LLM Inference Optimization

Autumn 2025

Lecturer: Yuedong (Steven) Xu

Shenzhen Loop Area Institute

yuedongxu@slai.edu.cn

Fudan University

ydxu@fudan.edu.cn

Disclaimer

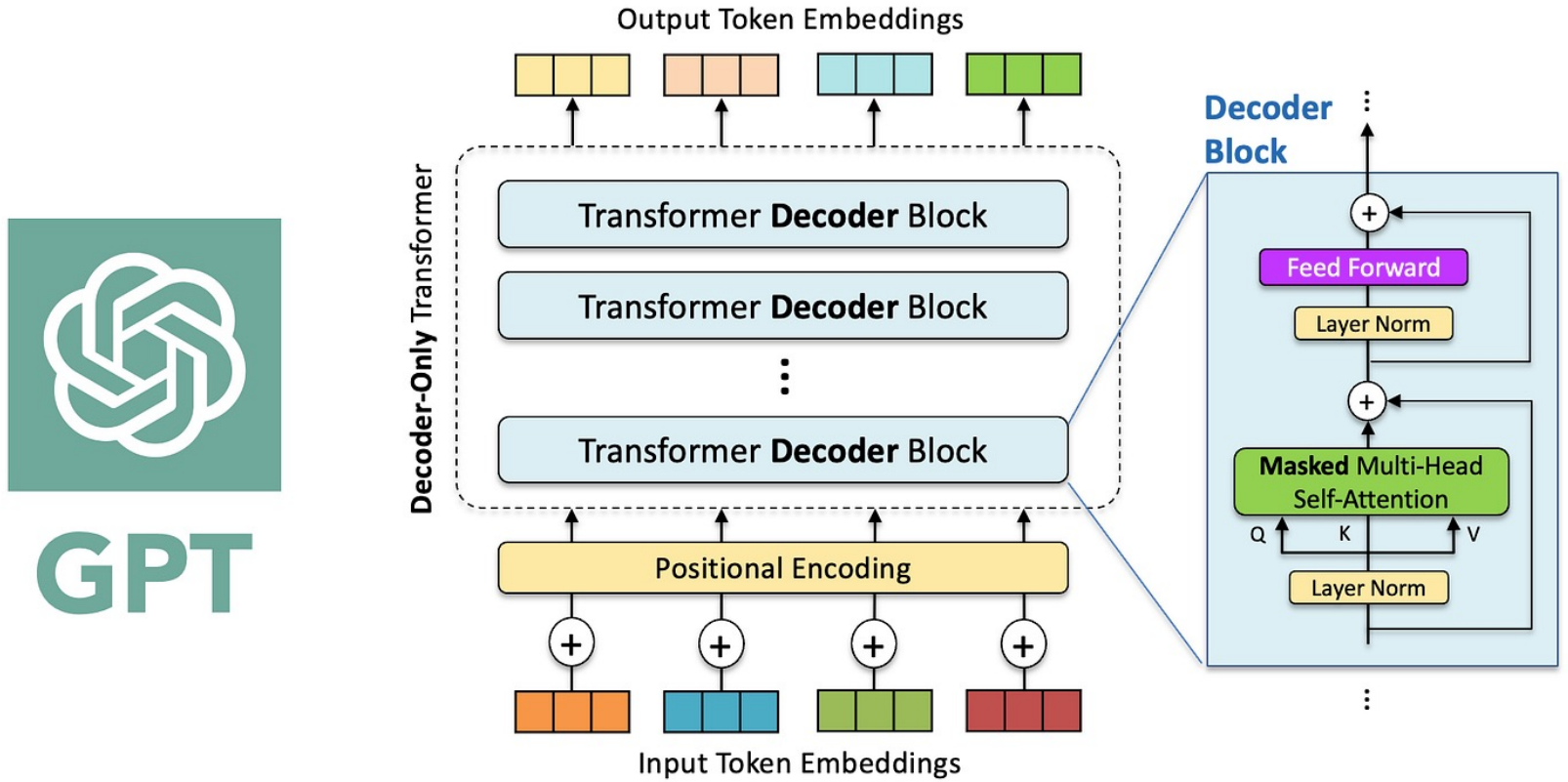
Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blogposts, research talks, tutorial videos, and other materials shared by the research community. Sometimes external animations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

Inference Optimization: Outline

- Overview
- Attention Optimization
- Continuous Batching
- KV Cache Optimization
- Speculative Decoding
- Distributed Serving

Overview

- Decoder-only Transformer

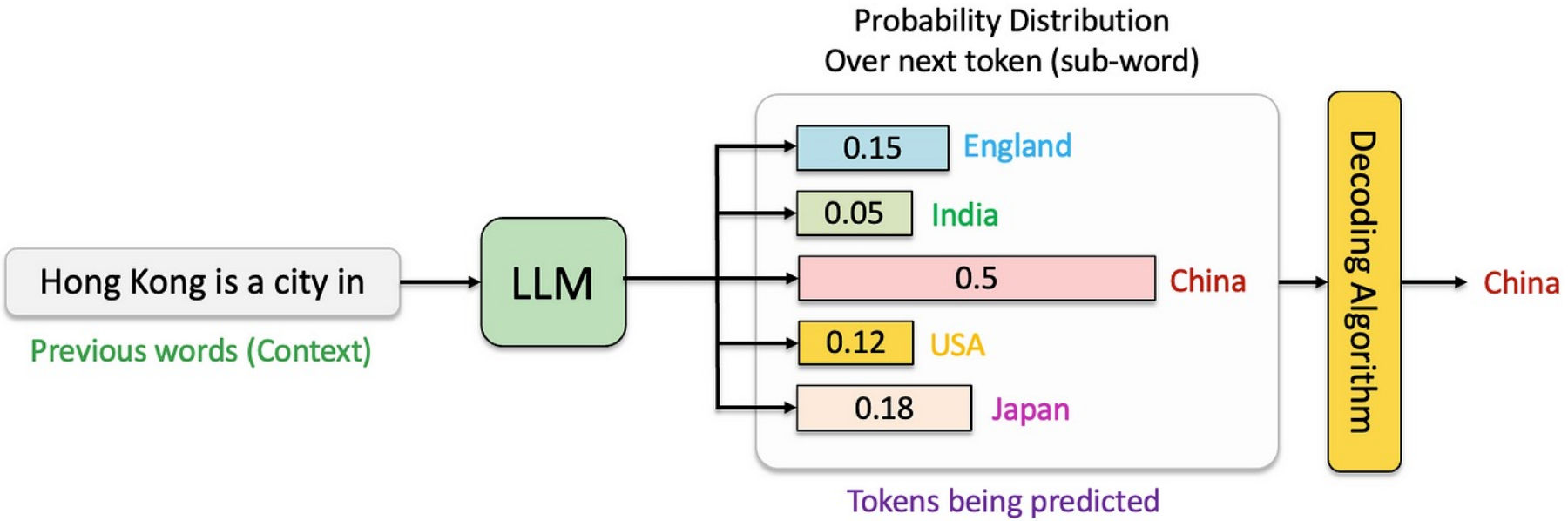


GPT (Generative Pre-trained Transformer) is the first decoder-only Transformer model

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 is employed to select the actual output token

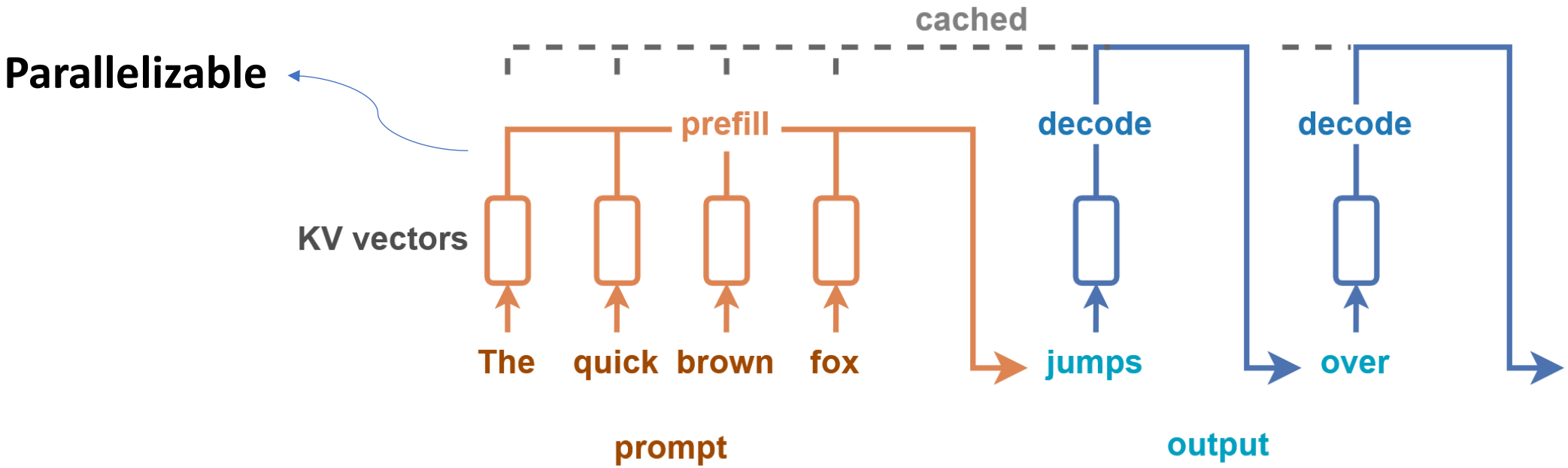


Next-token prediction, i.e. generating output tokens one by one

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

Overview

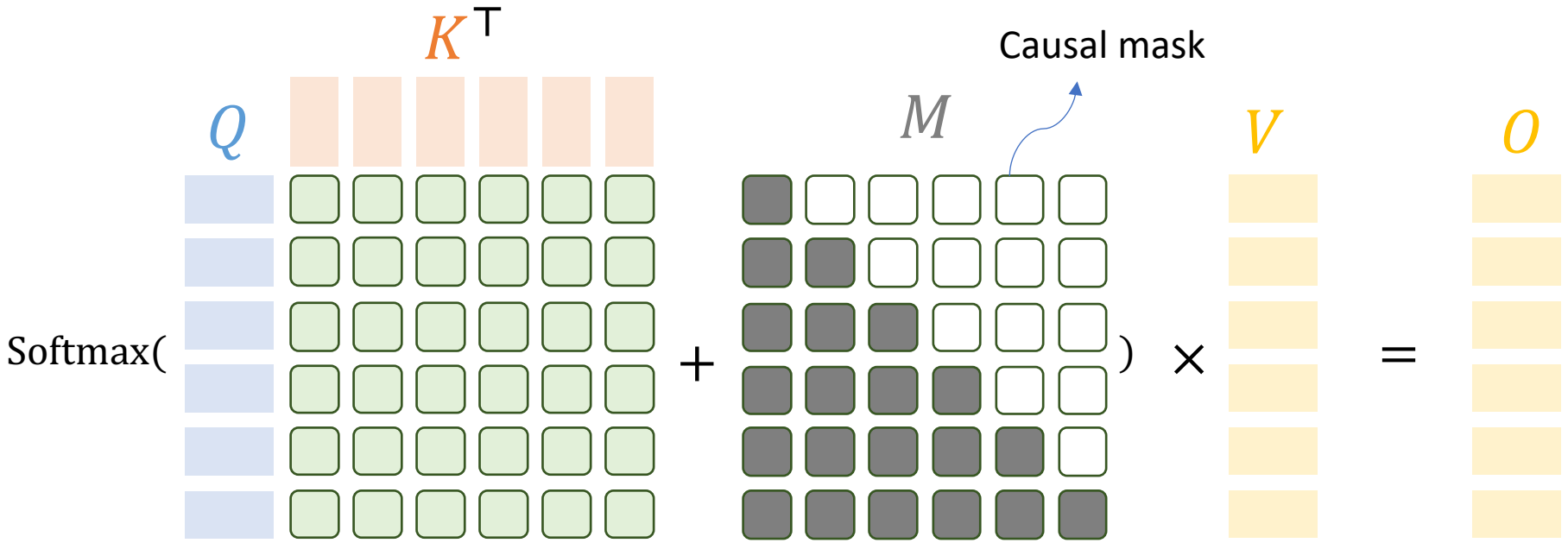
- Autoregressive Decoding
  - **“Prefill”** refers to the **initial parallel computation phase** where the model processes the entire input prompt for subsequent token-by-token generation



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

Overview

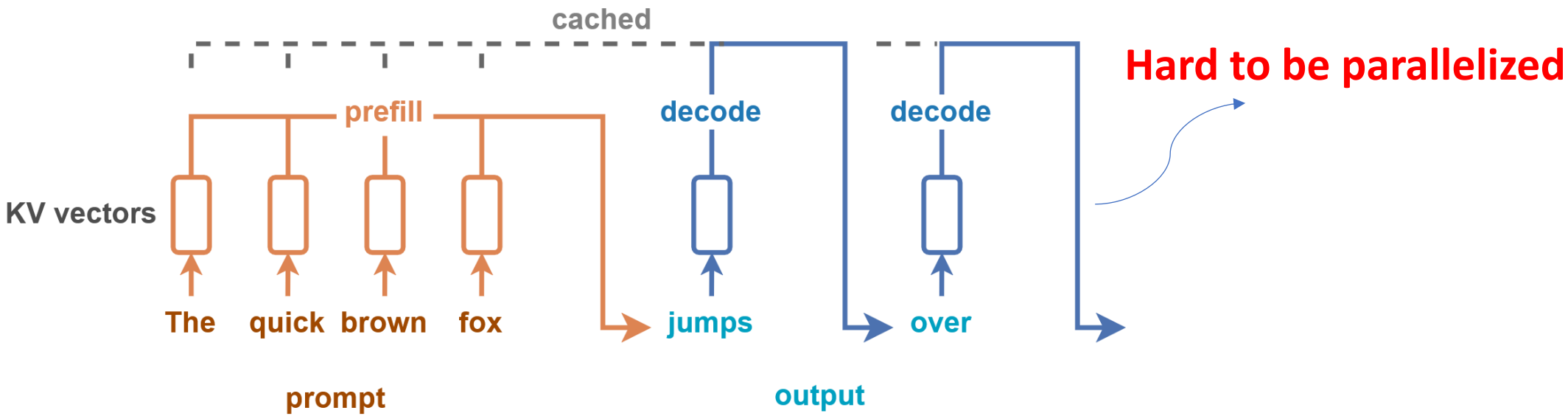
- Prefill: highly parallelizable
  - A small batch size can ”saturate” GPU computation
    - Parallelization over batch size, header size, sequence length and thread-block tiling



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

Overview

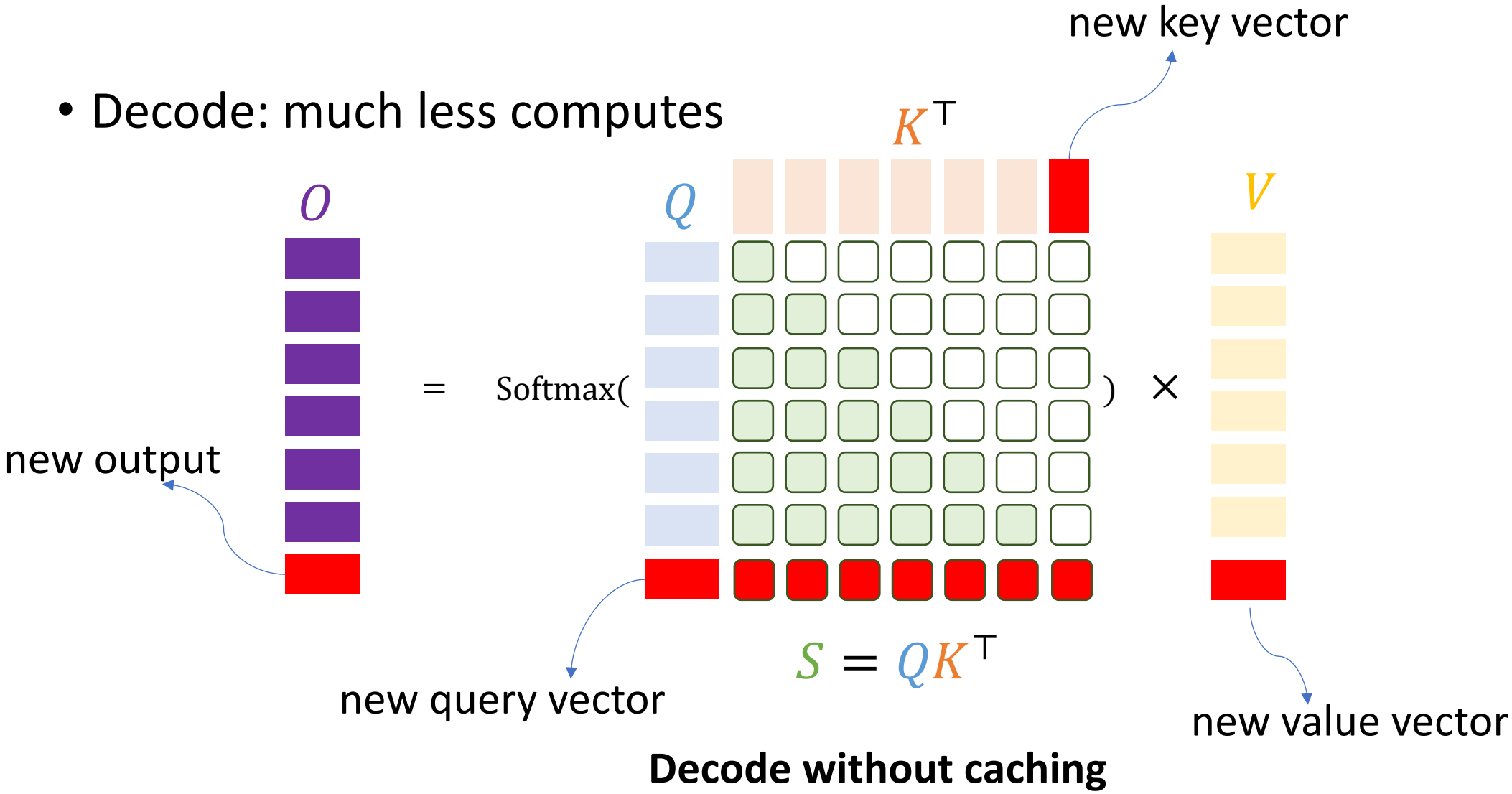
- Autoregressive Decoding
  - **“Decode”** refers to the **iterative process of generating output tokens** one at a time, where each new token is predicted based on the input prompt and all previously generated tokens



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

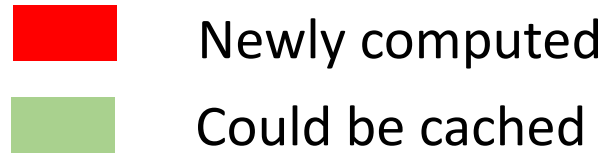
Overview

- Decode: much less computes

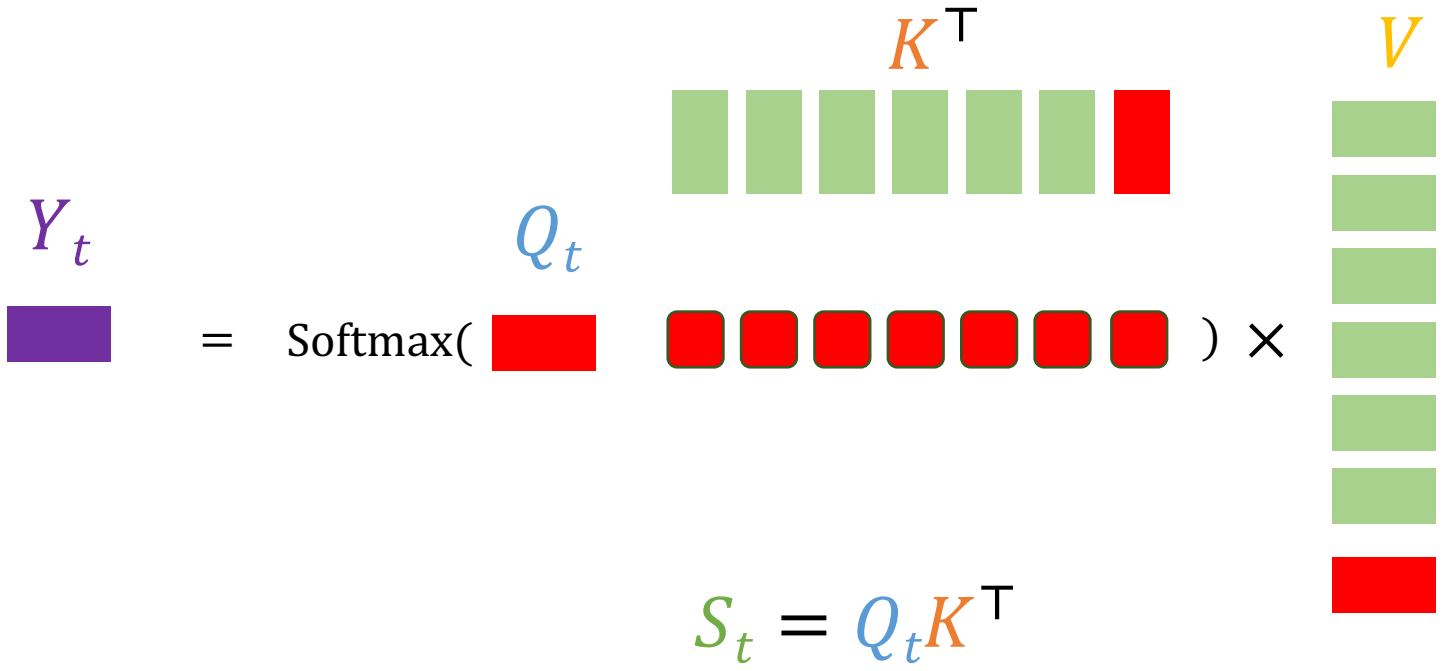


Decode without caching

Overview



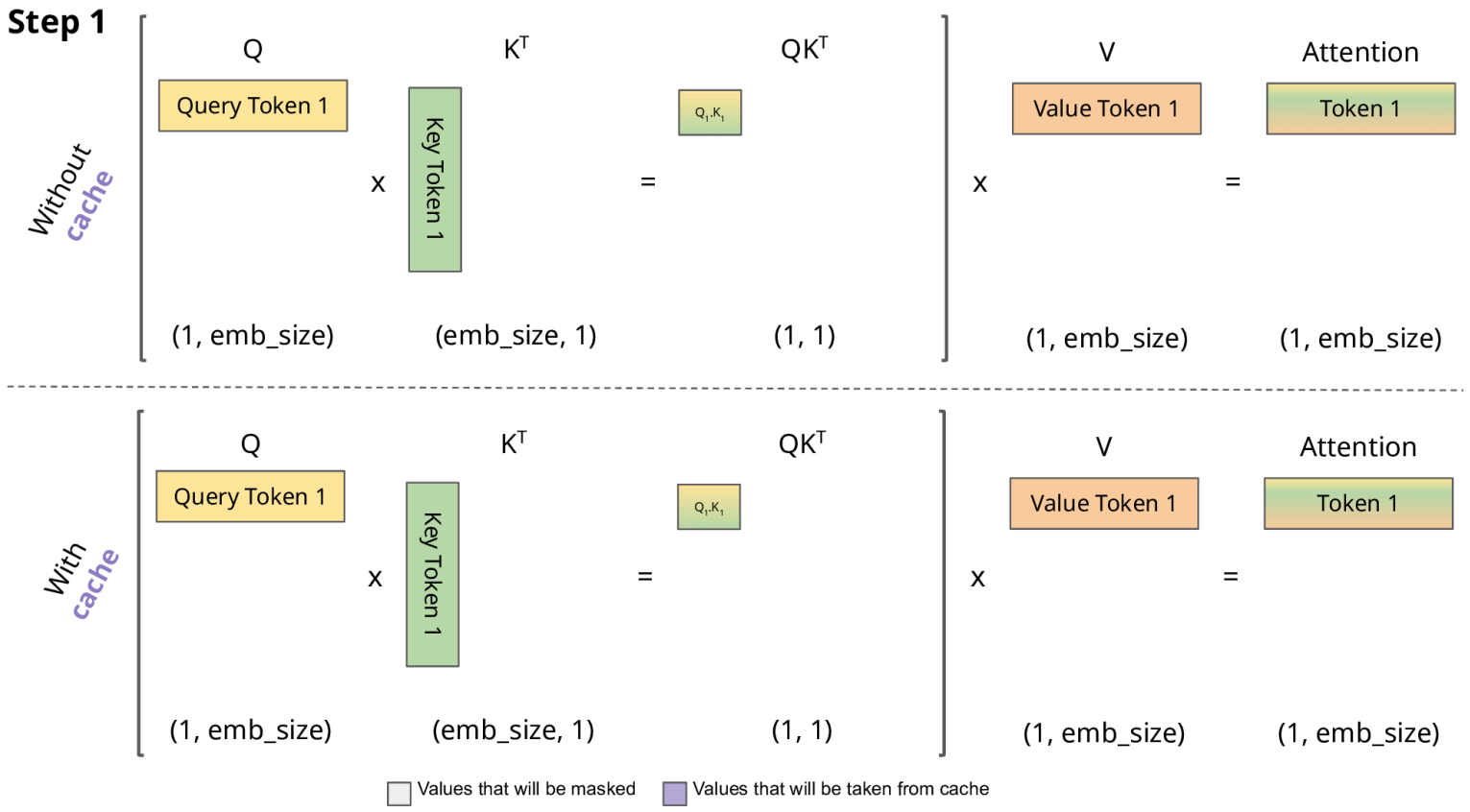
- Decode: much less computes
  - Why caching Key and Value, other than Query



Decode Compute output token one by one

Overview

- An animation



Overview

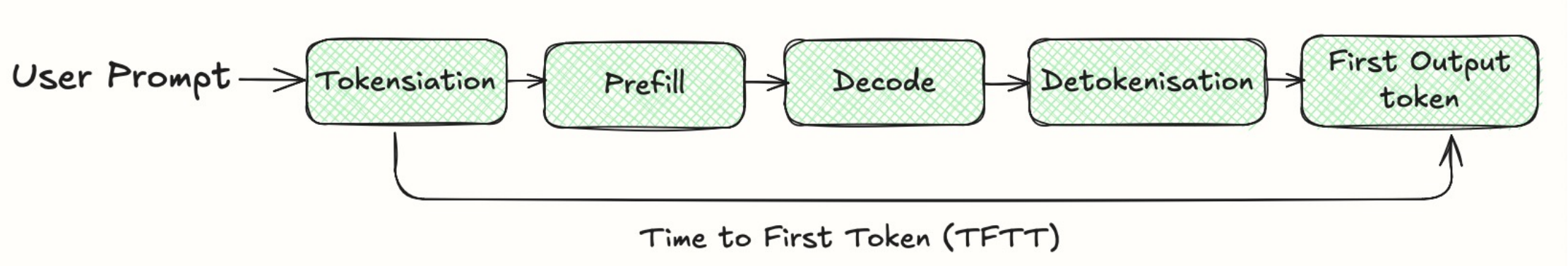
- Why Caching KV?
  - Reducing redundant computations for KEYs and VALUEs
  - Increasing memory consumption

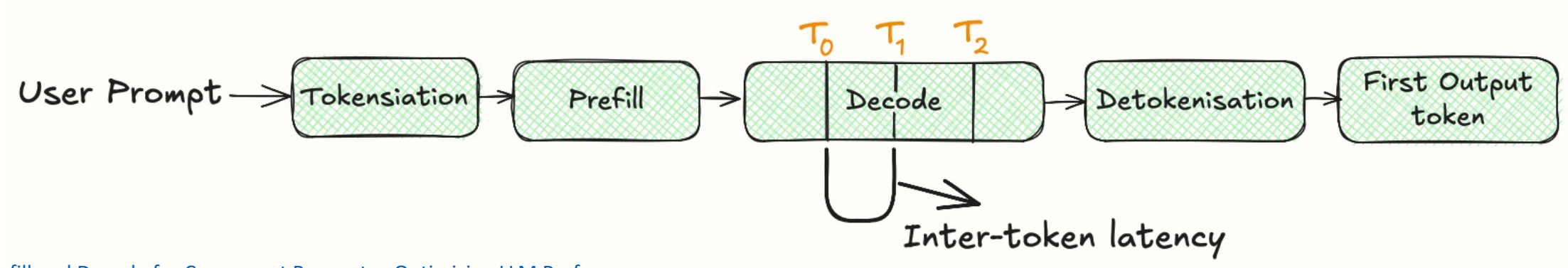
$$\underset{\text{K/V}}{2} \times \underset{\text{Float16}}{2} \times \underset{\text{Sequence length}}{8096} \times \underset{\text{\# of layers}}{80} \times \underset{\text{\# of heads}}{64} \times \underset{\text{dimension}}{64}$$

~10.6 GB!

Overview

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



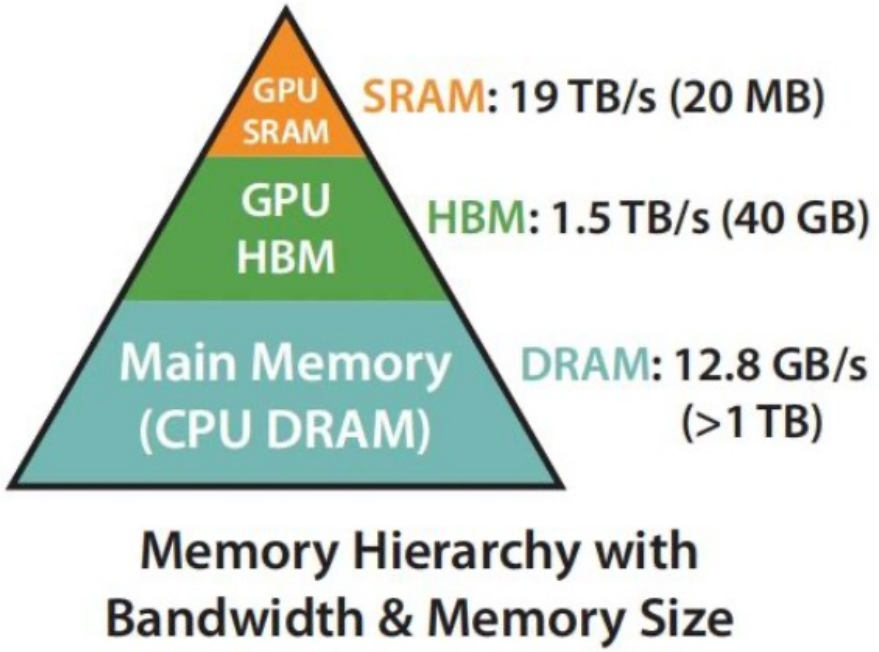


Source: [Prefill and Decode for Concurrent Requests - Optimizing LLM Performance]

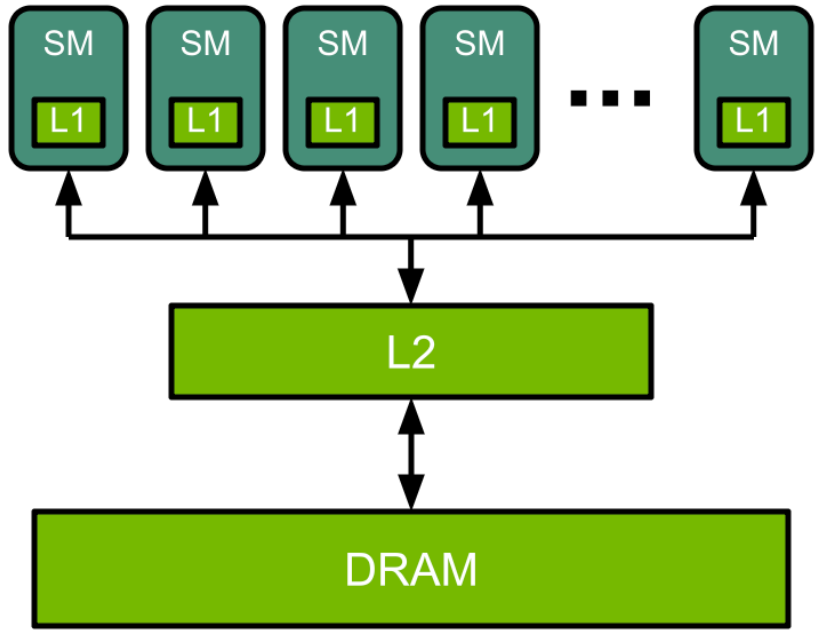
Overview

Insufficient to store data at SRAM: load from HBM to SRAM and write back to HBM

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



A macroscope view of I/O tradeoff (V100)

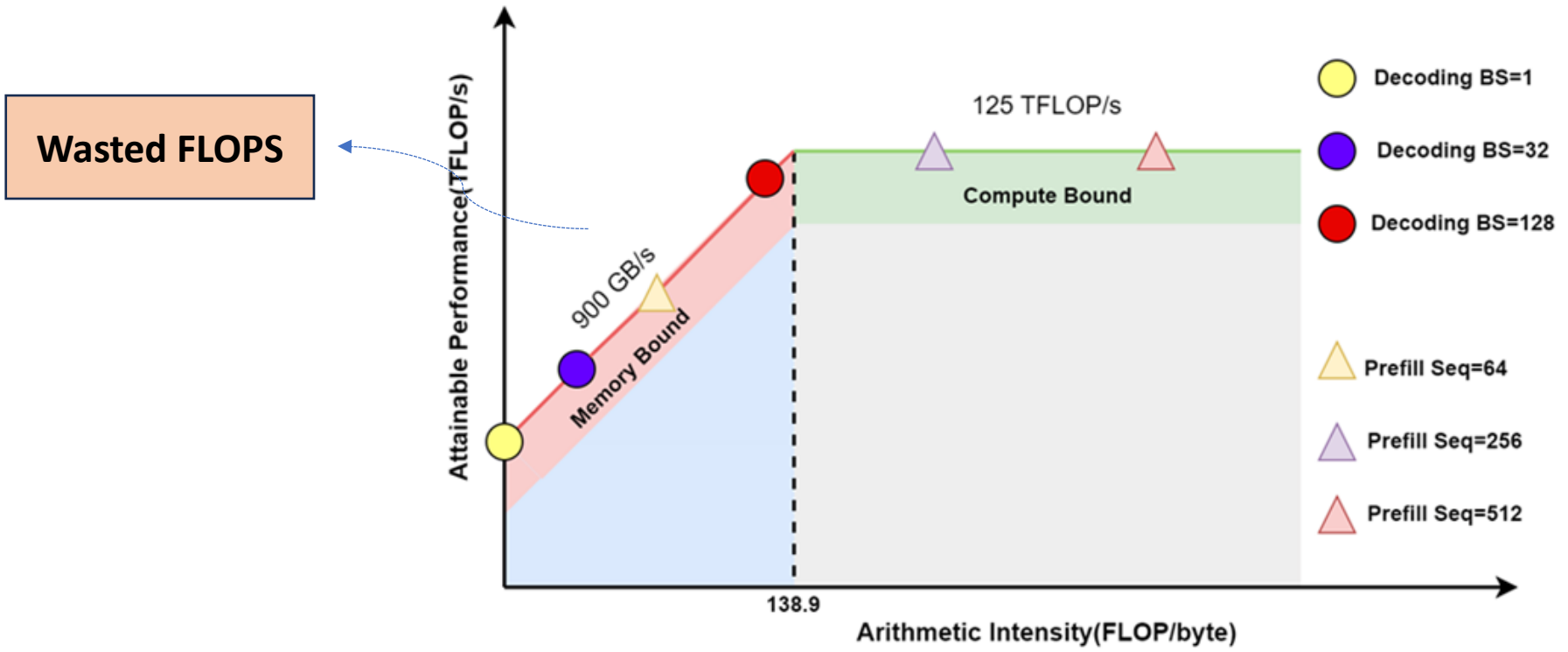


L1 instruction cache: 192KB per SM * 108 SM ~20MB
Data flow: DRAM/L2 Cache to/from L1 Cache, much slower than computing

