CS6216 Advanced Topics in Machine Learning (Systems)

Transformers, Attention and Optimizations

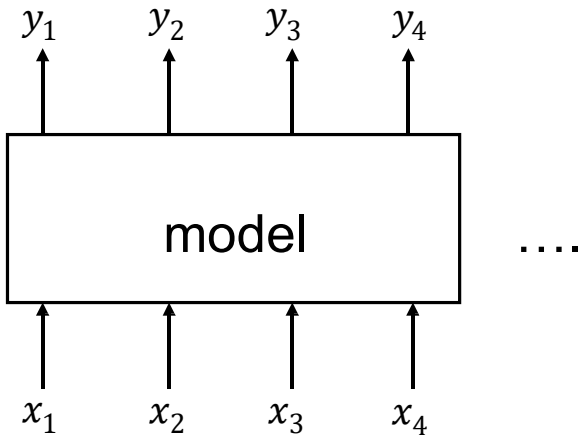
Yao LU

17 Sep 2026

National University of Singapore
School of Computing

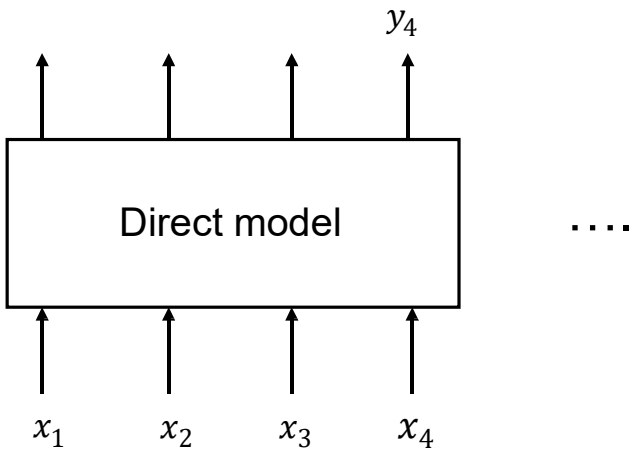
Sequence prediction

Take a set of input sequence, predict the output sequence



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

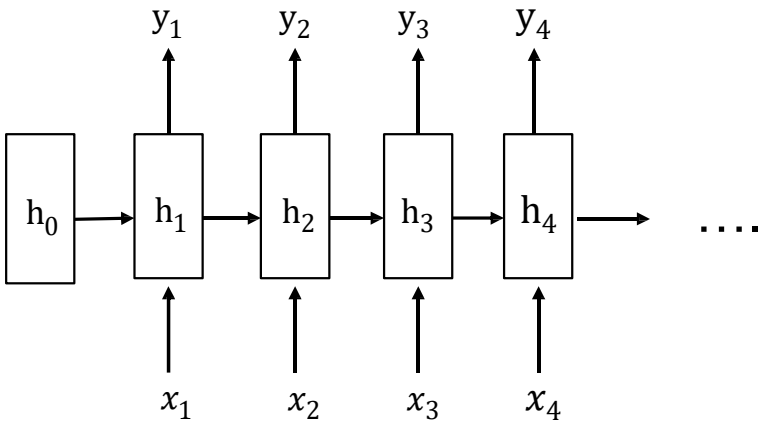
Method 1: direct / spot prediction



Challenge: inputs of different sizes.

Method 2: Recurrent Neural Networks

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



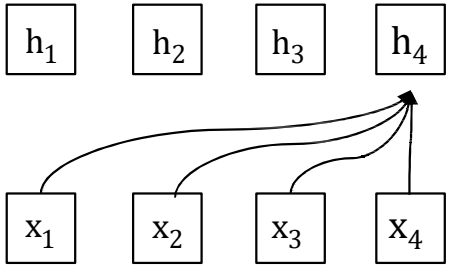
The information is carried only through h_t

“Attention” mechanism

Generally refers to the approach that weighted combine individual states

Attention output

Hidden states from previous layer



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

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

There are different methods to compute attention scores

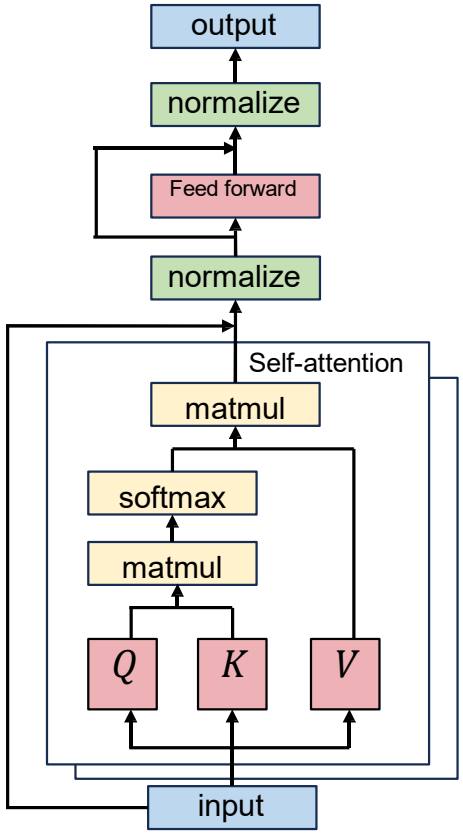
Transformer block and self attention

A typical transformer block

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

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

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



Transformer block and self attention

A typical transformer block

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

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

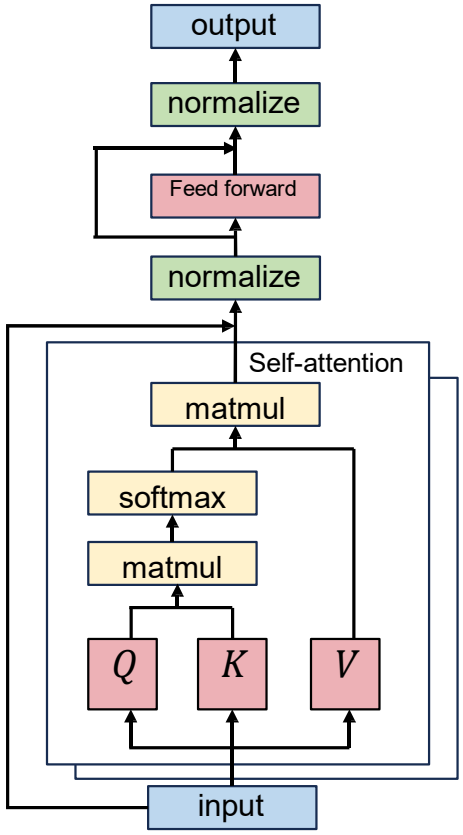
$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{\text{QK}^T}{\sqrt{d}}\right)V$$

Similarity

(Scaled Dot-Product Attention)

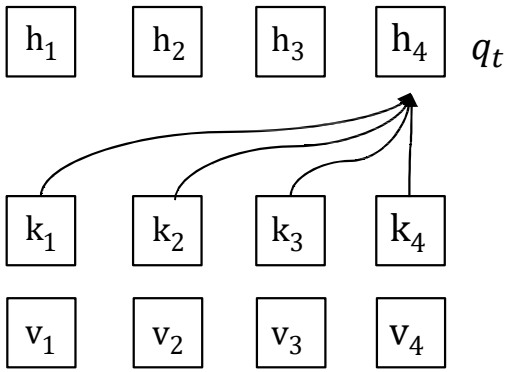
Select → attention weights

Scale factor



Self-attention operation

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

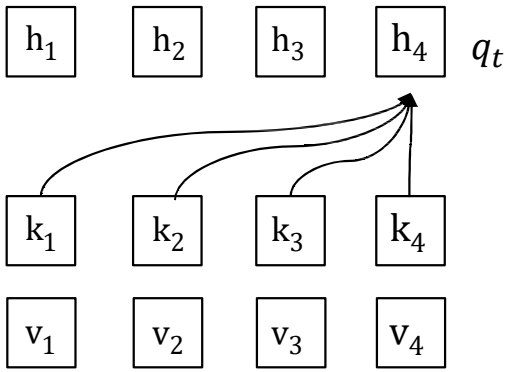


How to compute the output h_t, based on q_t, K, V one timestep t?

To keep it simple, we will drop suffix t and just use q to refer to q_t

Self-attention operation

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



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

- Pre-softmax "attention score"

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

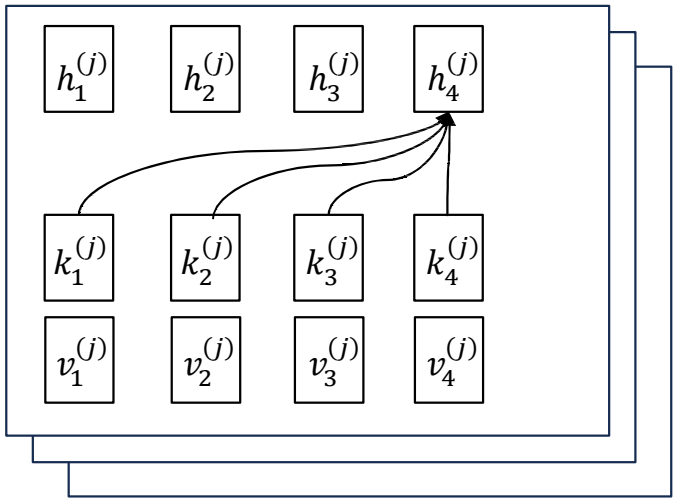
- Weighed average via softmax

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

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

Multi-head attention

Multiple “attention heads”, $Q^{(j)}, K^{(j)}, V^{(j)}$ denotes j-th attention head



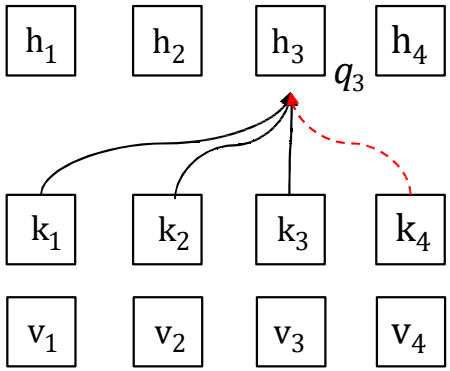
- Apply self-attention in each attention head
- Concatenate all output heads together as output
- Can compute all heads and Q, K, V together then split/reshape out into individual Q, K, V with multiple heads

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

(K, V cache)

Masked self-attention

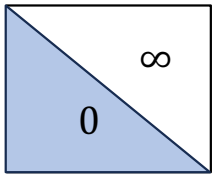
In the matrix form, we are computing weighted average over all inputs



To maintain casual relation and only attend to some of the inputs (e.g. skip the red dashed edge on the left), we can add "attention mask"

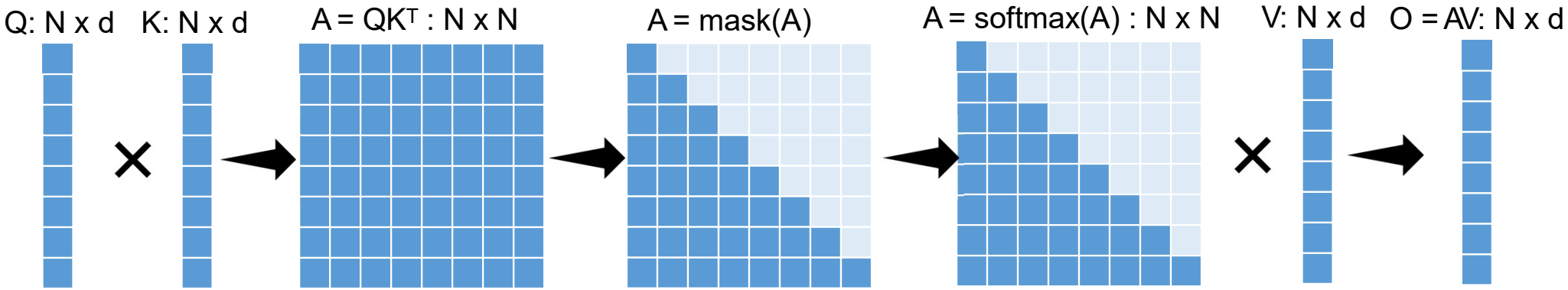
$$\text{MaskedSelfAttention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}} - M\right)V$$

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



Only attend to previous inputs. Skip the computation that are masked out.

