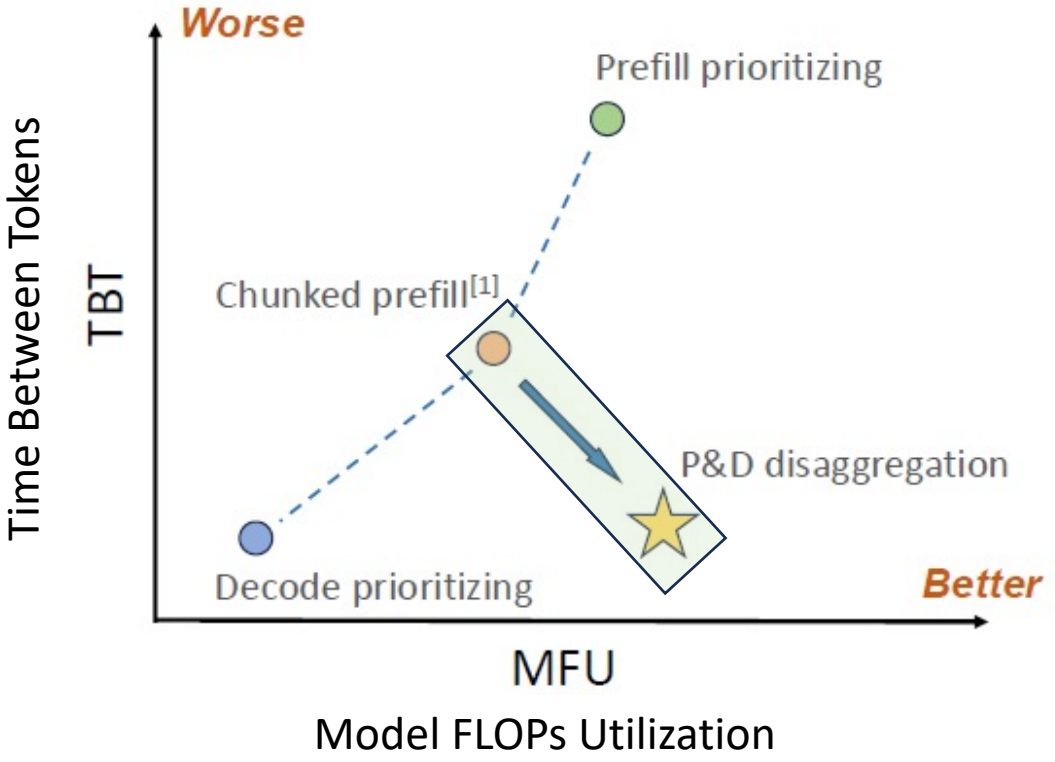
- Experimental Analysis

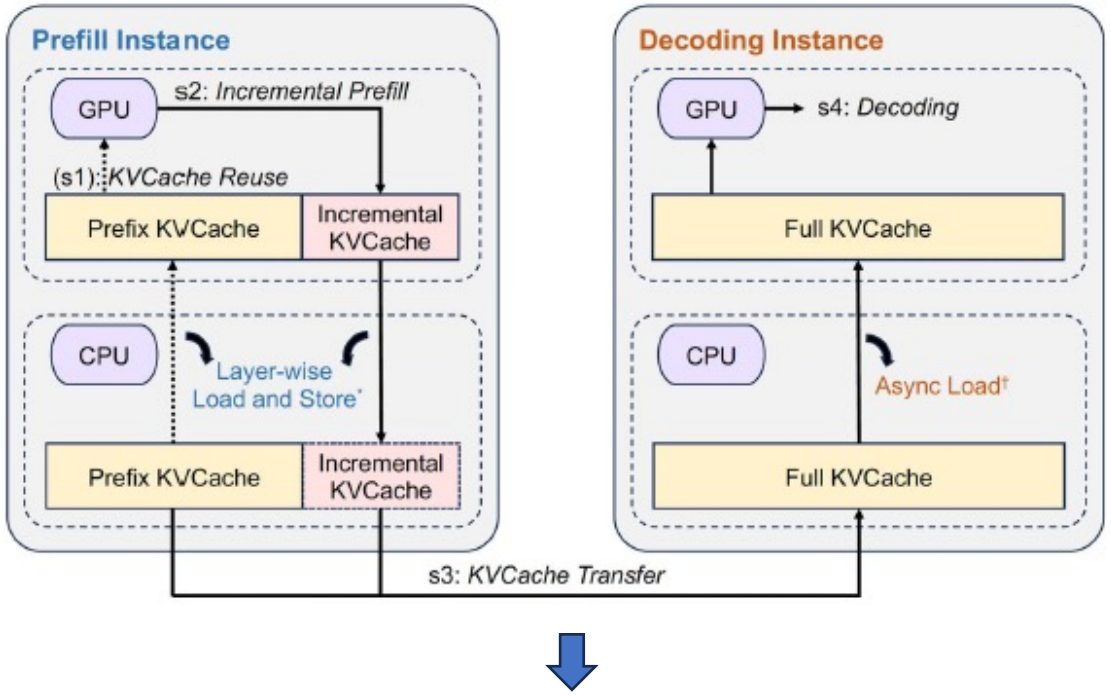


Evaluation of DistServe against vLLM on various datasets

Prefill-Decode Disaggregation

- Chunked Prefill versus PD Disaggregation