Source: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html

Overview

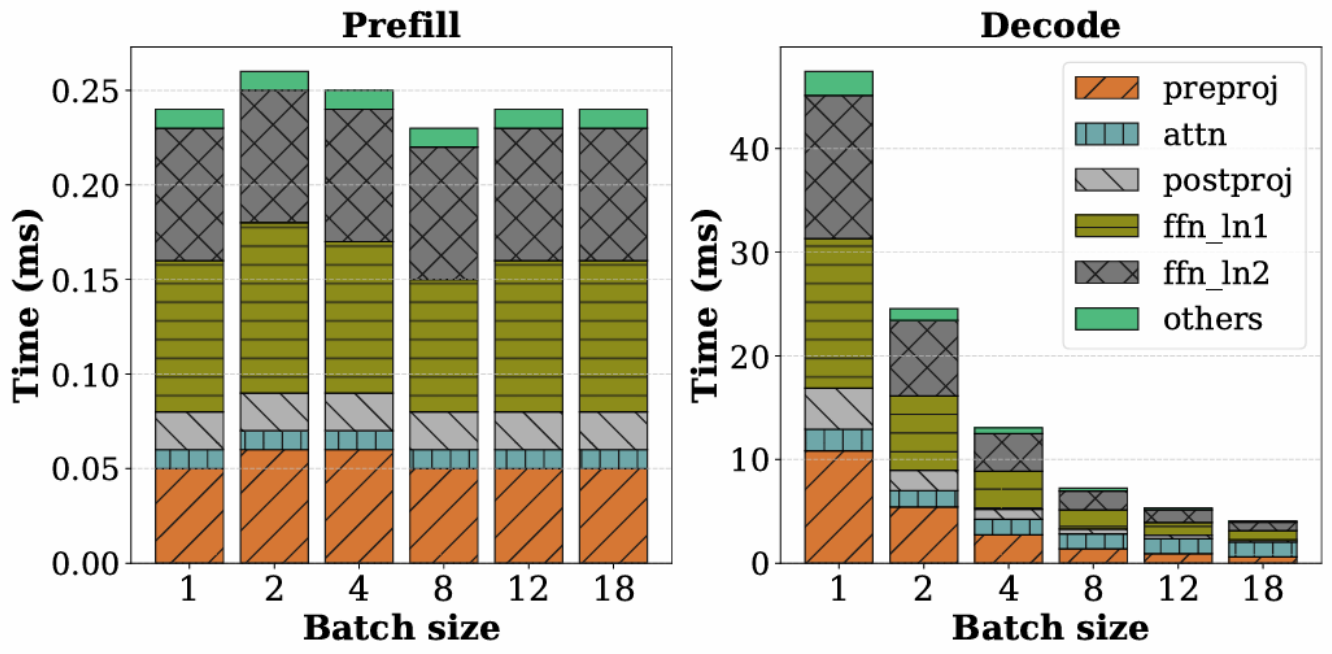
- Prefill and Decode
  - Prefill: Compute intensive with GEMM, Decode: memory I/O intensive with GEMV



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

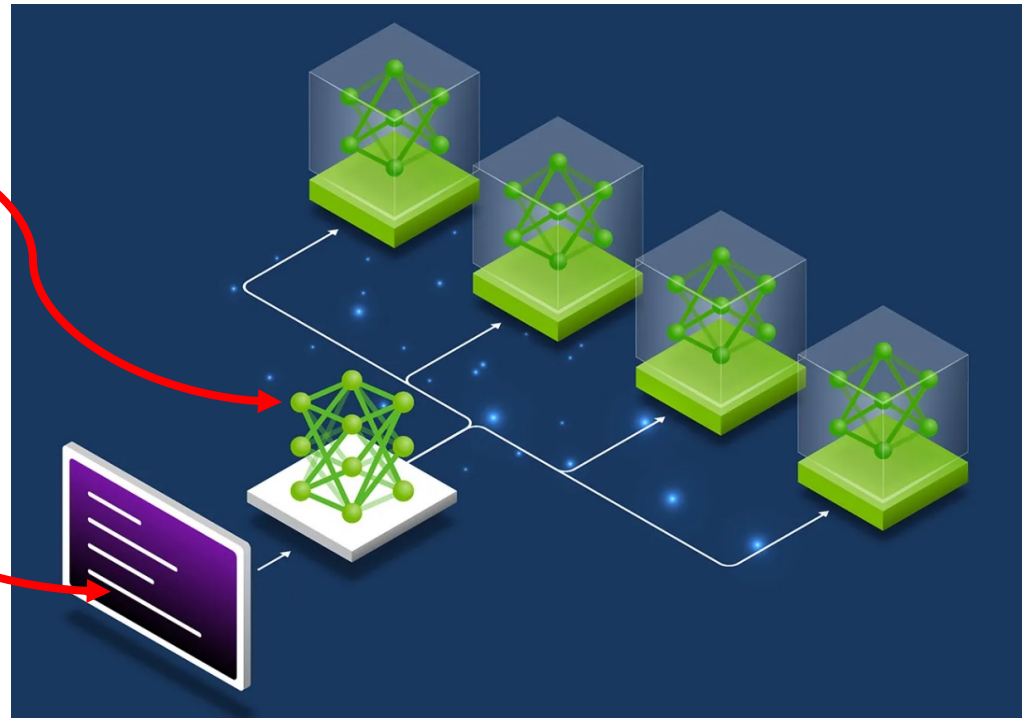
Overview

- Very Large-scale LLM Inference

deepseek

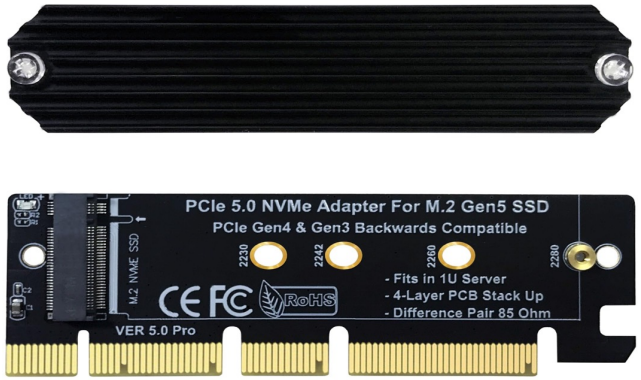
Full model size with 256 experts

Long context and high request loads

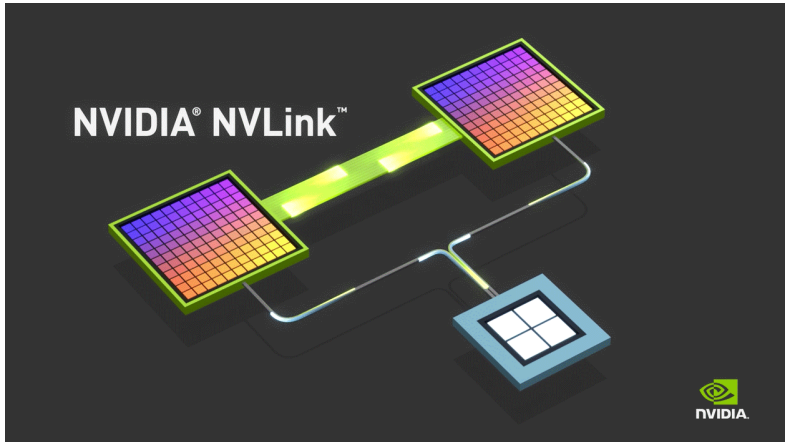


Overview

- Versatile communication medium
  - Transmission delay emerges



PCIe Link
e.g. CPU – GPU with
up to 128GB/s bandwidth



NVLink
e.g. GPU – GPU
in total 1.8TB/s bandwdith

RoCE
e.g. Machine – Machine
up to 800 Gbps

Overview

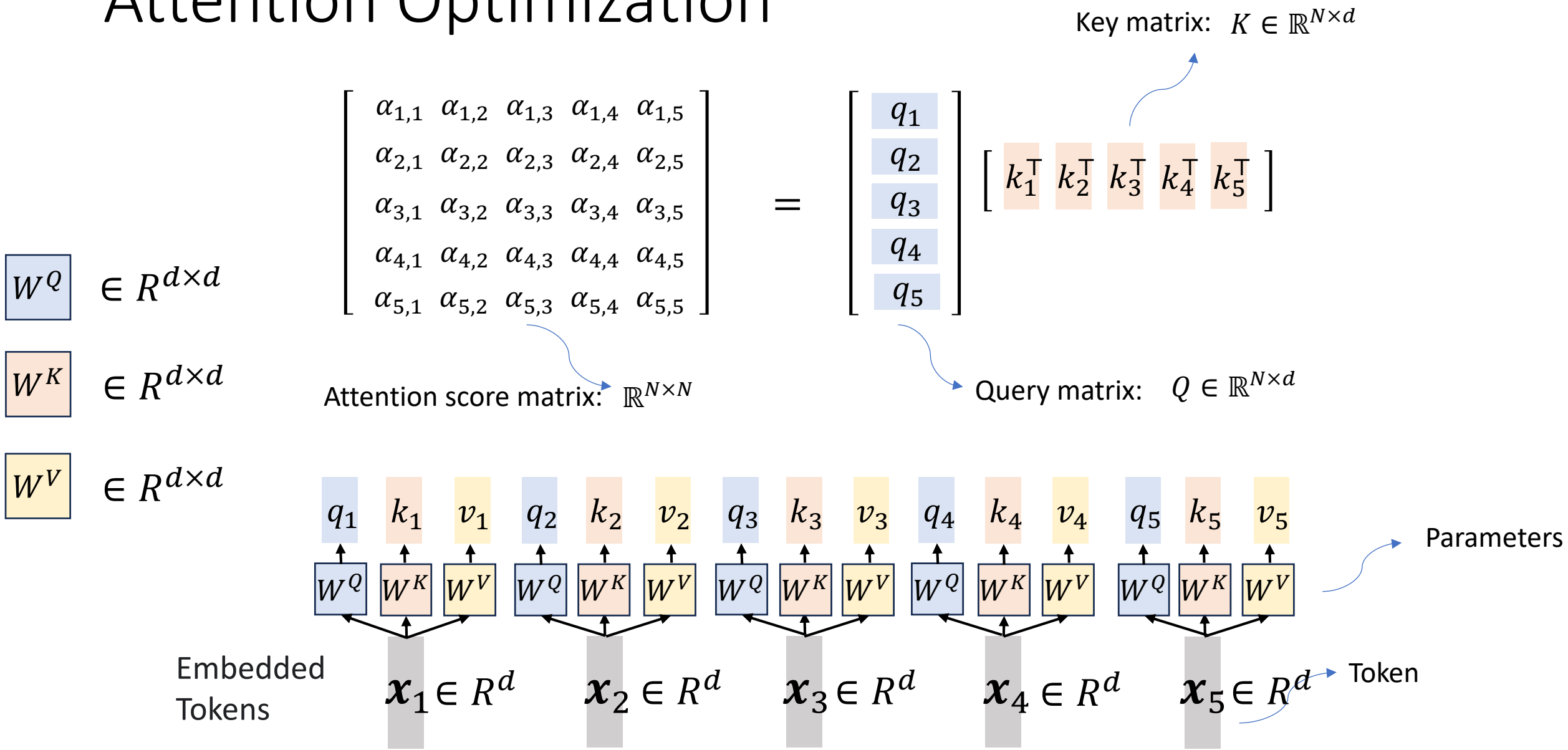
- Challenges of LLM Inference
  - New token generation paradigm
    - **Prefill** and **Decode**
  - Hierarchical memory
    - **Fast** I/O small size versus **slow** I/O large size
  - Heterogeneous bandwidth
    - **High** intra-machine bandwidth versus **low** inter-machine bandwidth
- To emphasize
  - 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: Outline

- Overview
- Attention Optimization
  - **Sparse Attention**
  - Linear Attention
  - Flash Attention
  - Continuous Batching
- Continuous Batching
- KV Cache Optimization
- Speculative Decoding
- Distributed Serving

Attention Optimization



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

Attention Optimization

- Real-world operations
  - Transferring both matrices from global memory to shared memory for computing, and write the result back to global memory progressively
- Complexity
  - 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

Attention Optimization

- Real-world operations
  - Transferring both matrices from global memory to shared memory for

Computation, I/O and storage should all be optimized!

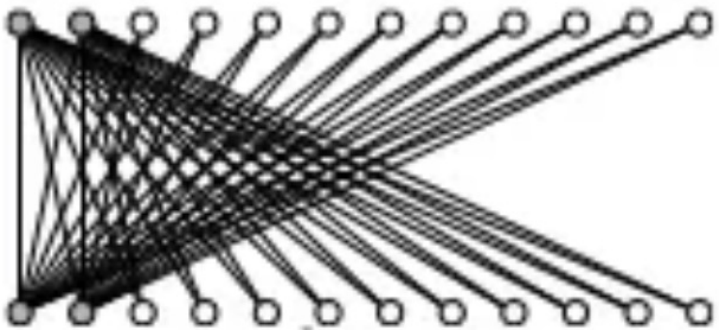
  - Communication: $O(N^2)$
    - Considering bidirectional I/O read/write → communication time $T_{I/O} = \frac{2*(N^2+2Nd)}{B}$
  - Global storage: $O(N^2)$

Sparse Attention

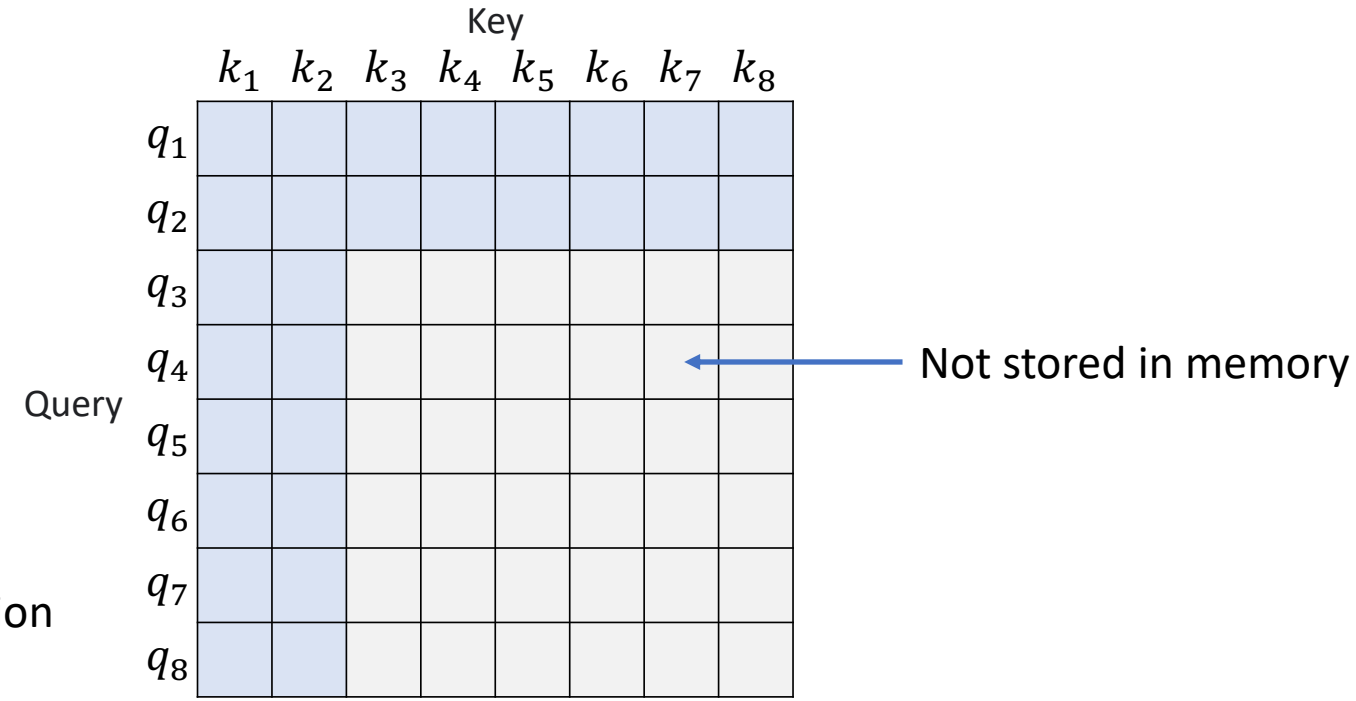
- Sparse attention
  - reduce computational and memory cost by only computing attention for a **subset of token pairs** instead of all pairs
  - evidence from machine learning community (e.g. Rewon Child, 2019)
- Sparse attention patterns
  - Position-based sparse attention: **rule-based or heuristic methods** to decide the positions where the interactions of these pair-wise tokens are important
  - Content-based sparse attention: tokens selectively attending only to other tokens that are **relevant based on their representations**

Position-based Sparse Attention

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

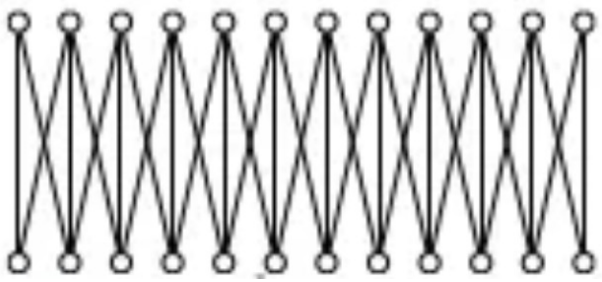


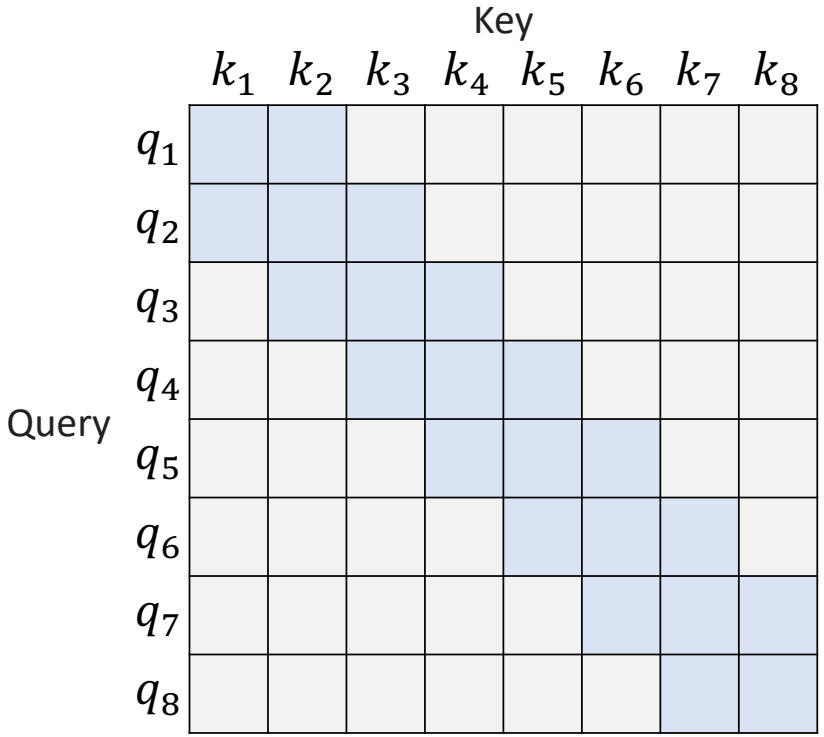
A bipartite graph illustration of QK^T computation



Position-based Sparse Attention

- 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

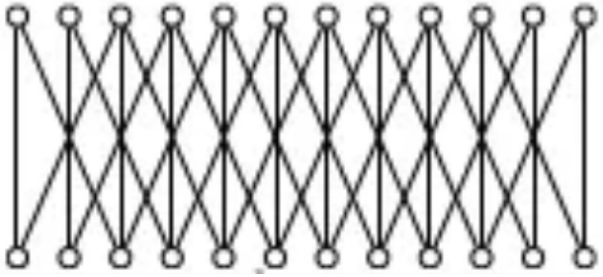


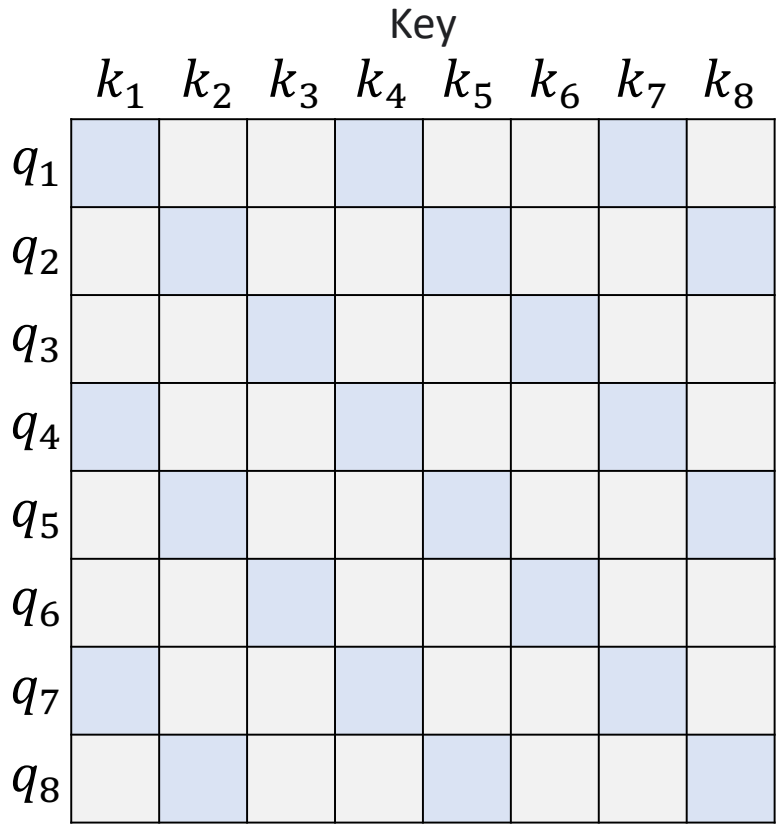


- Reducing complexity from $O(N^2)$ to $O(kN)$
- Limited receptive field (cannot capture long-range dependencies)
- Sacrificing context modeling ability: slow information propagation

Position-based Sparse Attention

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

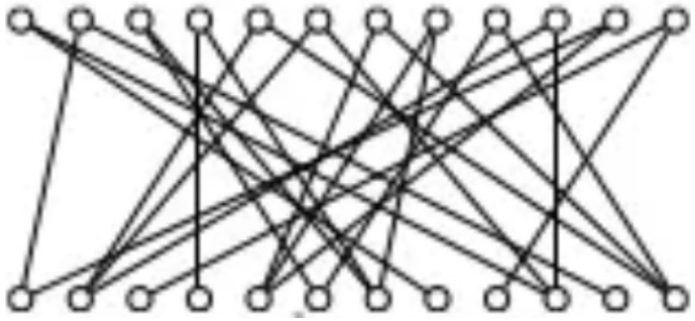


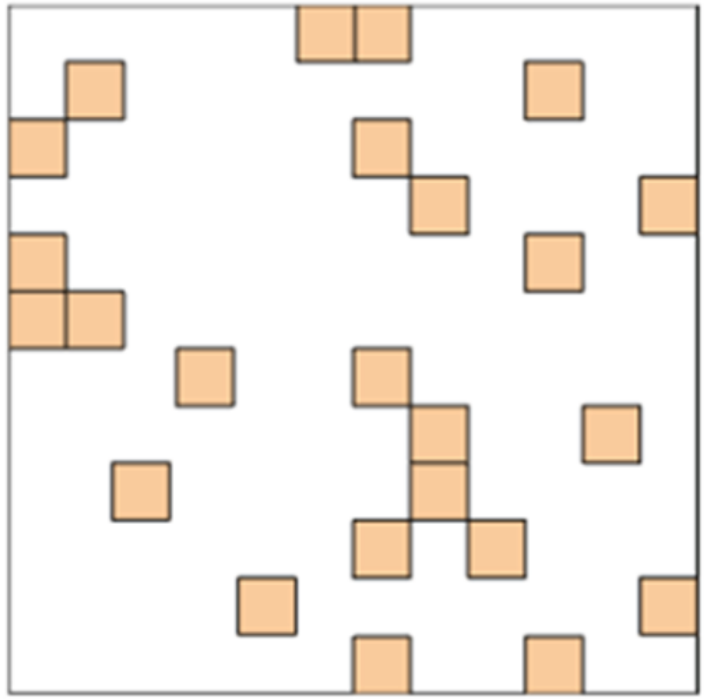


- Reducing complexity from $O(N^2)$ to $O(kN)$
- Missing fine-grained, short-range dependencies (failure to capture important neighbors in those gaps)

Position-based Sparse Attention

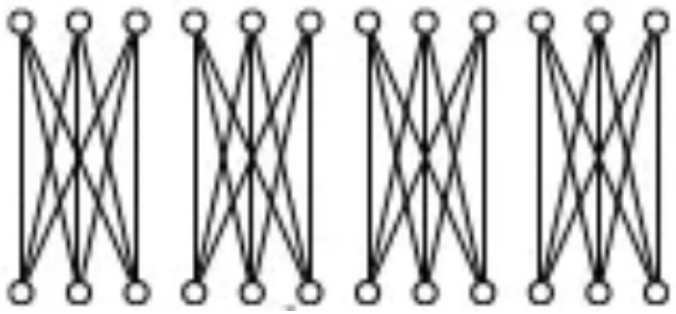
- Random attention
  - each query randomly samples a few keys for better capturing non-local interactions

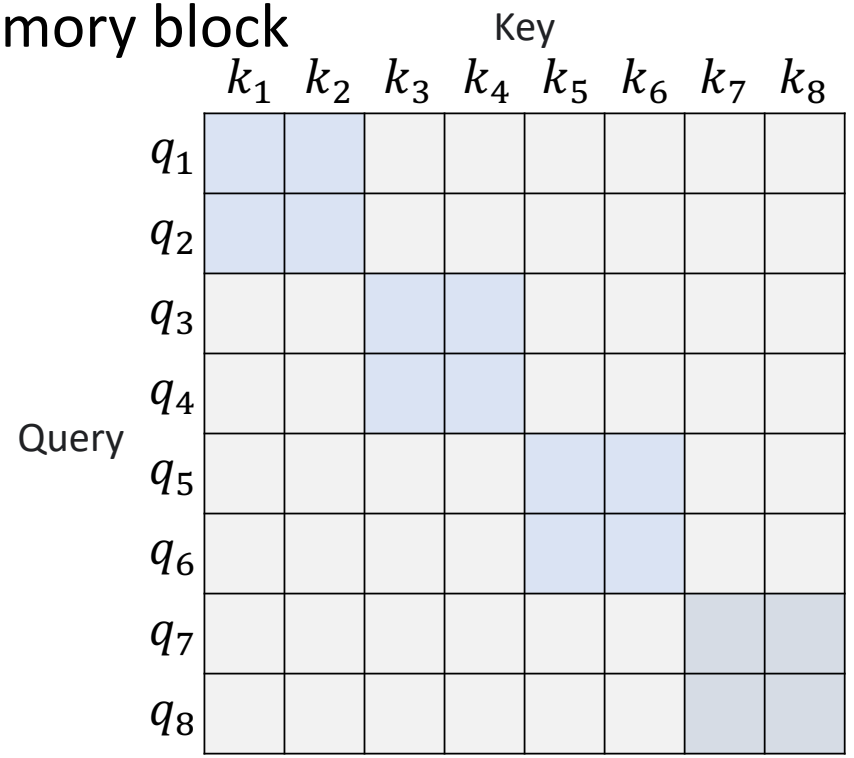




Position-based Sparse Attention

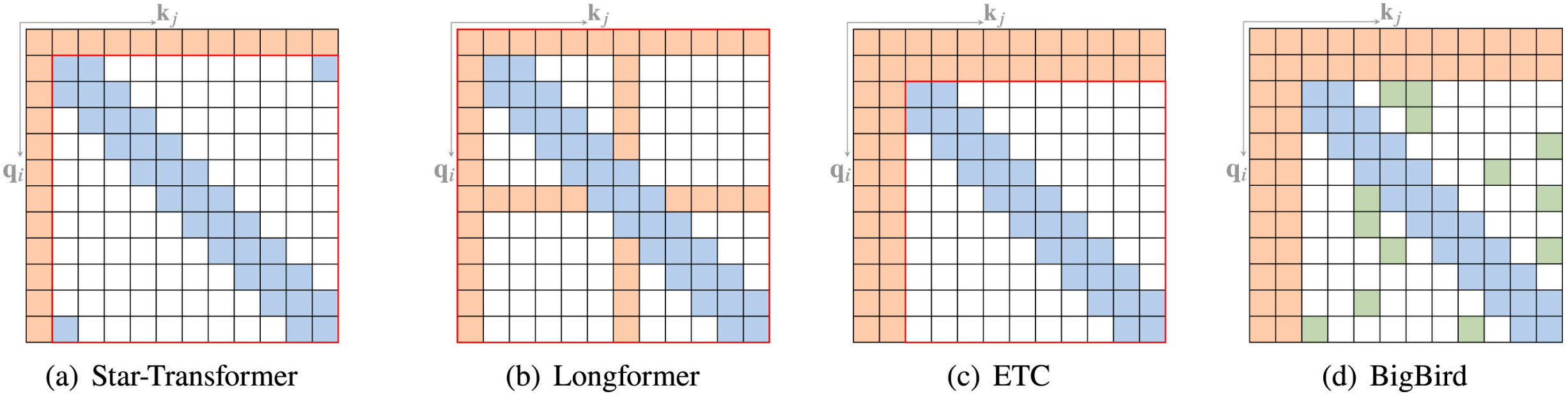
- Block attention
  - input sequence segmented into multiple non-intersection query blocks and each block assigned a local memory block





Position-based Sparse Attention

- COMPOUND attention



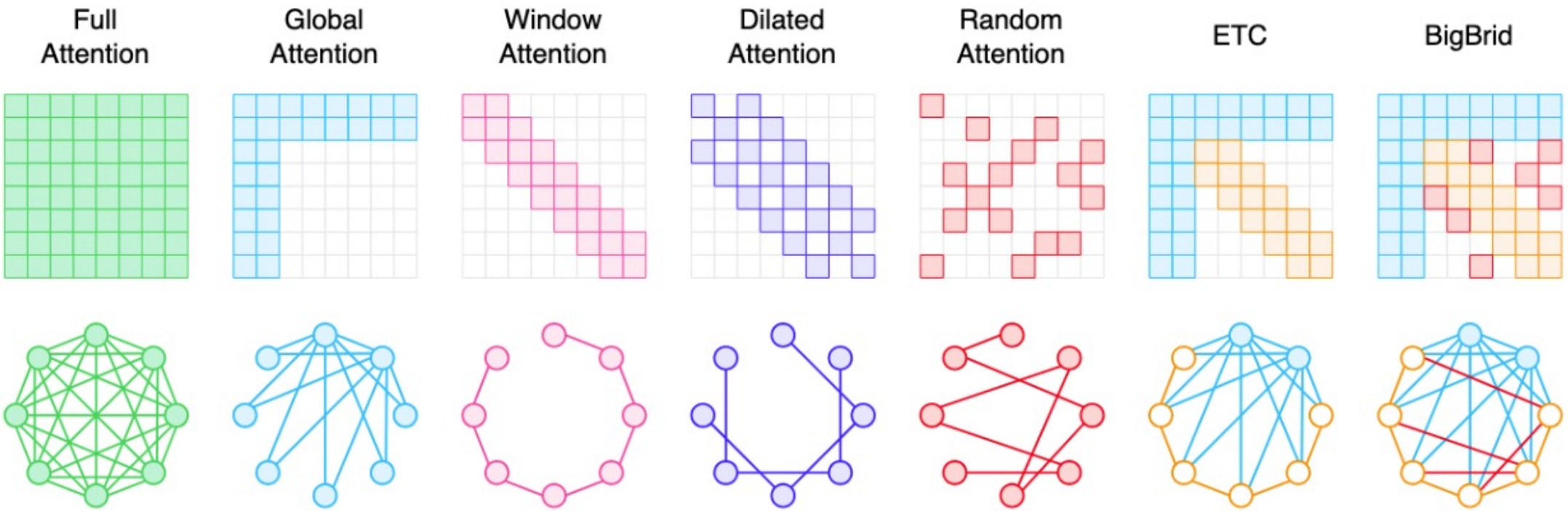
(a) Star-Transformer (b) Longformer (c) ETC (d) BigBird

- BigBird : local + global + random yields dense-equivalent performance
- Increased implementation complexity, more hyperparameters to tune and potential redundancy

https://wikidocs.net/198820

Position-based Sparse Attention

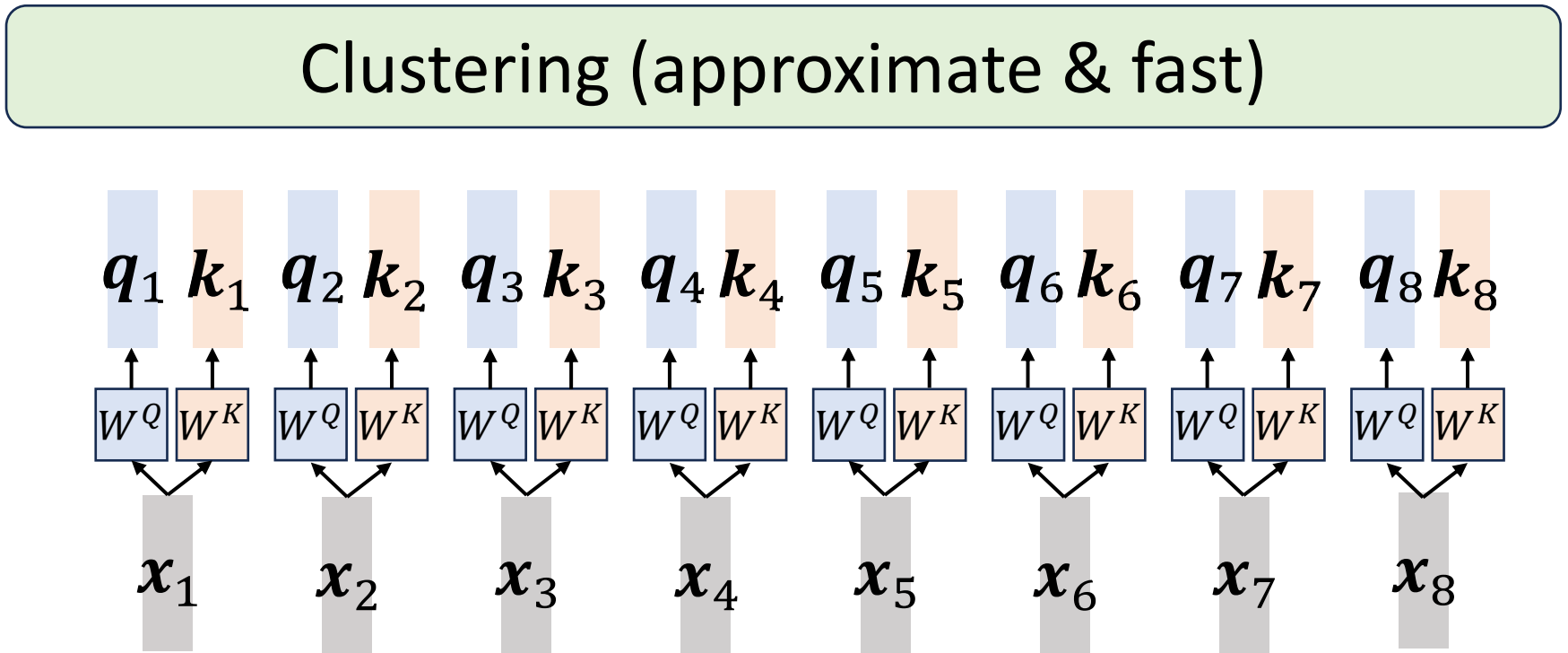
- Representative sparse attentions in a nutshell