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



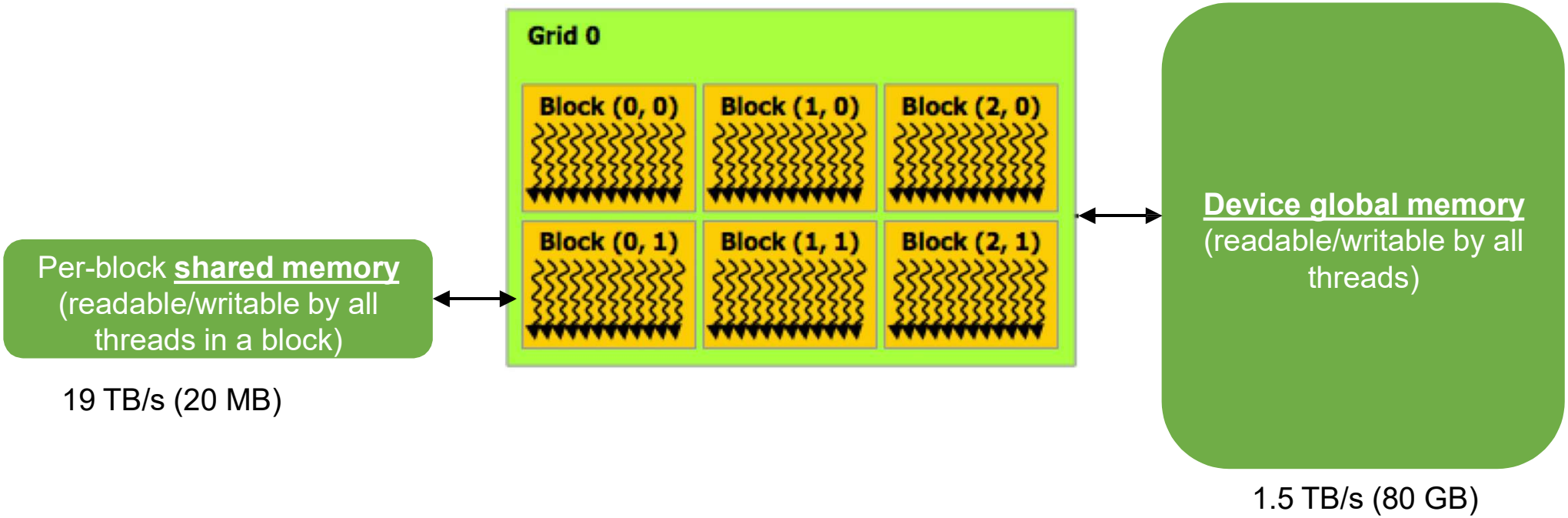
Challenges:

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

Attention optimizations

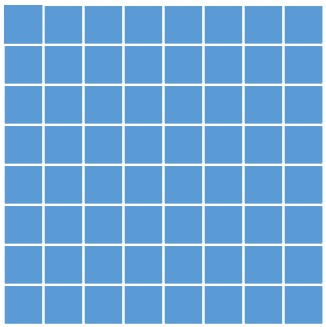
- LLM Training
  - FlashAttention
- LLM Inference
  - Recursive Attention
  - Flash Decoding
  - PagedAttention

Revisit: GPU memory hierarchy



FlashAttention

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



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

Two main Techniques:

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

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

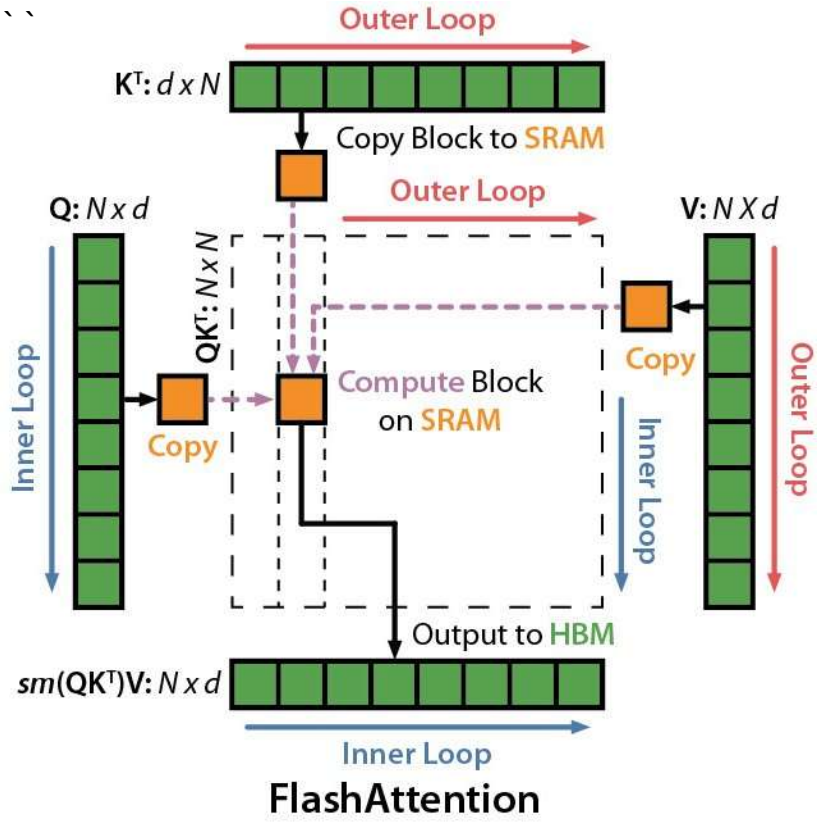
* FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tiling: decompose large softmax into smaller ones by scaling

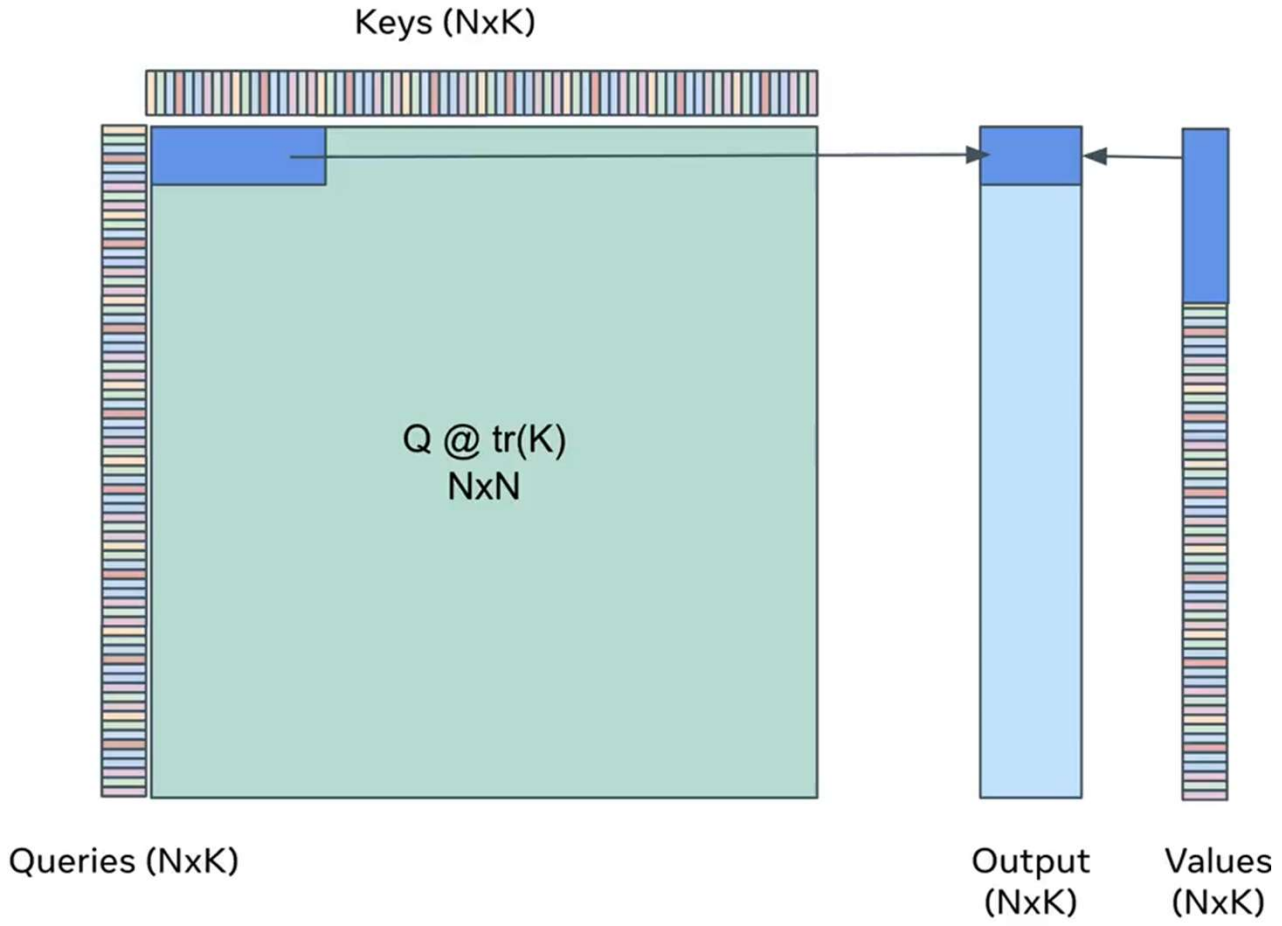
1. Load inputs by blocks from global to shared memory
2. On chip, compute attention output wrt the block
3. Update output in device memory by scaling

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

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



Tiling



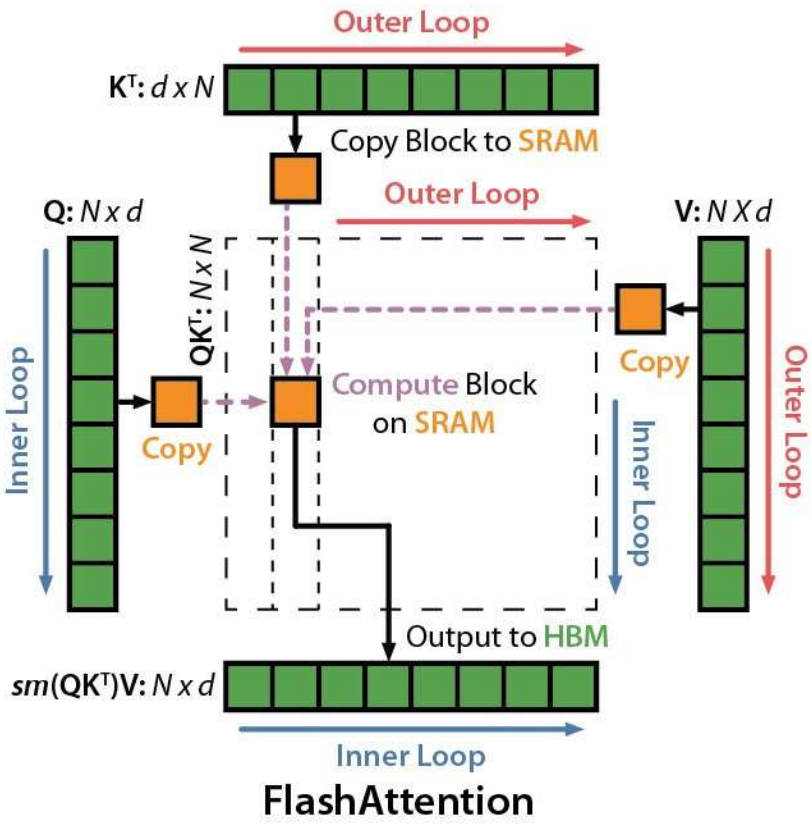
(animation) https://jacksoncakes.com/img/in-post/post-flash-attn/flash-attn-viz.mp4

(Here K is d from last slide)

Animation credit: Francisco Massa

Recomputation: backward pass

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



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

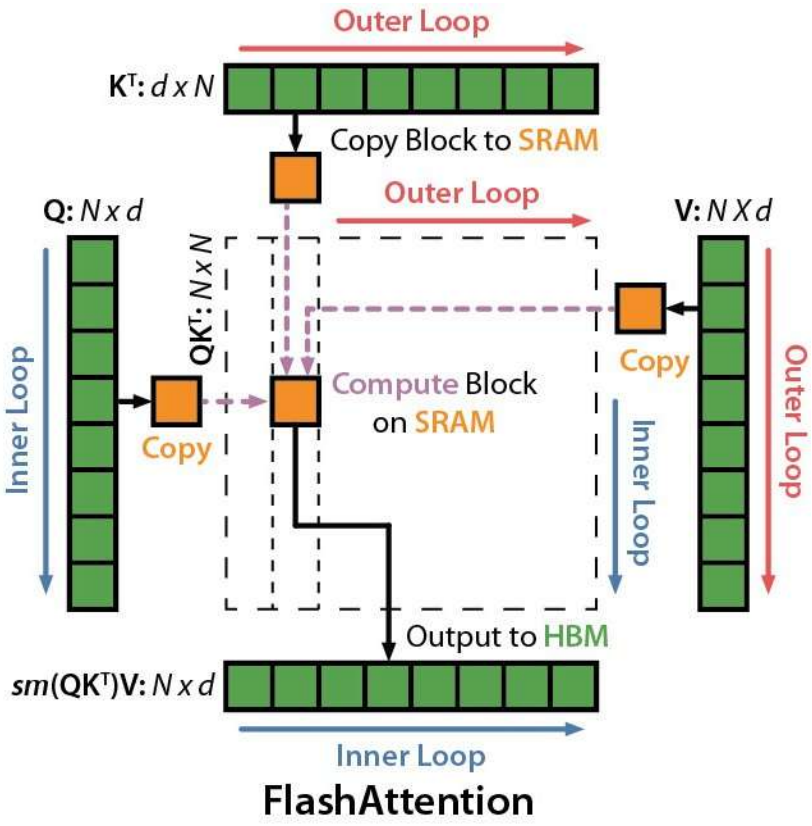
Speed up backward pass with increased FLOPs

FlashAttention v2: threadblock-level parallelism

How to partition FlasshAttention across thread blocks?