  - Best: high MFU low TBT



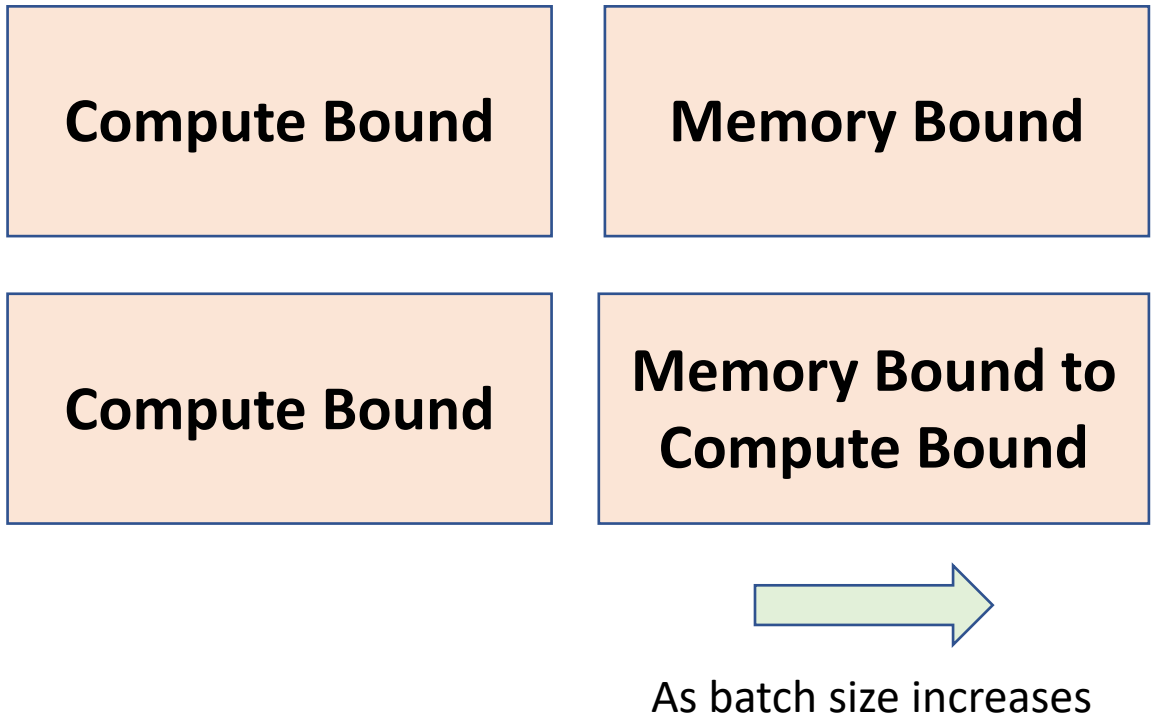


KV Cache transfer on the same layer

Source: Trading More Storage for Less Computation A KV Cache-centric Architecture for Serving LLM Chatbot