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

Content-based Sparse Attention

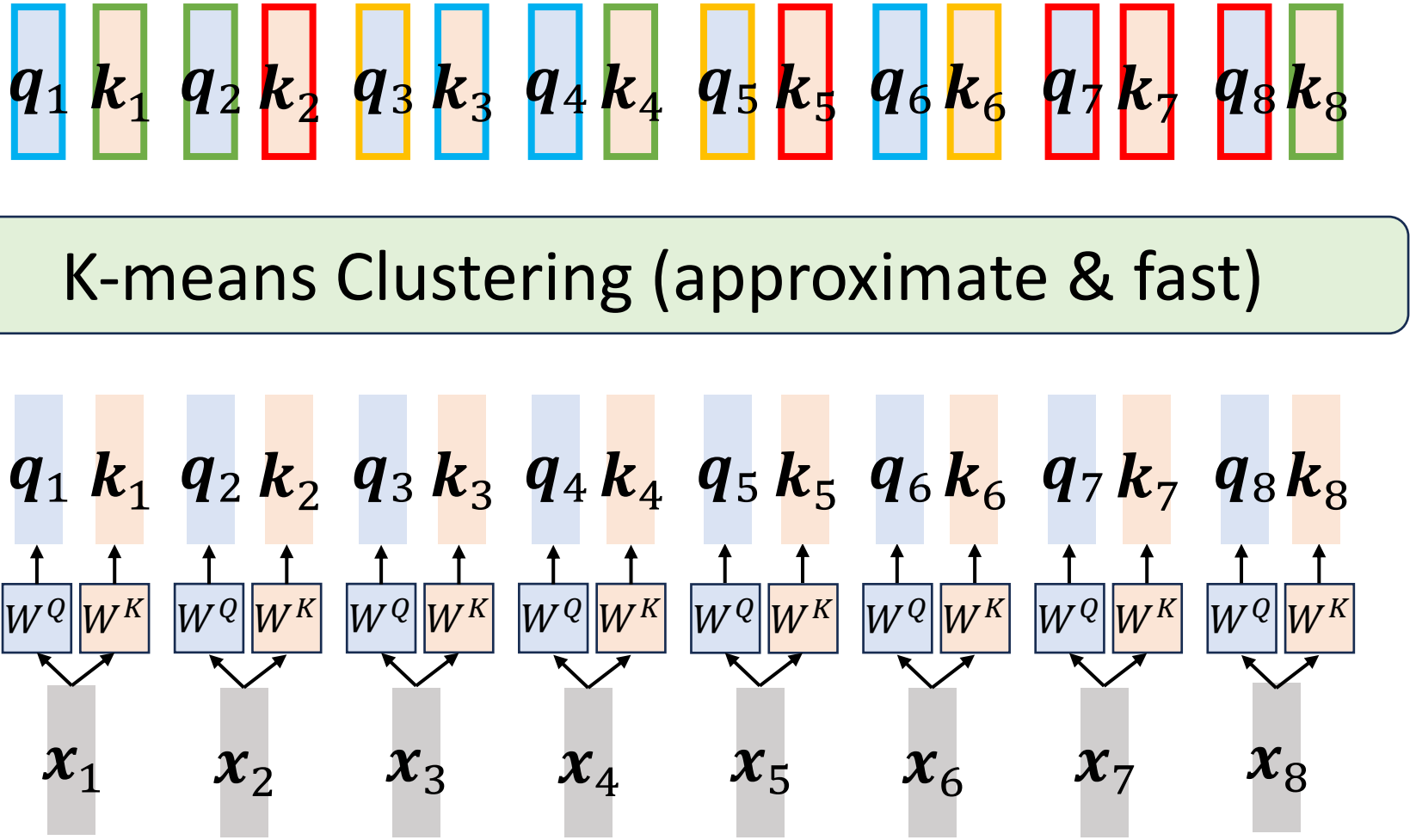
- Routing transformer



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

Content-based Sparse Attention

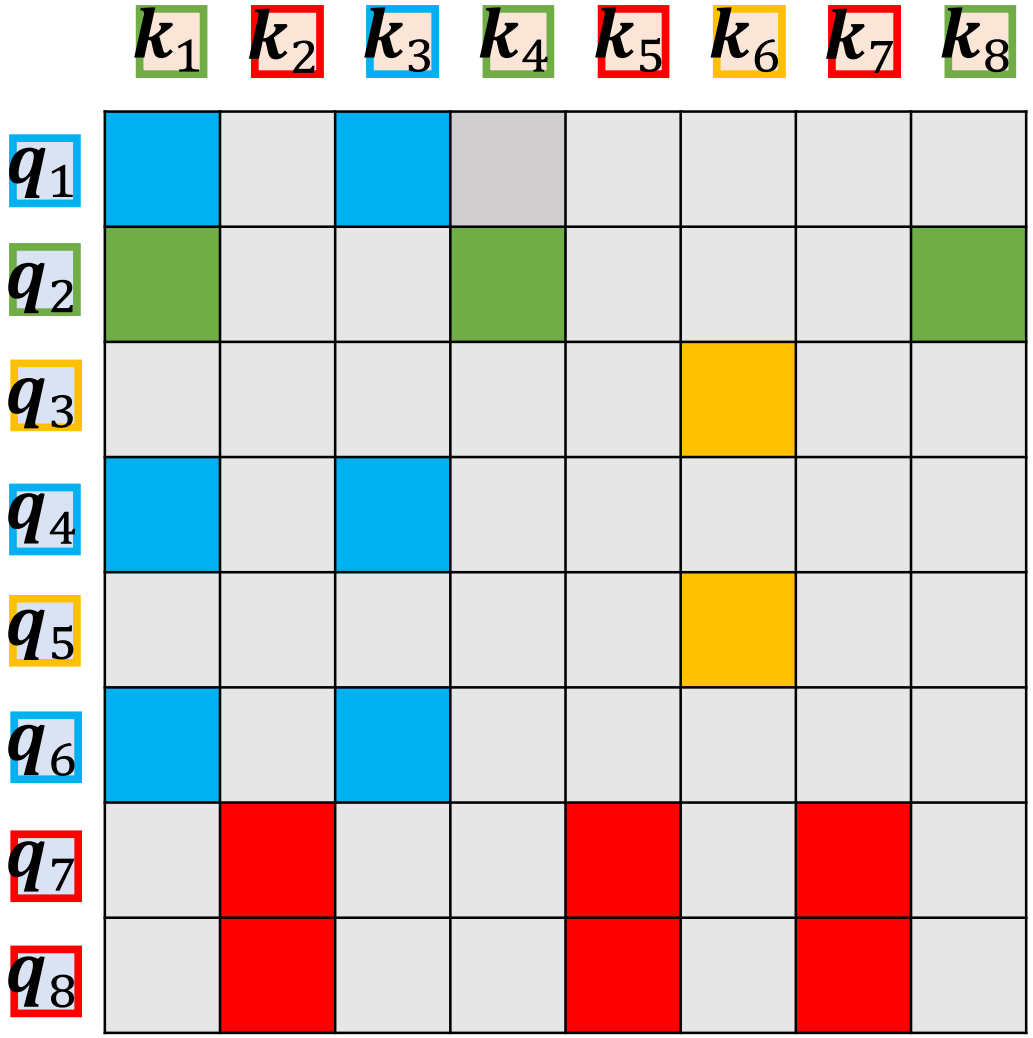
- Routing transformer



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

Content-based Sparse Attention

- Clusters
  - $\{q1, q4, q6, k1, k3\}$
  - $\{q2, k1, k4, k8\}$
  - $\{q3, q5, k6\}$
  - $\{q7, q8, k2, k5, k7\}$



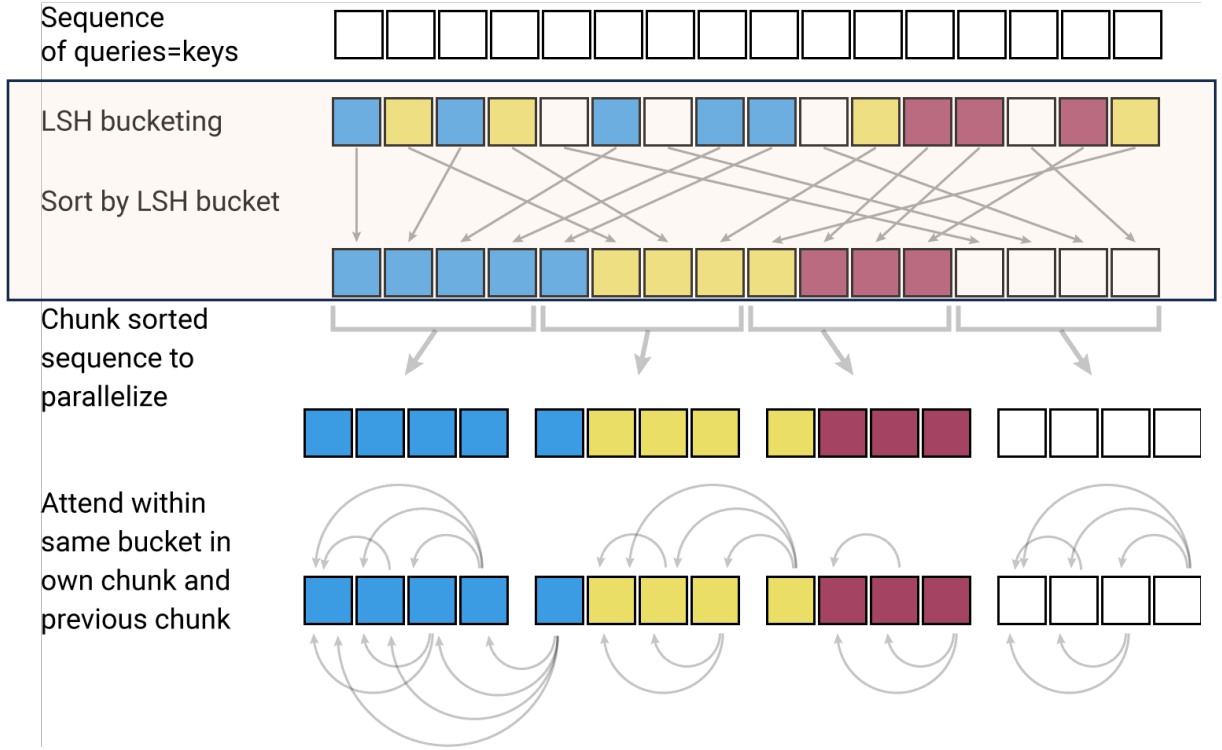
Source: Jia-Bin Huang from University of Maryland College Park

Content-based Sparse Attention

- Routing transformer: summary
  - attention scores in scaled dot-product rely on the similarity between Q and K → approximated by grouping them in a shared projection space
    - needs to be **double checked** by our students manually
  - each query only needs to attend to keys within its assigned cluster (typically $\sqrt{n}$ in size), reducing complexity to $O(n\sqrt{n})$ while preserving relevant long-range dependencies
  - cluster centroids are learned during training, allowing the model to adaptively group based on content

Content-based Sparse Attention

- Reformer: using locality-sensitive hashing (LSH) instead of dot-product attention

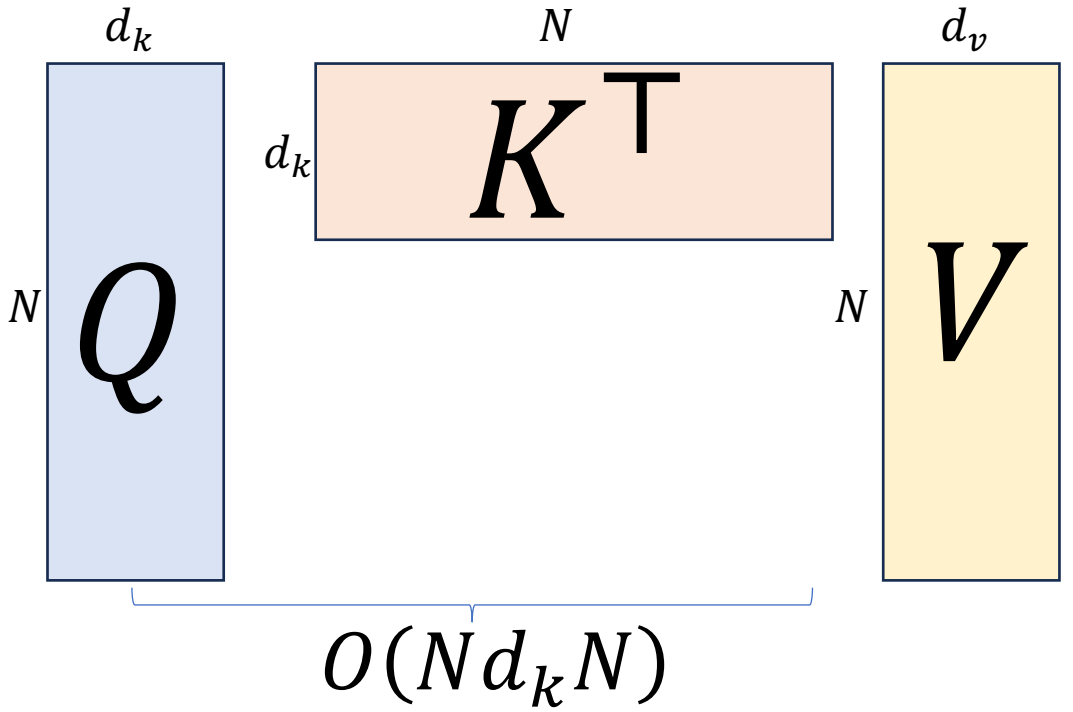


- Using **locality-sensitive hashing** (LSH) to select key-value pairs for each query
- Using Reversible Transformer to reduce memory usage during training
- Similar items (queries and keys) falling in the same bucket with high probability
- Processing sequences of length **64K tokens** or more efficiently

Source: Reformer: The Efficient Transformer

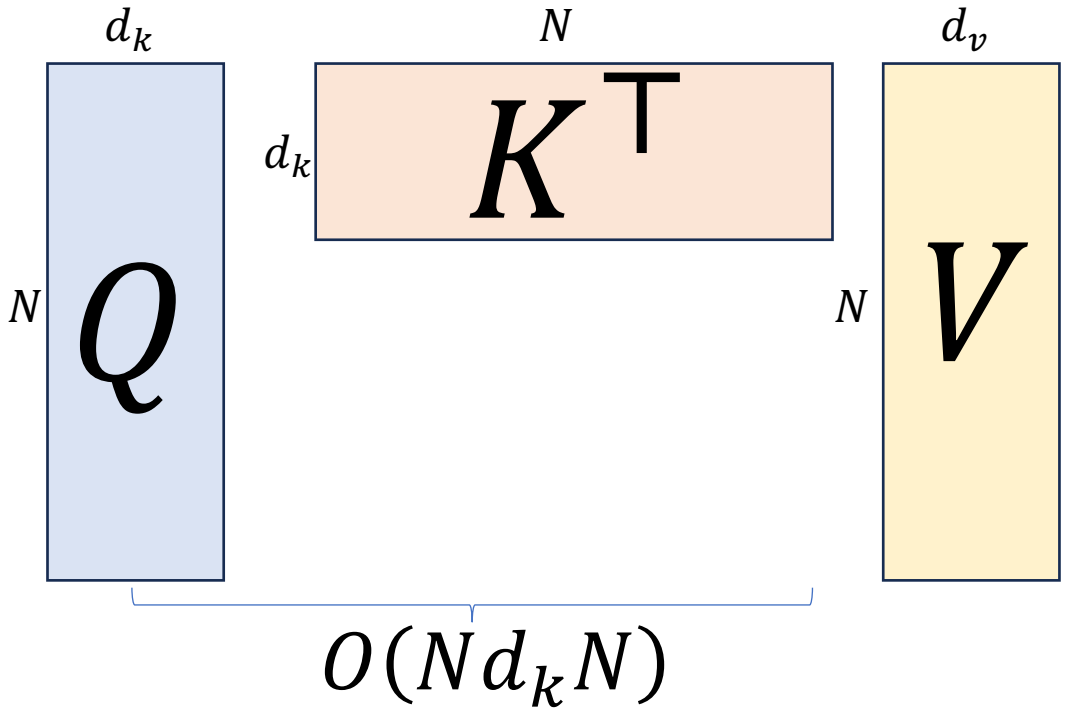
Inference Optimization: Outline

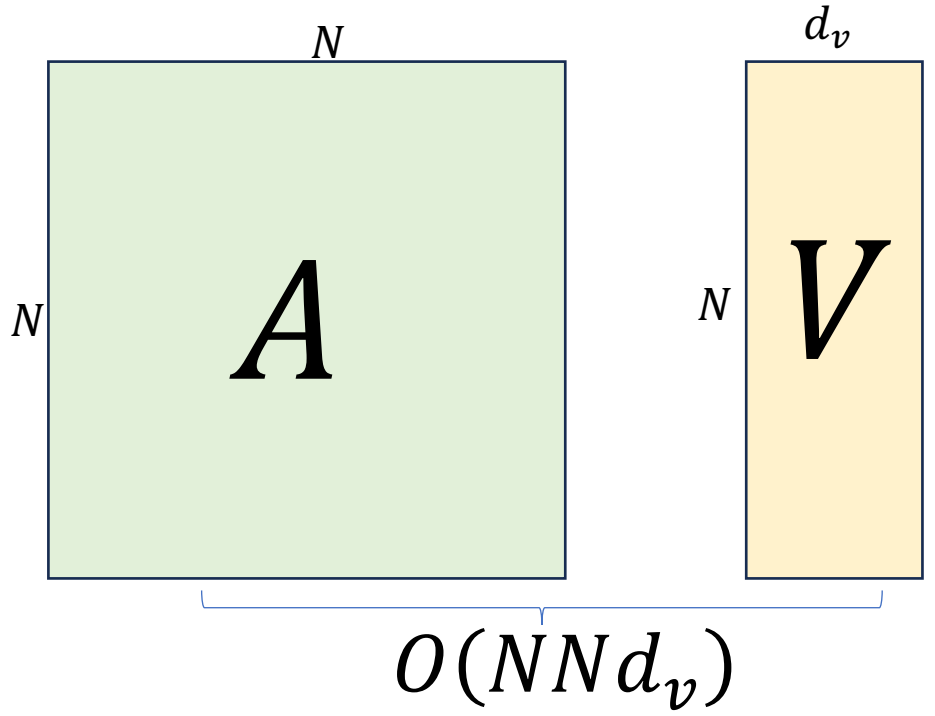
- Overview
- Attention Optimization
  - Sparse Attention
  - **Linear Attention**
  - Flash Attention
- Continuous Batching
- KV Cache Optimization
- Speculative Decoding
- Distributed Serving



"optimal parenthesization" via dynamic programming

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

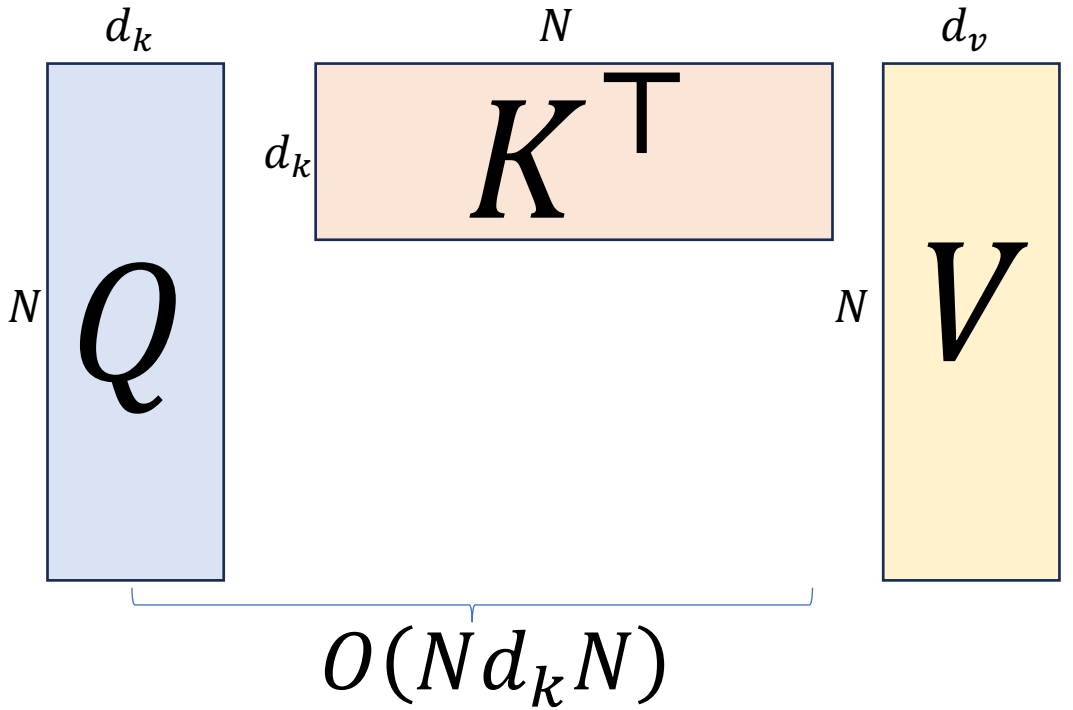


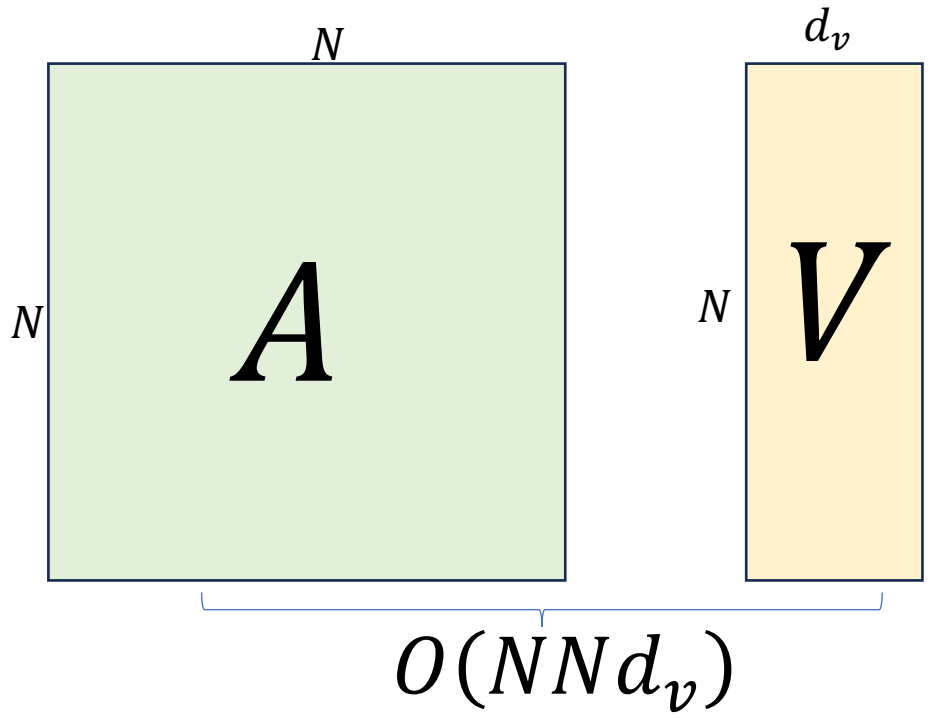


Source: Jia-Bin Huang from University of Maryland College Park

Original

$O(N^2(d_v+d_k))$

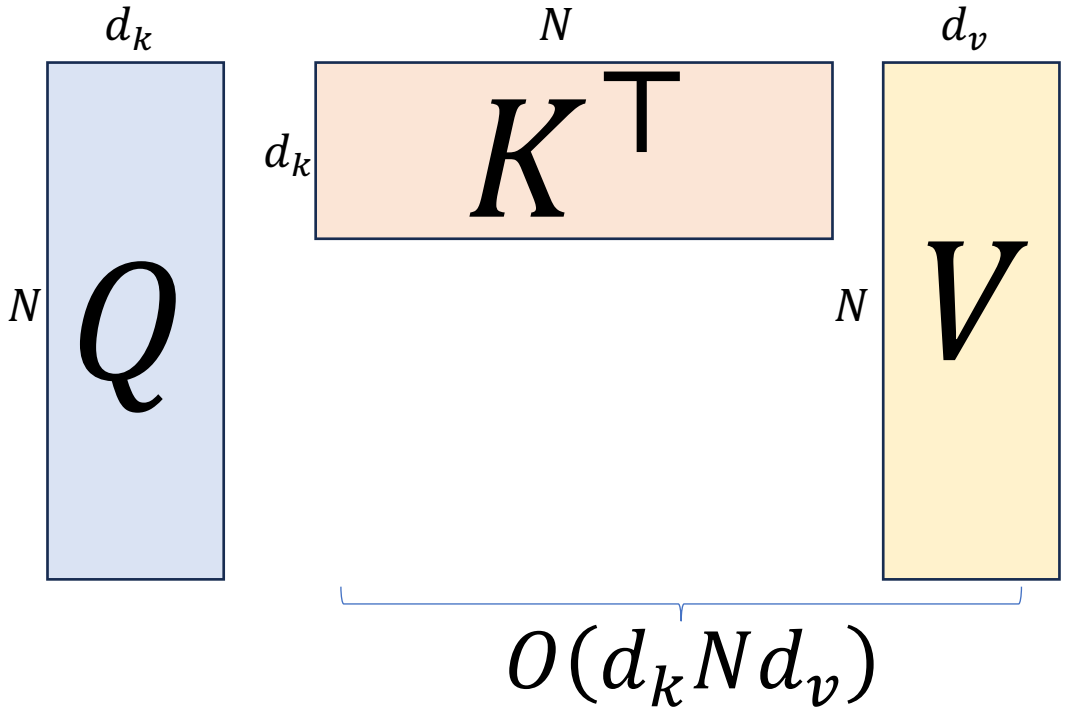




Source: Jia-Bin Huang from University of Maryland College Park

Original

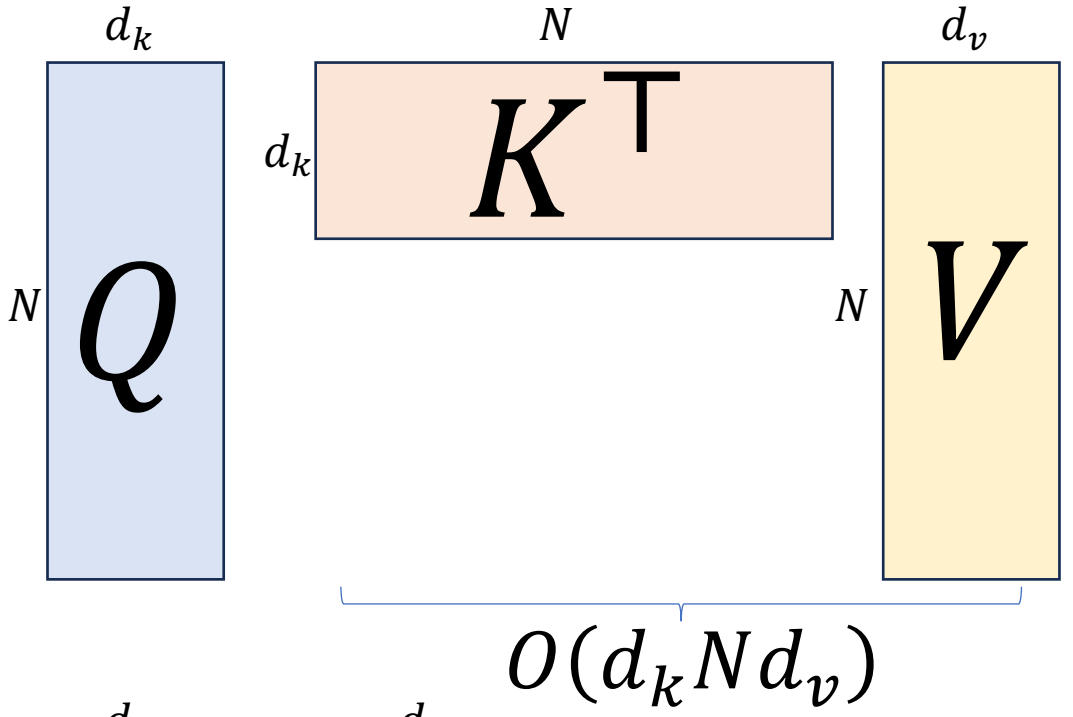
$$O(N^2(d_v+d_k))$$



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

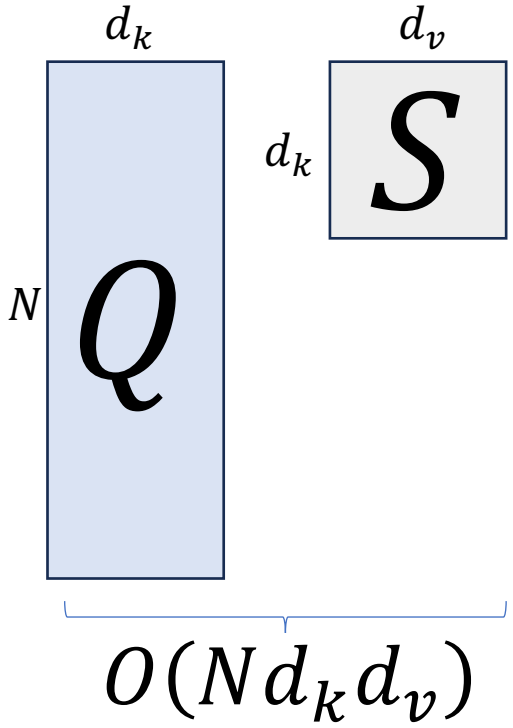
Original

$$O(N^2(d_v+d_k))$$



New

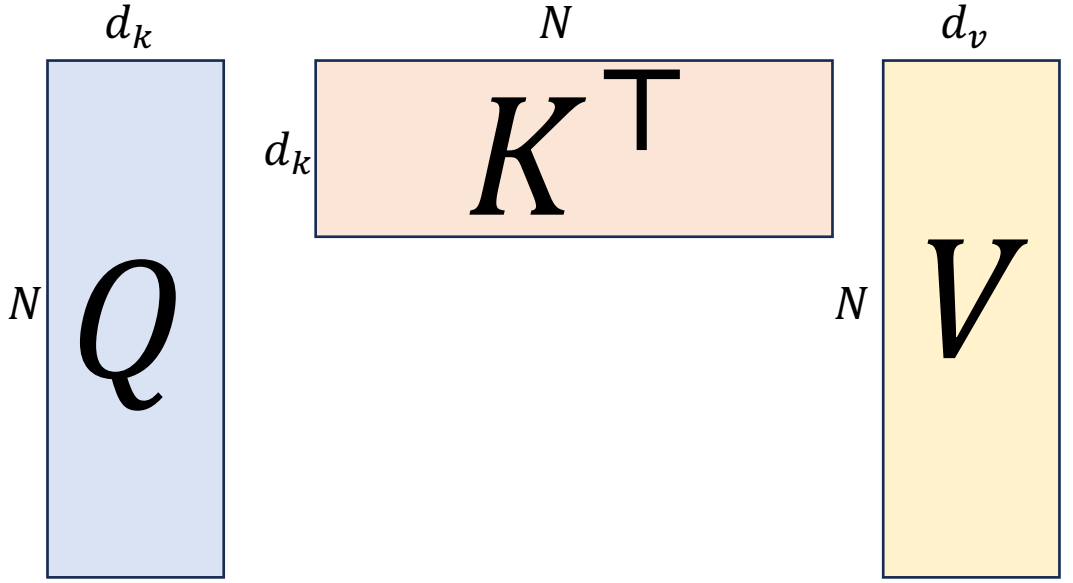
$$O(N d_k d_v)$$



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

Letting $N = 2048$

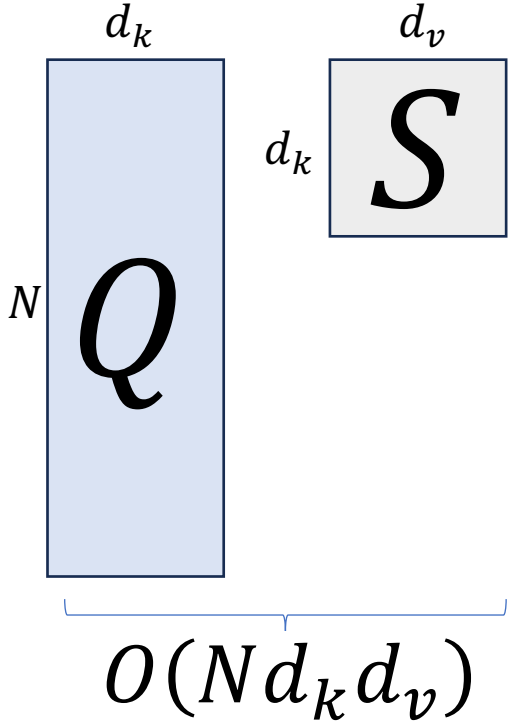
$d_k = d_v = 64$



Original

$O(N^2(d_v+d_k))$ = 536,870,912

1.56%!



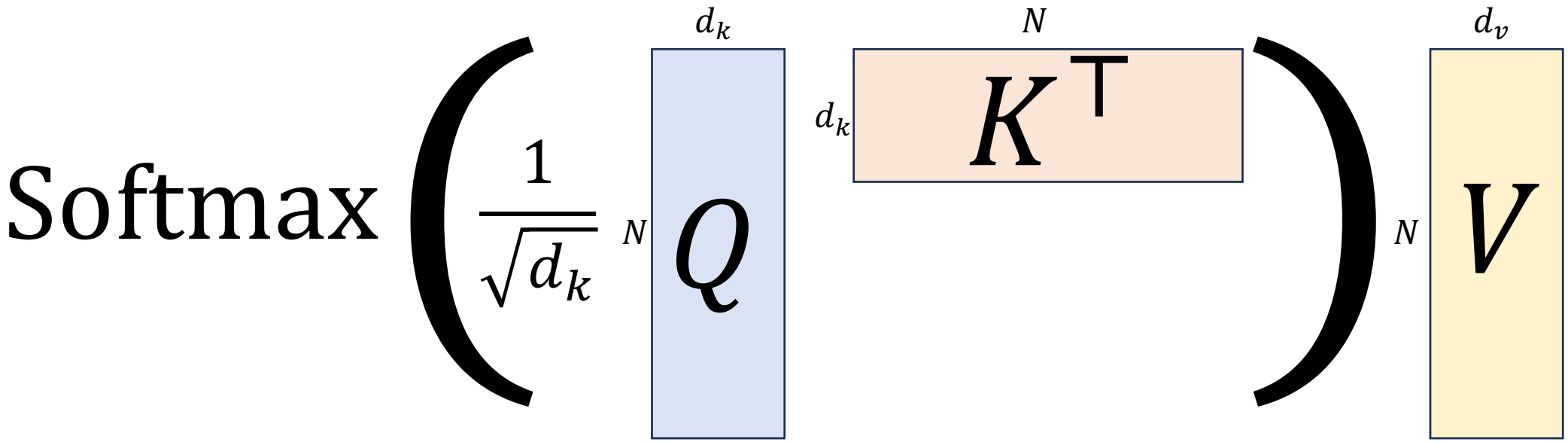
New

$O(N d_k d_v)$ = 8,388,608

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

Obstacle

- Softmax prevents this reordered GEMM



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

Linear Attention

- Computing Softmax (Katharopoulos et al., 2020)

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

- Substituting *exp* by

$$\exp(q \cdot k) \approx \phi(q) \cdot \phi(q)$$

Kernel function

  - there exists

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

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

Linear Attention

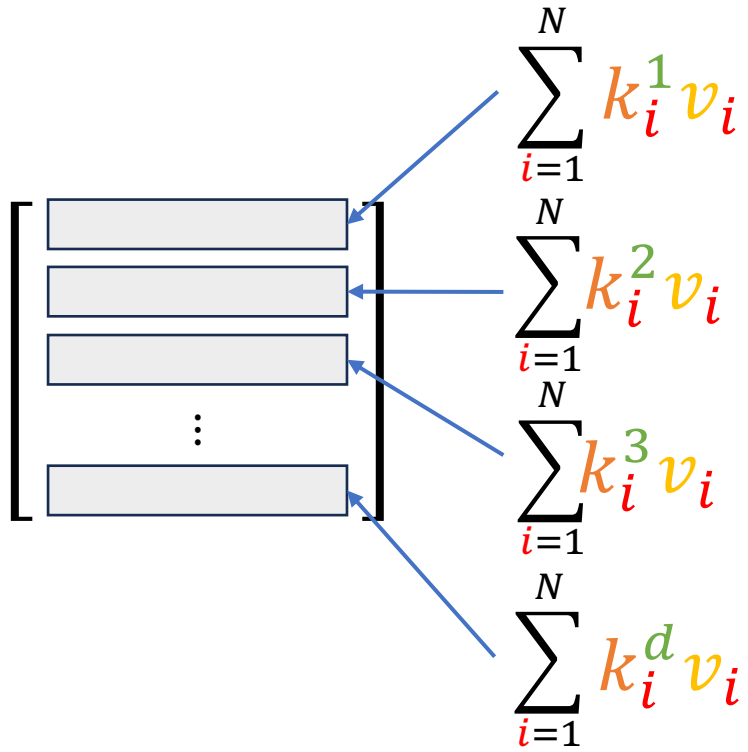
- Overall Output: complexity order $O(N)$

$$O \approx \phi(Q) \cdot (\phi(K)\, V), \quad O, Q, K, V \in \mathbb{R}^{N \times d}$$

- An example: denominator of o_1

$$\phi(q_1) = \begin{bmatrix} q_1^1 \\ q_1^2 \\ q_1^3 \\ \vdots \\ q_1^d \end{bmatrix} \quad \phi(k_i) = \begin{bmatrix} k_i^1 \\ k_i^2 \\ k_i^3 \\ \vdots \\ k_i^d \end{bmatrix},$$

$$\begin{bmatrix} q_1^1 & q_1^2 & q_1^3 & \cdots & q_1^d \end{bmatrix}$$



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

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: Outline

- Overview
- Attention Computation Optimization
  - Sparse Attention
  - Linear Attention
  - **Flash Attention**
  - Continuous Batching
- KV Cache Optimization
- Speculative Decoding
- 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-trival

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 $\mathbf{O}$.