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

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

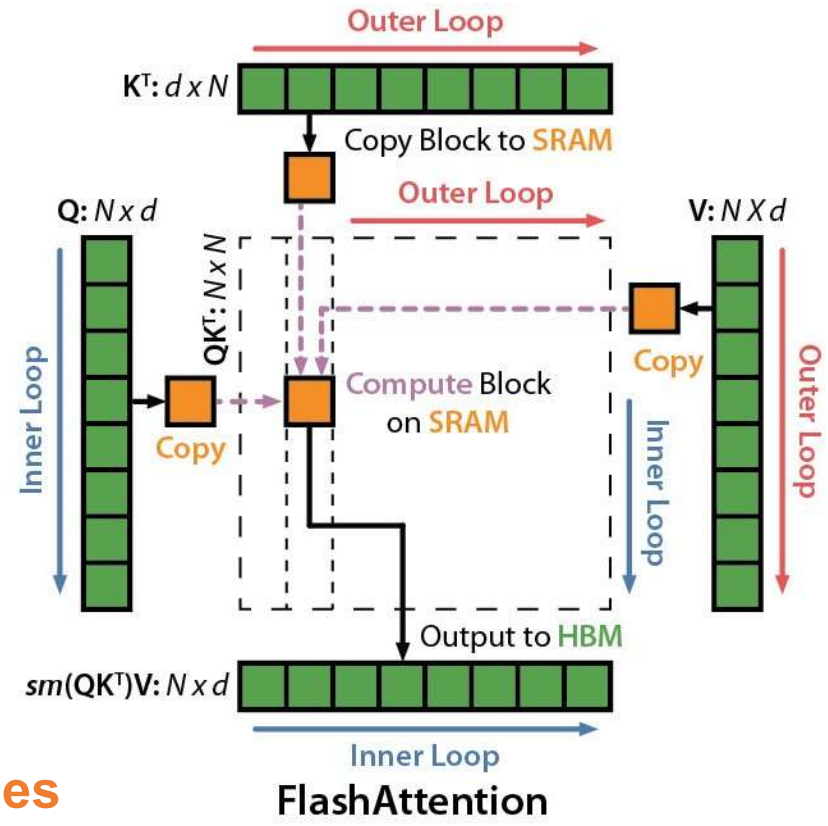


FlashAttention v2: threadblock-level parallelism

How to partition FlasshAttention across thread blocks?

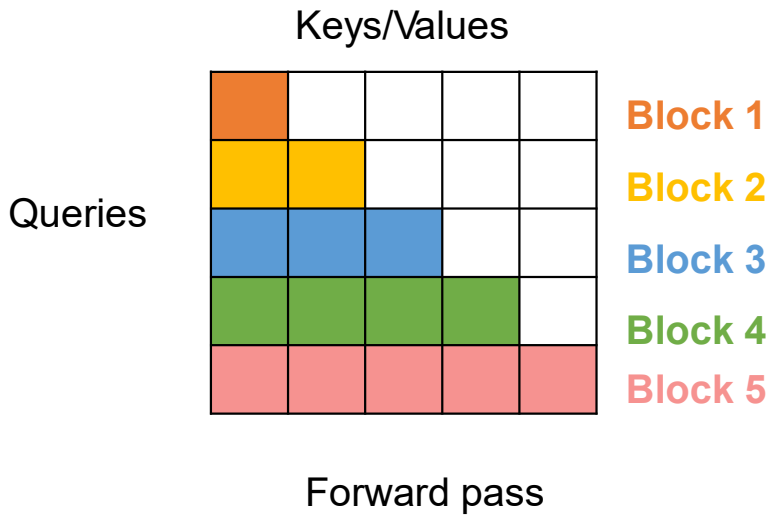
(An A100 has 108 SMMs -> 108 thread blocks)

- Step 1: assign different heads to different thread blocks (16-64 heads)
- Step 2: assign different queries (not K/V) to different thread blocks



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

FlashAttention v2: threadblock-level parallelism

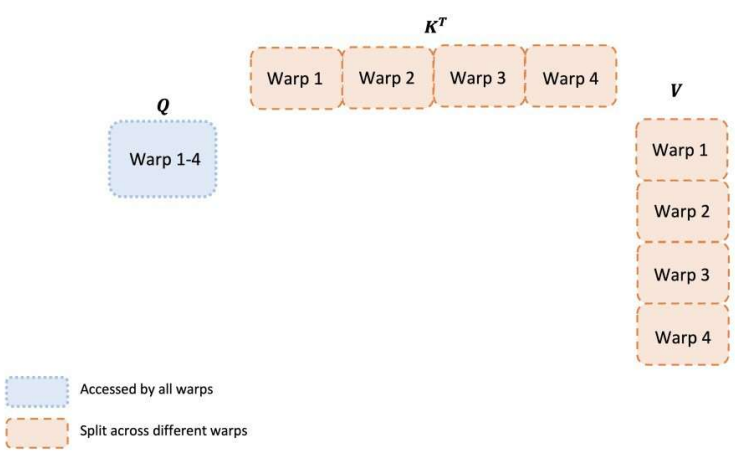


No need to handle workload imbalance.

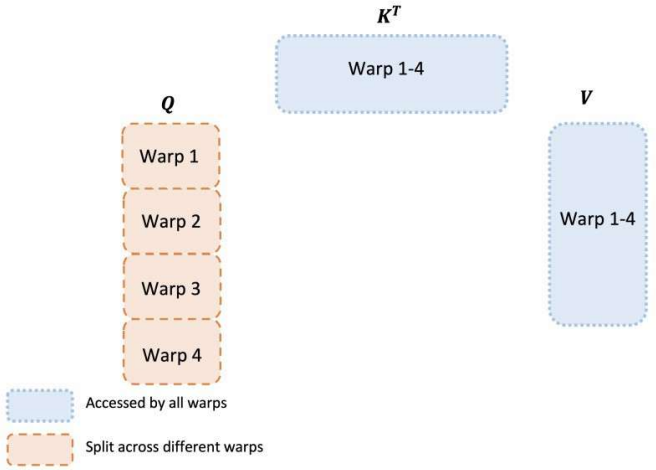
GPU scheduler automatically loads the next block once the current one completes.

FlashAttention v2: warp-level parallelism

- How to partition FlashAttention across warps within a thread block?



(a) FlashAttention

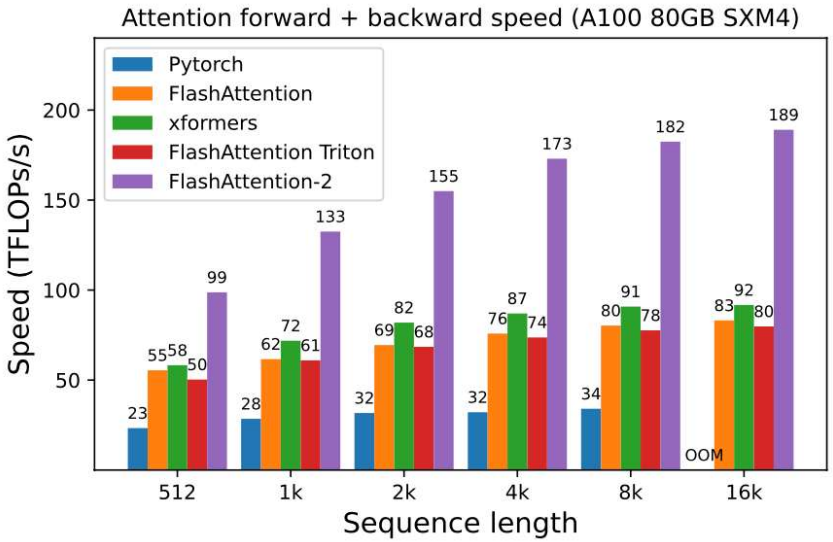


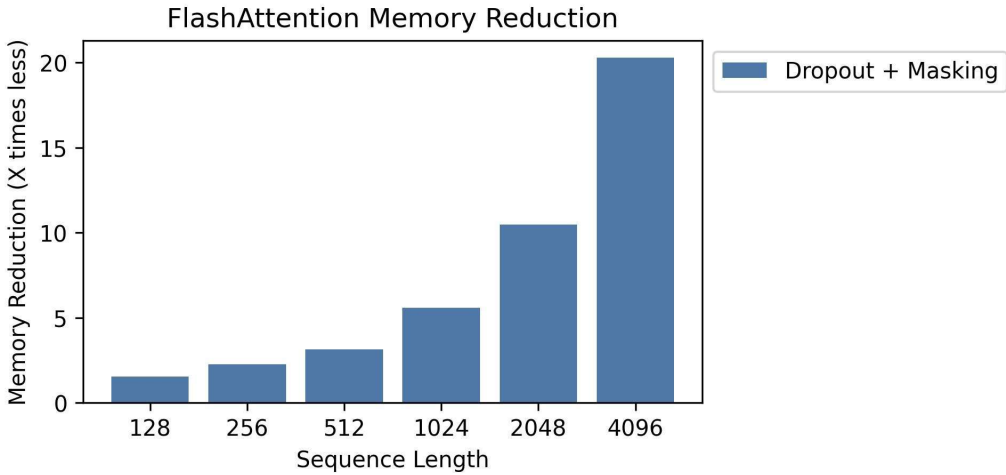
(b) FlashAttention-2

Splitting across K/V requires communication to add results

Splitting across Q avoids communications

FlashAttention v2: 2-4x speedup, 10-20x memory reduction





Memory linear in sequence length

Attention optimizations

- LLM Training
  - FlashAttention
- LLM Inference
  - Recursive Attention
  - Flash Decoding
  - PagedAttention

Generalizing attention score and value vector

Pre-softmax "attention score" $s_i = \frac{1}{\sqrt{d}} q k_i^T$

Define the following "attention weight" for an index set I

$$s(I) = \log\left(\sum_{i \in I} \exp(s_i)\right)$$

Generalize the value vector v for index set I

$$v(I) = \sum_{i \in I} \mathrm{softmax}(s)_i v_i = \frac{\sum_{i \in I} \exp(s_i) v_i}{\exp(s(I))}$$

When index set $I = \{i\}$, $\mathrm{s}(\{i\}) = s_i$, $\mathrm{v}(\{i\}) = v_i$

When index set $I = \{1,2, \ldots t\}$, $\mathrm{v}(I)$ is the final output of the attention

FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. MLsys 2025

FlashInfer: recursive attention

$$s(I) = \log\left(\sum_{i\in I} \exp(s_i)\right), v(I) = \sum_{i\in I} \text{softmax}(s)_i v_i = \frac{\sum_{i\in I} \exp(s_i) v_i}{\exp(s(I))}$$

For any partition $\{I_j\}$ of I such that $I = \cup_{j=1}^{n} I_j$, the following relation holds

$$s\left(\cup_{j=1}^{n} I_j\right) = \log \sum_j \exp\left(s\left(I_j\right)\right), v\left(\cup_{j=1}^{n} I_j\right) = \sum_j \text{softmax}([s(I_1), s(I_2), \ldots])_j v\left(I_j\right)$$

Attention computation is **communicative** and **associative,** can be done by divide-and-conquer.