Attention-FFN Disaggregation

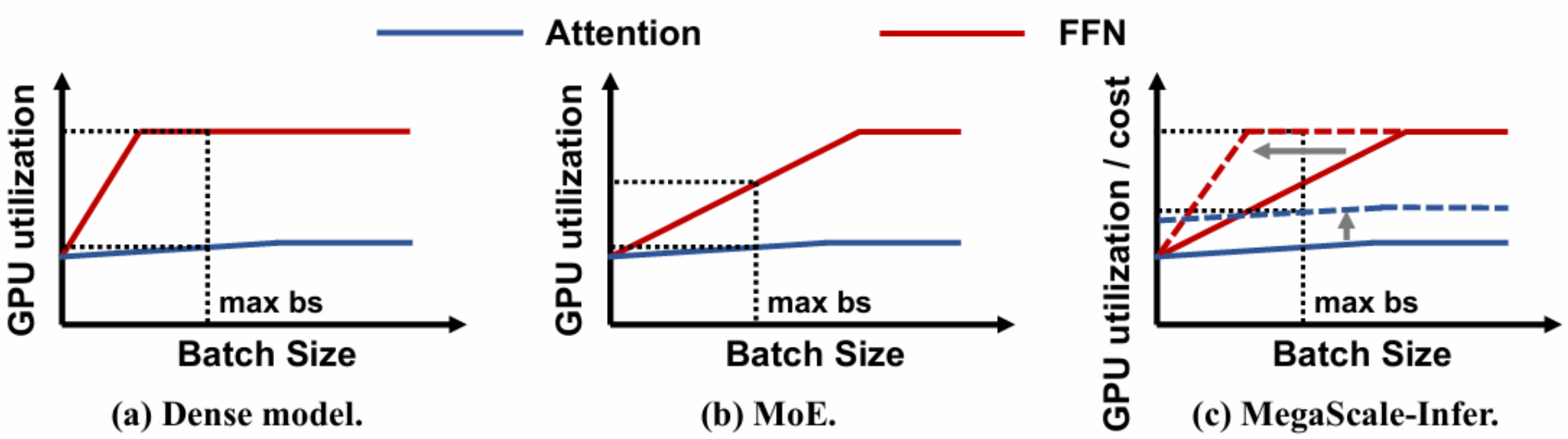
- MoE Inference



Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Attention-FFN Disaggregation

- Motivation of AF Disaggregation
  - Attention is memory intensive and FFN is compute intensive in dense models
  - Sparse MoE models reduce the batch size on an expert

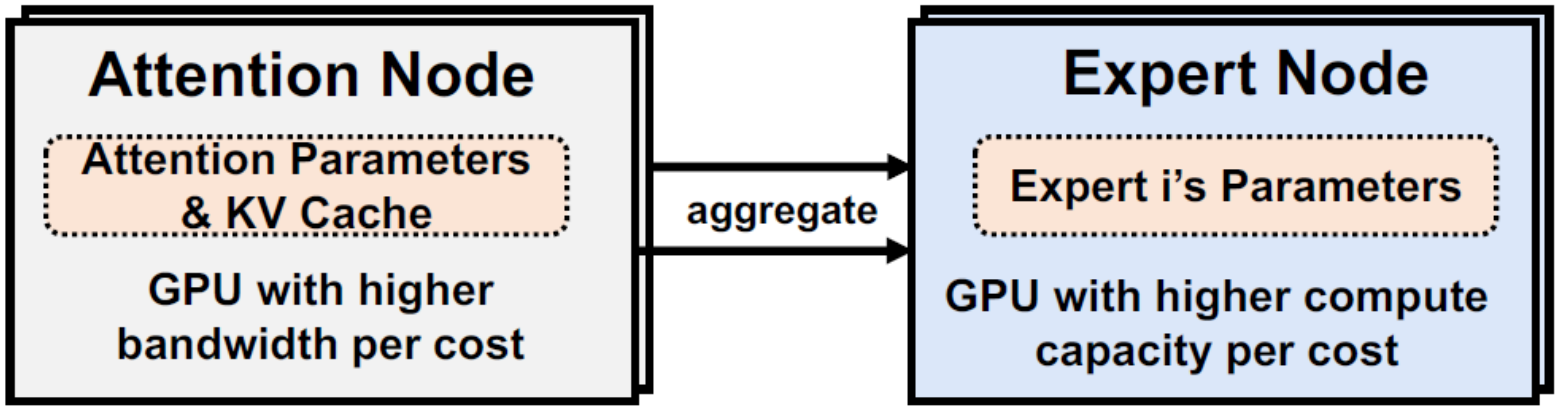


GPU utilization of Attention and FFN vs. batch size in dense model, MoE

Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Attention-FFN Disaggregation