Flash Attention

- From Softmax to Safe Softmax

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

- Online Safe Softmax: not enough space to store entire x
  - Alg1: three-pass algorithm

m_i: $\max_{j=1}^{i}\{x_j\}$ with initial value $m_0 = -\infty$

d_i: $\sum_{j=1}^{i} e^{x_j - m_N}$ is the denominator of Softmax

a_i: is the final Softmax output

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

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

end

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

$$d_i \leftarrow d_{i-1} + e^{x_i - m_N}$$

end

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

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d_N}$$

end

Flash Attention

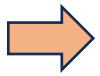
- Online Safe Softmax
  - d_i can be computed iteratively

$$d'_i = \sum_{j=1}^{i} e^{x_j - m_i}$$

$$= \left(\sum_{j=1}^{i-1} e^{x_j - m_i}\right) + e^{x_i - m_i}$$

$$= \left(\sum_{j=1}^{i-1} e^{x_j - m_{i-1}}\right) e^{m_{i-1} - m_i} + e^{x_i - m_i}$$

$$= d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}$$



Algorithm 2-pass online softmax

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

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

end

Flash Attention

- Online Safe Softmax
  - One-pass safe softmax algorithm does not exist
  - Safe softmax is only our intermediate result:

$$O = \text{Softmax}(QK^T)V$$

NOTATIONS

$Q[k,:]$: the k-th row vector of Q matrix.
$K^T[:,i]$: the i-th column vector of K^T matrix.
$O[k,:]$: the k-th row of output O matrix.
$V[i,:]$: the i-th row of V matrix.
$\{\boldsymbol{o}_i\}$: $\sum_{j=1}^{i} a_j V[j,:]$, a row vector storing partial aggregation result $A[k,:i] \times V[:i,:]$

BODY

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

$$\begin{aligned} x_i &\leftarrow Q[k,:]\, K^T[:,i] \\ m_i &\leftarrow \max(m_{i-1}, x_i) \\ d'_i &\leftarrow d'_{i-1} e^{m_{i-1}-m_i} + e^{x_i - m_i} \end{aligned}$$

end
for $i \leftarrow 1, N$ **do**

$$\begin{aligned} a_i &\leftarrow \frac{e^{x_i - m_N}}{d'_N} \\ \boldsymbol{o}_i &\leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:] \end{aligned}$$

end

$$O[k,:] \leftarrow \boldsymbol{o}_N$$

Flash Attention

- One-pass Attention
  - Define an “intermediate” output
  - Compute it iteratively

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

$$\begin{aligned}
\boldsymbol{o}'_i &= \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \frac{d'_{i-1}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} V[j,:] \right) \frac{d'_{i-1}}{d'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]
\end{aligned}$$

$$\begin{aligned}
&for\, i \rightarrow 1, N \\
&\quad x_i \leftarrow Q[k,:] K^T[:,i] \\
&\quad m_i \leftarrow \max(m_{i-1}, x_i) \\
&\quad d'_i \leftarrow d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&\quad o'_i = o'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&end \\
&O[k,:] \leftarrow o'_N
\end{aligned}$$

Flash Attention

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 $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.

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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

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

5: **for** $1 \le j \le T_c$ **do** — Outer Loop on Key/Value

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

7: **for** $1 \le i \le T_r$ **do** — Inner Loop on Key/Value

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.

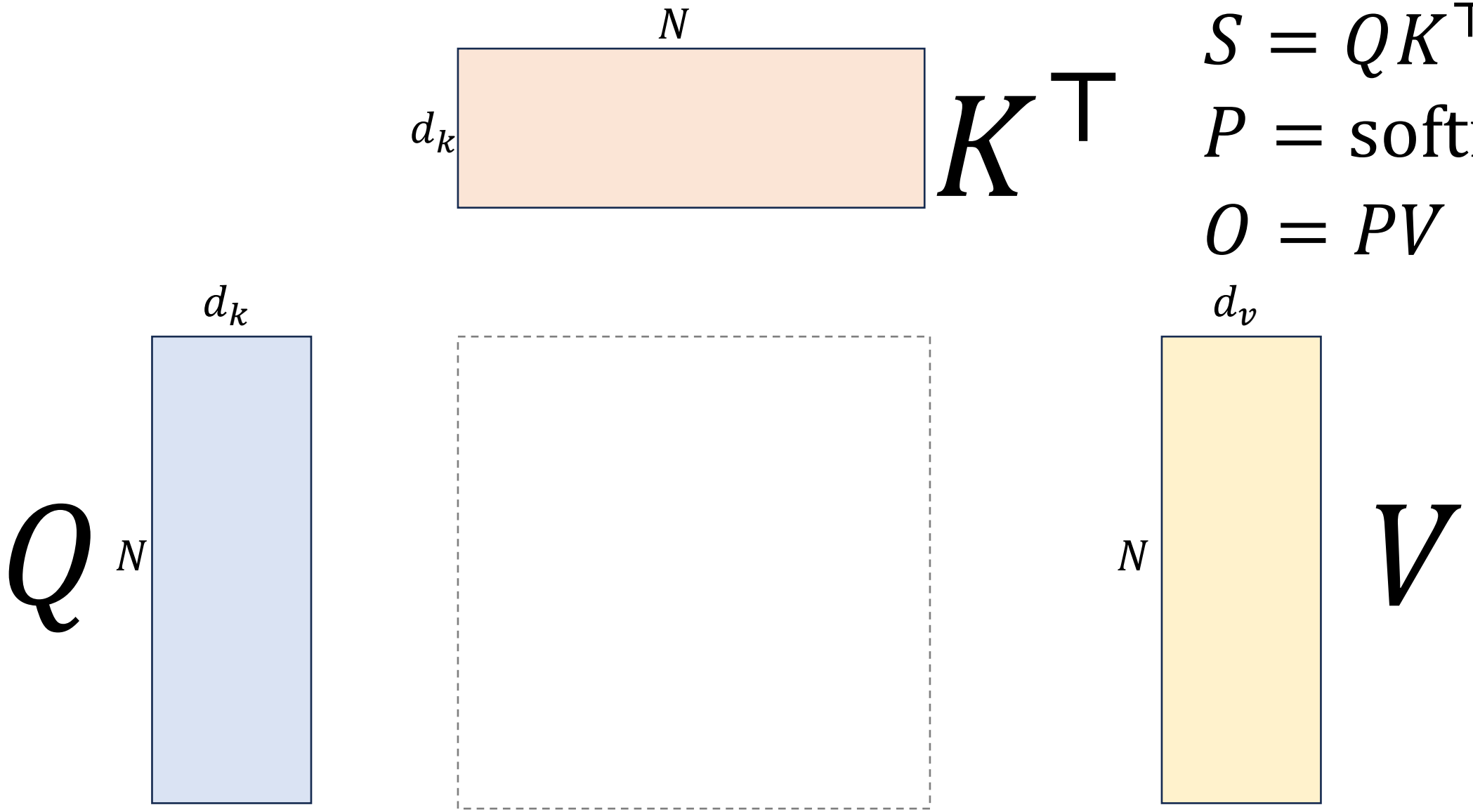
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

14: **end for**

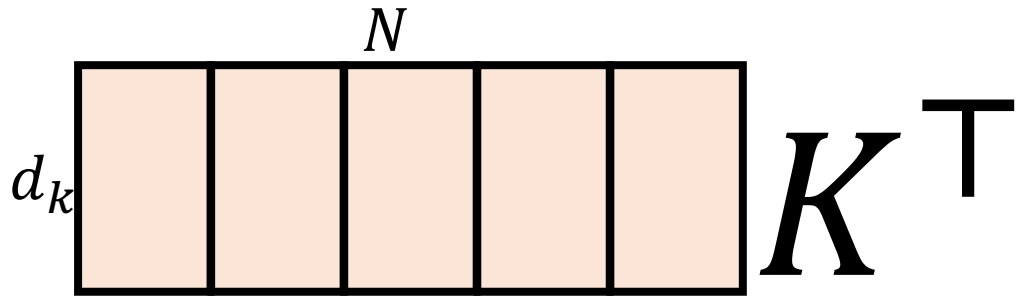
15: **end for**

16: Return $\mathbf{O}$.

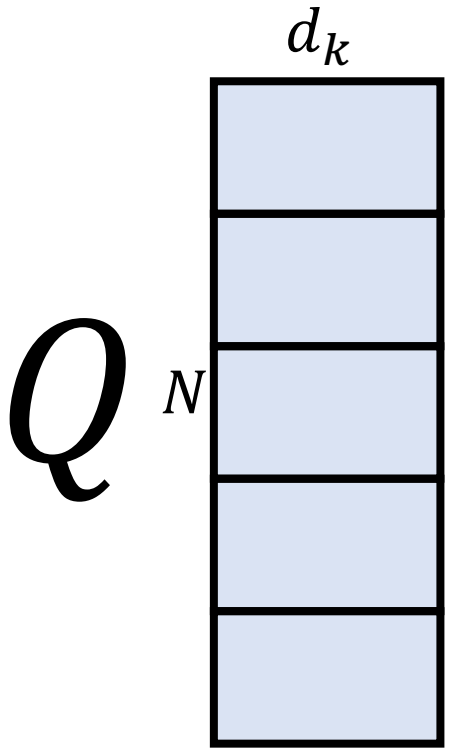
$$S = QK^\top$$
$$P = \text{softmax}(S)$$
$$O = PV$$

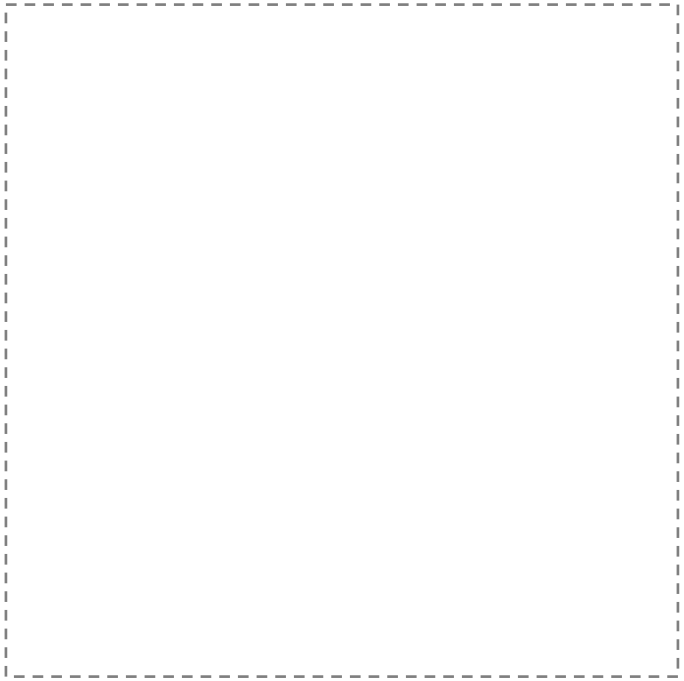


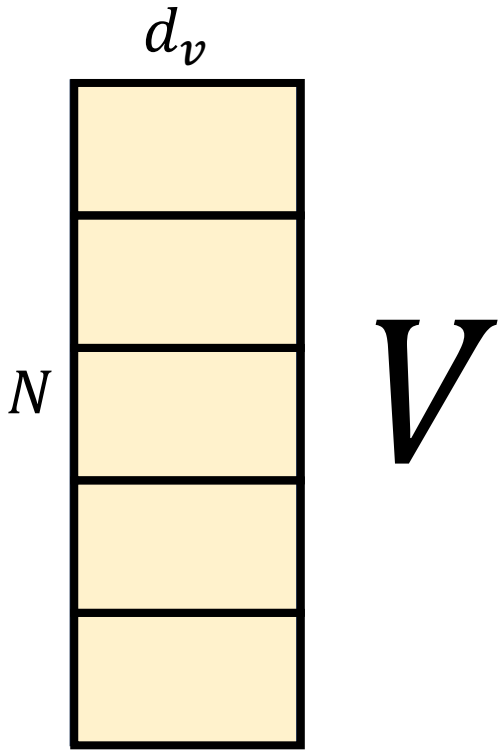
Source: Jia-Bin Huang from University of Maryland College Park



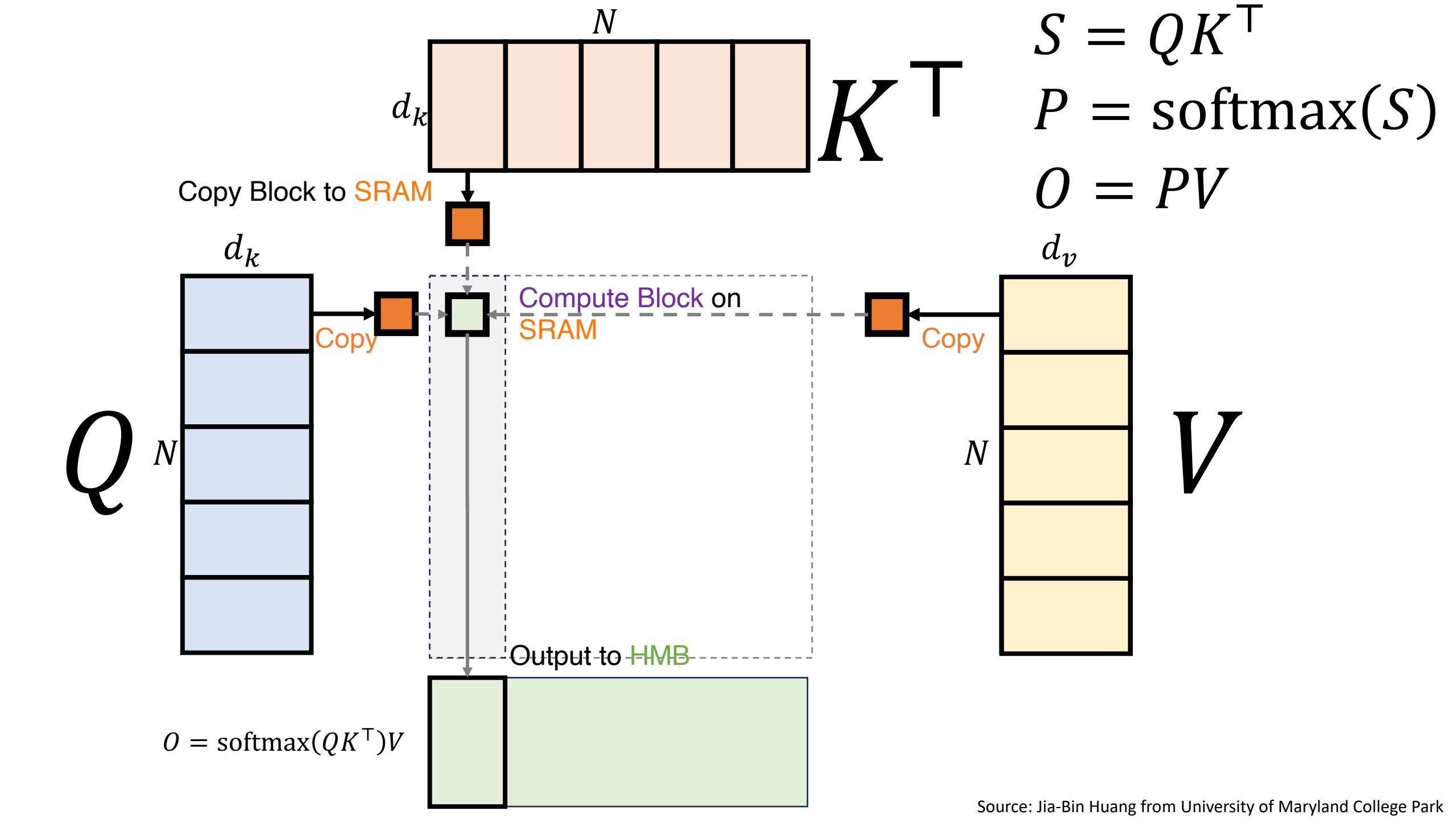
$$S = QK^\top$$
$$P = \text{softmax}(S)$$
$$O = PV$$



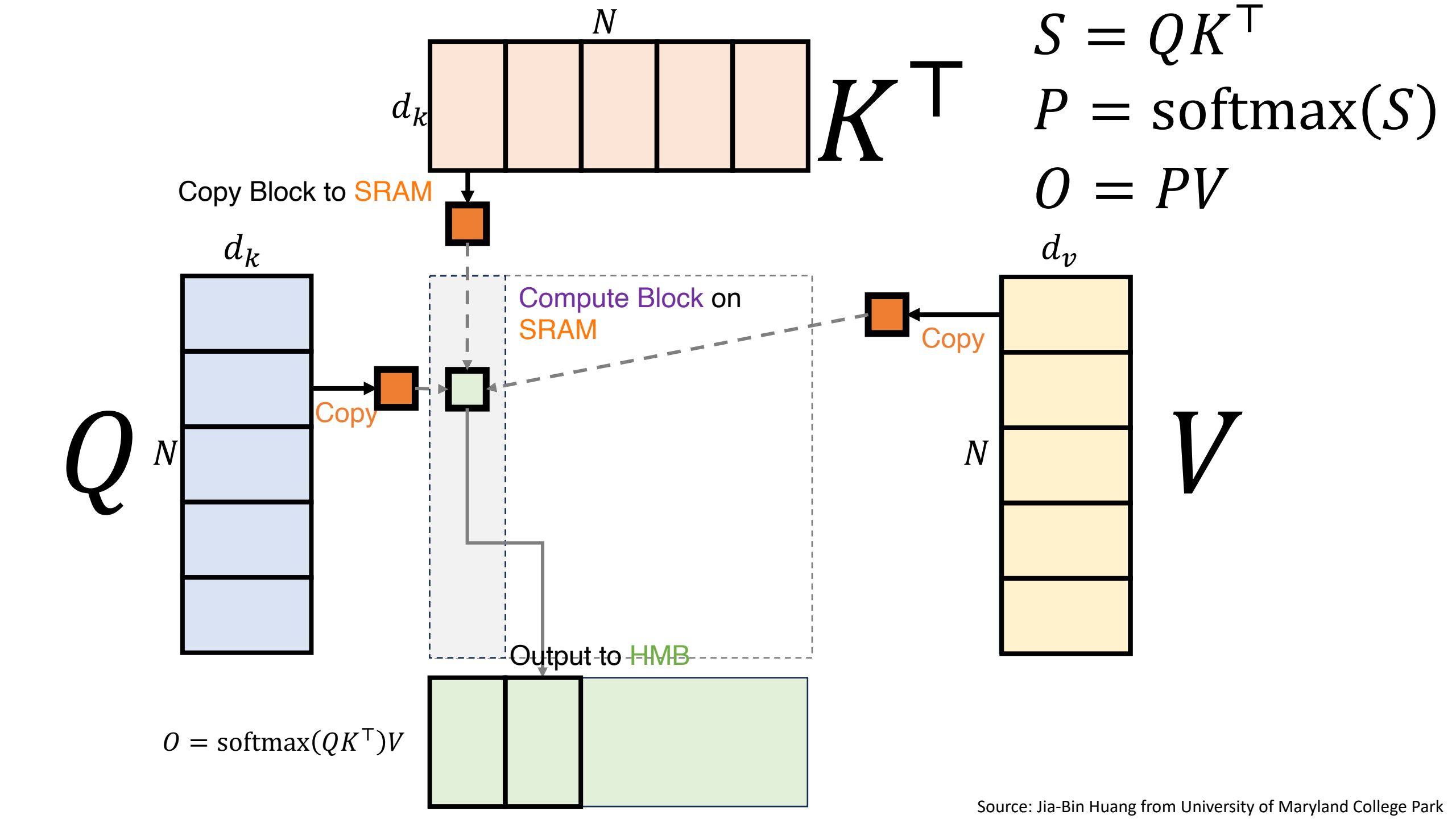




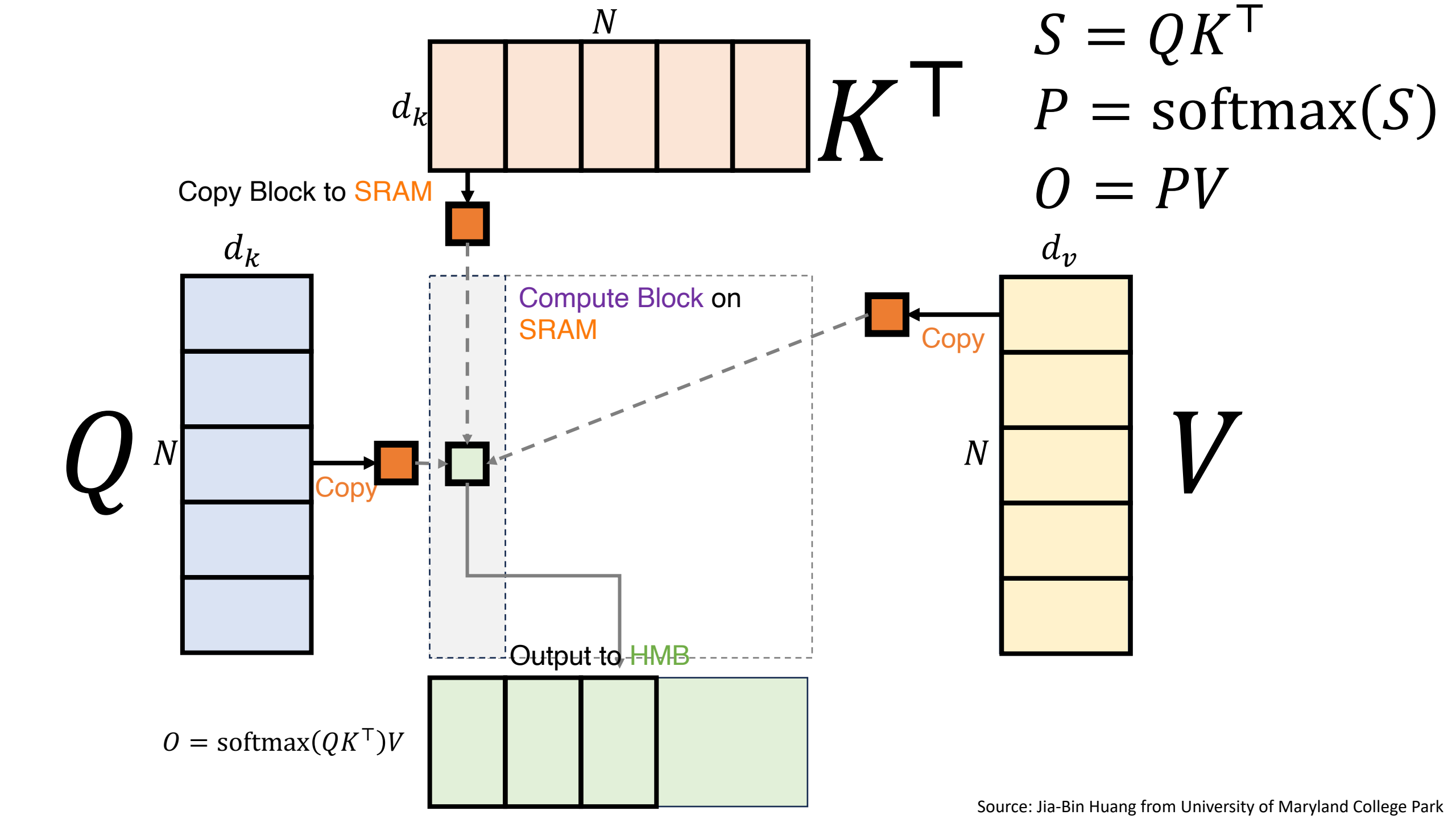
Source: Jia-Bin Huang from University of Maryland College Park



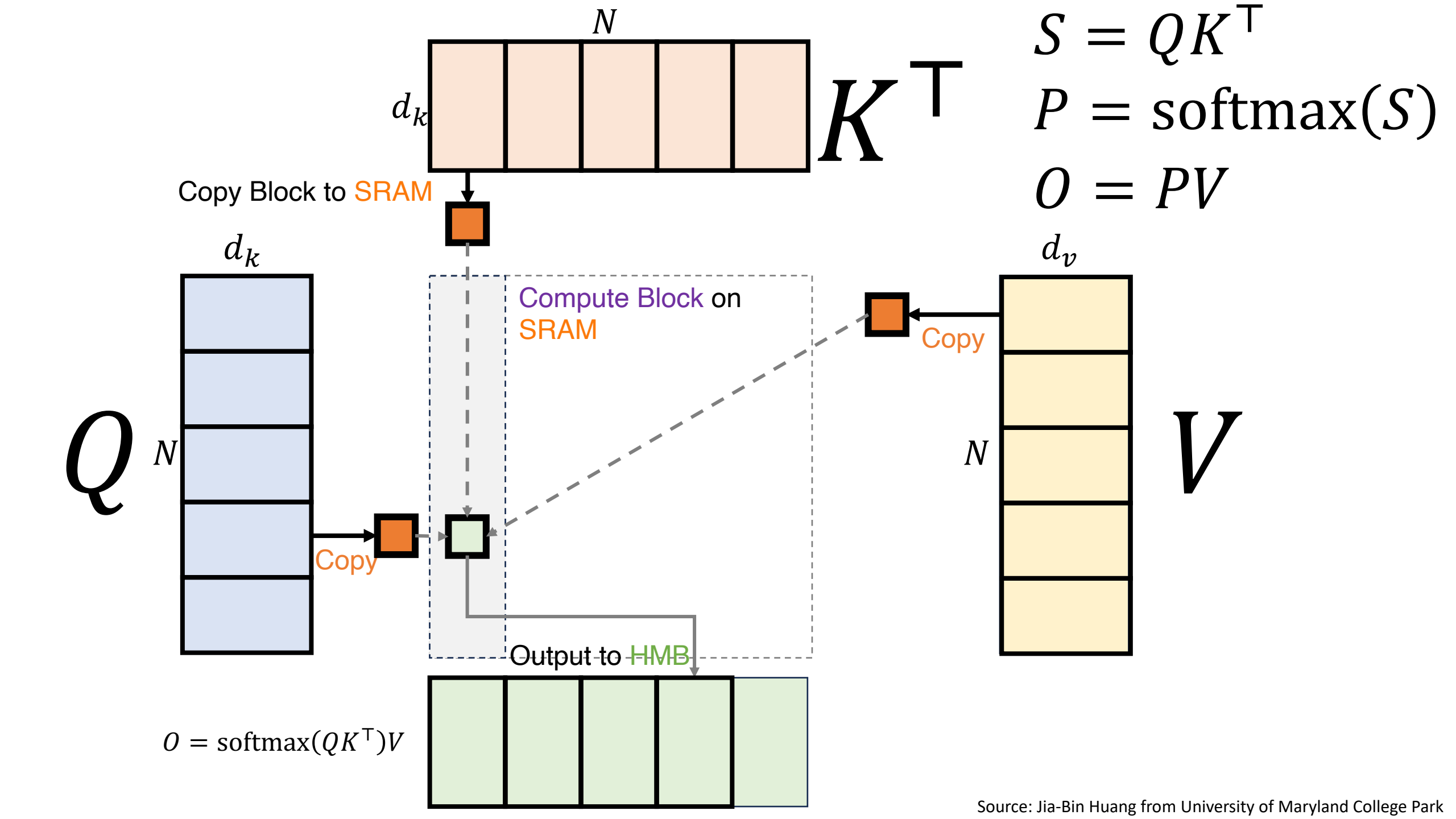
Source: Jia-Bin Huang from University of Maryland College Park



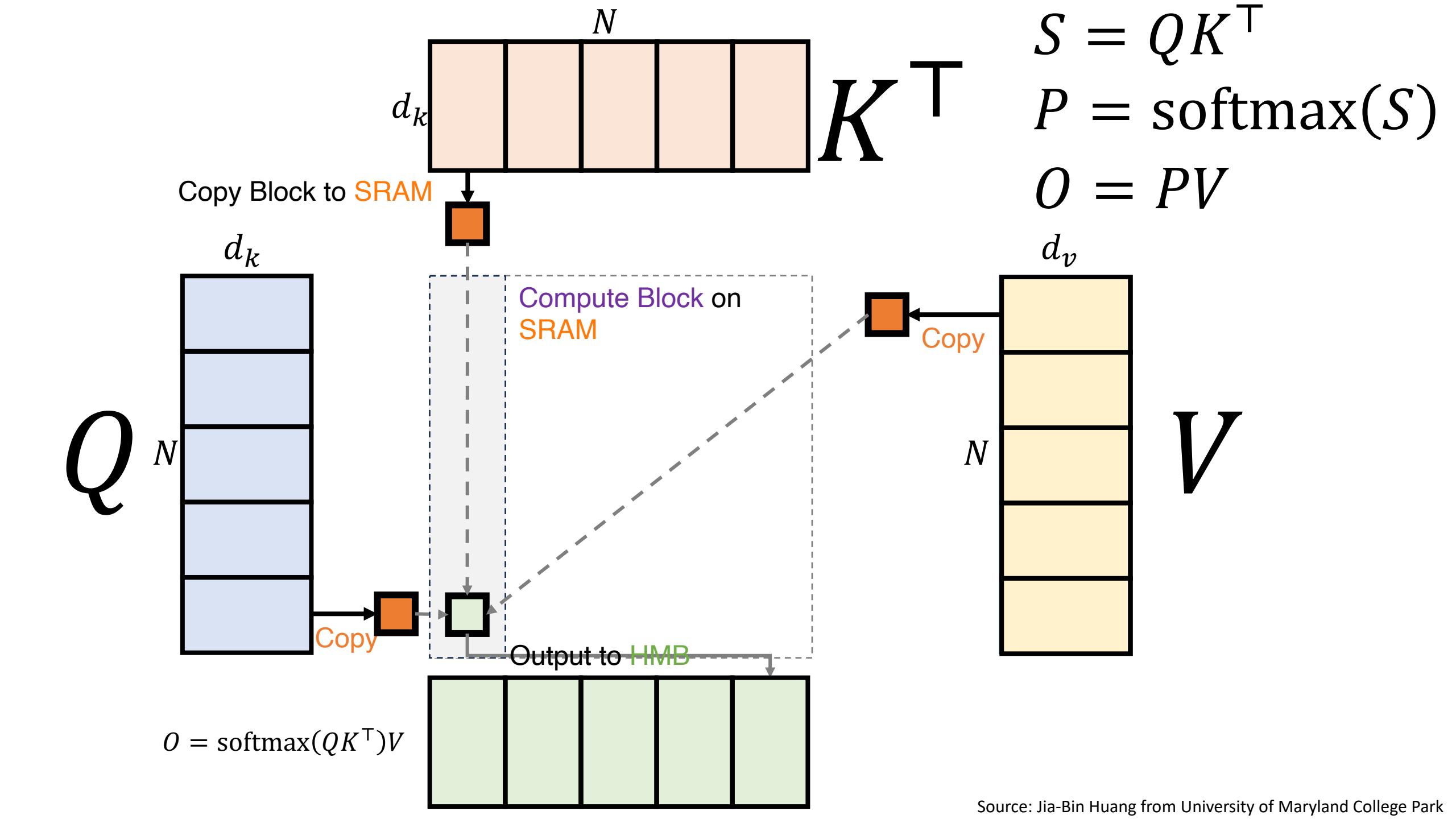
Source: Jia-Bin Huang from University of Maryland College Park



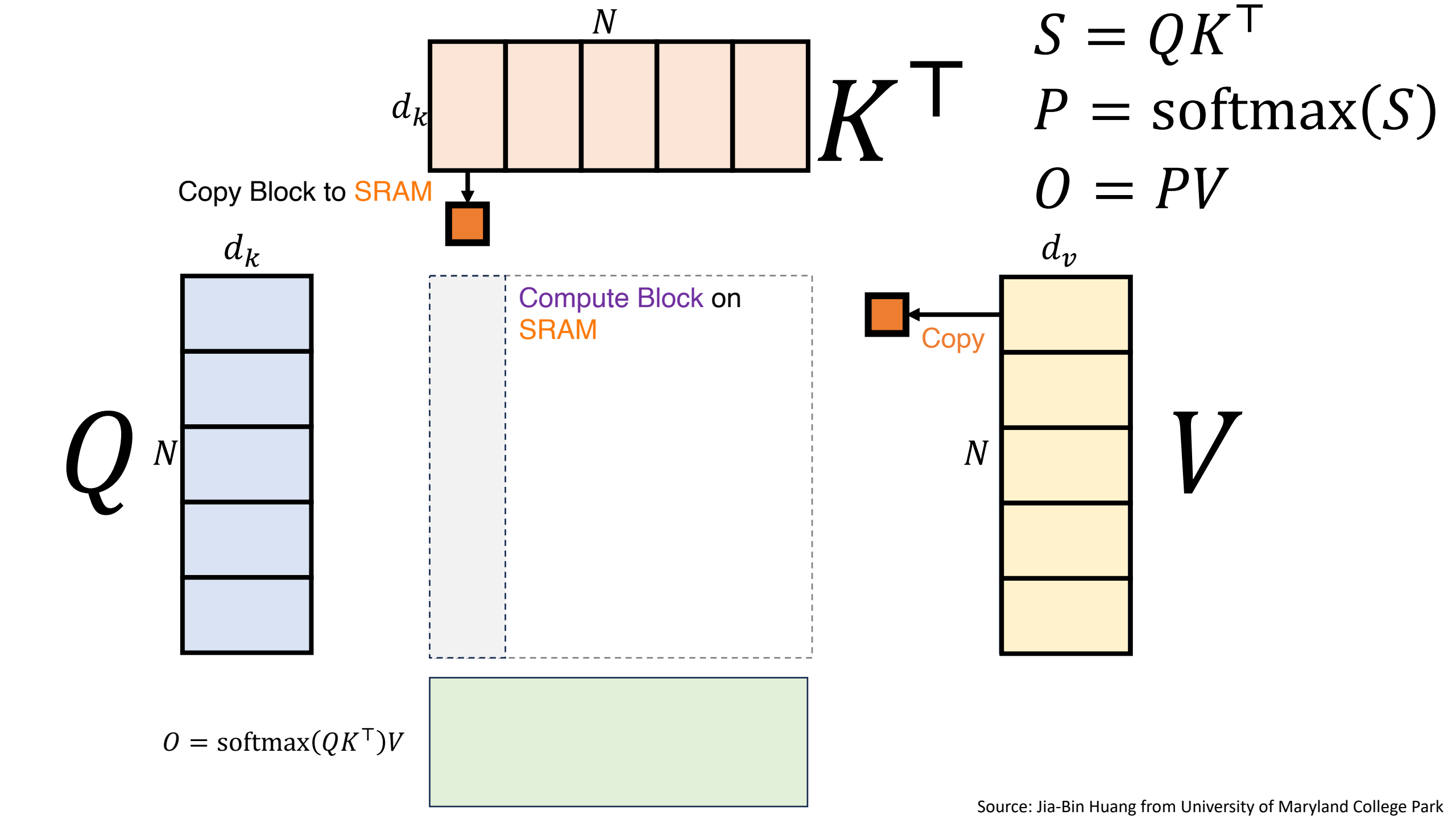
Source: Jia-Bin Huang from University of Maryland College Park



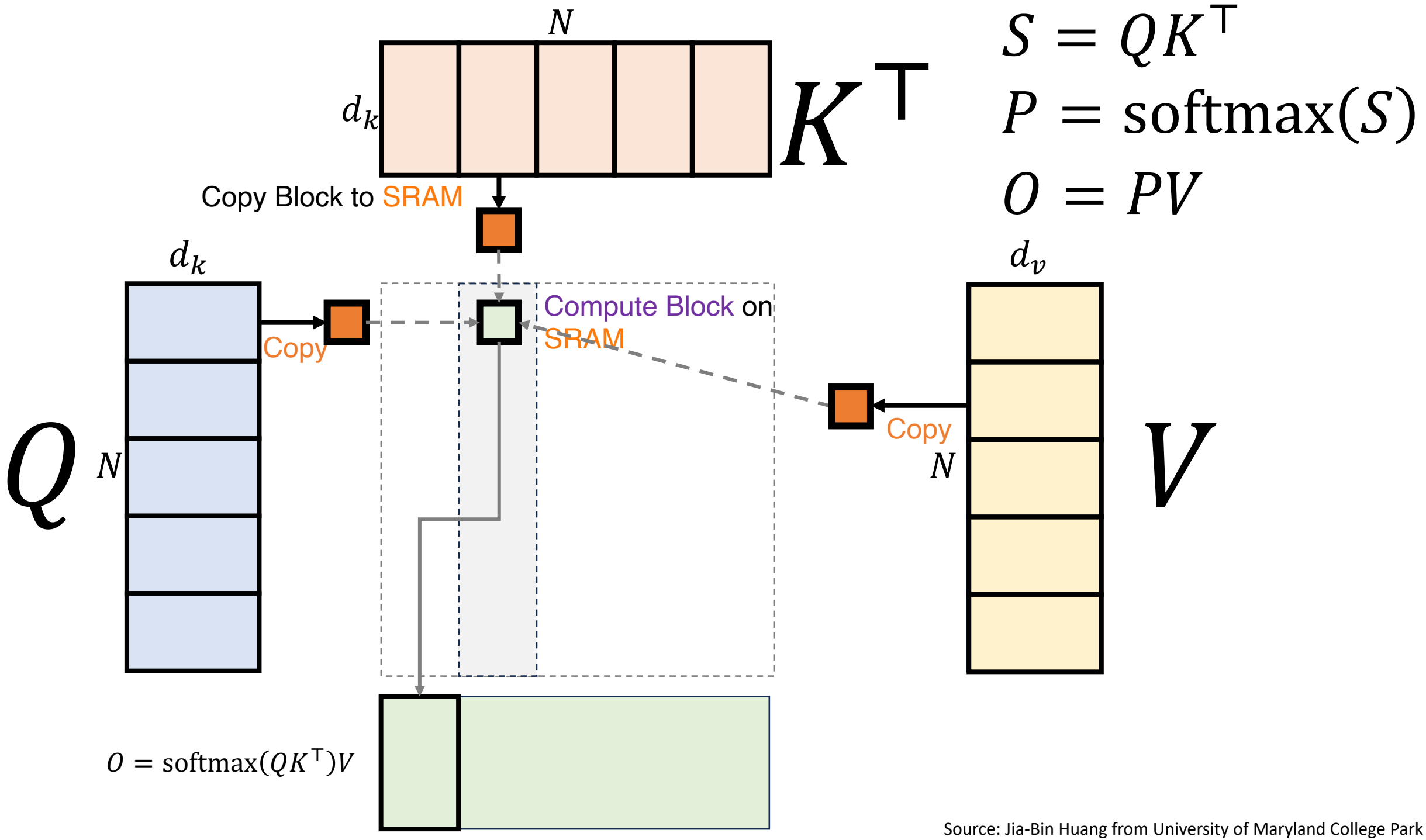
Source: Jia-Bin Huang from University of Maryland College Park