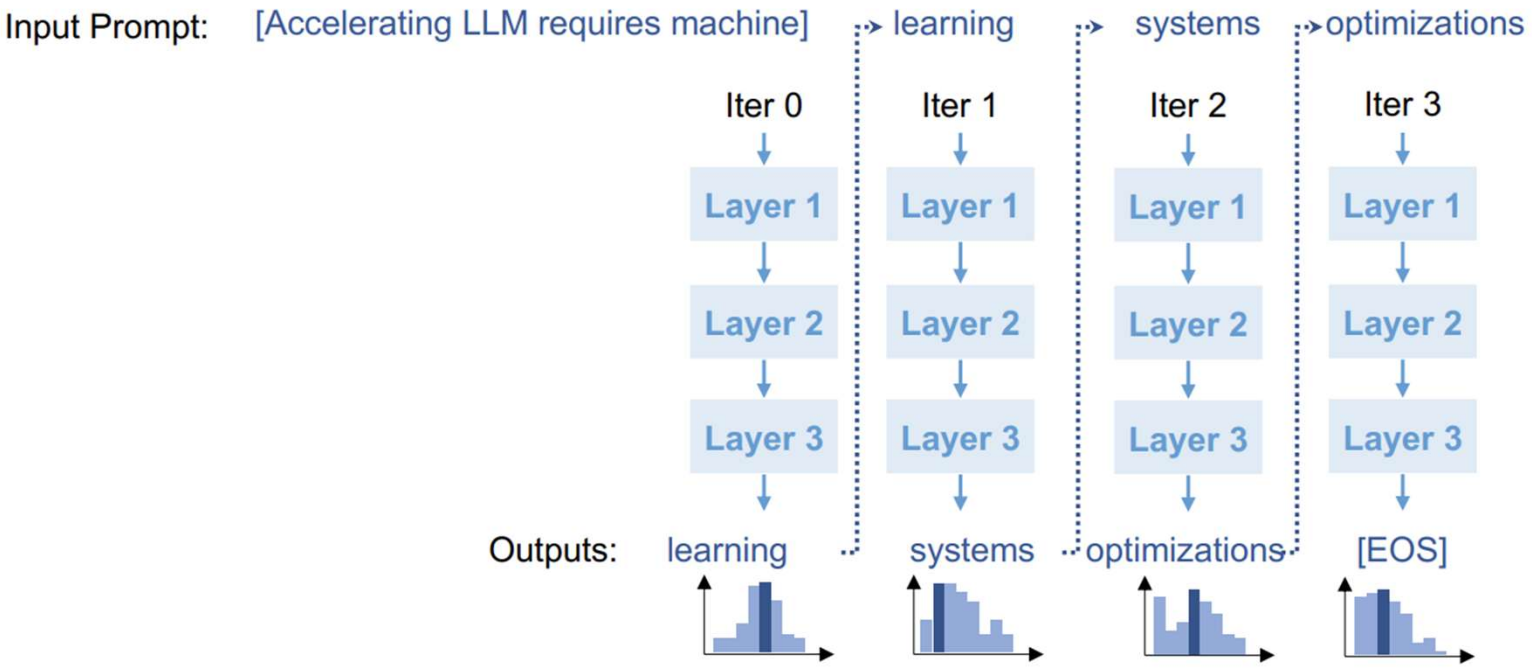
This is an important property for a lot of system optimization:
We can recursively combine the vector and "attention score" of any subsets of indices.

FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. MLsys 2025

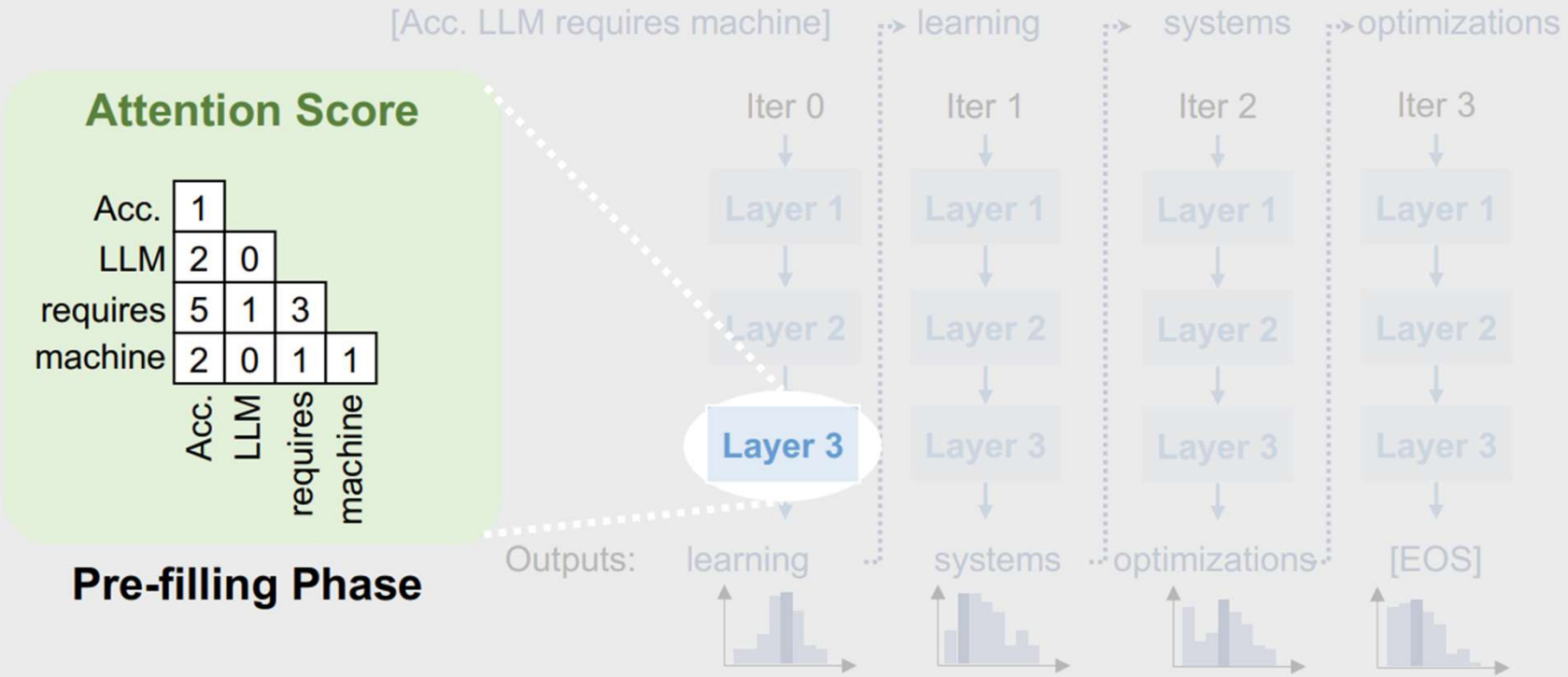
Attention optimizations

- LLM Training
  - FlashAttention
- LLM Inference
  - Recursive Attention
  - Flash Decoding
  - PagedAttention

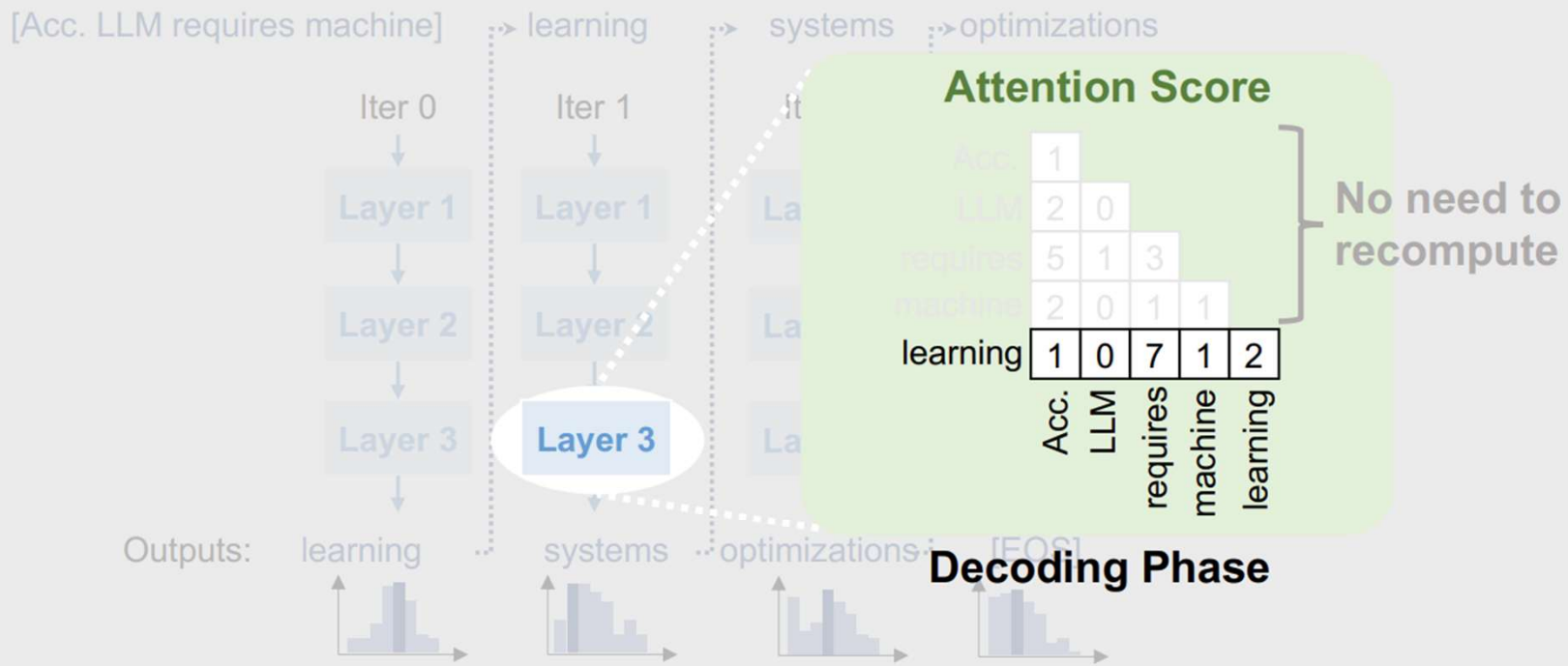
Generative LLM inference: autoregressive decoding



Generative LLM inference: autoregressive decoding



Generative LLM inference: autoregressive decoding

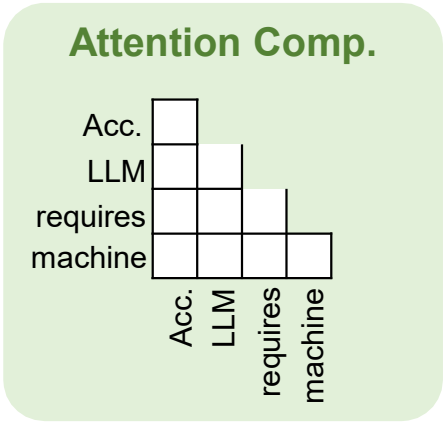


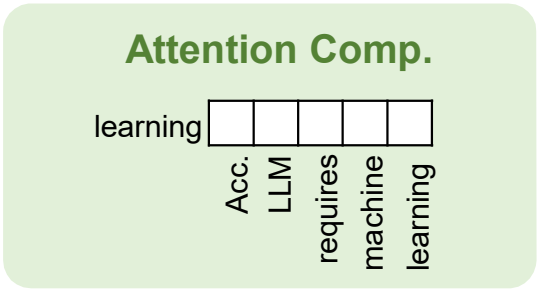
Generative LLM inference: autoregressive decoding

- **Pre-filling phase** (0-th iteration):
  - Process ***all*** input tokens at once
- **Decoding phase** (all other iterations):
  - Process a ***single*** token generated from previous iteration
  - Use attention keys & values of all previous tokens

- Key-value cache:
  - Save attention keys and values for the following iterations to avoid recomputation