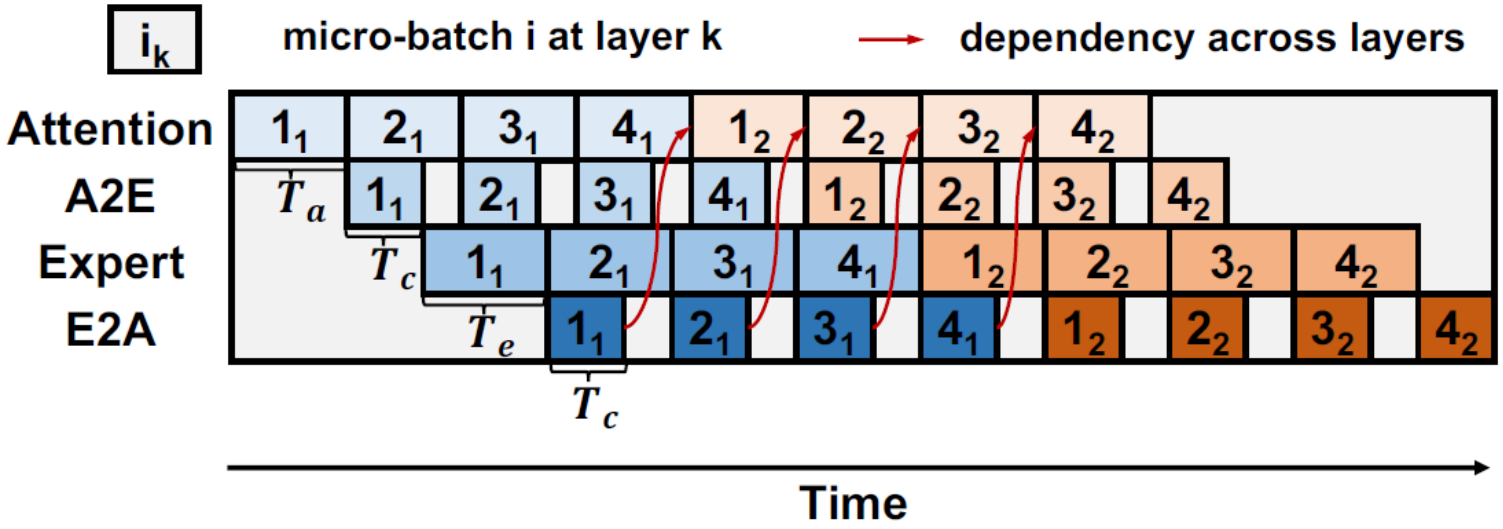
- Key Idea of MegaScale-Infer
  - Independent scaling: feed more requests to FFN
  - Cost-effective, heterogeneous hardware for Attention and FFN



Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Attention-FFN Disaggregation

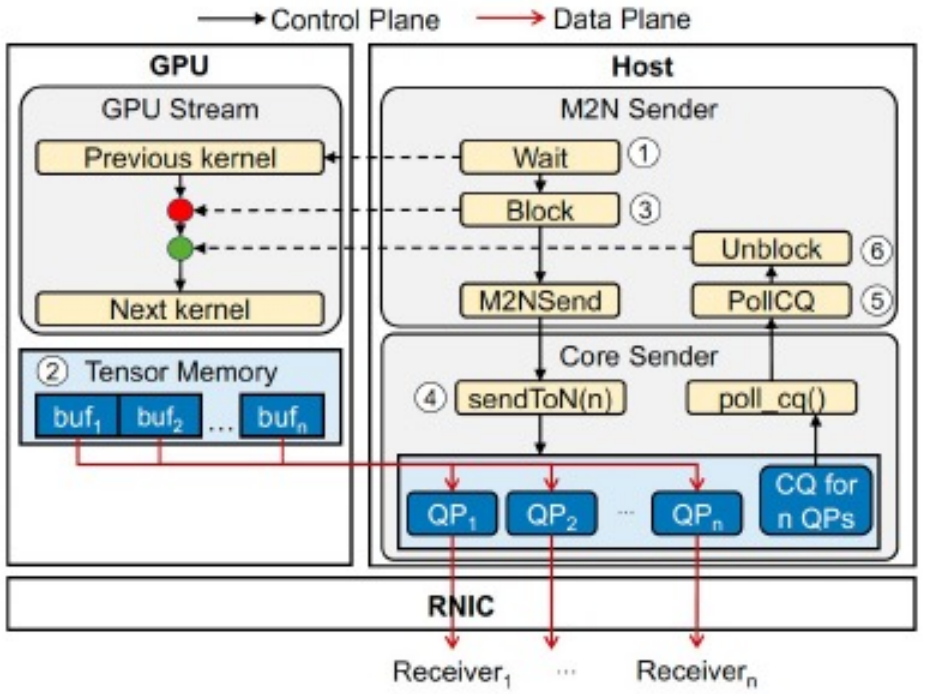
- Challenges
  - Pipeline bubbles: sequential computation of a batch will result in pipeline bubbles, i.e. not all the GPUs being utilized at the same time
  - Ping-Pong pipeline parallelism: dividing a large batch into multiple microbatches



Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Attention-FFN Disaggregation

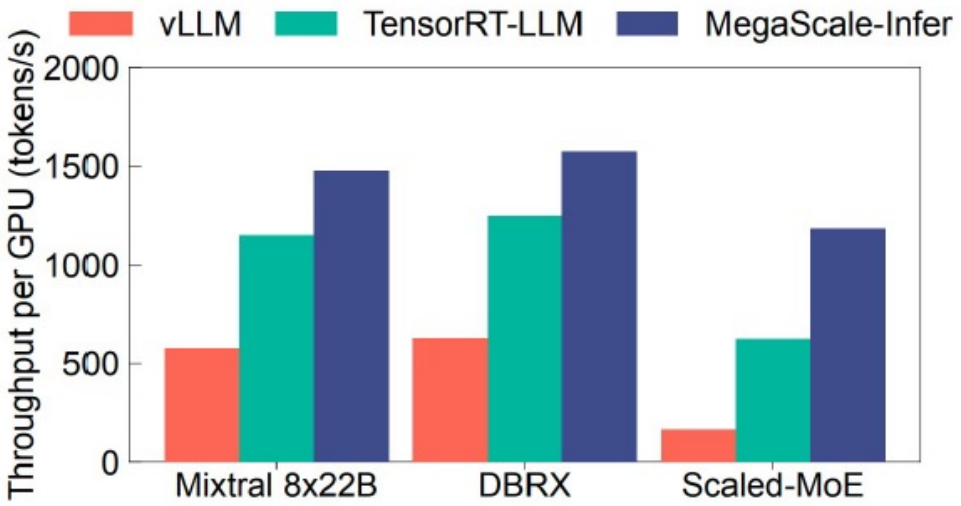
- Challenges
  - Simple idea, yet definitely non-trivial implementation
    - Minimize overhead such as data copies
    - CPU manages send/receive operations
    - GPU Direct over RDMA data transfer



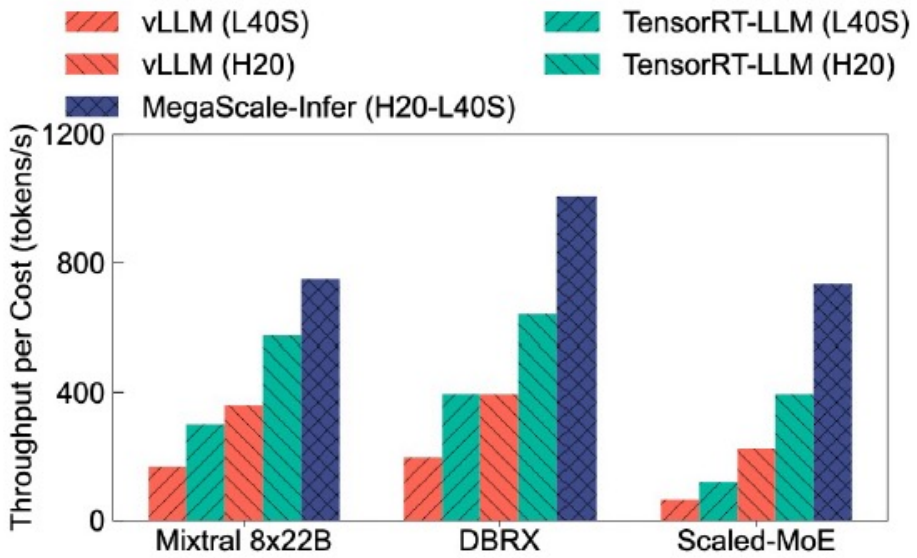
Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Attention-FFN Disaggregation

- Experimental Analysis



Homogeneous clusters



Heterogeneous clusters

Source: MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism

Thanks!