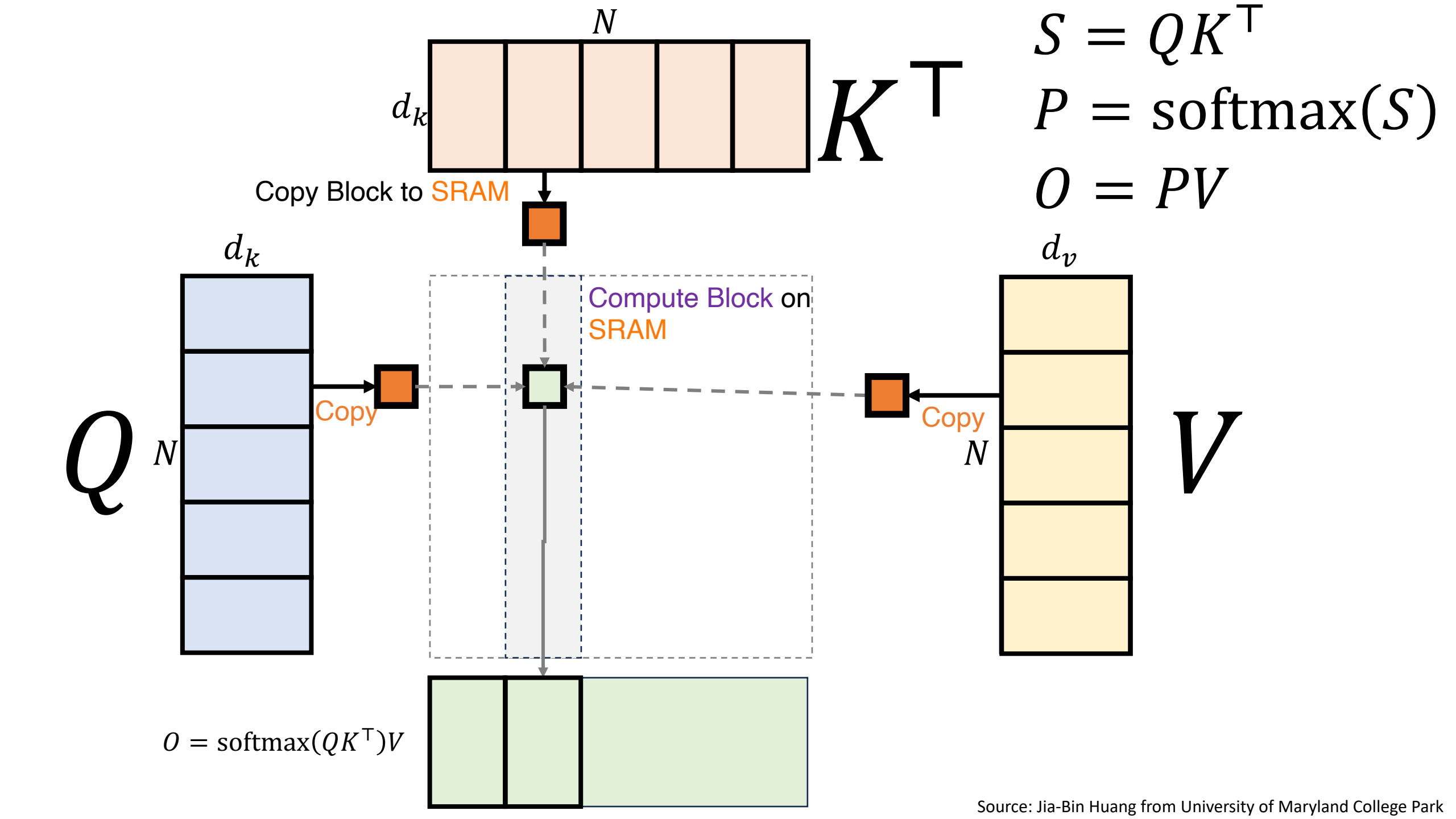
Source: Jia-Bin Huang from University of Maryland College Park



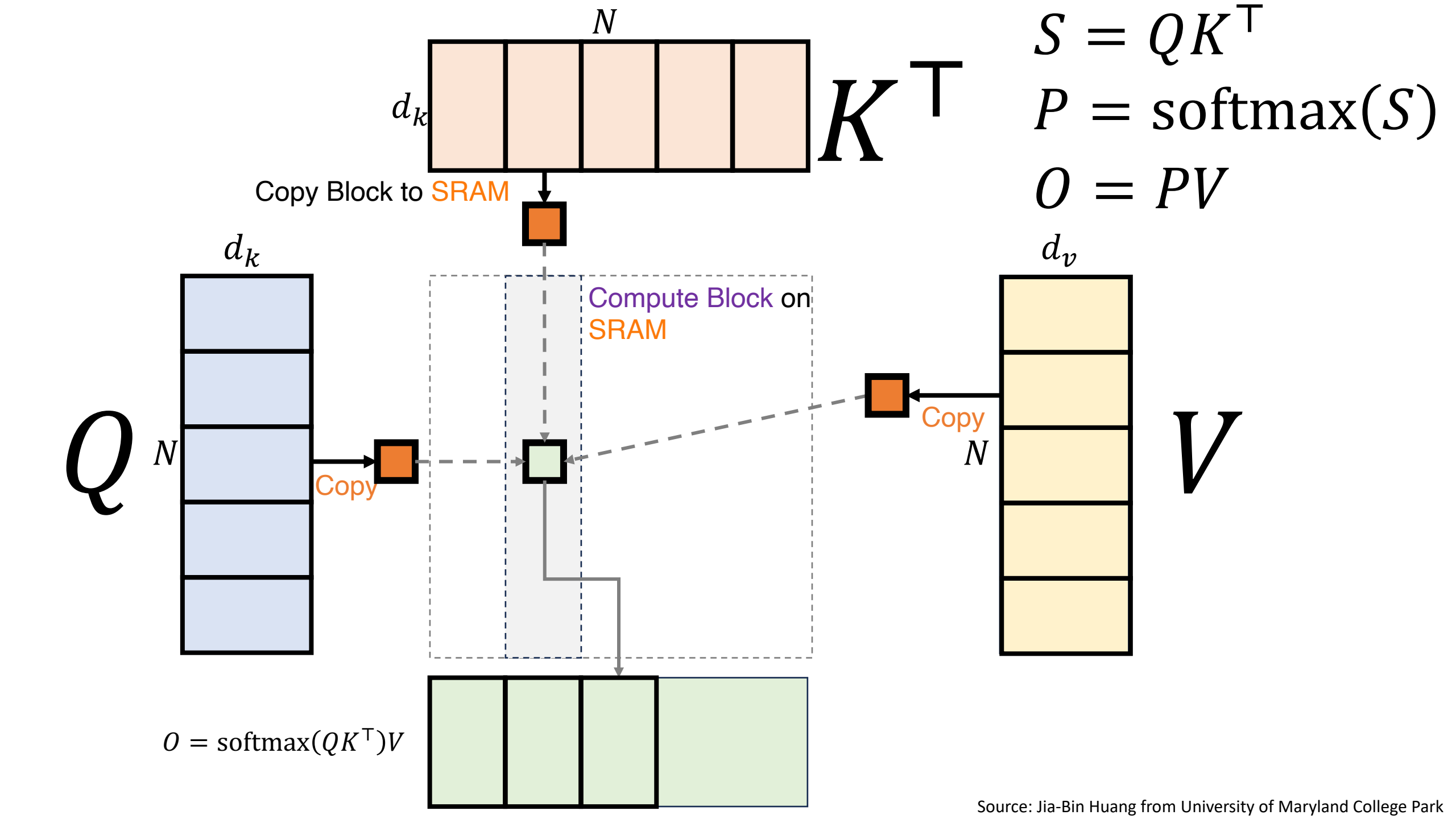
Source: Jia-Bin Huang from University of Maryland College Park



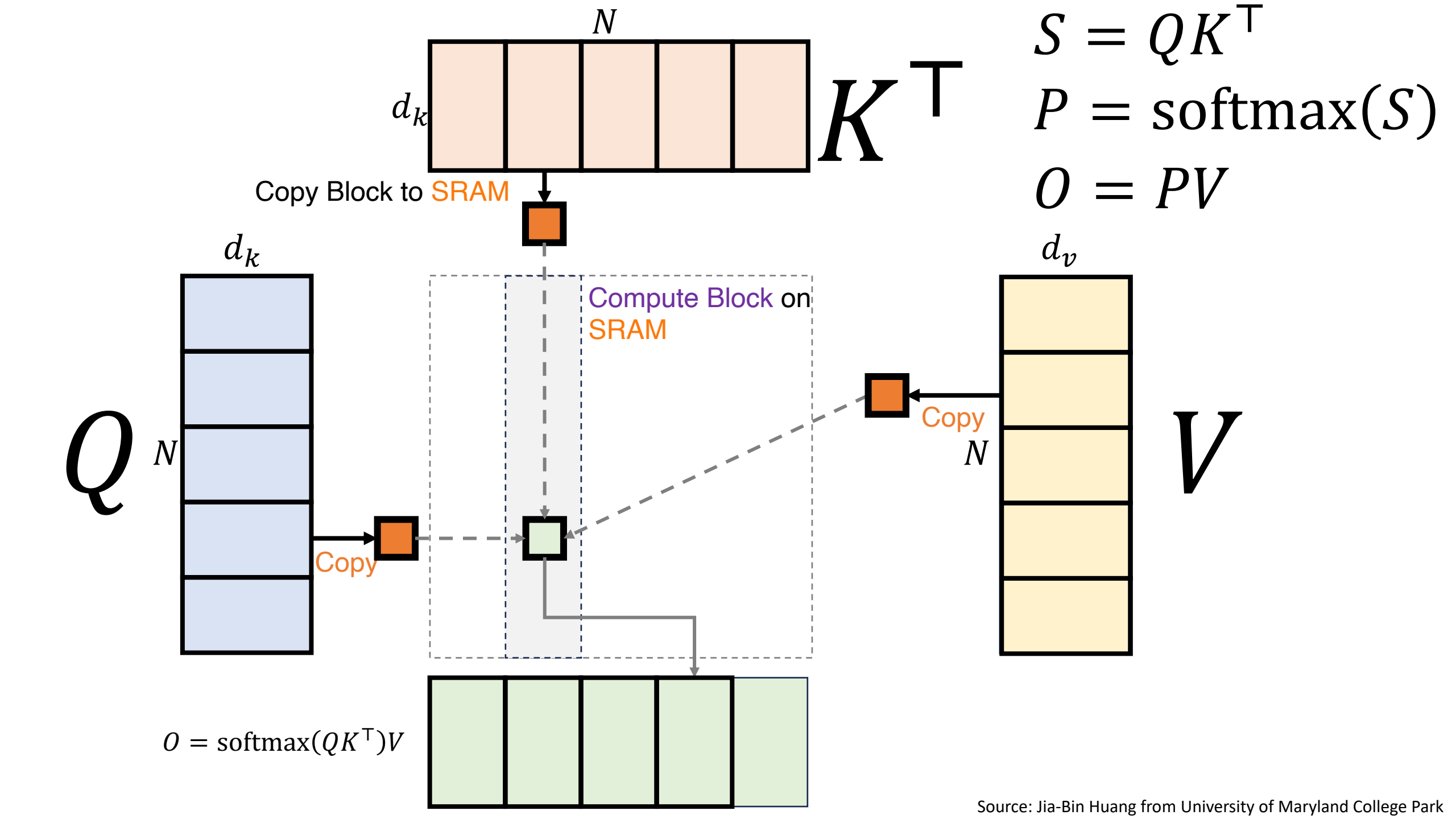
Source: Jia-Bin Huang from University of Maryland College Park



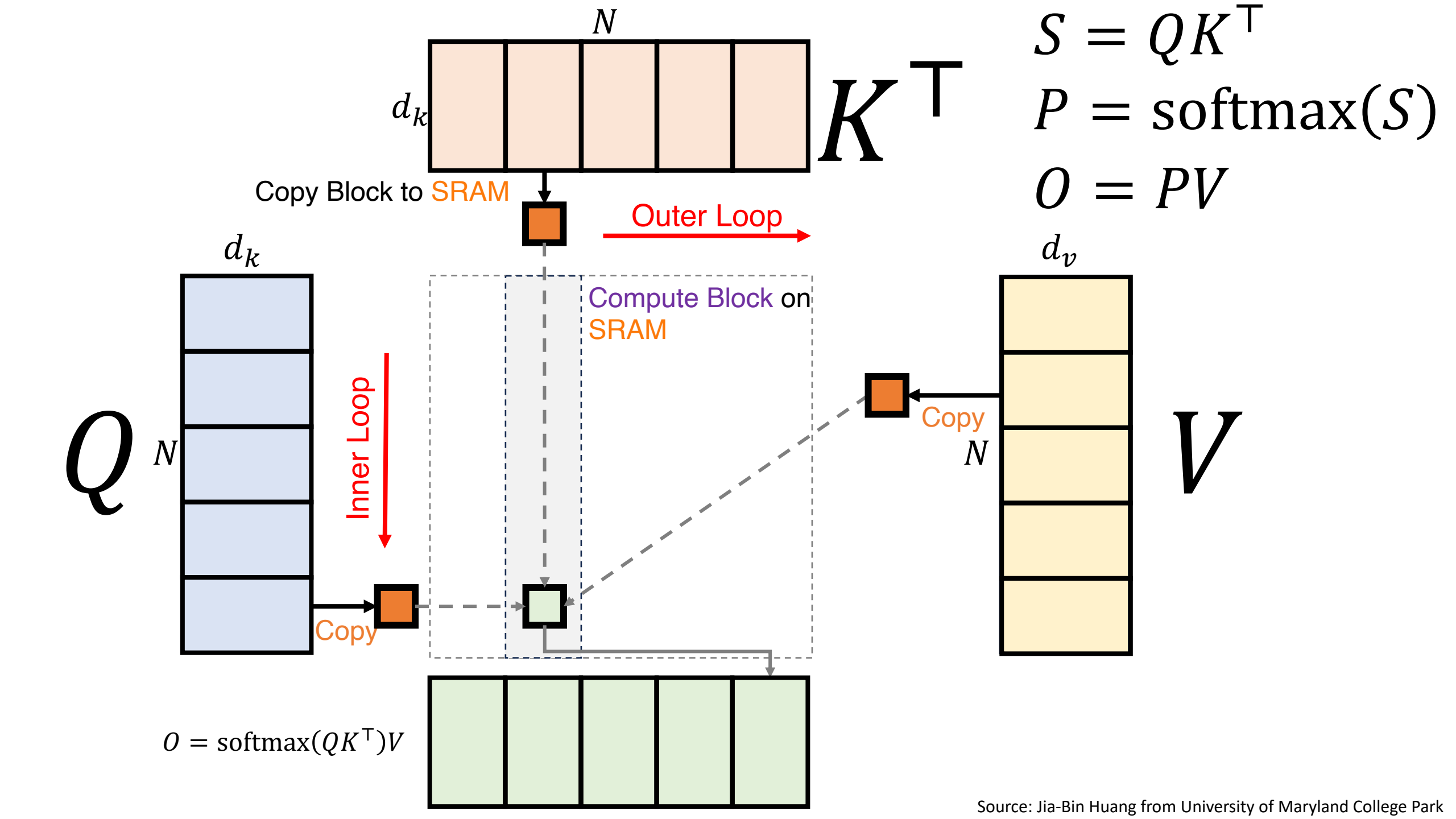
Source: Jia-Bin Huang from University of Maryland College Park



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



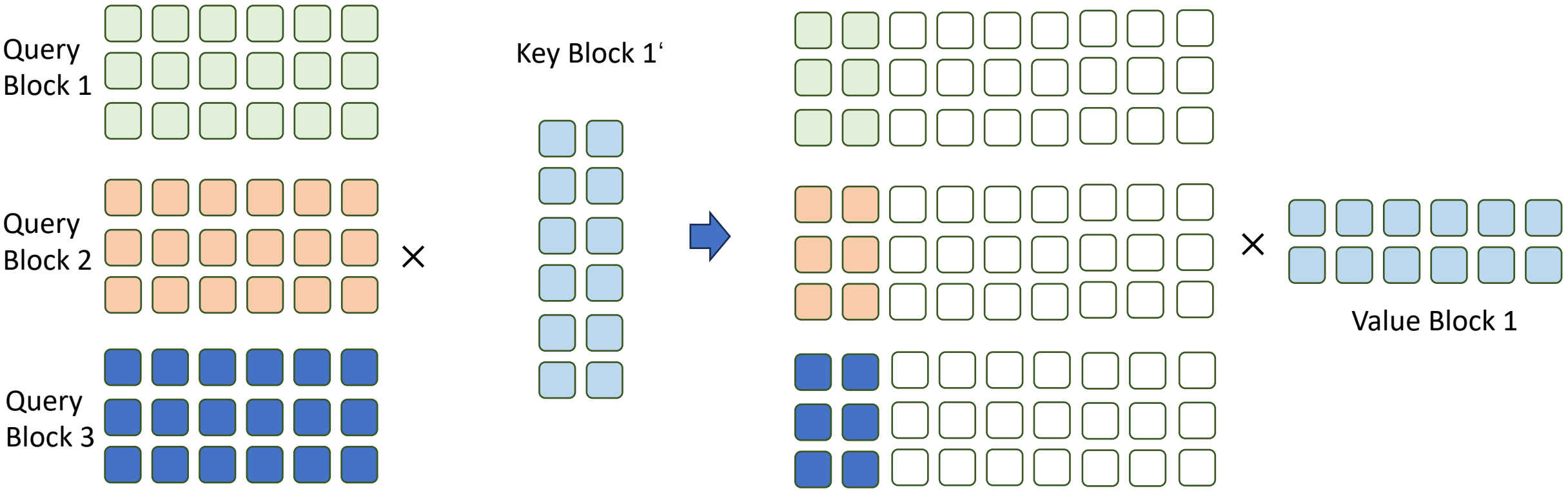
Source: Jia-Bin Huang from University of Maryland College Park



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

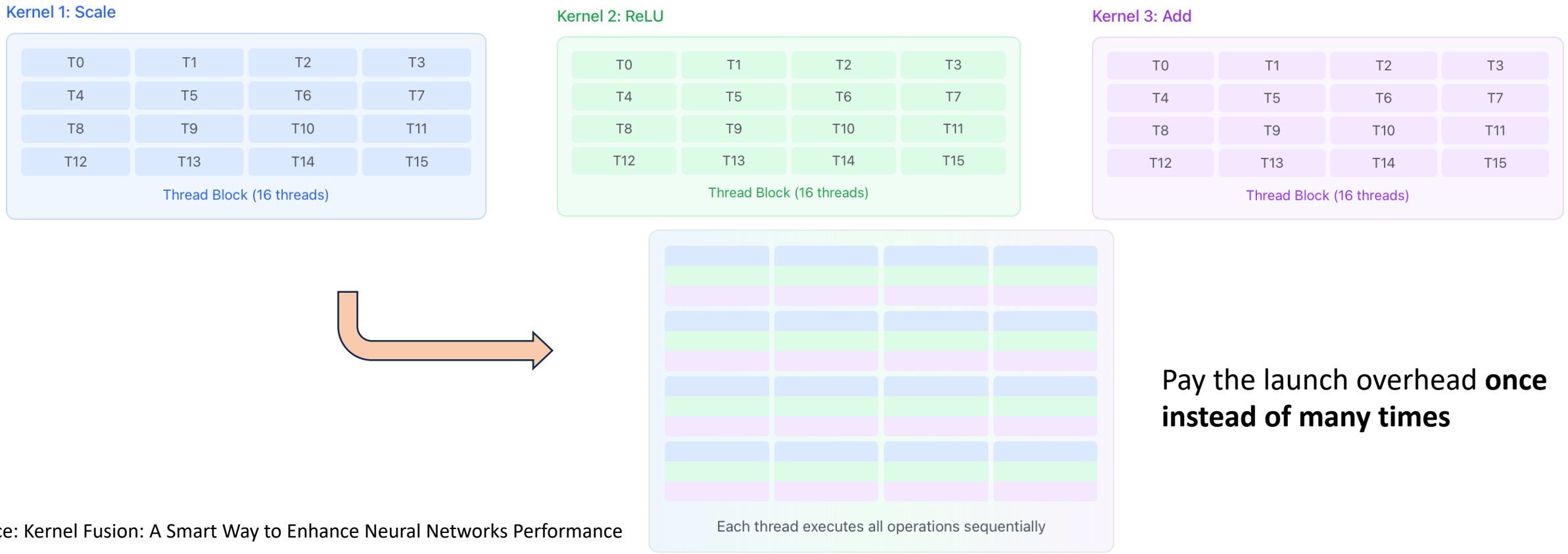
Flash Attention

- Flash Attention (Tiling)



Flash Attention

- Kernel Fusion
  - Multiple separate operations or kernels (small, independent programs that run on hardware like GPUs) are combined or "fused" into a single, more efficient kernel.

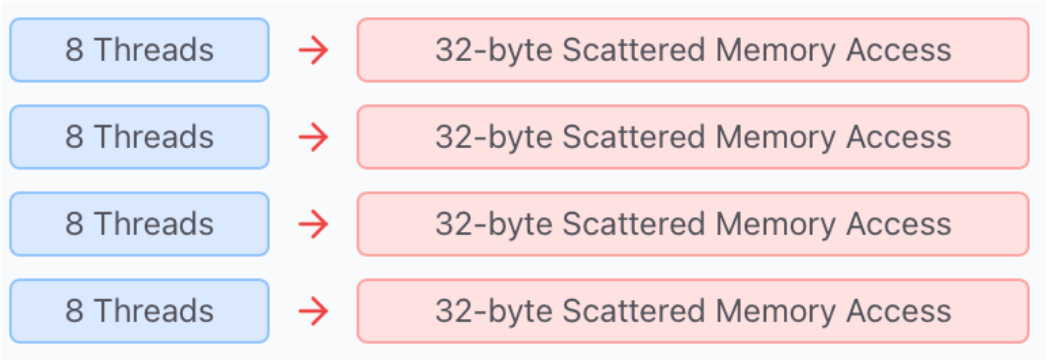


Pay the launch overhead **once instead of many times**

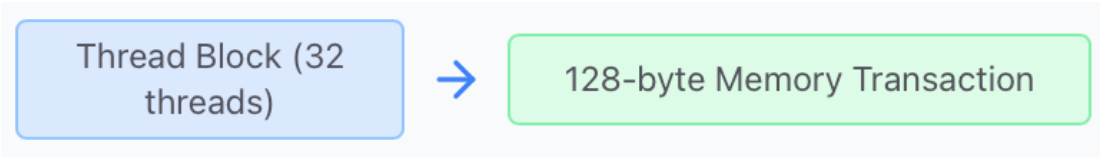
Source: Kernel Fusion: A Smart Way to Enhance Neural Networks Performance

Flash Attention

- Kernel Fusion
  - Minimizing data movement between slow global memory and faster on-chip memory (like SRAM or registers), avoids storing intermediate results



Multiple smaller memory transactions

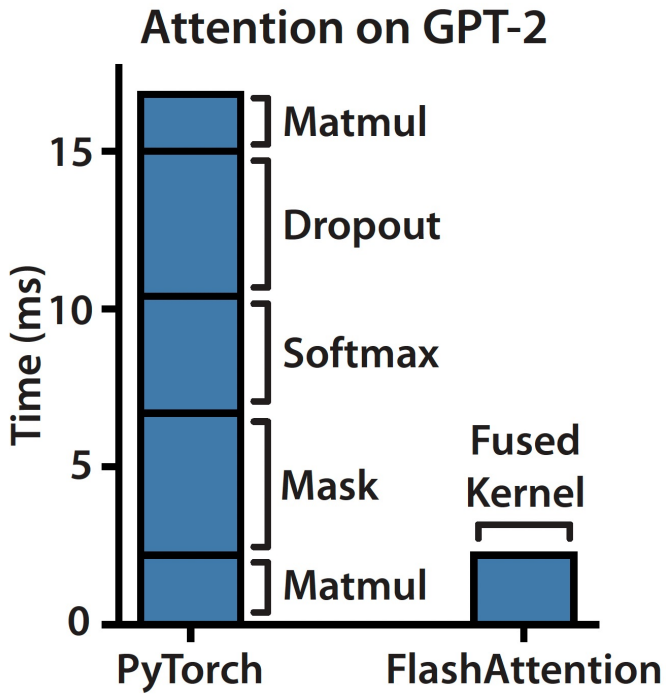


Single memory transaction for 32 consecutive elements

Source: Kernel Fusion: A Smart Way to Enhance Neural Networks Performance

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



6 - 7 × improvement

Flash Attention

- Complexity Analysis
  - GPT2 Medium (seq. length 1024, head dim. 64, 16 heads, batch size 64) on A100

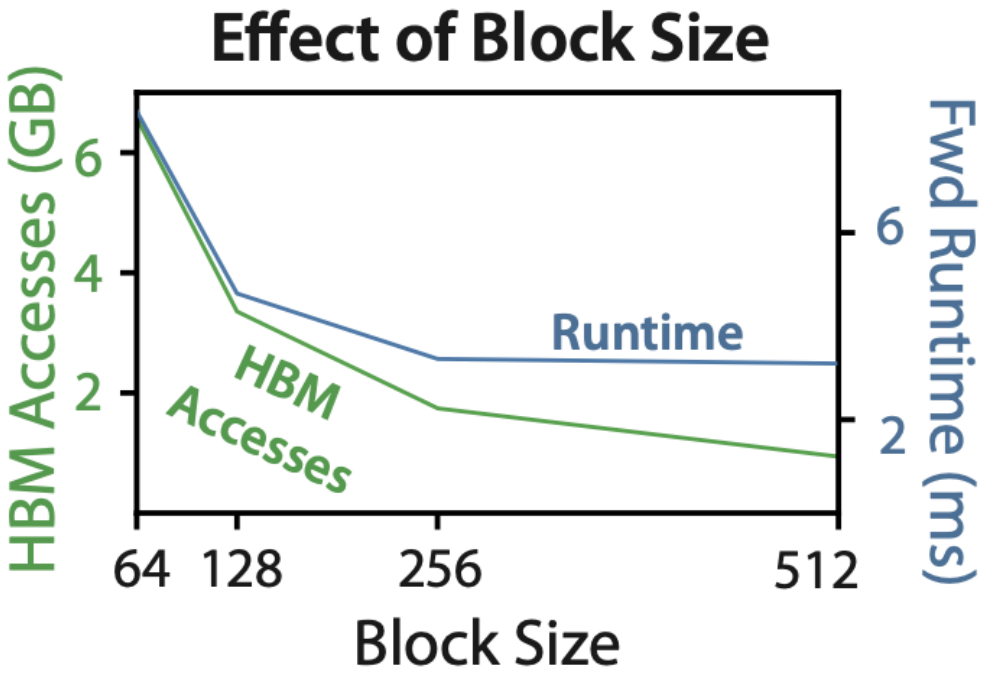
Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

  - Computational complexity: Flash Attention is slightly higher
  - Memory complexity: Flash Attention is significantly better $O(Nd)$
  - I/O complexity
    - Standard Attention $O(N^2d)$ versus Flash Attention $O(N^2d^2M^{-1})$
      - Outer loop runs $T_c = N/B_c = 4d/M$ where M is SRAM size of a SM, and reads K and V once
      - Inner loop reads Q, O, and write O back to HBM
      - Total I/O complexity: $O(Nd) \times T_c = O(N^2d^2M^{-1})$

Flash Attention

- Flash Attention in practice
  - Impact of hyper parameters:

    Larger block size reduces the number of data loading, thus improving the forward runtime

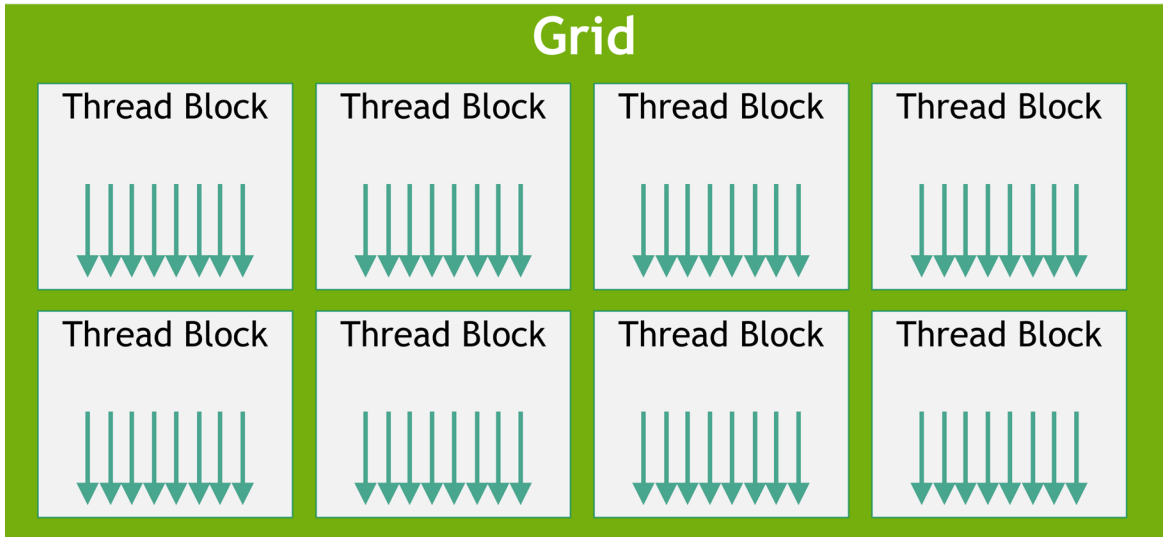


Forward runtime of FlashAttention

  - As hidden dimension becomes large, the block size shrinks
    - $O(N^2 d^2 M^{-1})$ is quadratic to the hidden dimension d, while M is constrained by the hardware

Flash Attention

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

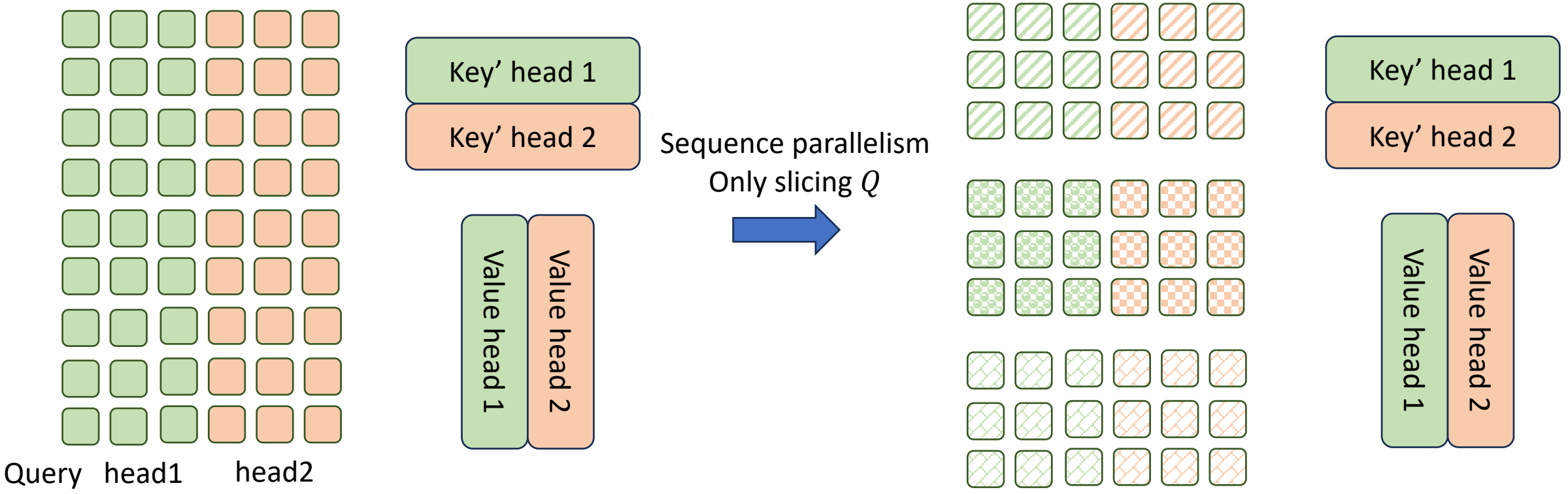


Grid of Thread Blocks

- 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 16 out of 108 SMs are used on A100

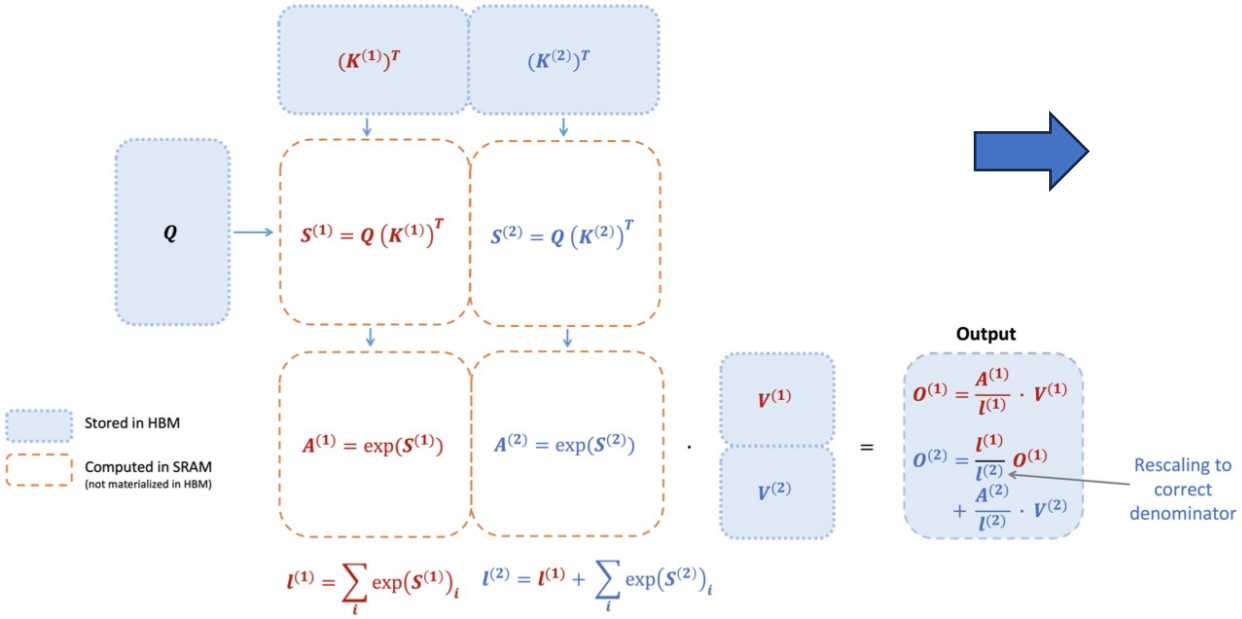
Flash Attention

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



Flash Attention

- Flash Attention v2: Loop Reordering



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

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

Source: Flash Attention v2

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

$$
\begin{aligned}
S_{ij} &\leftarrow Q_i K_j^T \\
m_{ij} &\leftarrow \mathrm{rowmax}(S_{ij}) \\
m_i^{new} &\leftarrow \max(m_i, m_{ij}) \\
P_{ij} &\leftarrow \exp(S_{ij} - m_i^{new}) \\
l_{ij} &\leftarrow \mathrm{rowsum}(P_{ij}) \\
l_i^{new} &\leftarrow \exp(m_i - m_i^{new}) l_i + l_{ij} \\
O_i^{new} &\leftarrow (l_i^{new})^{-1} (l_i \exp(m_i - m_i^{new}) O_i + P_{ij} V_j)
\end{aligned}
$$