Apply FlashAttention to LLM inference





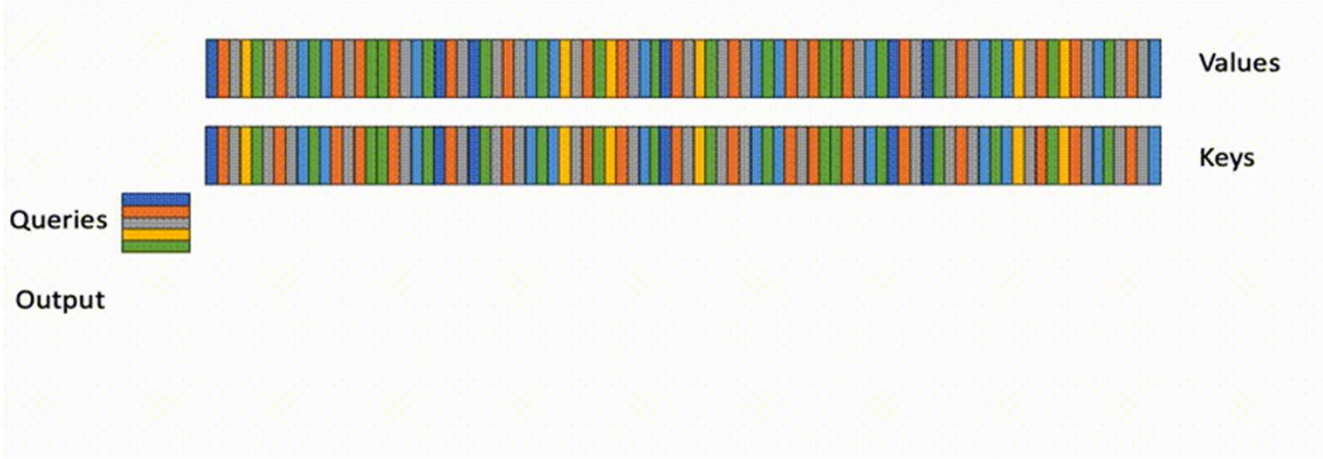
Pre-filling phase:

- Yes, compute different queries using different thread blocks/warps

Decoding phase:

- No, there is a single query in the decoding phase

FlashAttention processes K/V sequentially

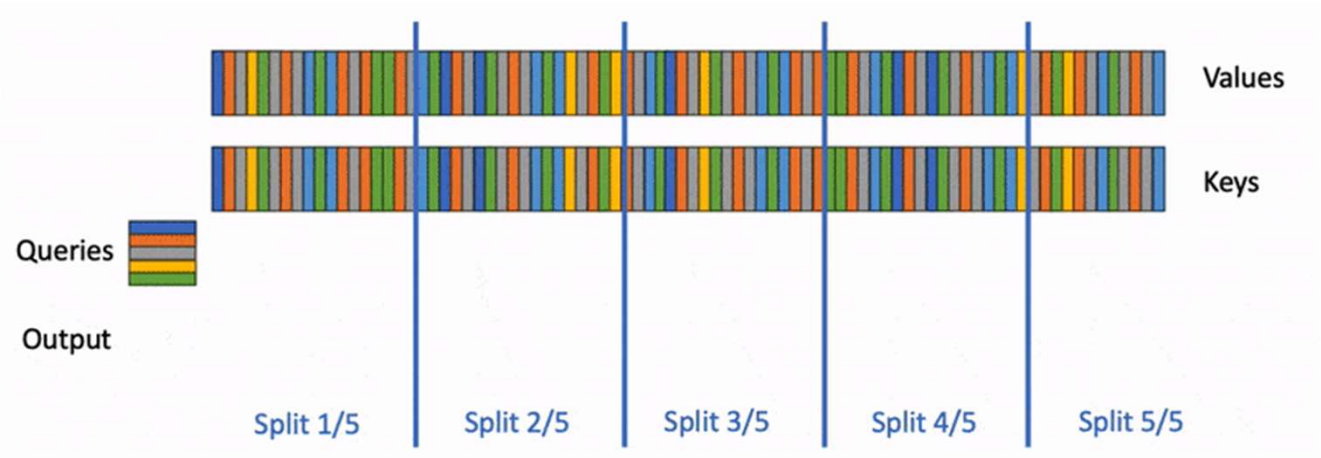


(animation) https://pytorch.org/assets/images/Inference_regular_attn.gif

Inefficient for requests with long context (many keys/values)

Flash-decoding parallelizes across keys/values

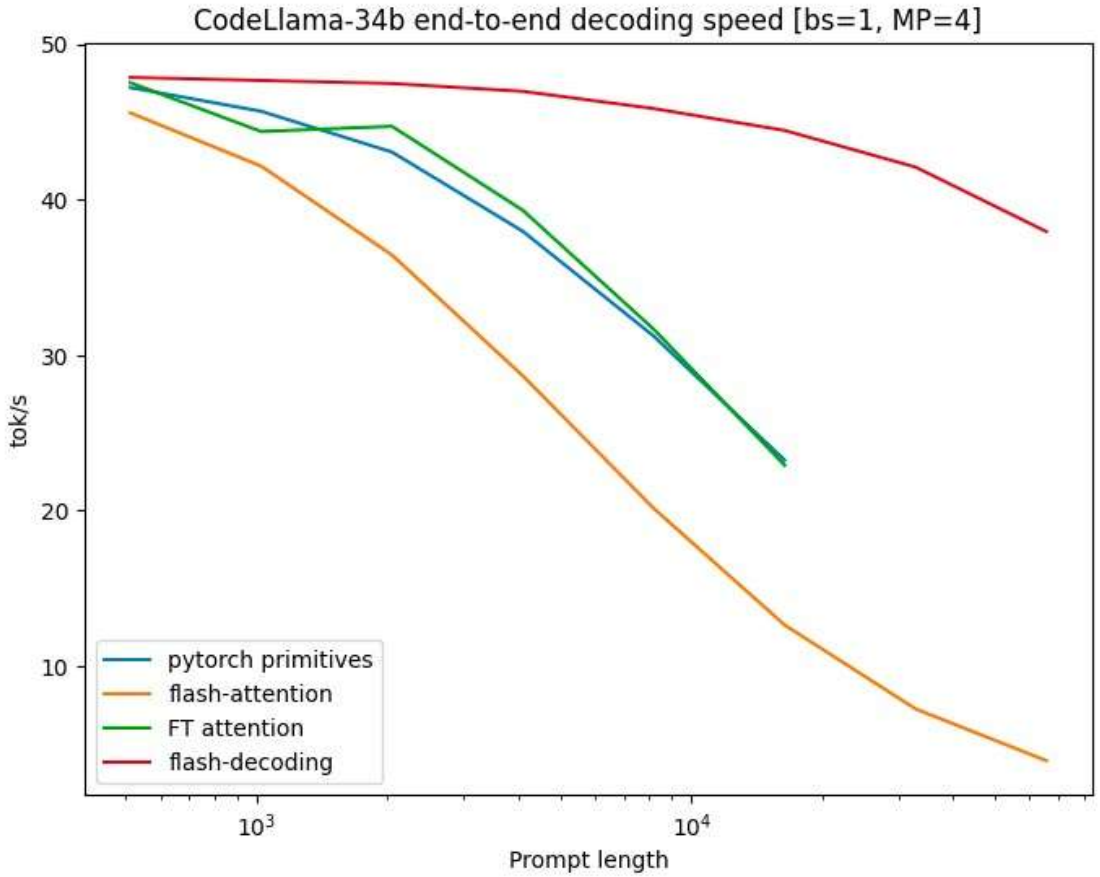
1. Split keys/values into small chunks
2. Compute attention with these splits using FlashAttention
3. Reduce overall all splits



(animation) https://pytorch.org/assets/images/inference_splitkv.gif

Key insight: attention is associative and commutative (recall Recursive Attention)

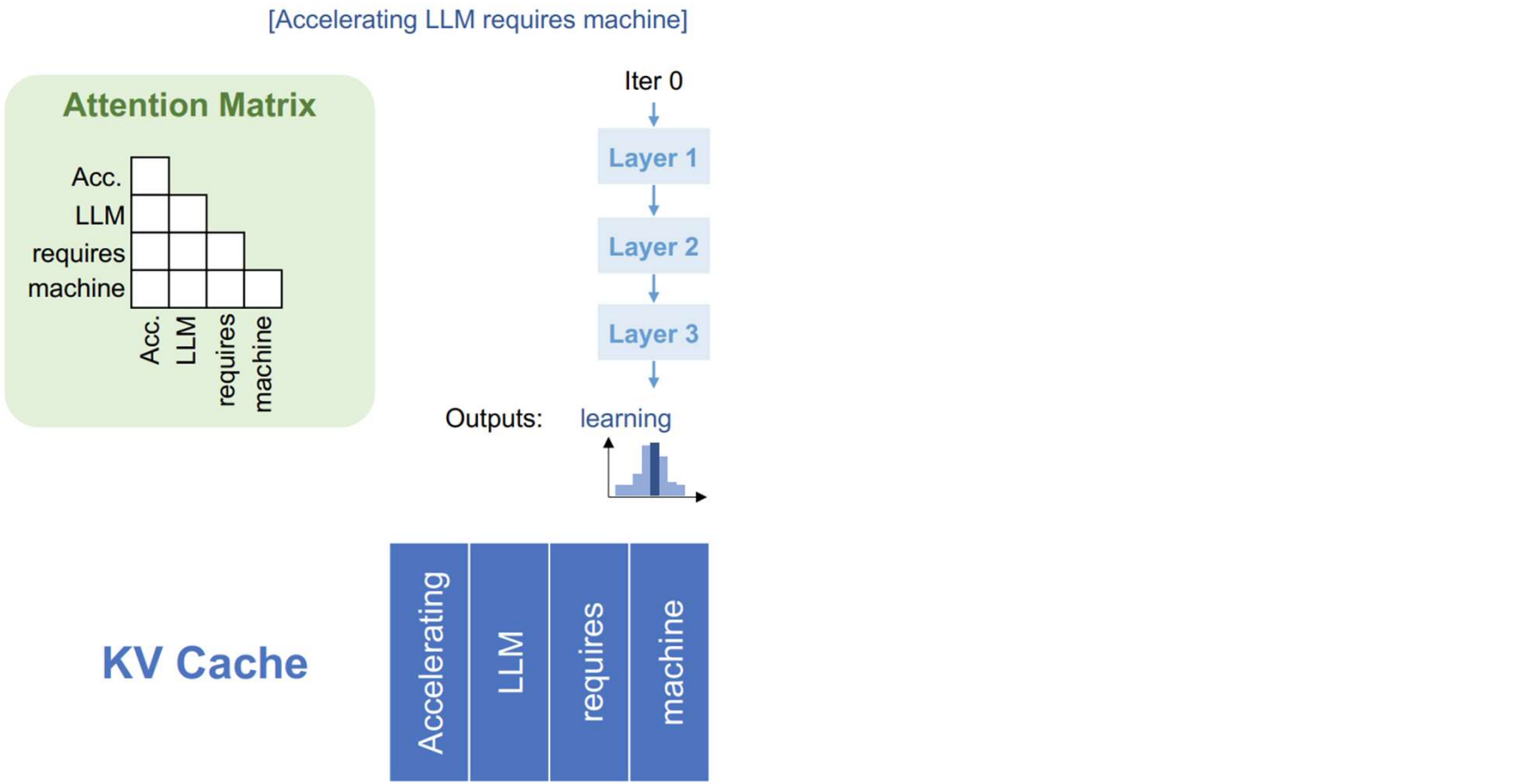
Flash-decoding is up to 8x faster than prior work



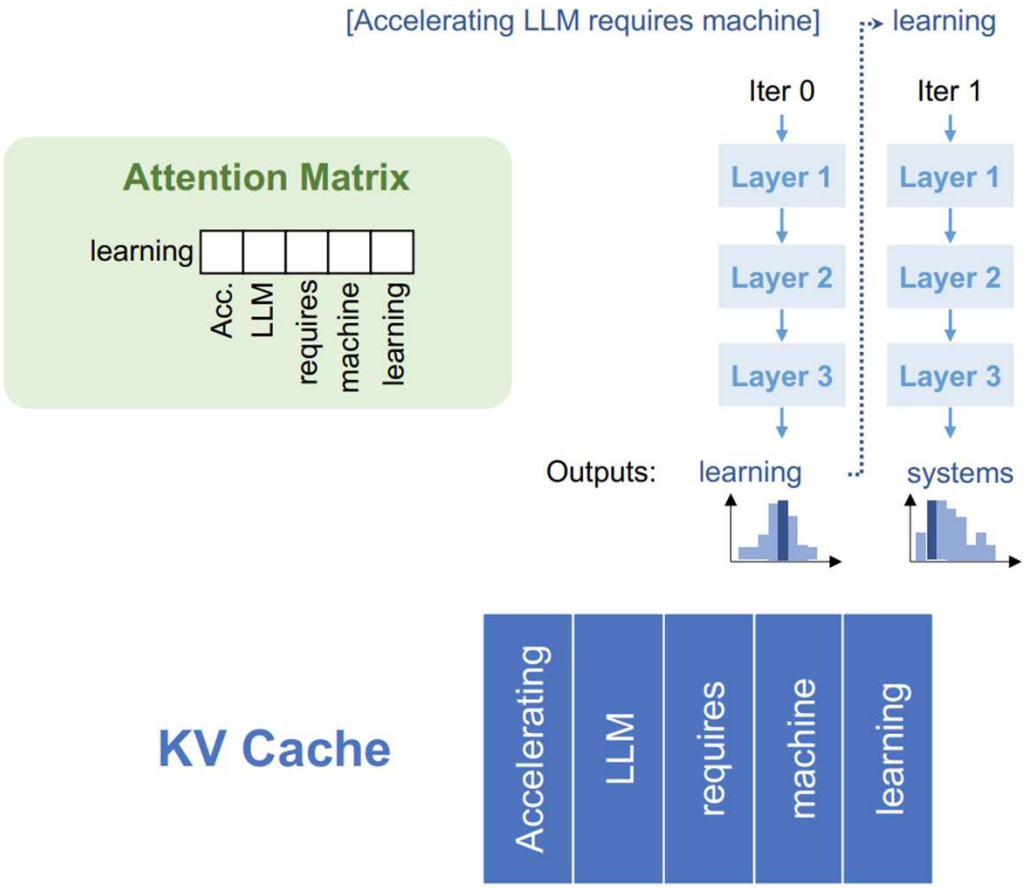
Attention optimizations

- LLM Training
  - FlashAttention
- LLM Inference
  - Recursive Attention
  - Flash Decoding
  - PagedAttention

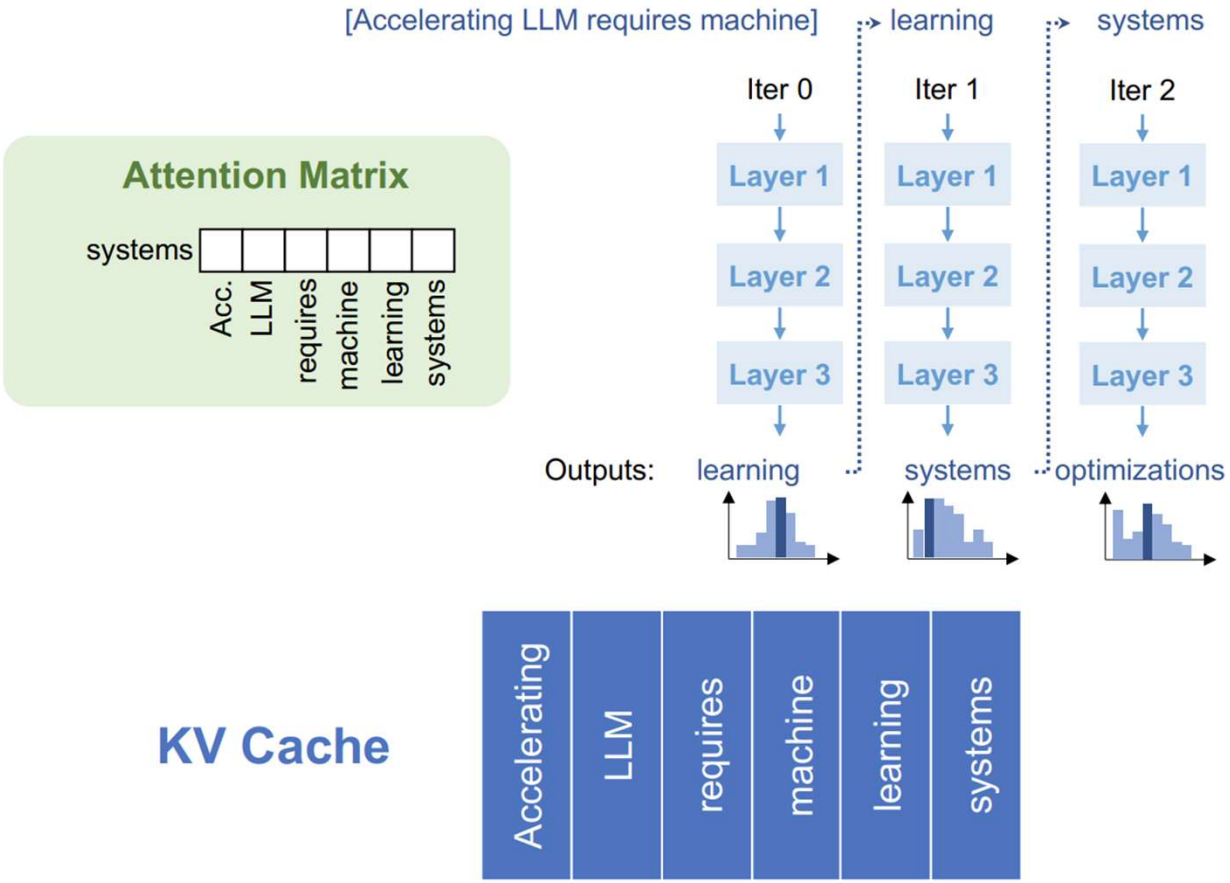
KV cache dynamically grows and shrinks



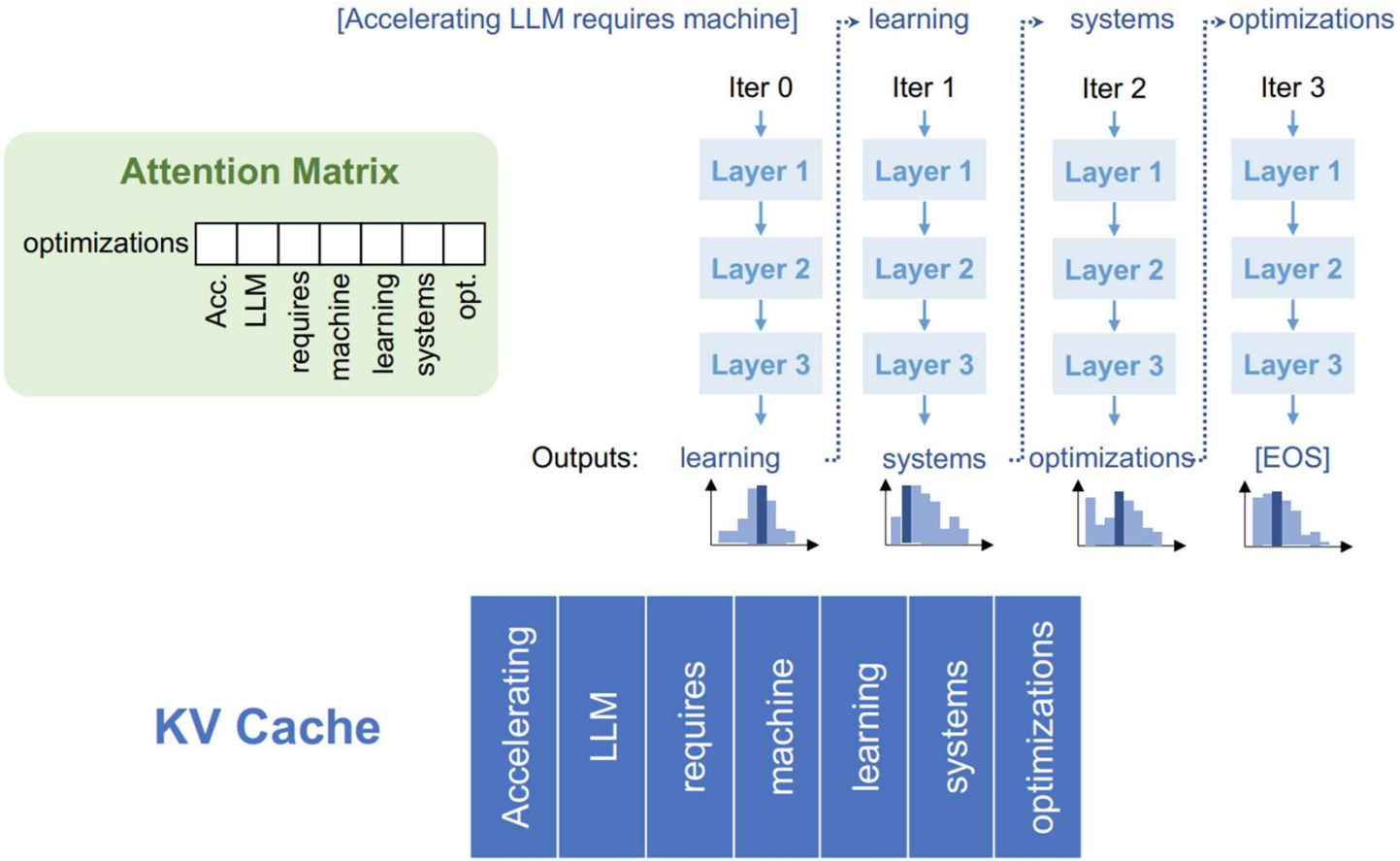
KV cache dynamically grows and shrinks



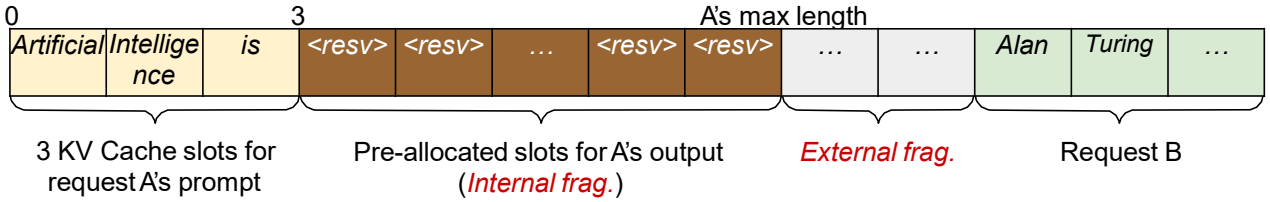
KV cache dynamically grows and shrinks



KV cache dynamically grows and shrinks



Static KV cache management wastes memory



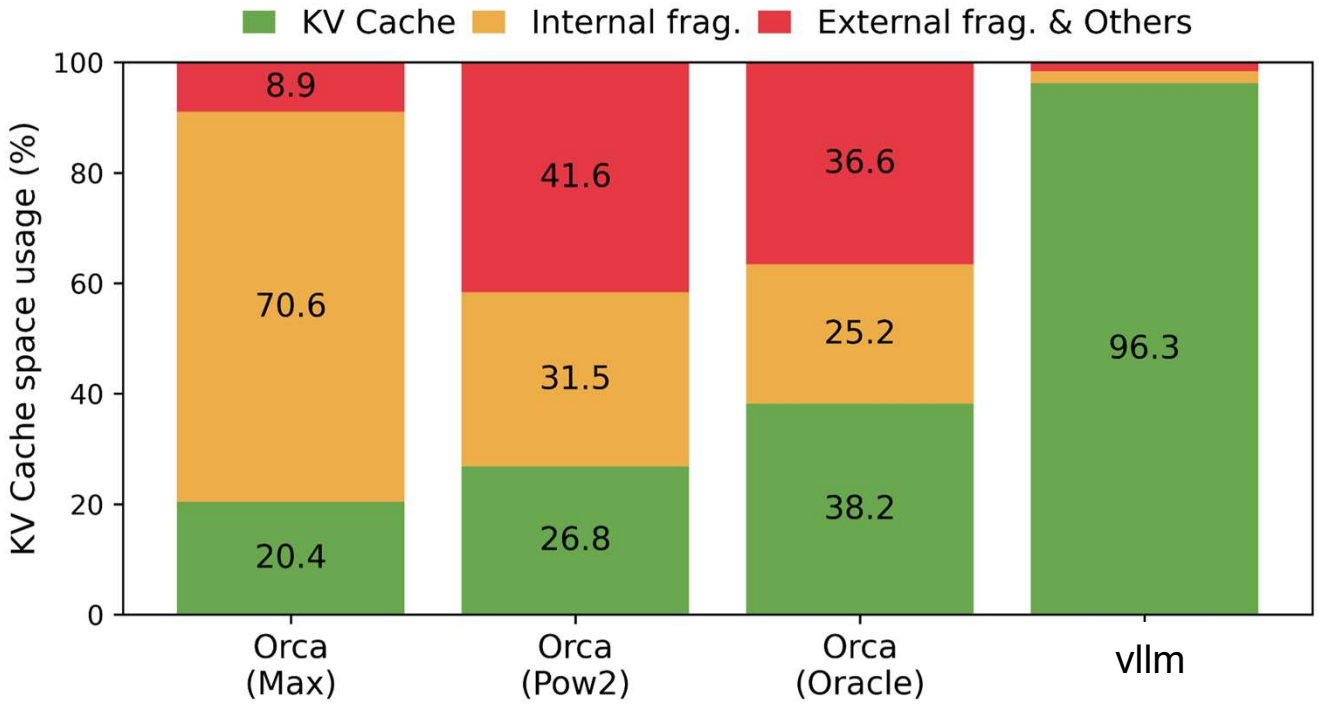
- **Pre-allocates contiguous** space of memory to the request's maximum length