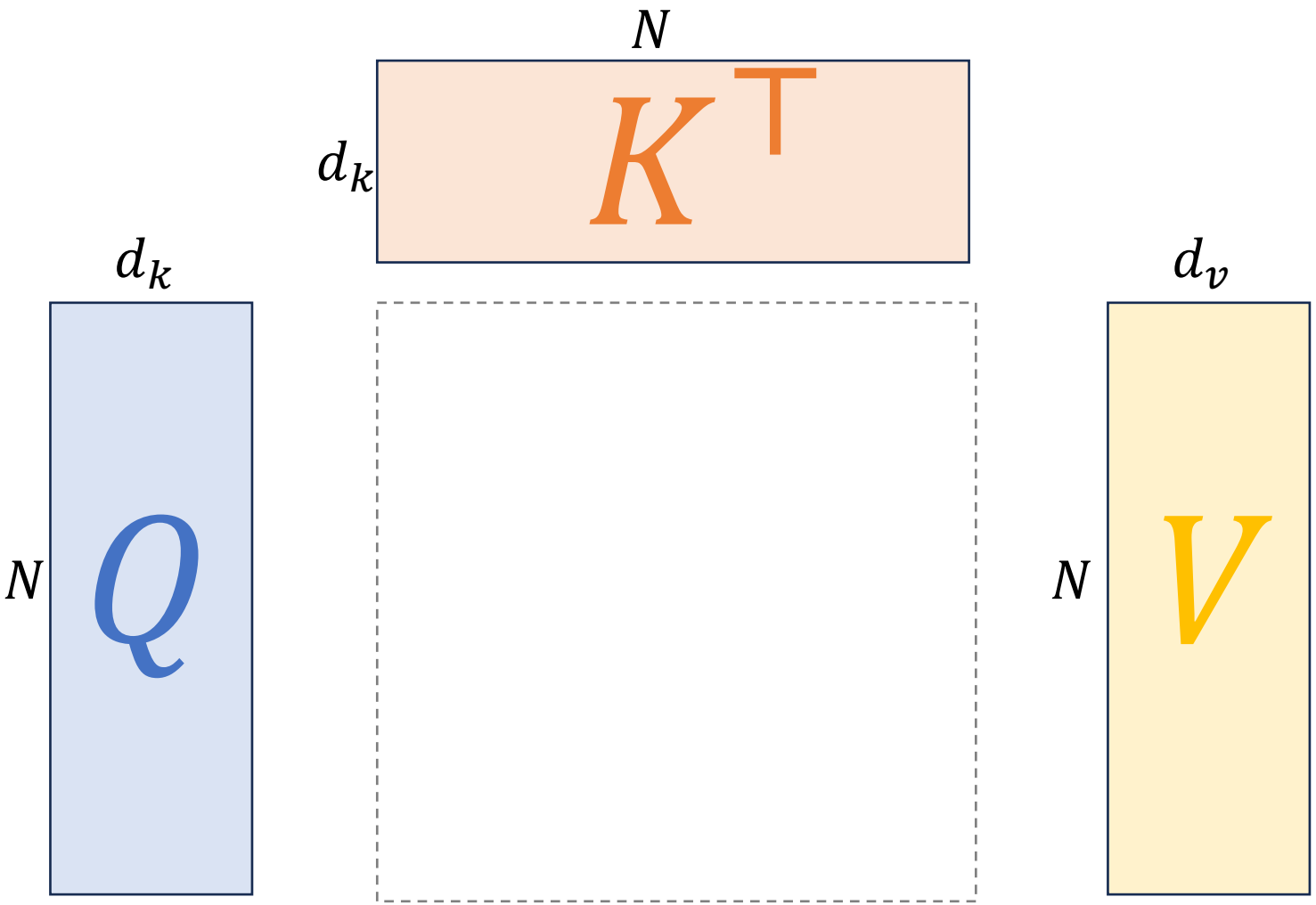
Scaling at next round

Scaling at this round in FlashAttention v1

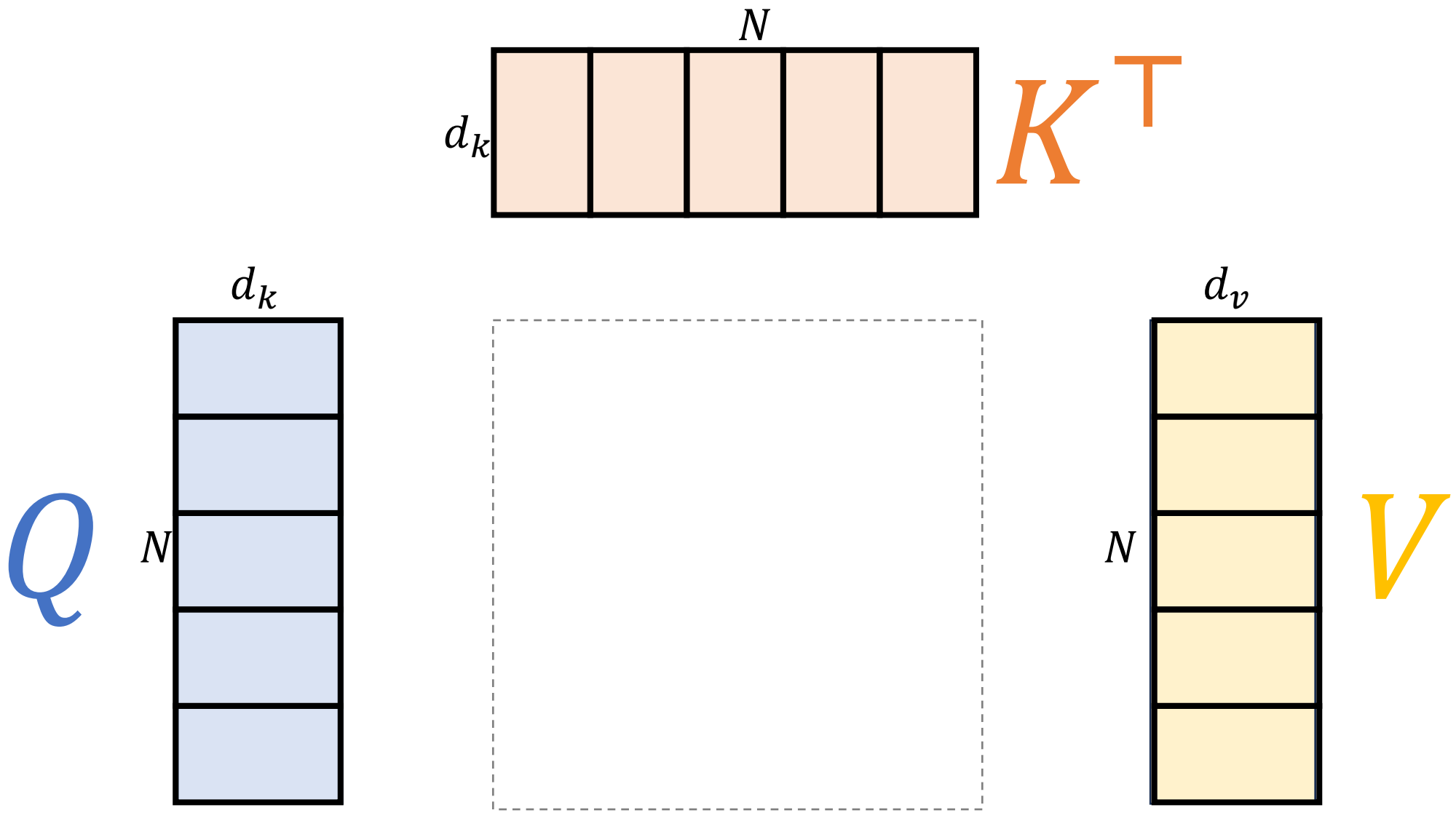
$$
\begin{aligned}
\tilde{O}_i^{new} &\leftarrow \exp(m_i - m_i^{new}) \tilde{O}_i + P_{ij} V_j \\
O_i^{last} &\leftarrow (l_i^{last})^{-1} \tilde{O}_i^{last}
\end{aligned}
$$

Only scaling the final result in FlashAttention v2

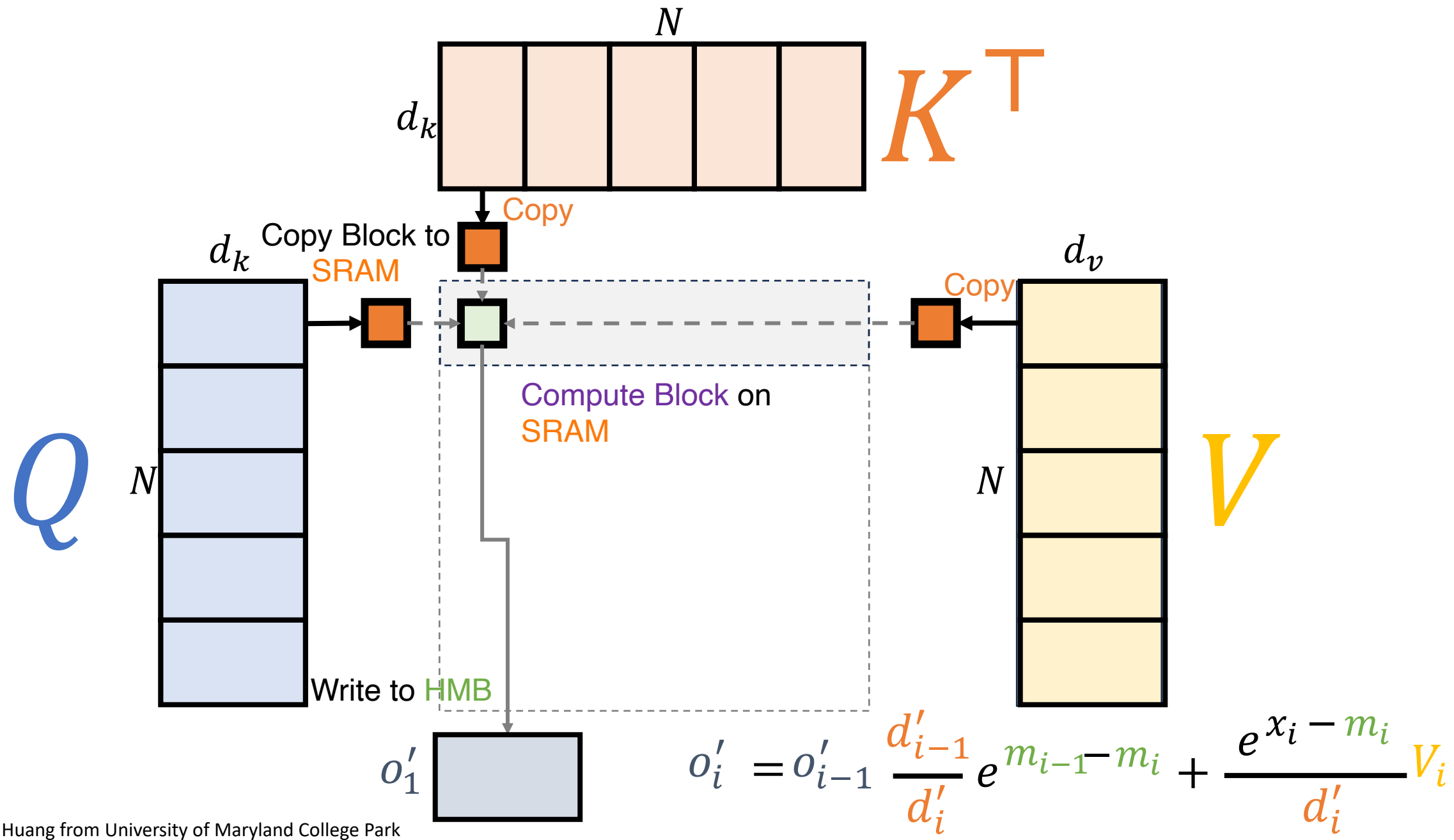
FlashAttention v2: reduce the number of rescaling ops, as well as bound-checking and causal masking operations



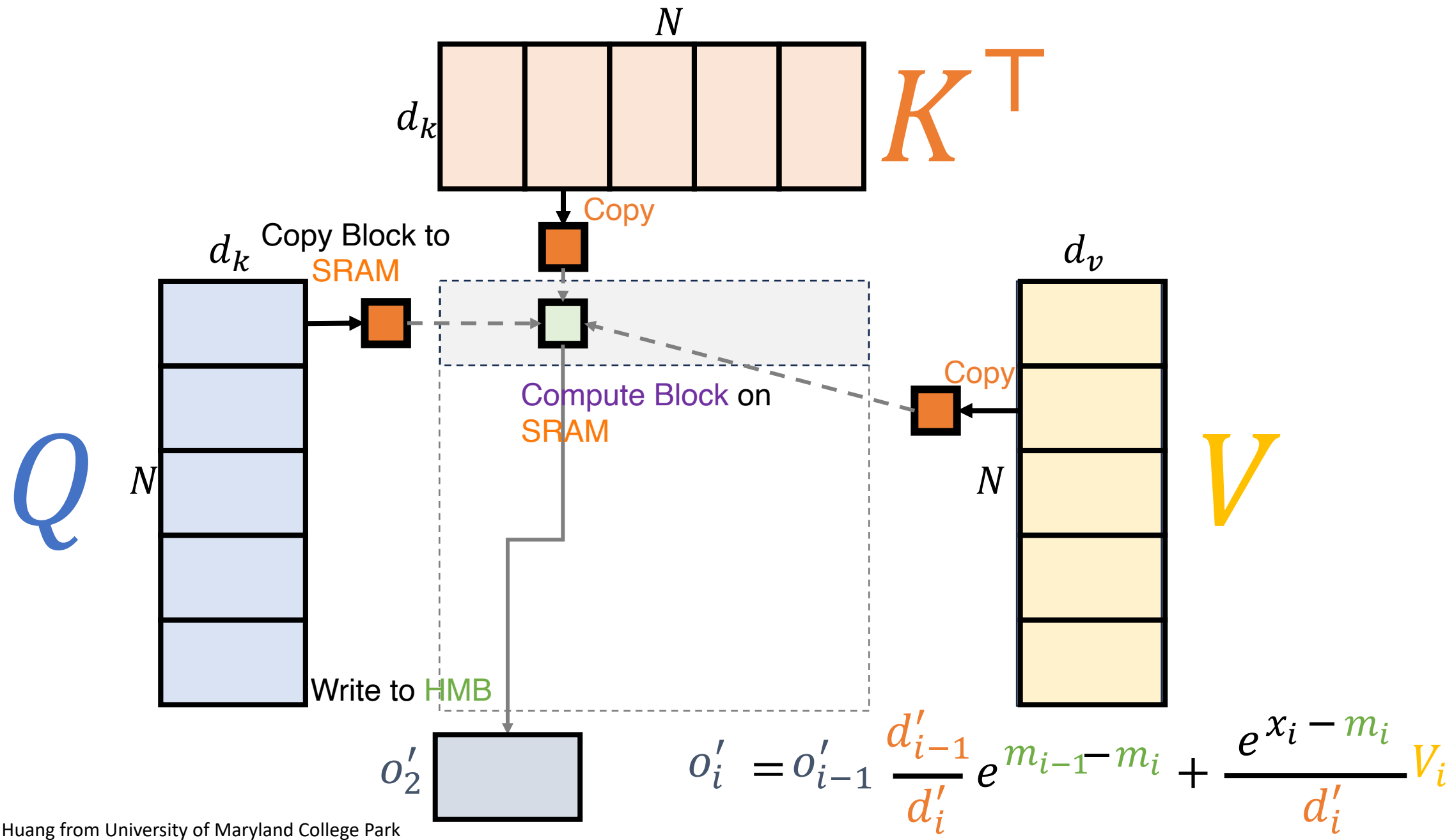
Source: Jia-Bin Huang from University of Maryland College Park



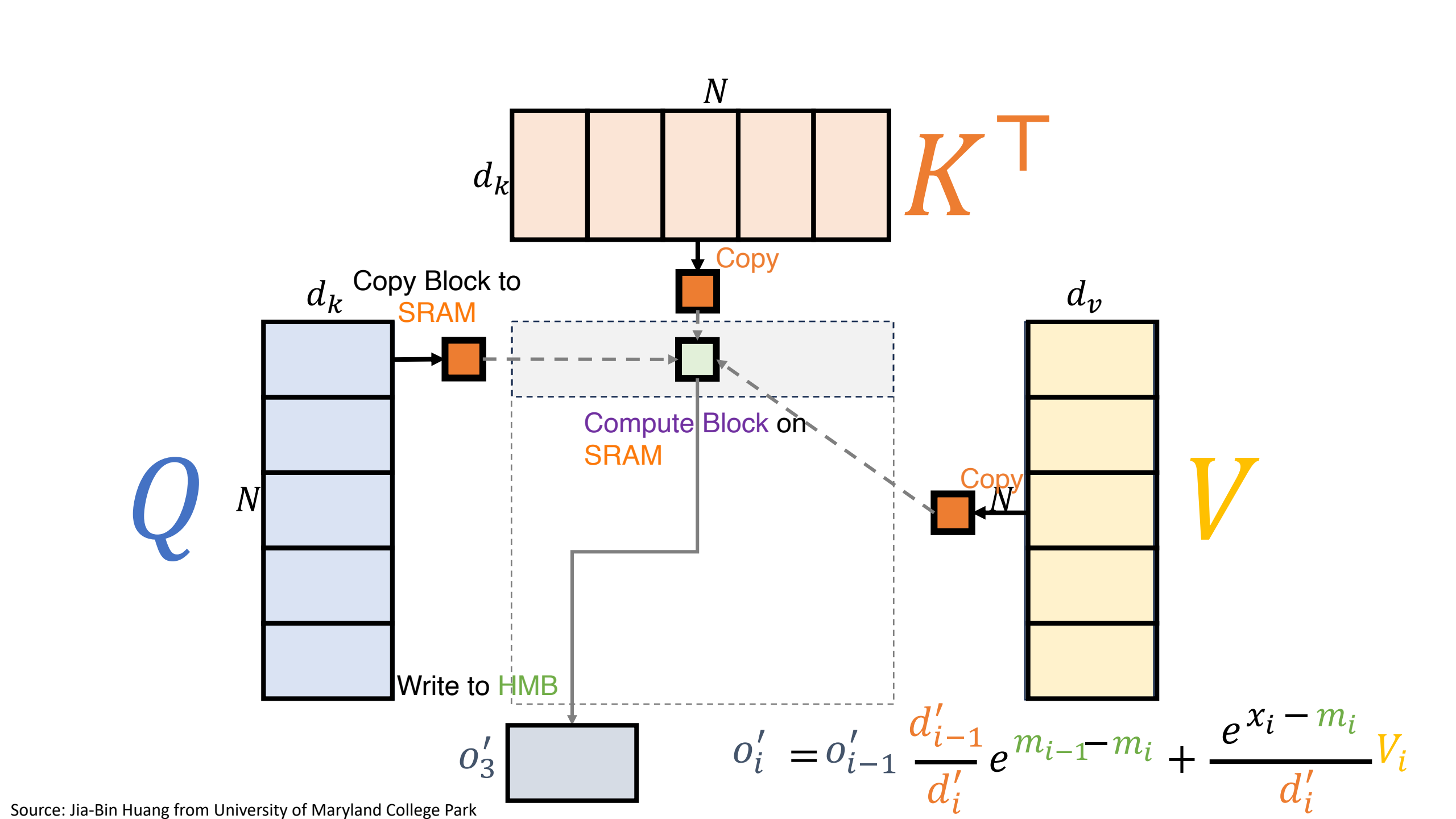
Source: Jia-Bin Huang from University of Maryland College Park



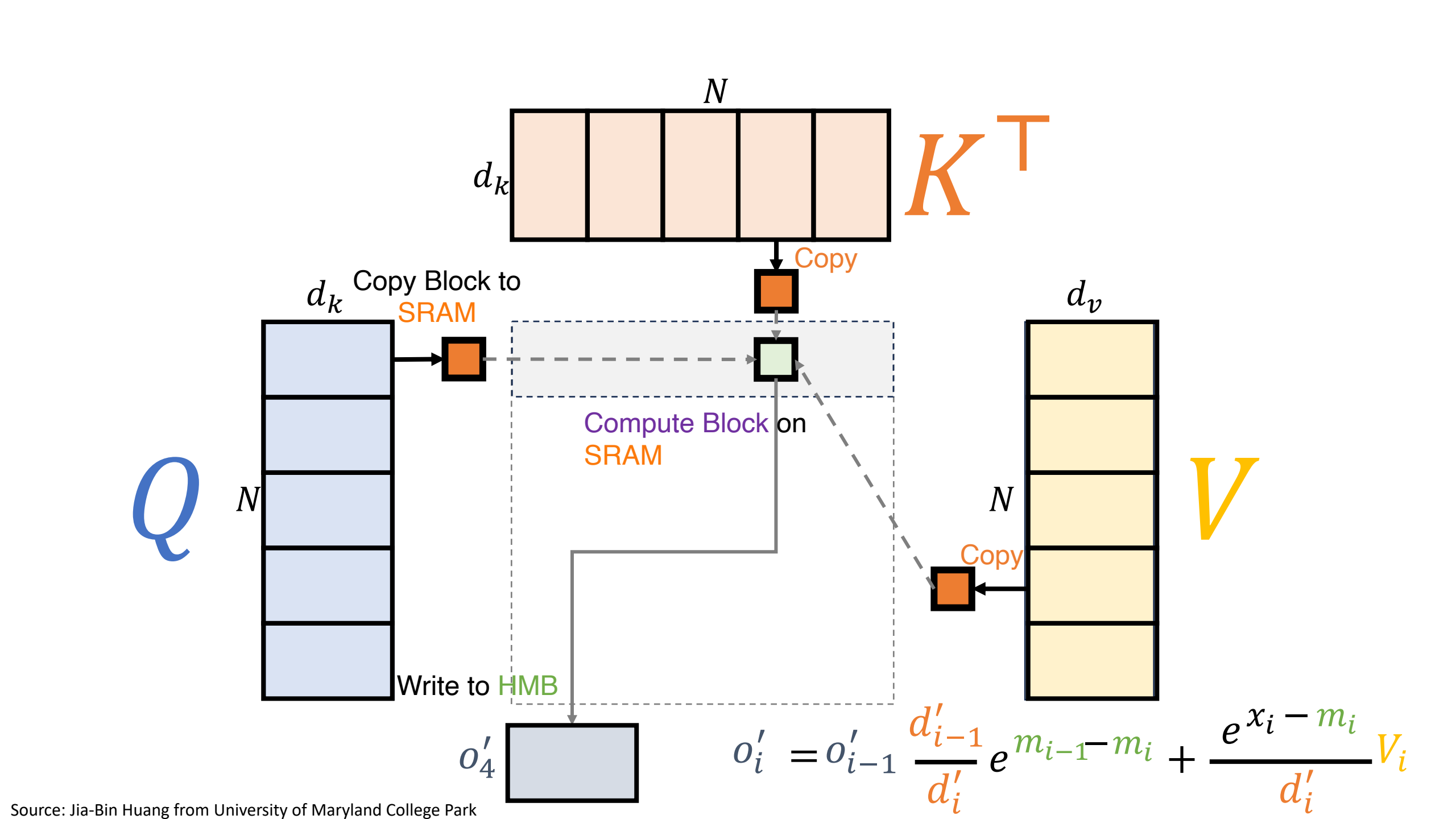
Source: Jia-Bin Huang from University of Maryland College Park



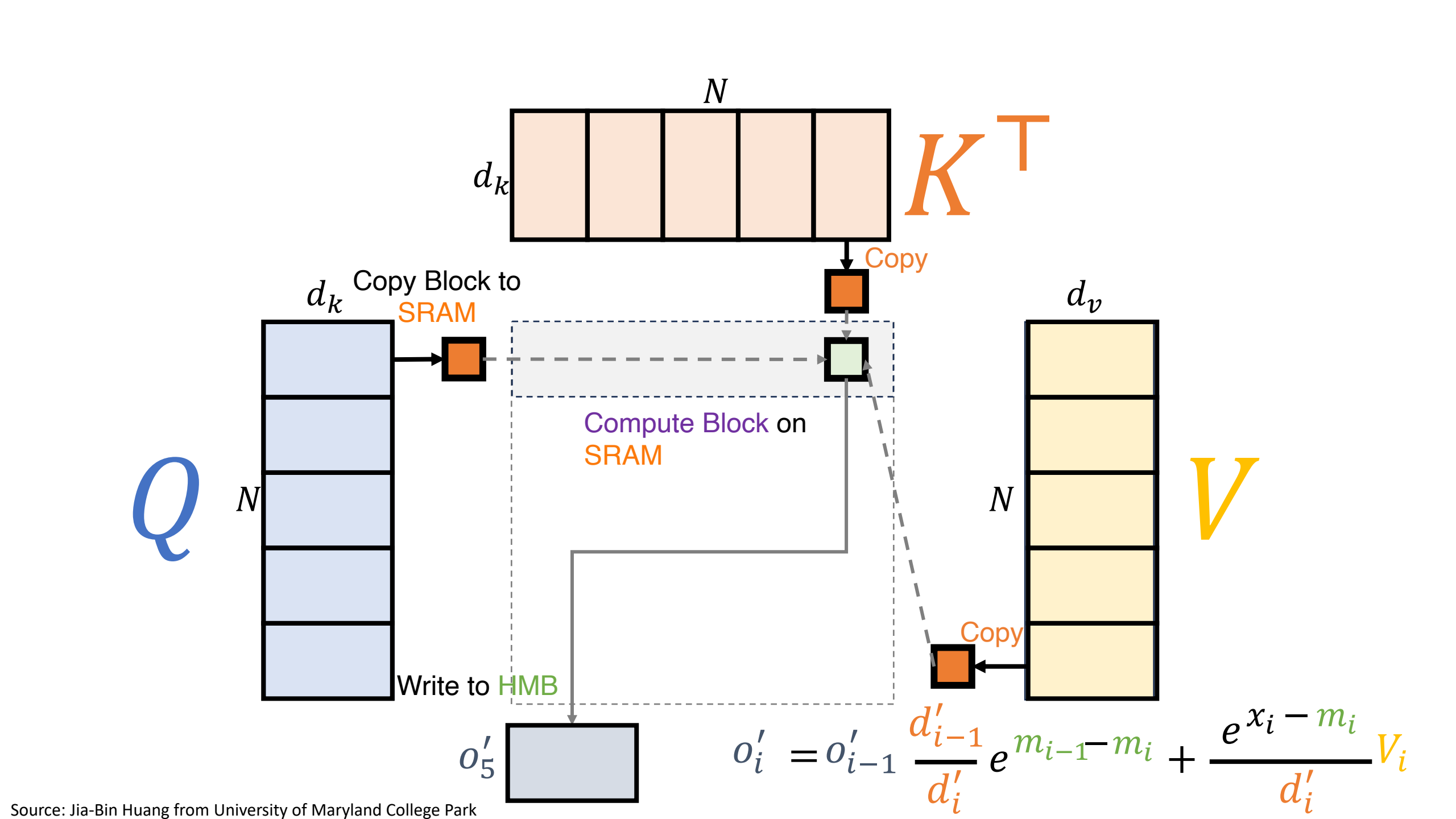
Source: Jia-Bin Huang from University of Maryland College Park



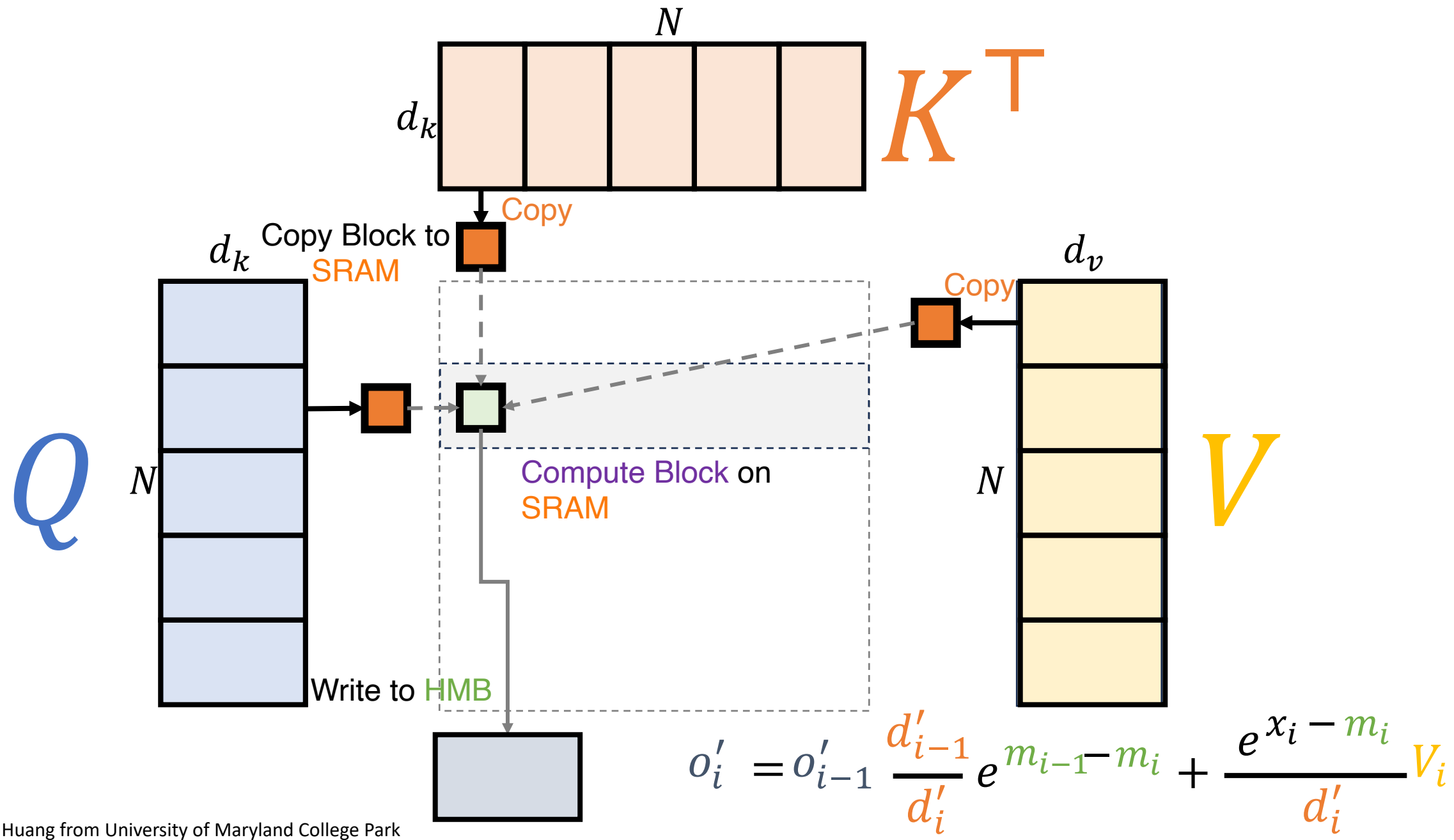
Source: Jia-Bin Huang from University of Maryland College Park



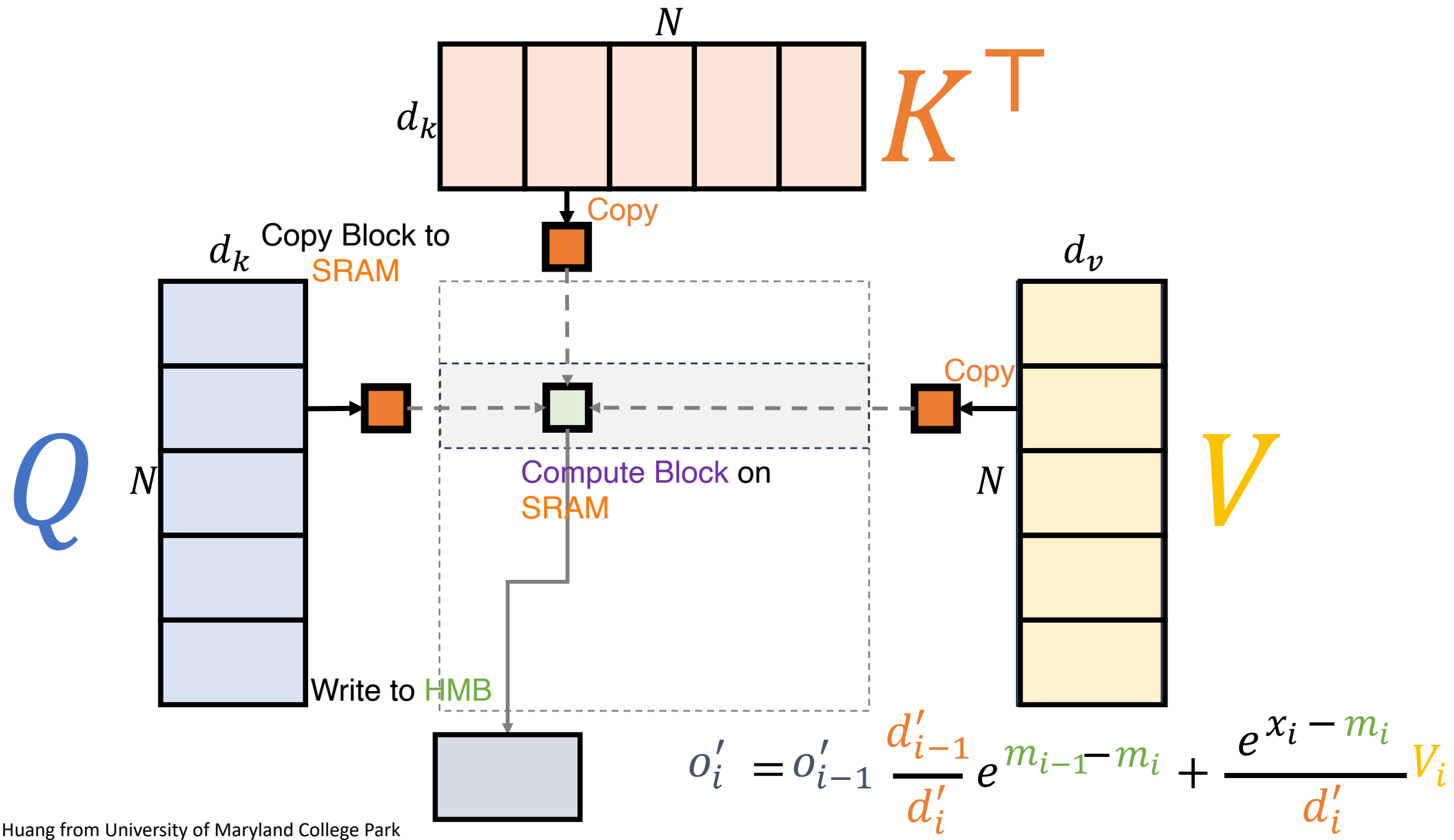
Source: Jia-Bin Huang from University of Maryland College Park



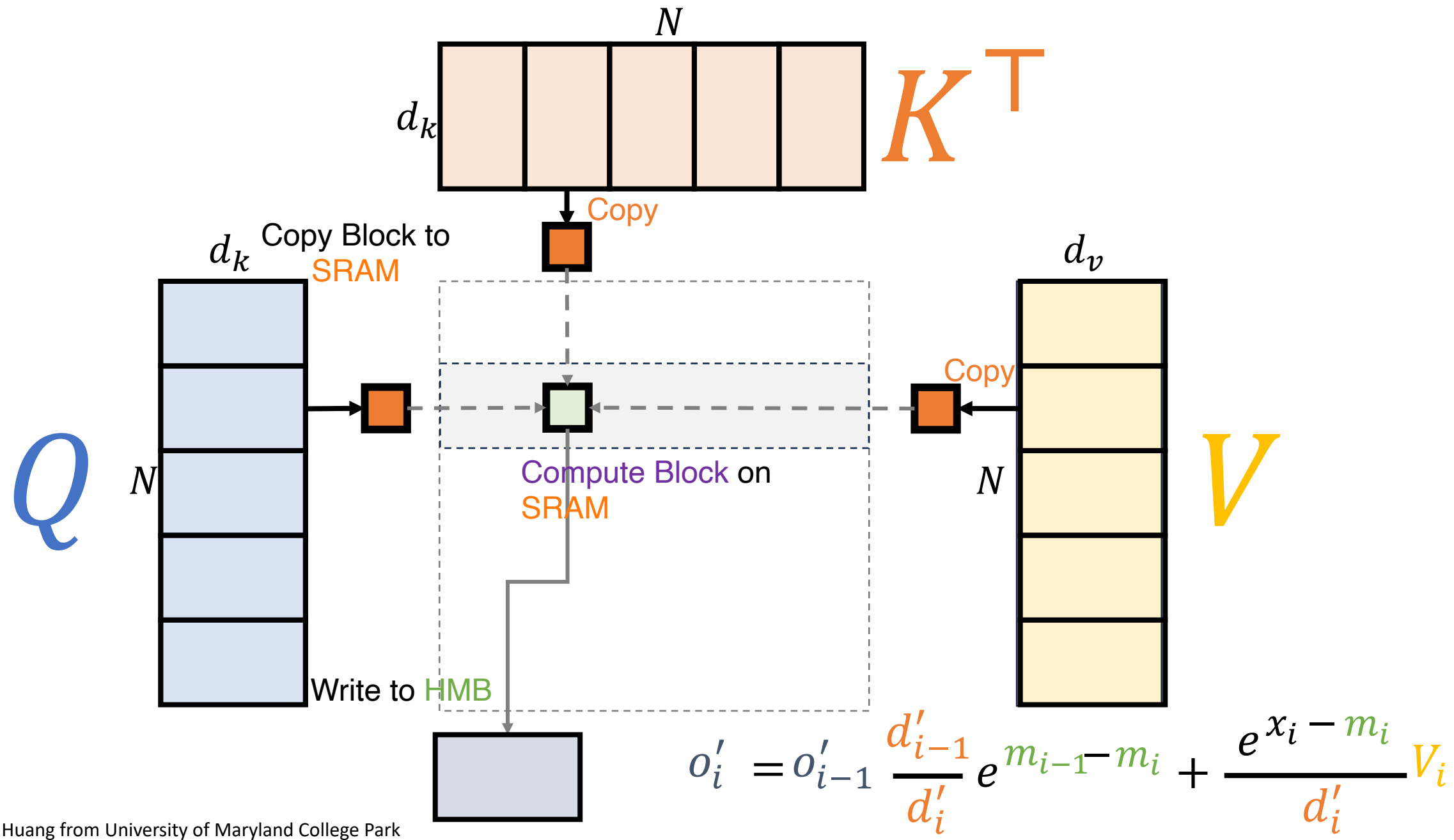
Source: Jia-Bin Huang from University of Maryland College Park