- Memory fragmentation
  - **Internal fragmentation** due to unknown output length
  - **External fragmentation** due to non-uniform per-request max lengths

Slides from vllm: Efficient Memory Management for Large Language Model Serving with PagedAttention

Significant memory waste in KV cache

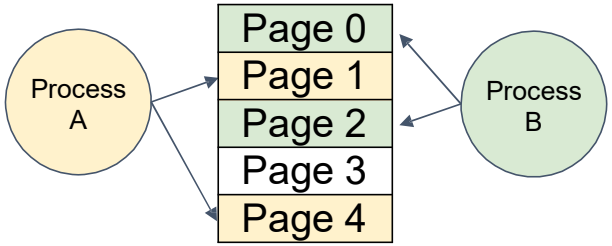
- Only 20-40% of KV cache is utilized to store actual token states



PagedAttention

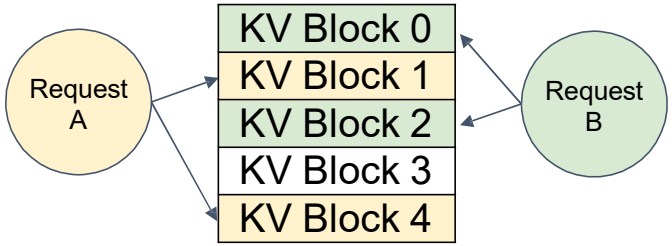
- Application-level memory paging and virtualization for KV cache

Memory management in OS



Physical Memory

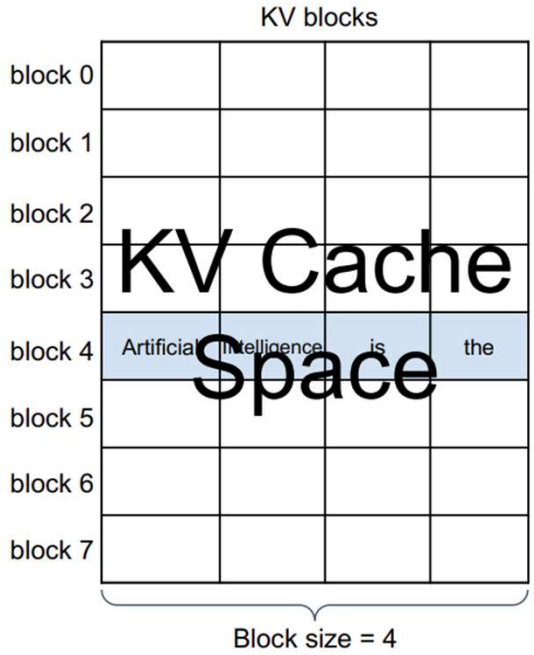
PagedAttention



KV Cache

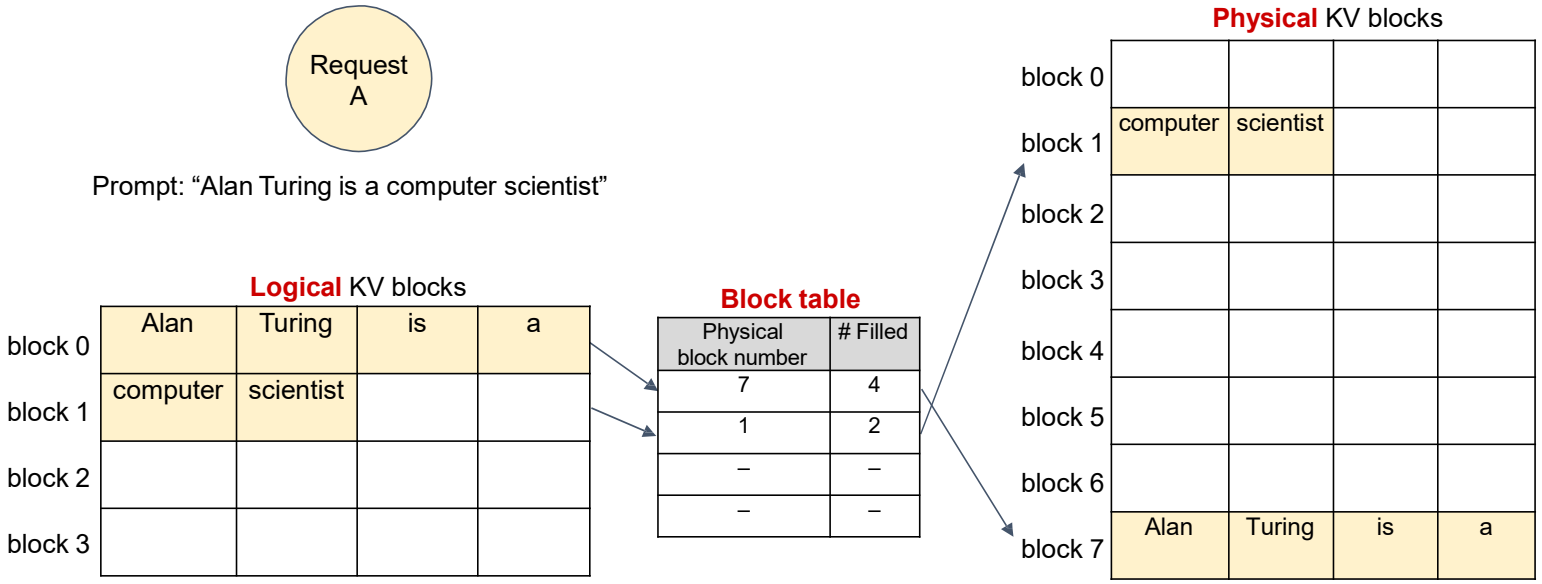
Paging KV cache space into KV blocks*

- KV block is a **fixed-size** contiguous chunk of memory that stores KV states from **left to right**



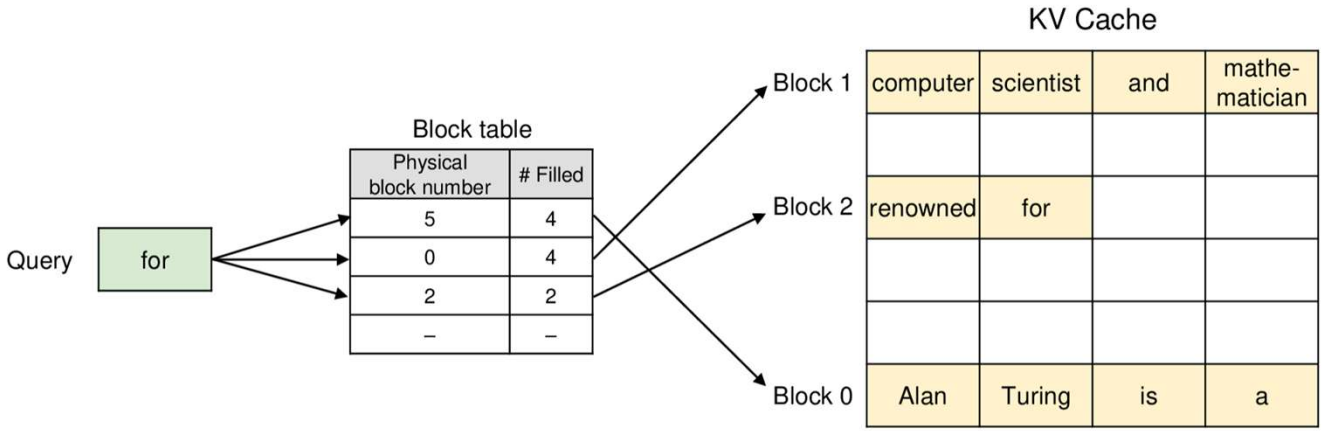
* overloaded in PagedAttention

Attention with virtualized KV cache



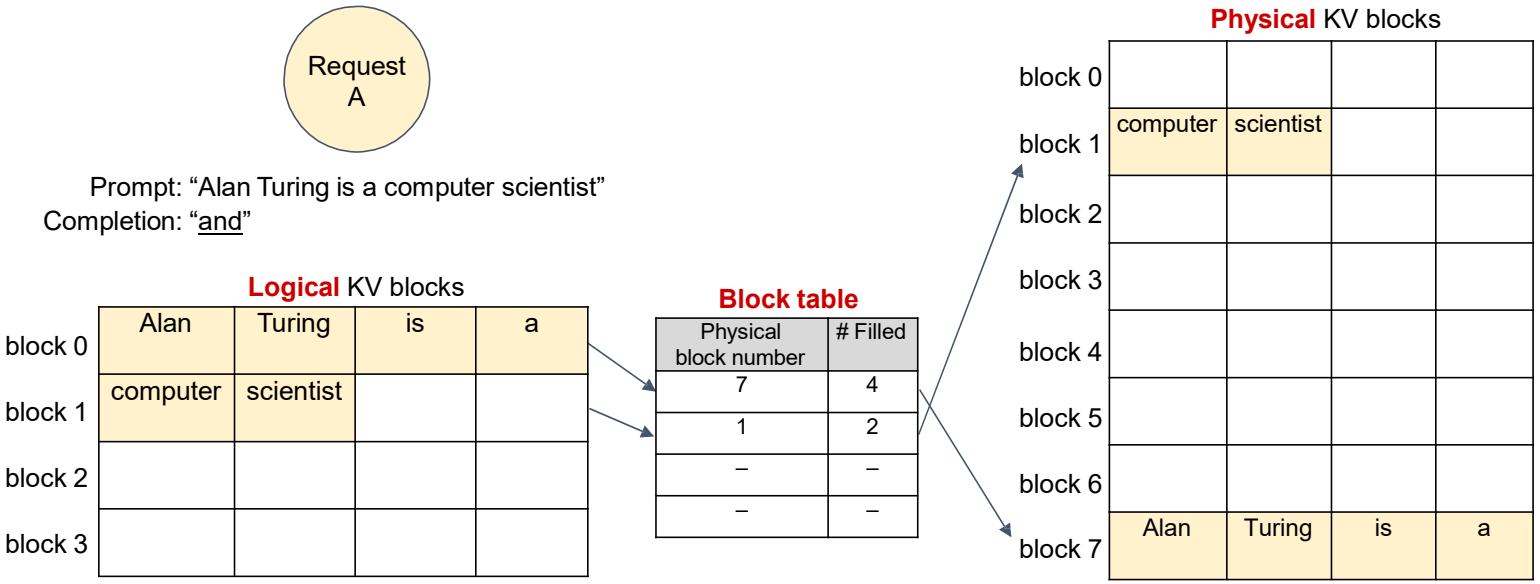
Attention with virtualized KV cache

1. Fetch non-contiguous KV blocks using the block table
2. Apply attention on the fly

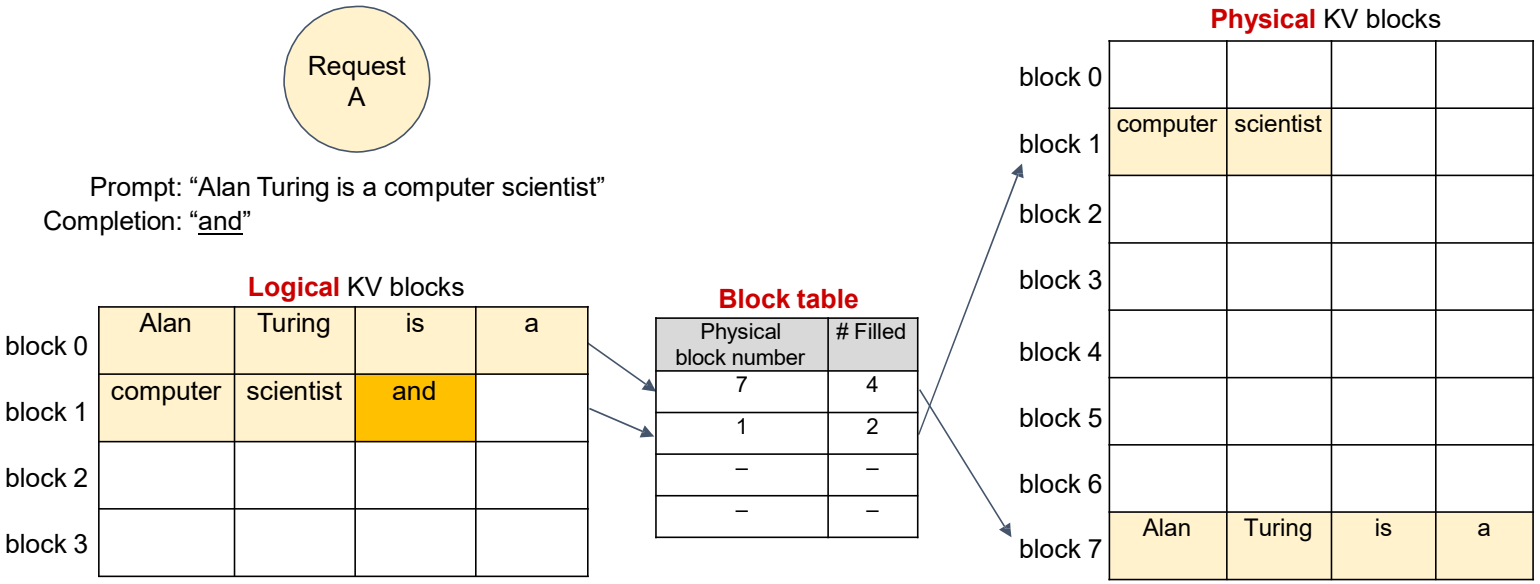


Key insight: attention is associative and commutative

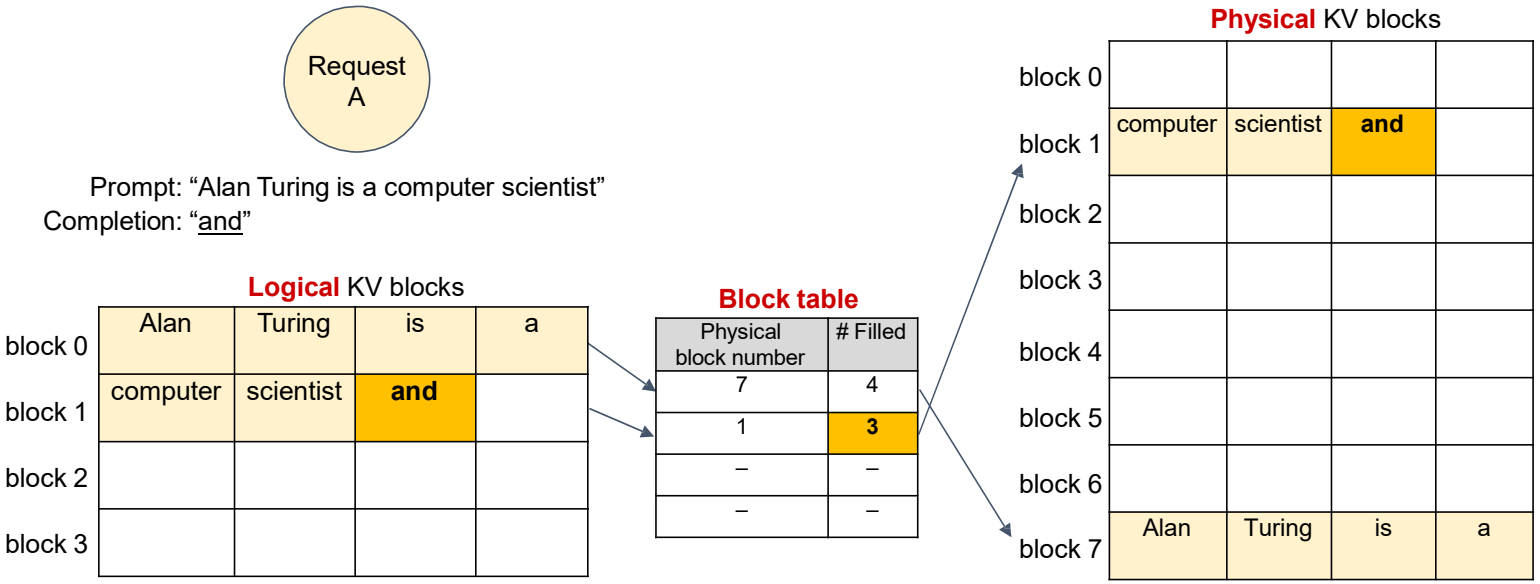
Memory management with PagedAttention



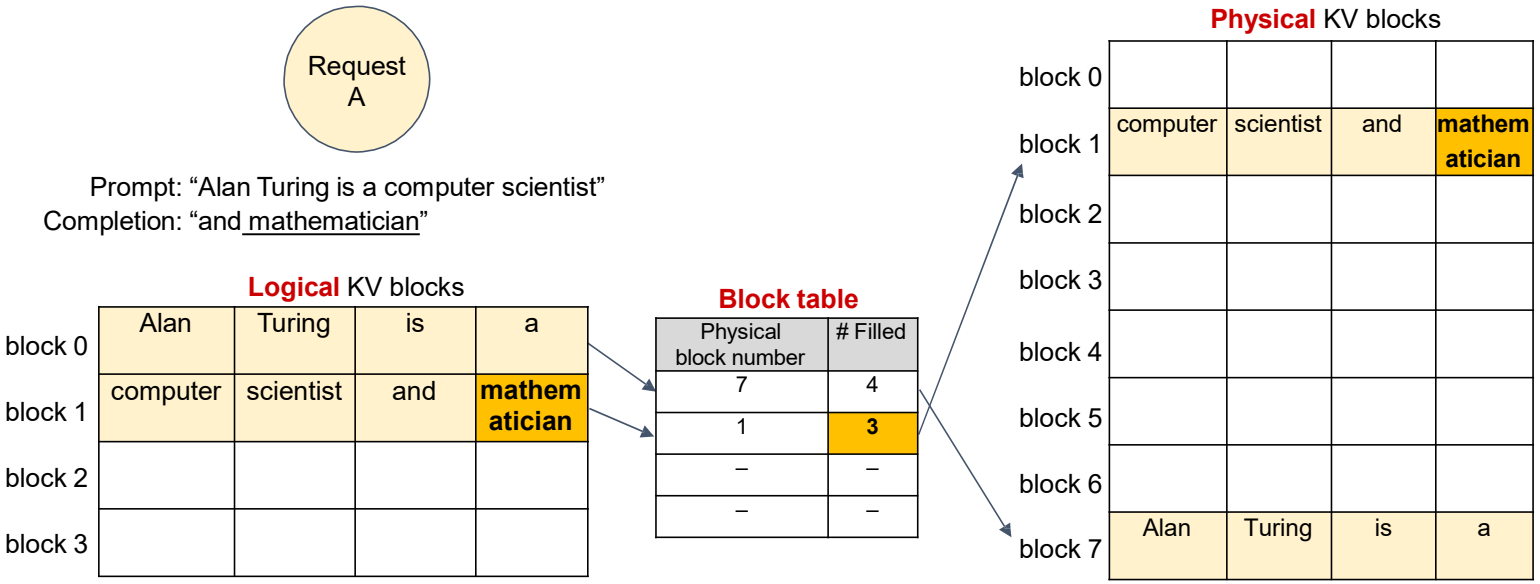
Memory management with PagedAttention



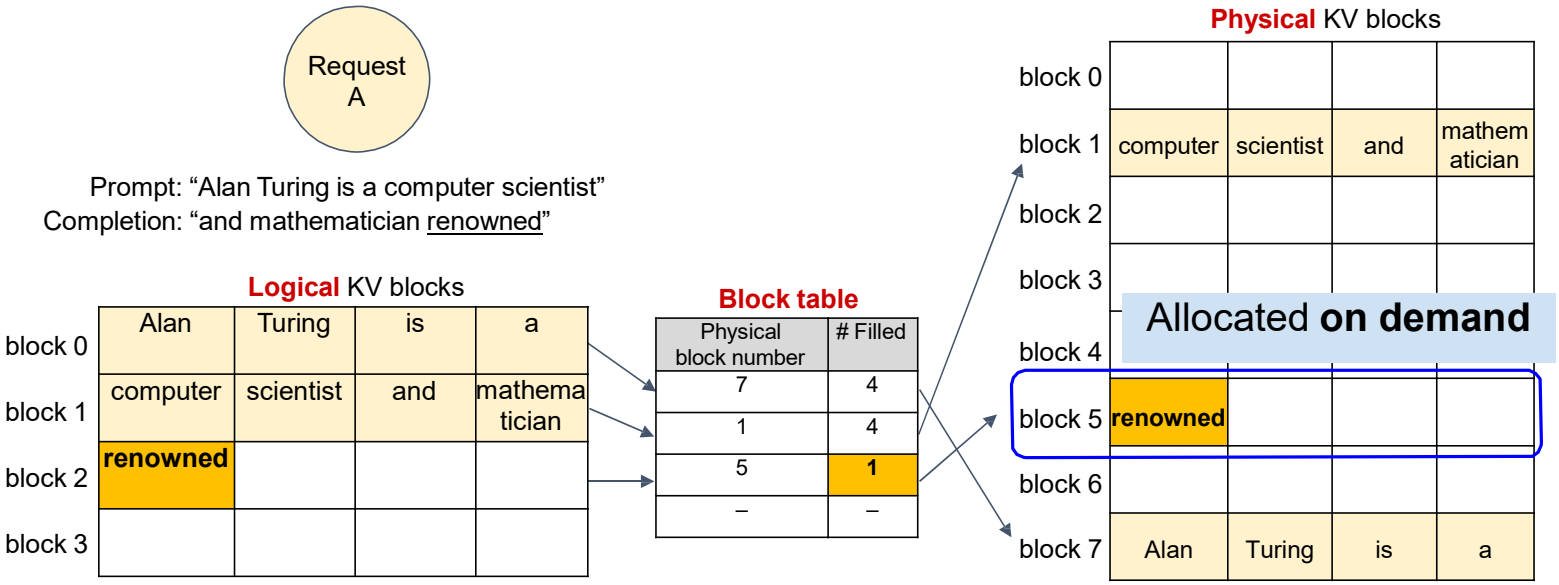
Memory management with PagedAttention



Memory management with PagedAttention



Memory management with PagedAttention



Memory efficiency of PagedAttention

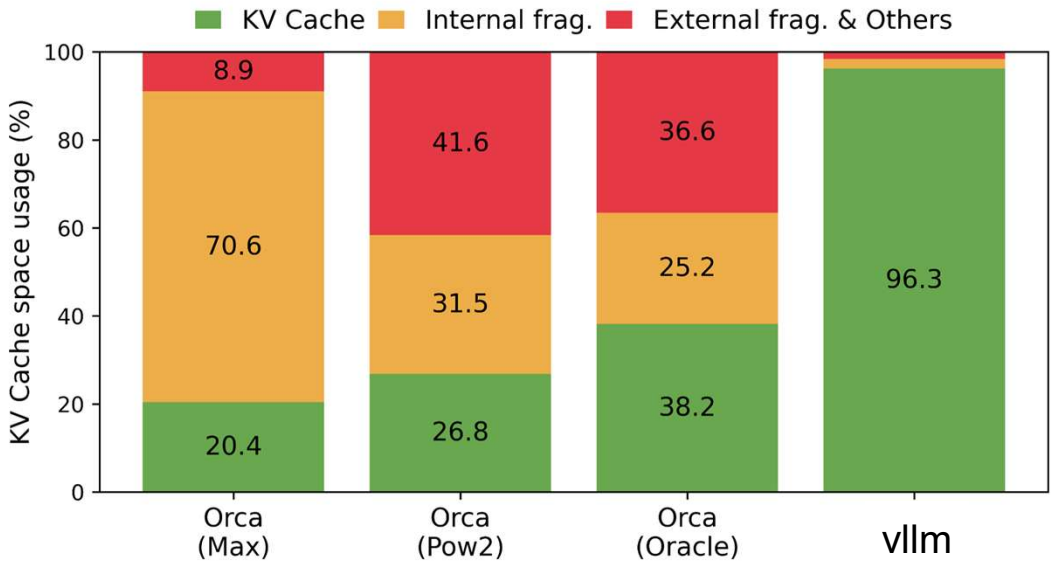
Minimal internal fragmentation

- Only happens at the last block of a sequence
- # wasted tokens / seq < block size

No external fragmentation

Alan	Turing	is	a
computer	scientist	and	mathemati cian
renowned			

Internal fragmentation



Summarize: techniques for optimizing attention

- **FlashInfer:** incremental / divide-and-conquer attention compute
- **FlashAttention**: tiling to reduce GPU global memory access
- **Auto-regressive Decoding**: pre-filling and decoding phases, KV cache
- **FlashDecoding**: improving attention's parallelism by splitting keys/values
- **PagedAttention**: paging and virtualization to reduce KV cache's memory requirement

Recess next week

- **No lecture**