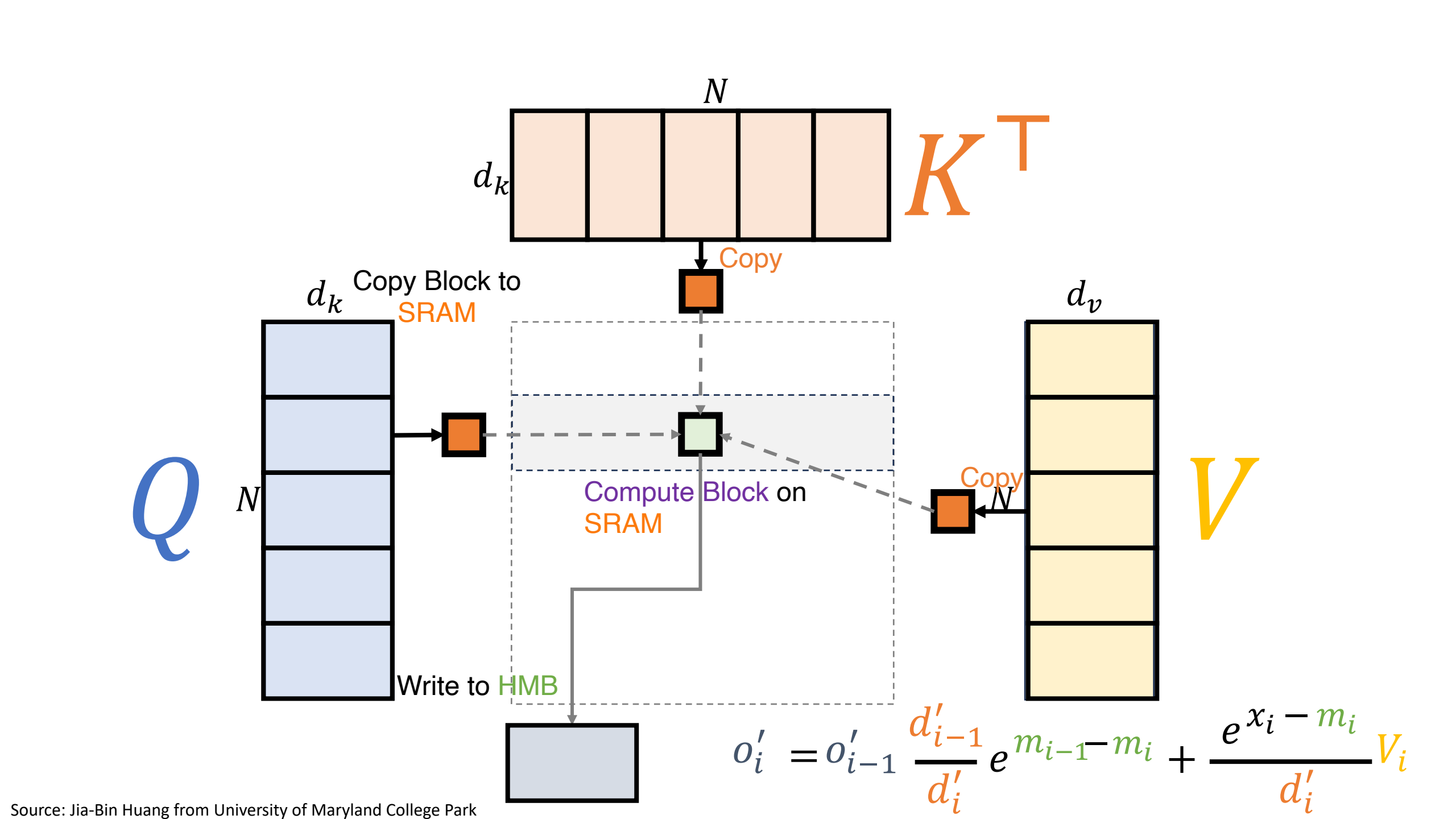
Source: Jia-Bin Huang from University of Maryland College Park



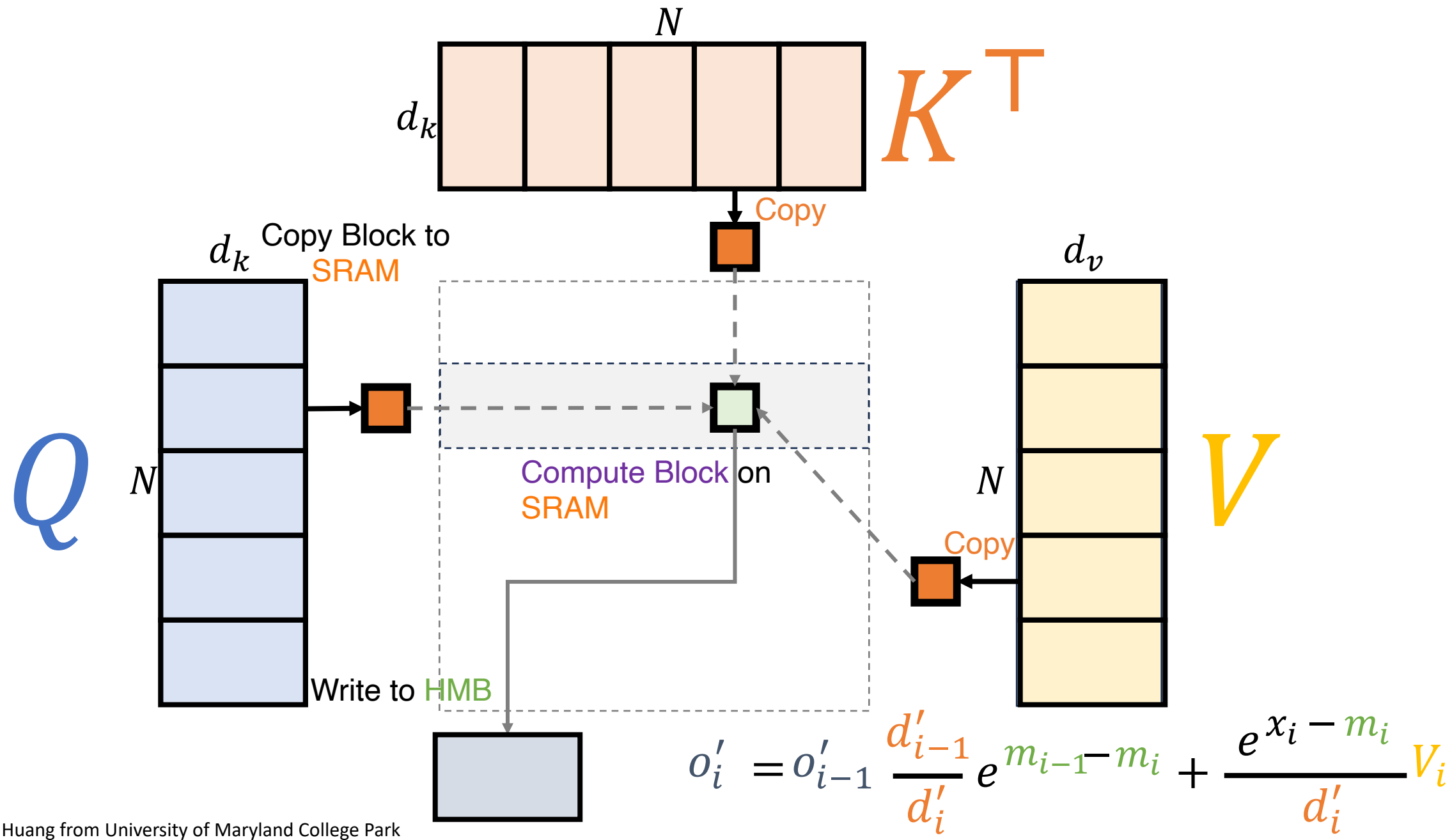
Source: Jia-Bin Huang from University of Maryland College Park



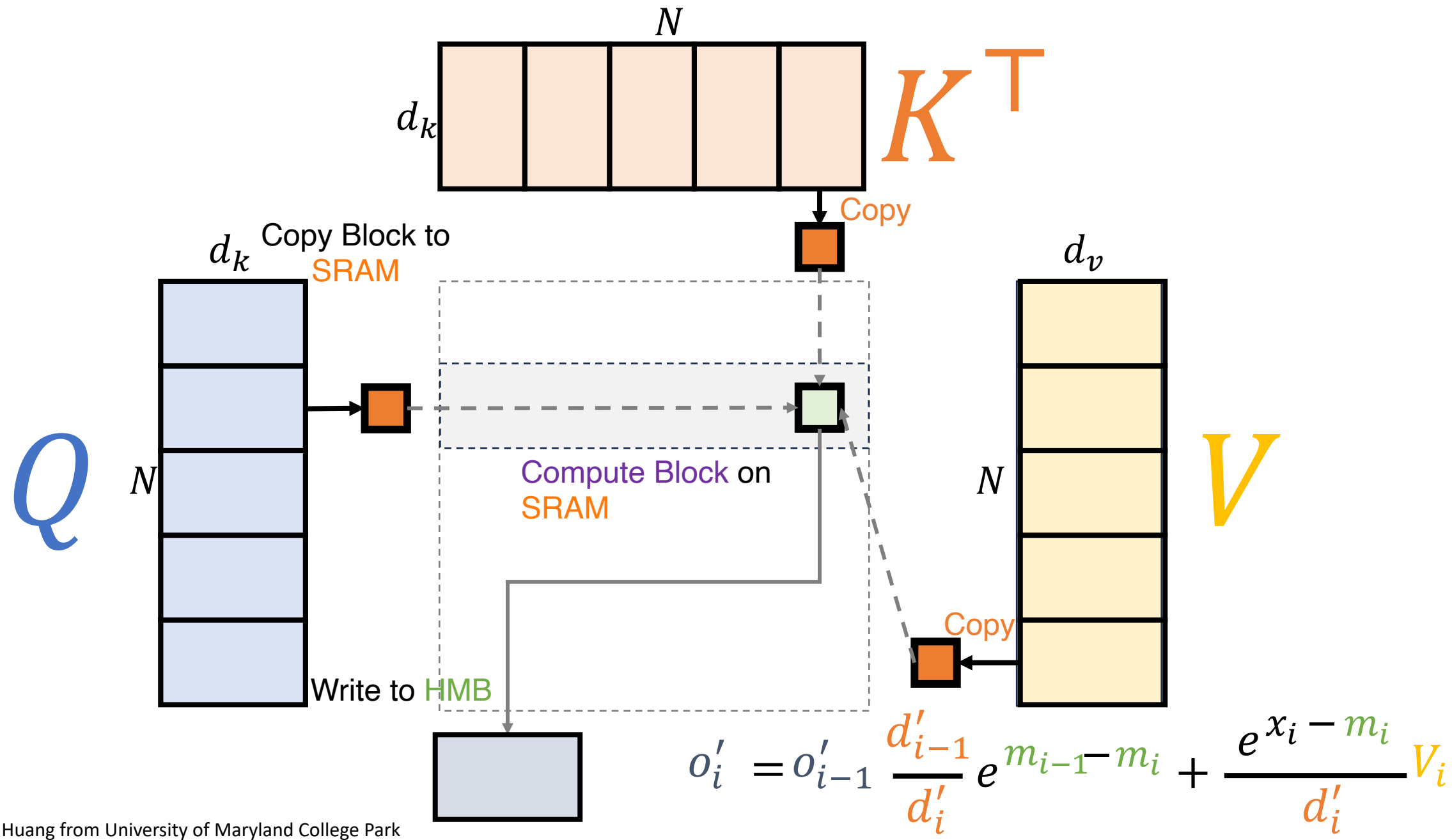
Source: Jia-Bin Huang from University of Maryland College Park



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

Algorithm 1 FLASHATTENTION-2 forward pass

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

1: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ 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.

2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \ldots, L_{T_r}$ of size B_r each.

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

4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.

5: On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.

6: **for** $1 \le j \le T_c$ **do**

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

8: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

9: On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

10: On chip, compute $\mathbf{O}_i^{(j)} = \text{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.

11: **end for**

12: On chip, compute $\mathbf{O}_i = \text{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.

13: On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.

14: Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.

15: Write L_i to HBM as the i-th block of L.

16: **end for**

17: Return the output $\mathbf{O}$ and the logsumexp L.

Outerloop: traversing Q submatrices first;

Innerloop: traversing K and V submatrices

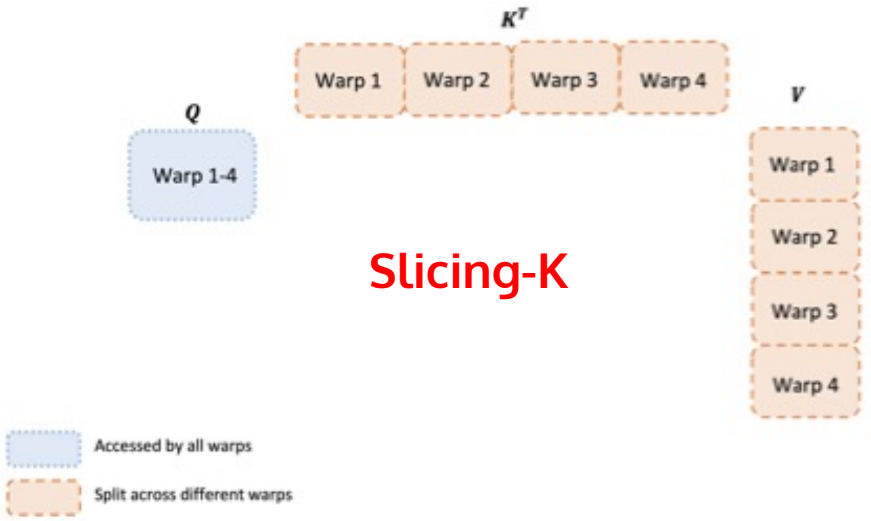
Write back to HBM after the inner-loop: reducing HBM read/write compared with v1

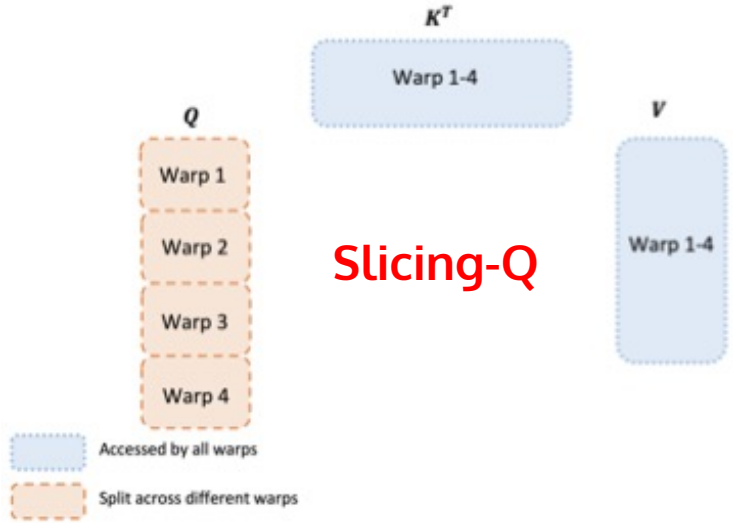
Flash Attention v2: Loop Reordering

- Maintaining the same order of I/O complexity: $O(N^2 d^2 M^{-1})$
- Reducing read/write of O_i: write O_i to HBM after traversing all K/V blocks at the inner loop

Flash Attention

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



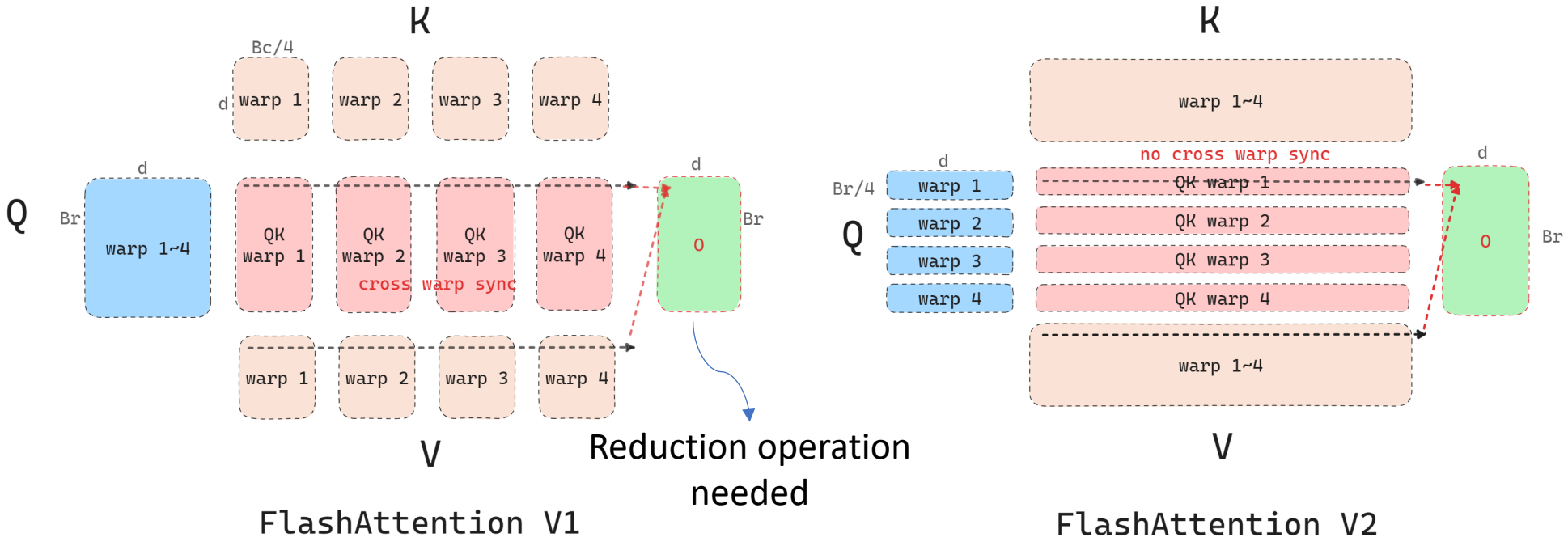


- Slicing K/V at each Warp while maining complete Q
- QK^T has four partitions, and $(QK^T)V$ needs REDUCTION (write to shared memory, synchronize, and add-up intermediate results)

- 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

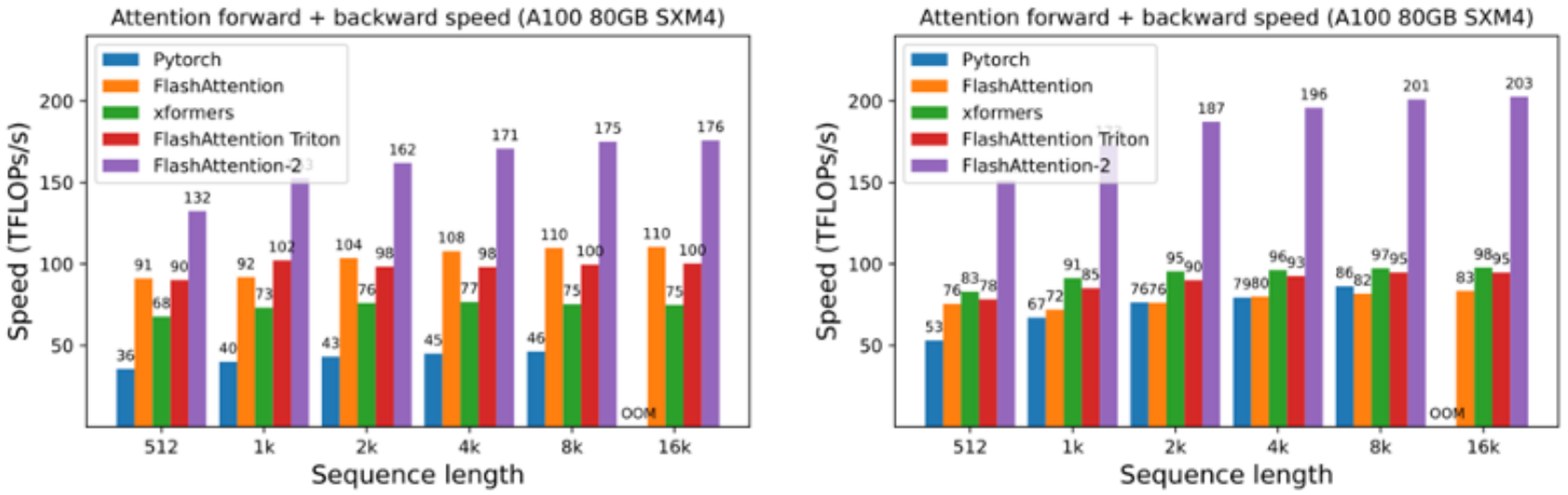
- FlashAttention v2
  - Better Work Partitioning
    - A thread block is partitioned into multiple warps (32 threads per warp, 4~8 warps per block)



By Zhihu blogger: Antinomi

Flash Attention

- FlashAttention v2
  - Performance speedup



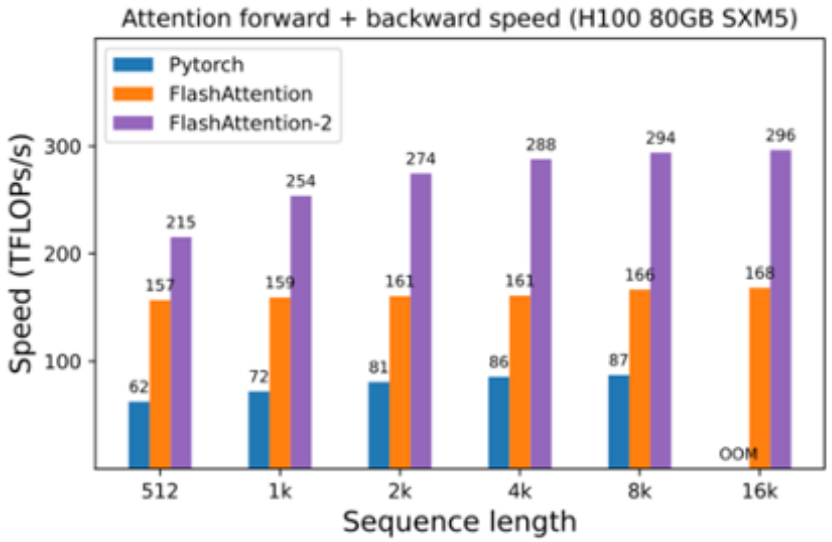
(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

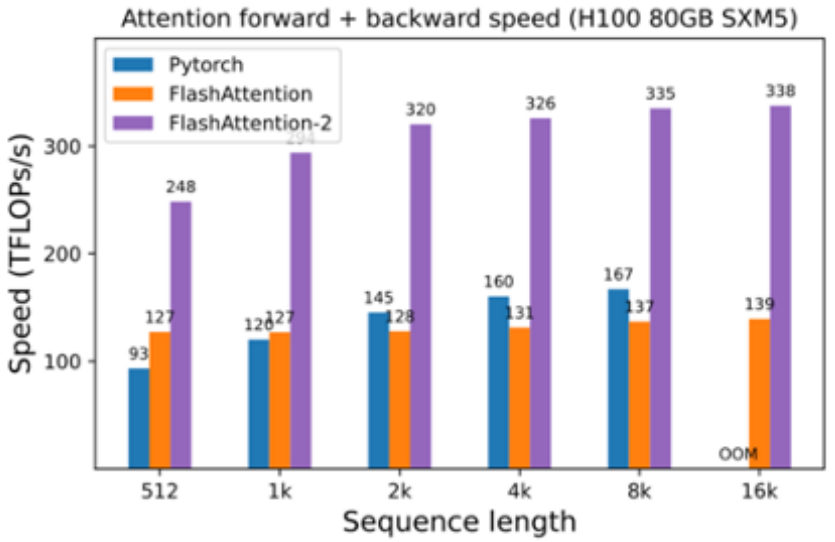
Attention forward + backward speed on A100 GPU

Flash Attention

- FlashAttention v2
  - Performance speedup



(a) Without causal mask, head dimension 64

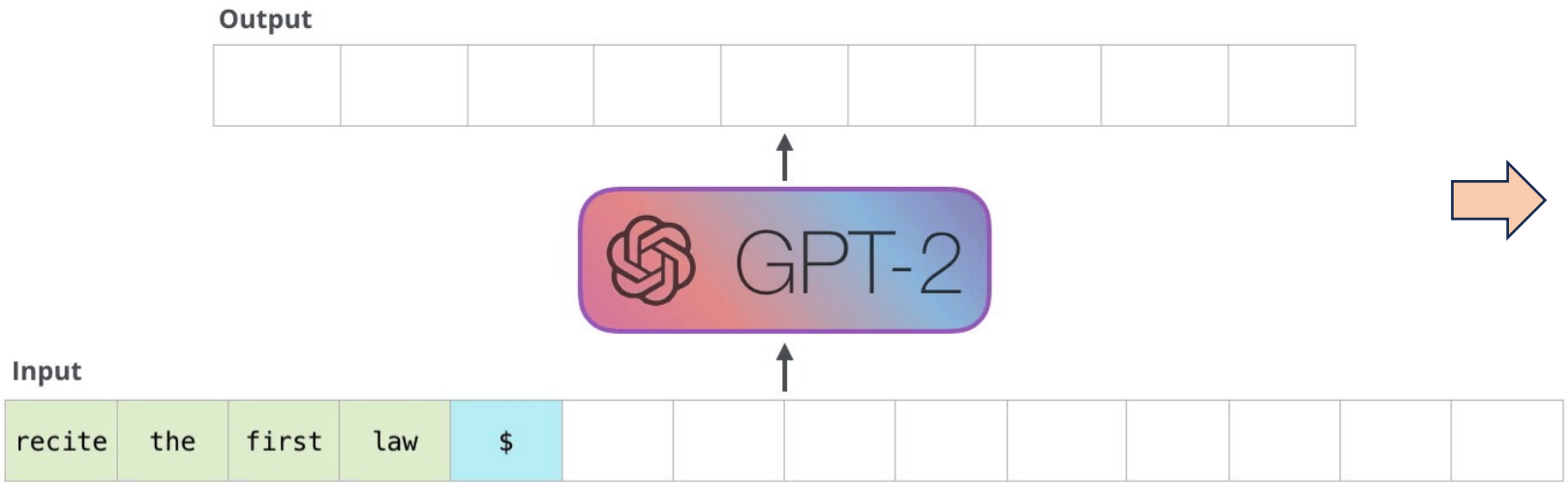


(b) Without causal mask, head dimension 128

Attention forward + backward speed on H100 GPU

Flash Attention

- Flash Decoding
  - New challenges in autoregressive decoding



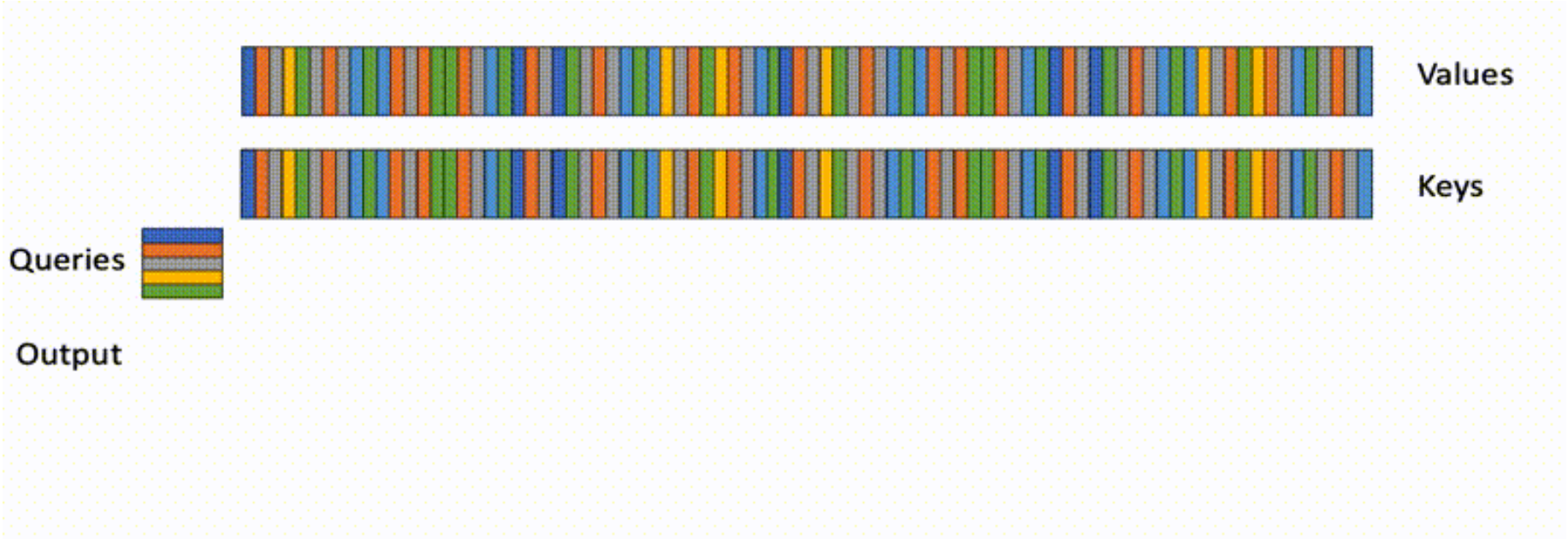
- Key (K) and Value (V) keep growing to be extraordinarily large
  - Out of Memory error
  - Swapping to CPU memory which is very slow
  - Truncating the input causes poor performance

Query (Q) length is always 1, i.e. the newly generated token

Source: https://sh-tsang.medium.com/review-gpt-2-nlp-26e7ce92d234

Flash Attention

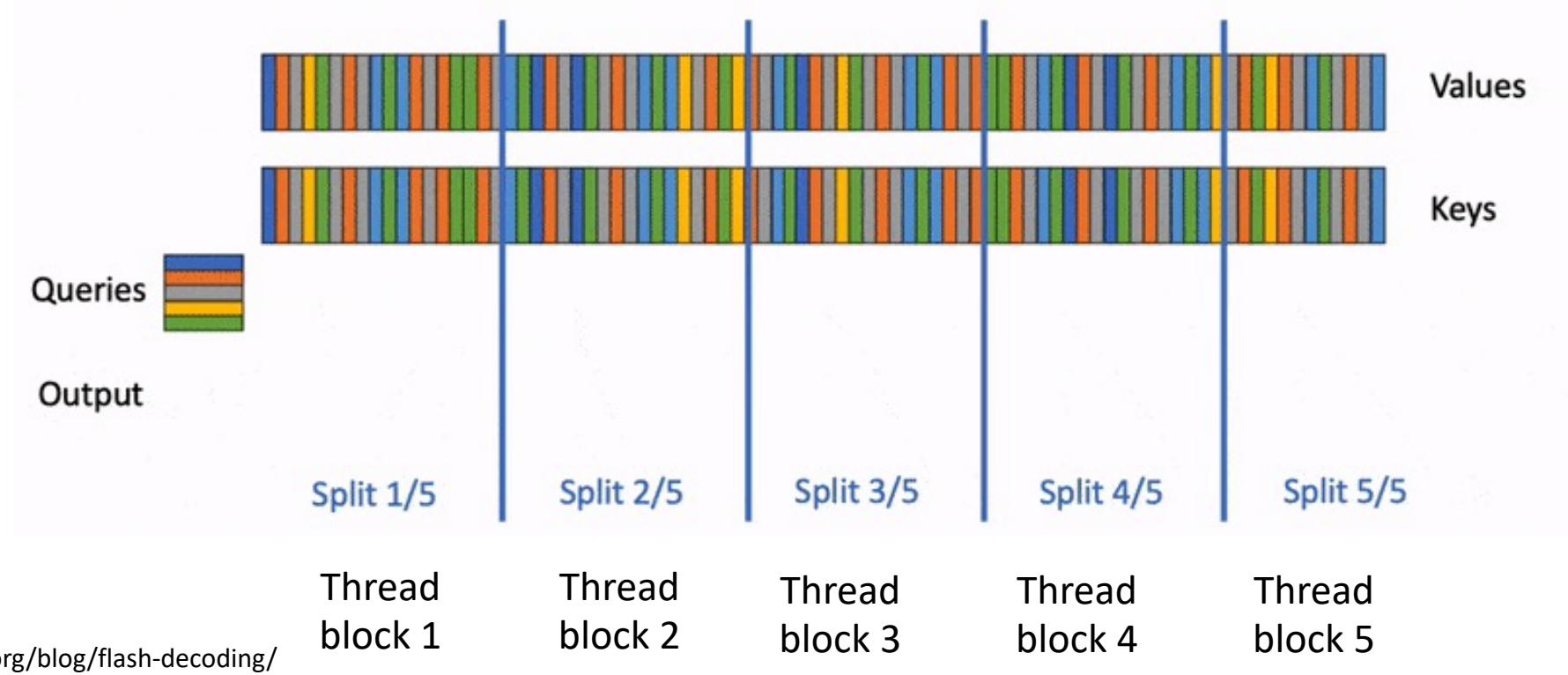
- Flash Decoding
  - Low efficiency due to small Q (i.e. sequence length of 1 in decoding)



Source: https://pytorch.org/blog/flash-decoding/

Flash Attention

- Flash Decoding
  - Sharding Key and Value matrices (e.g. 5 pieces) in order to create many more thread blocks



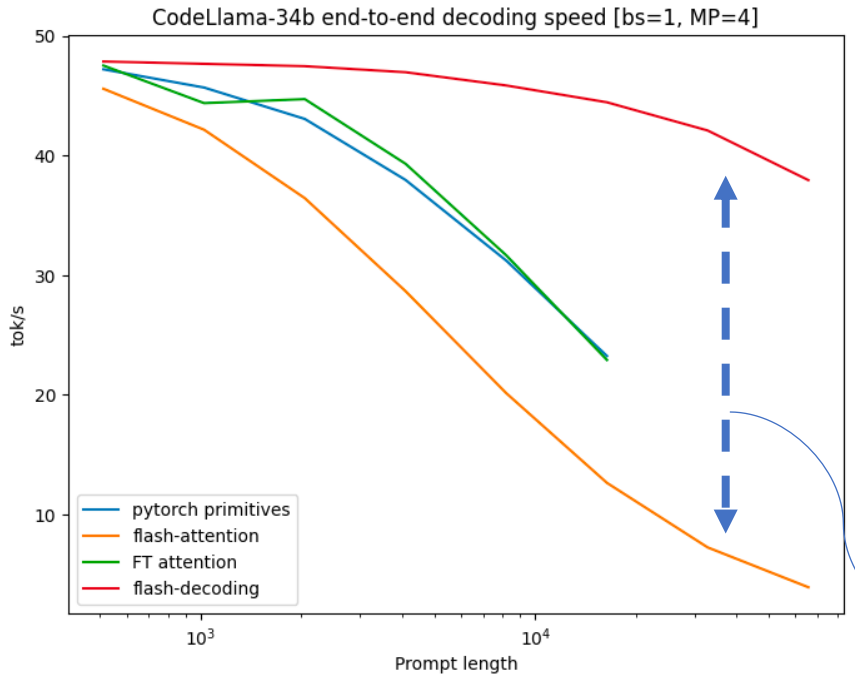
Source: https://pytorch.org/blog/flash-decoding/

Flash Attention

- Flash Decoding
  - Computing attention scores with K/V splits using FlashAttention v1/v2
    - Only obtaining local ***max*** and local ***sum***
    - Output O not properly scaled
  - Launching an independent ***REDUCE*** kernel
    - Obtaining global ***max*** and global ***sum***
    - Rescaling local O to obtain the final output

Flash Attention

- Flash Decoding

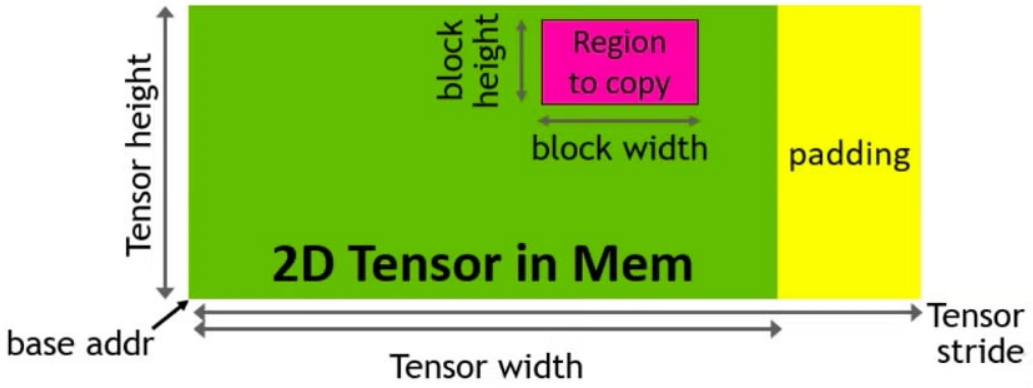


Setting \ Algorithm	PyTorch Eager	Flash-Attention v2.0.9	Flash-Decoding
B=256, seqlen=256	3058.6	390.5	63.4
B=128, seqlen=512	3151.4	366.3	67.7
B=64, seqlen=1024	3160.4	364.8	77.7
B=32, seqlen=2048	3158.3	352	58.5
B=16, seqlen=4096	3157	401.7	57
B=8, seqlen=8192	3173.1	529.2	56.4
B=4, seqlen=16384	3223	582.7	58.2
B=2, seqlen=32768	3224.1	1156.1	60.3
B=1, seqlen=65536	1335.6	2300.6	64.4
B=1, seqlen=131072	2664	4592.2	106.6

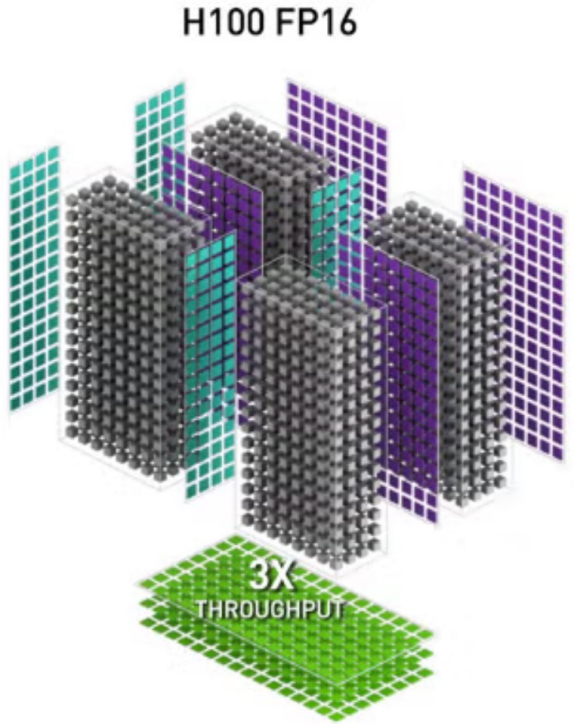
Flash Decoding versus Flash Attention

Flash Attention v3

- Hardware improvement (only for your reference)
  - TMA (Tensor Memory Accelerator)



  - Warpgroup Matrix Multiply-Accumulate



  - Asynchrony
    - Overlap overall computation and data movement via warp-specialization and interleave block-wise ***matmul*** and ***softmax*** operations

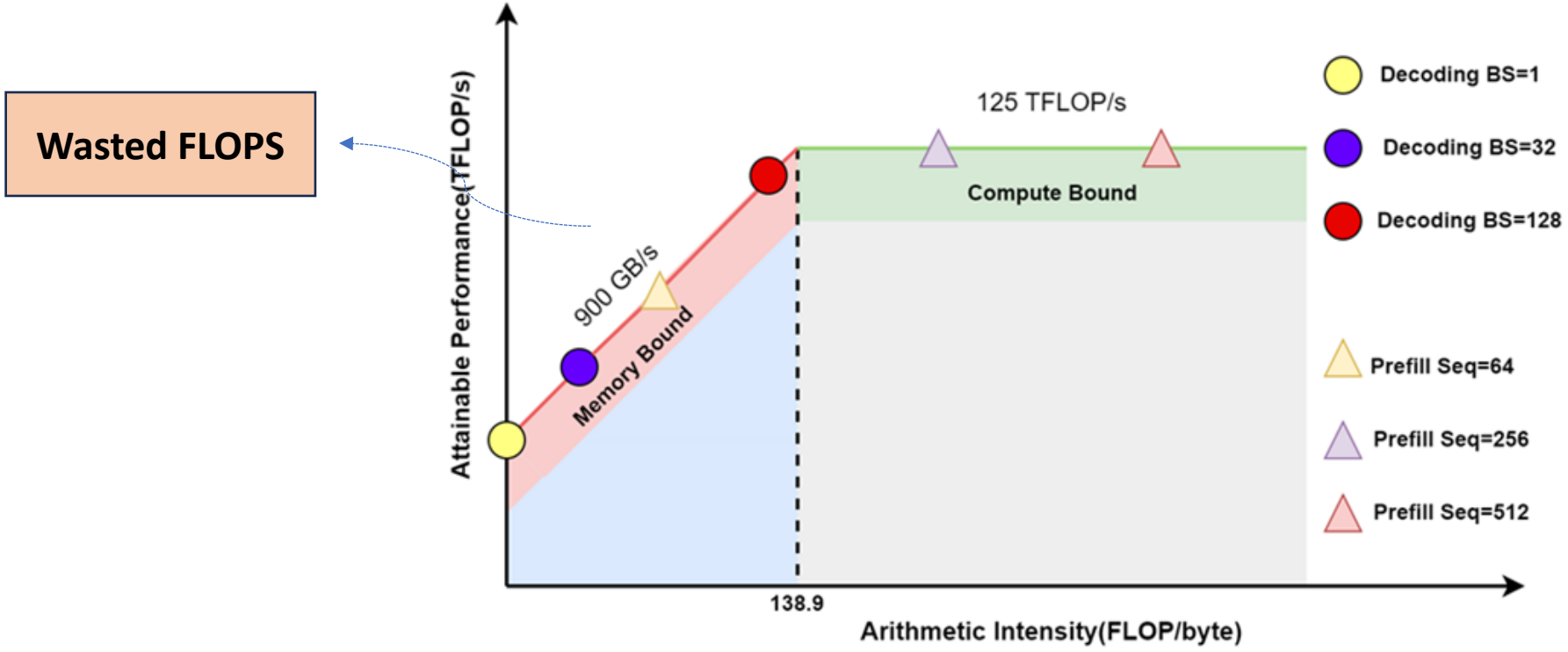
Source: FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision

Inference Optimization: Outline

- Overview
- Attention Computation Optimization
  - Sparse Attention
  - Linear Attention
  - Flash Attention
- **Continuous Batching**
- KV Cache Optimization
- Speculative Decoding
- Distributed Serving

Batching to Meet Compute-Bound

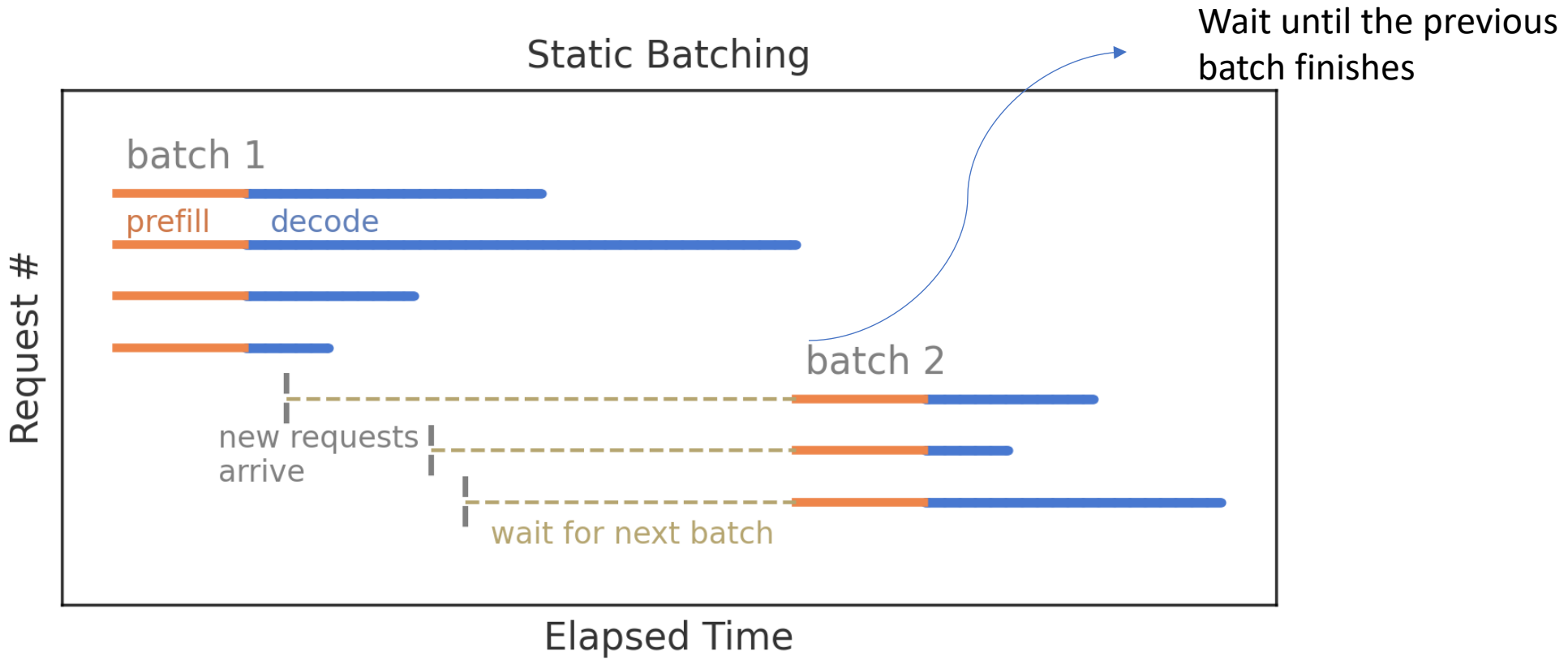
- Why batching multiple requests?
  - Generating tokens for a large number of prompts to match the compute bound



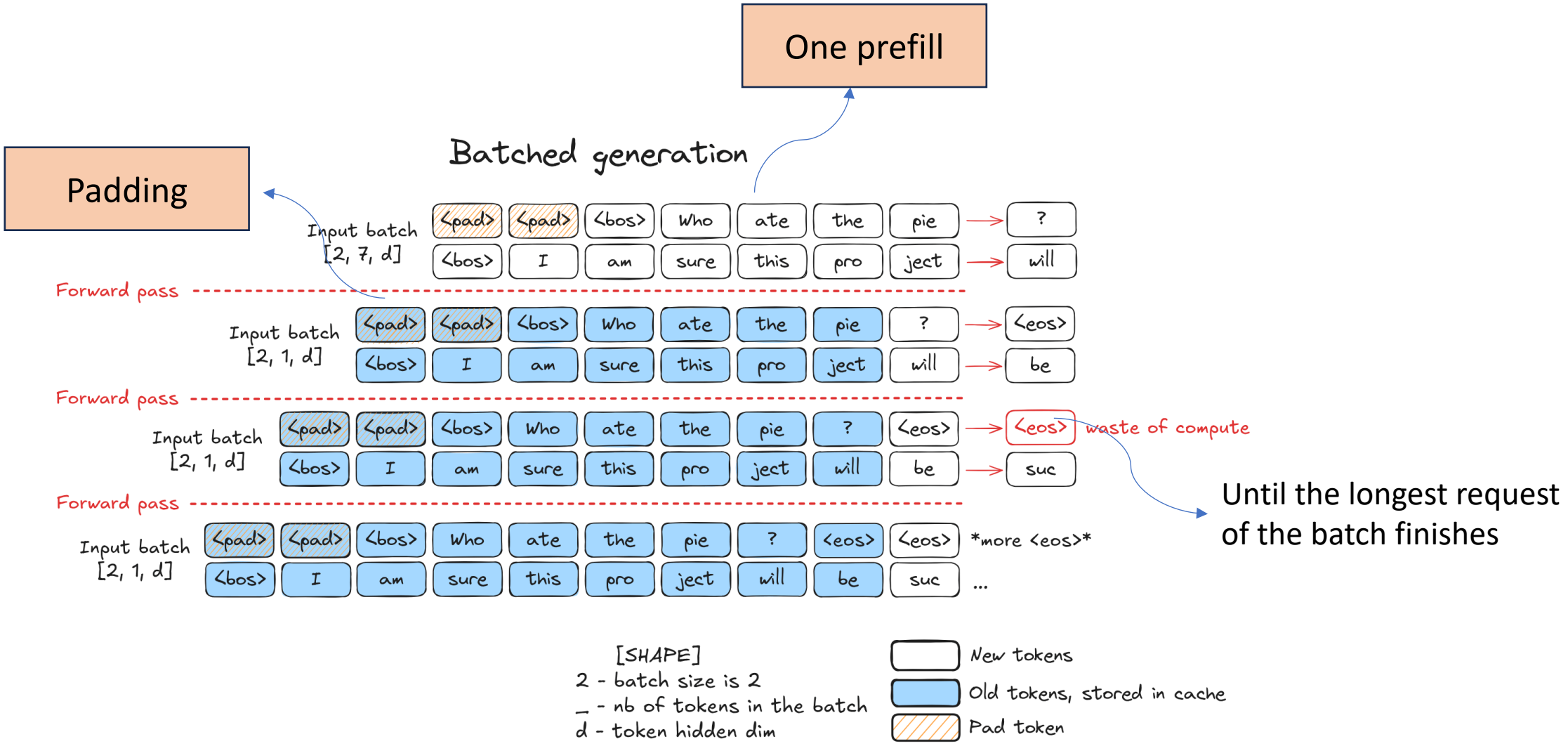
Roofline model of NVIDIA V100 GPU

Static Batching

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



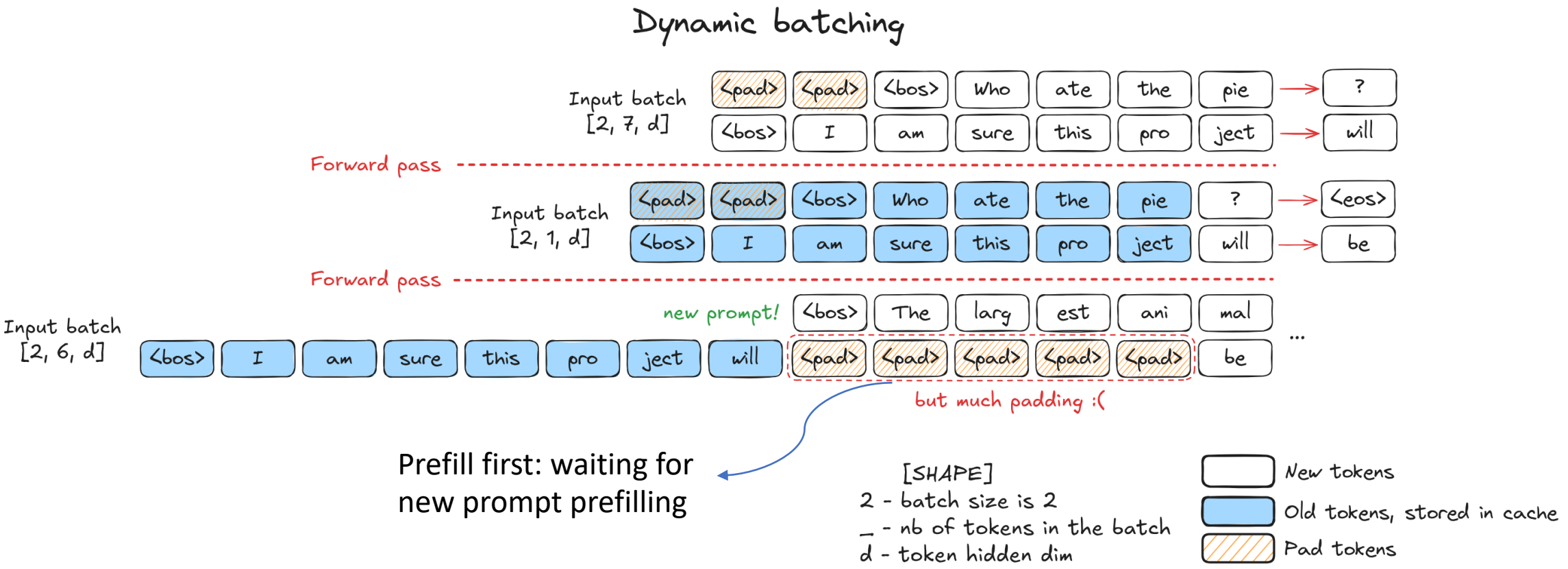
Static Batching



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

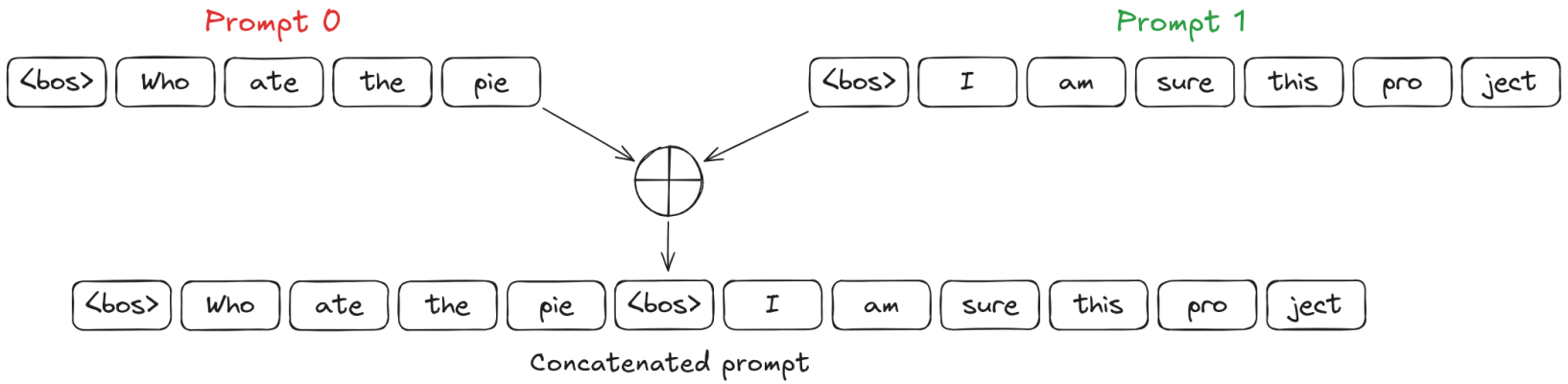
Dynamic Batching

- Insert an inference request in the queue



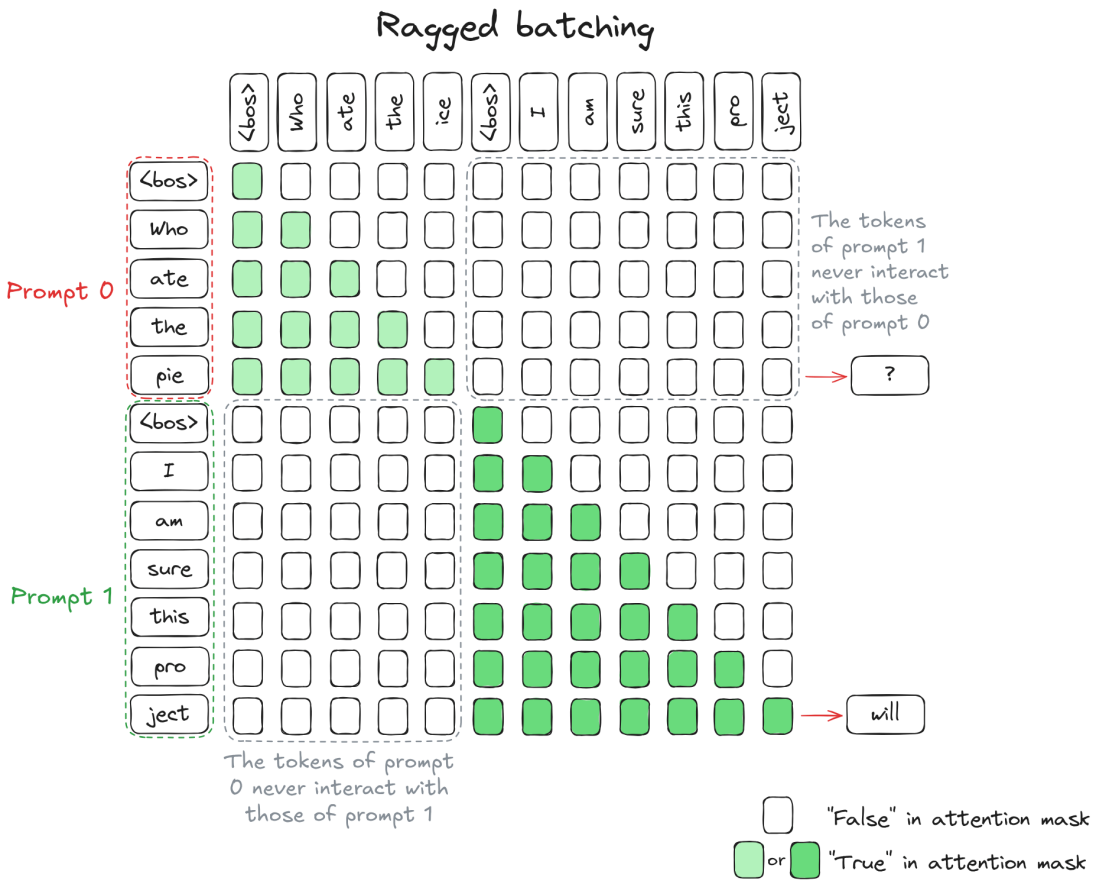
Idea Batching

- Concatenating prompts
  - ***Decode*** is equivalent to ***prefill*** with sequence length of 1
  - ***Decode*** and ***Prefill*** of different prompts operate simultaneously without bubbles



Continuous Batching

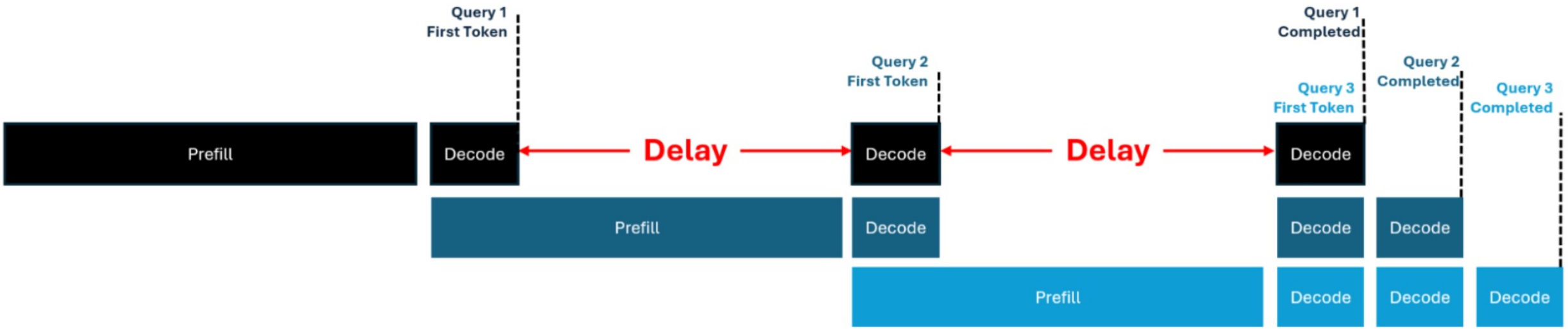
- Continuous batching = ragged batching + ideal dynamic batching
  - Attention scores of tokens belonging to *prompt 0* and *prompt 1* should not be computed together
  - Ragged batching because sequence lengths are 'ragged' or uneven, no need for padding tokens



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

Continuous Batching

- Question remains unsolved: how to make ***prefill*** of new requests and ***decode*** of existing requests operate ***in parallel***?

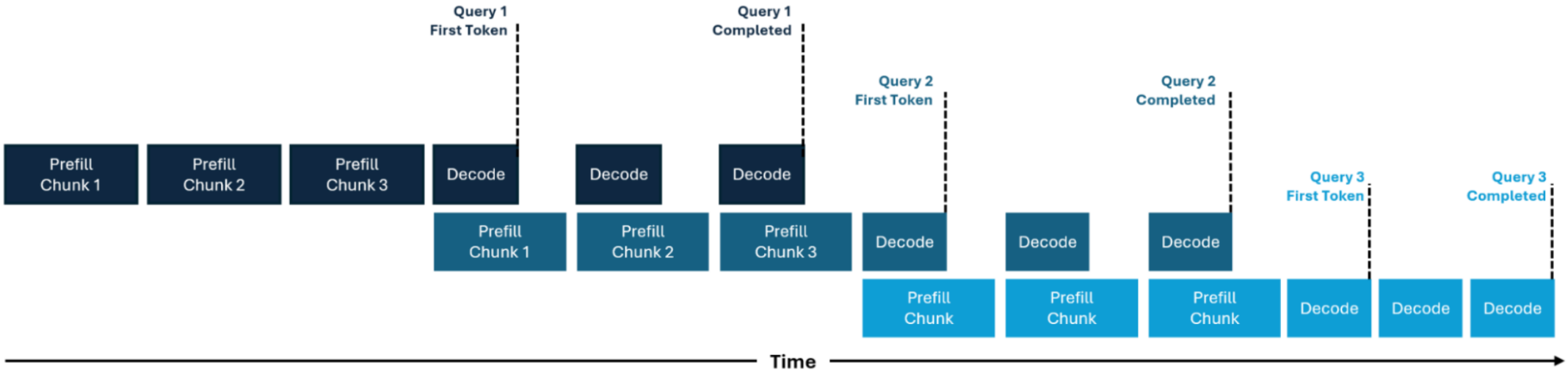


Continuous batching without chunked prefill

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

Chunked Prefill

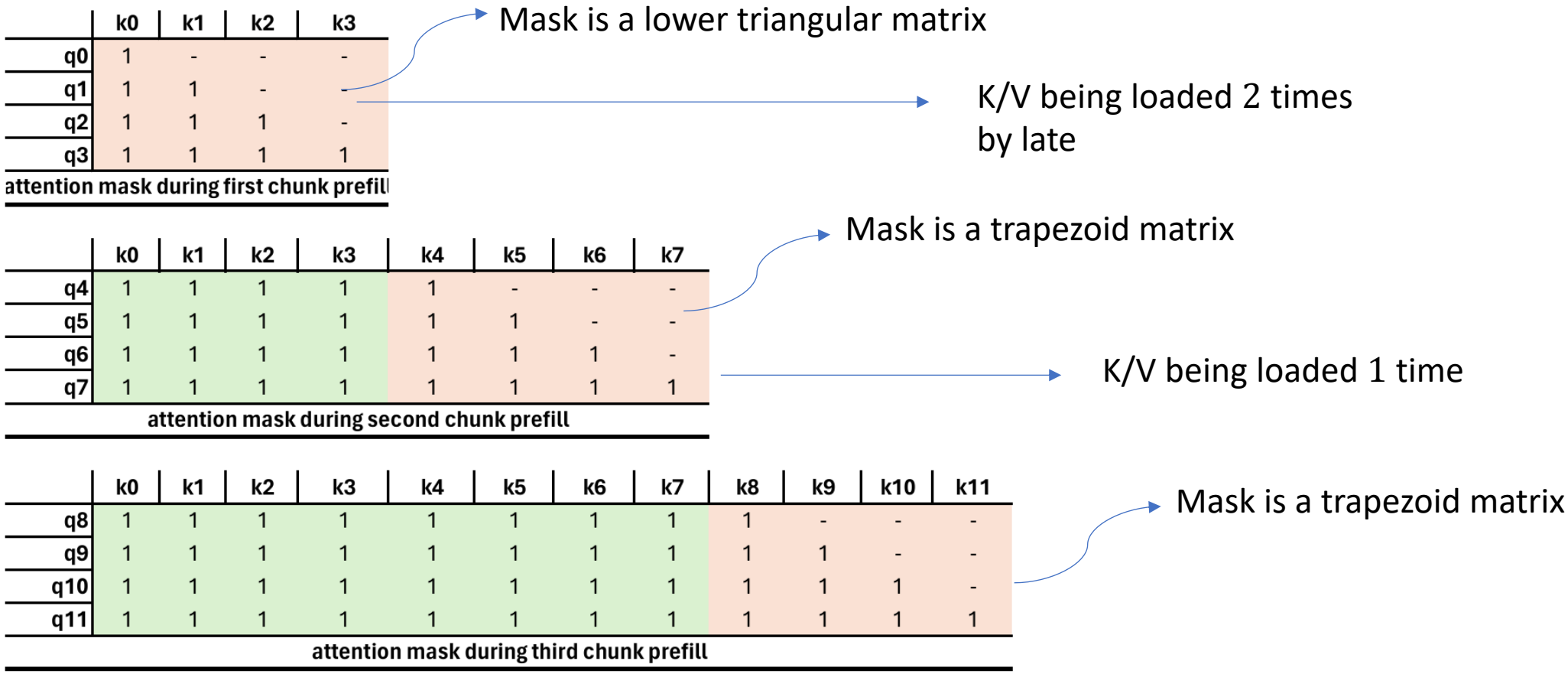
- Chunked Prefill
  - **Elastic Scheduling**: split the entire prompt into smaller chunks: more scheduling flexibility and enables mixing prefill and decode.
  - **Decode-Maximal Batching**: combine a single prefill chunk with multiple decode requests in the same batch, allowing decode operations to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



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

Chunked Prefill

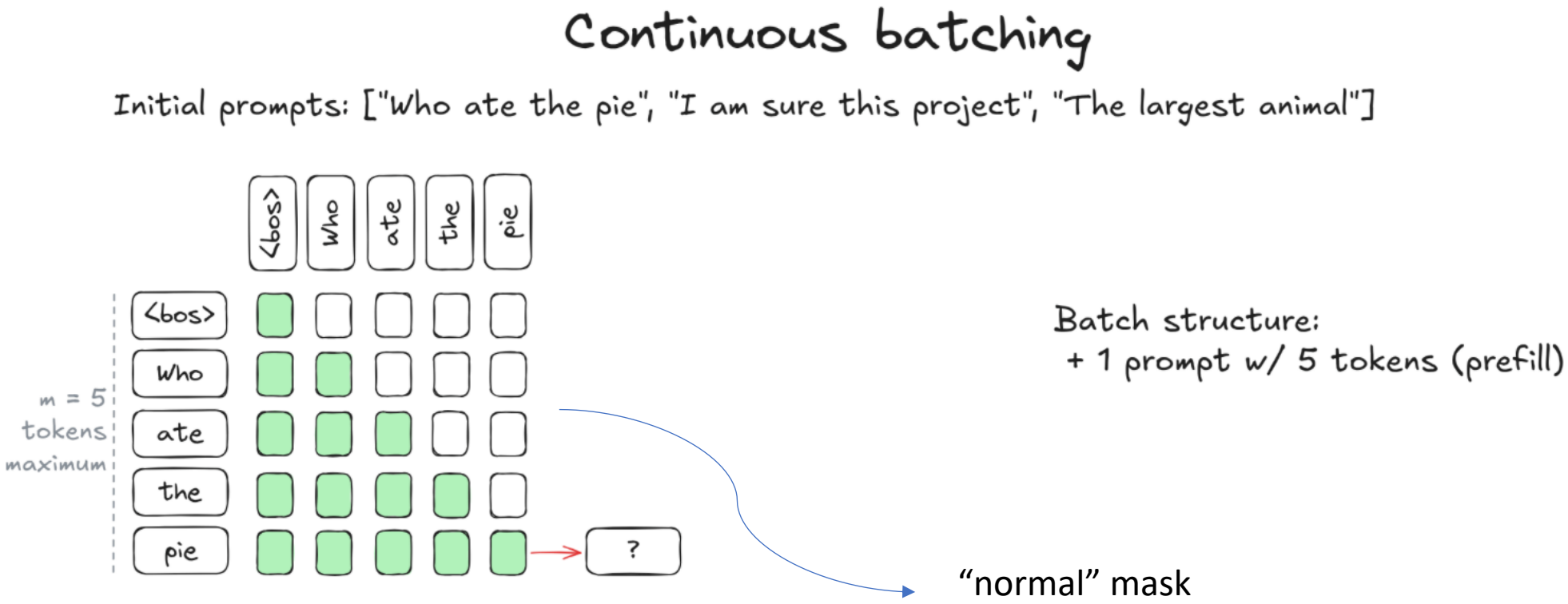
- Combined together to acquire a global view of complete prefill



Attention mask matrices for successive prefill chunks

Continuous Batching

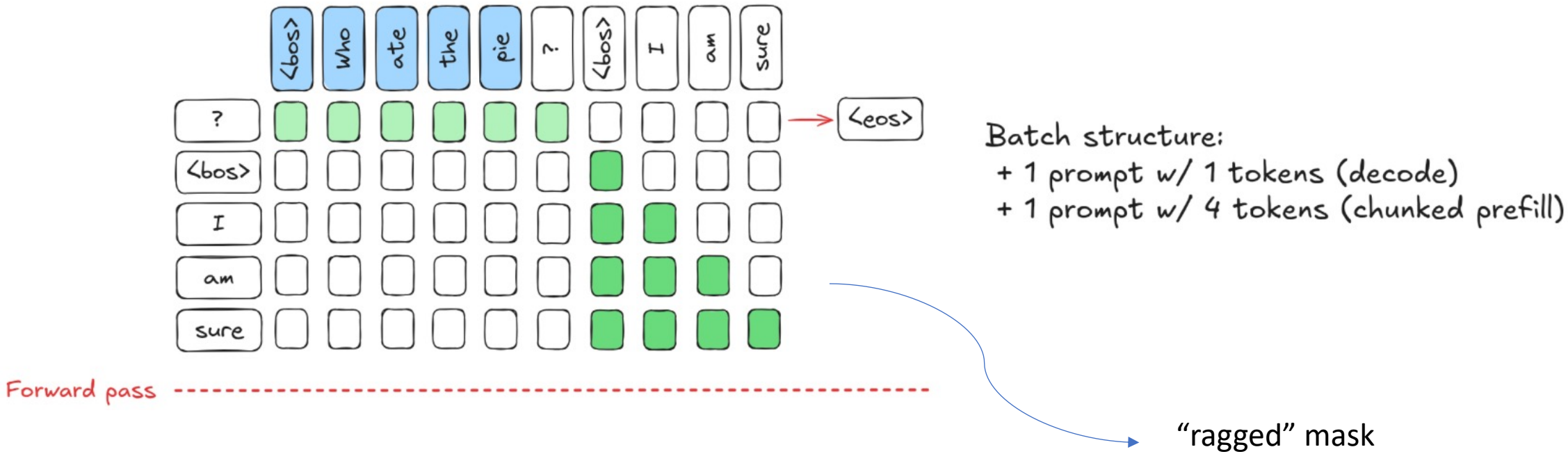
- Continuous batching = ragged batching + dynamic batching



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

Continuous Batching

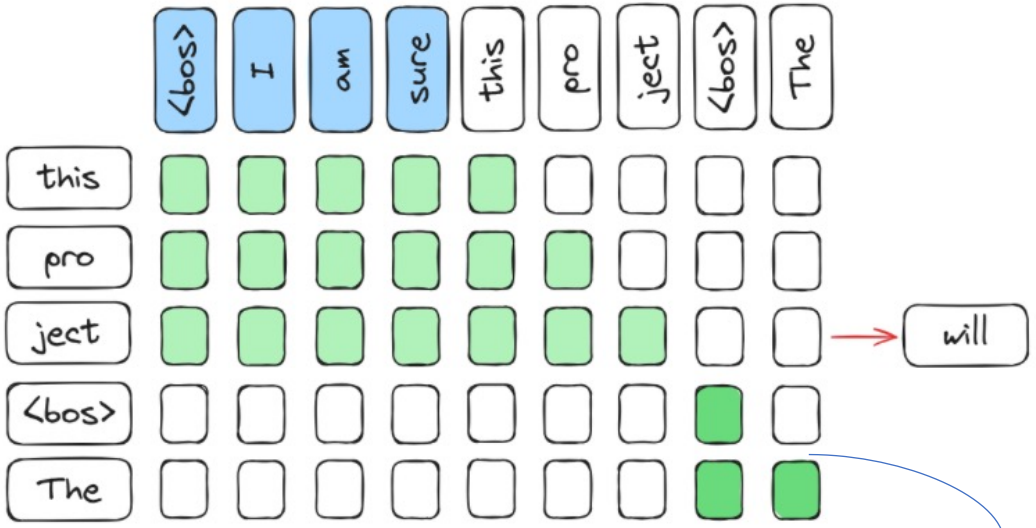
- Continuous batching = ragged batching + dynamic batching



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

Continuous Batching

- Continuous batching = ragged batching + dynamic batching



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

Thanks!