Making Long-Context Attention Efficient

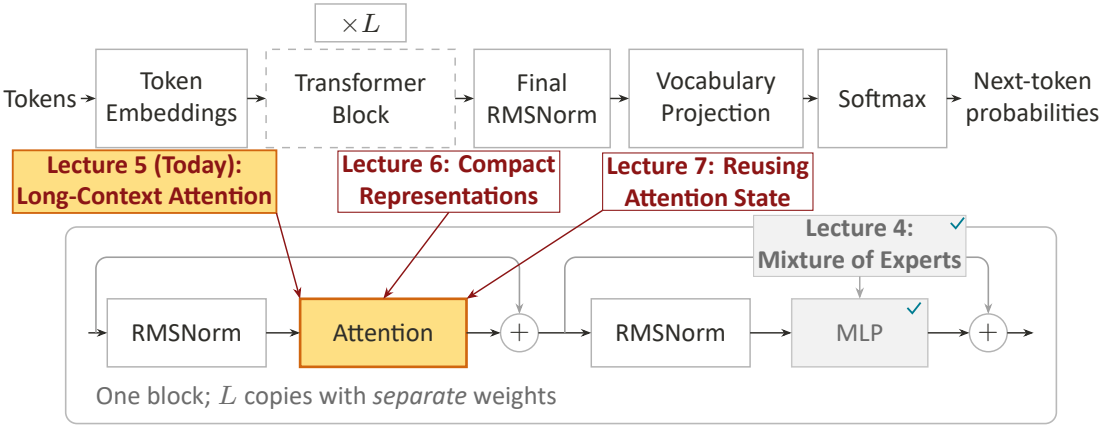
Lecture 5: KV Caching, Sparse Attention, and Compression

Anay Mehrotra

MS&E 319

Autumn 2026

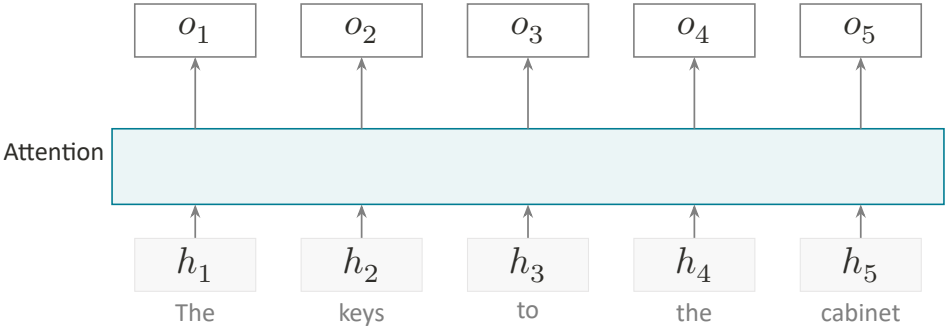
Topic 2: Innovations in Architecture



Keep quality fixed. Can each token cost less?

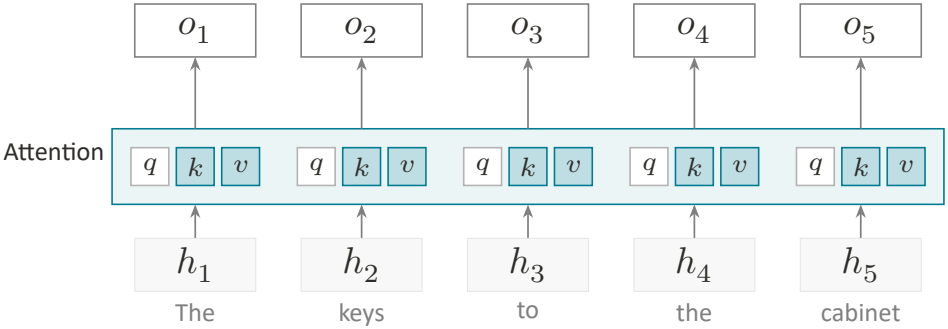
Attention: Inputs and Outputs

Recall (Lecture 2): attention takes T vectors and returns T vectors.



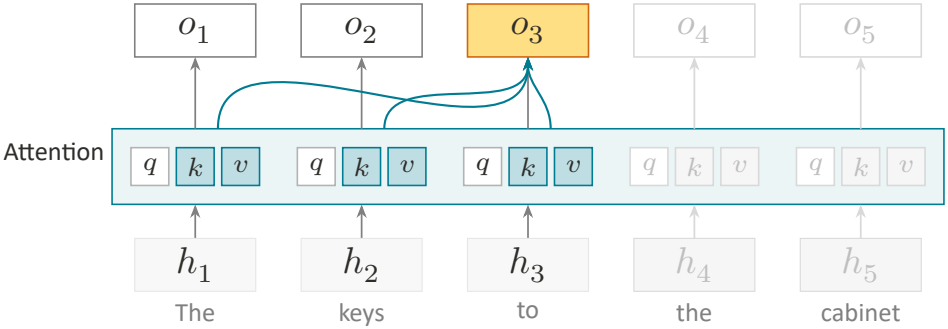
Attention: Inputs and Outputs

Recall (Lecture 2): attention takes T vectors and returns T vectors.



Attention: Inputs and Outputs

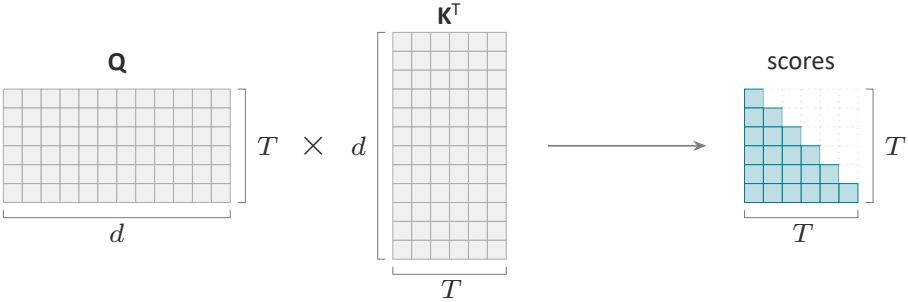
Recall (Lecture 2): attention takes T vectors and returns T vectors.



Computing All Attention Scores

One causal pass: compute all T outputs in one layer.

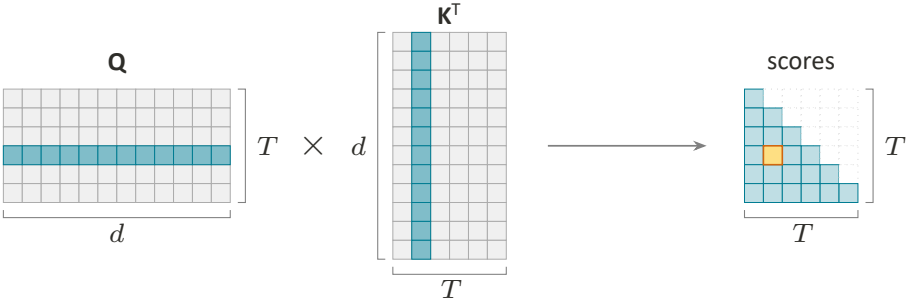
Each query compares with the keys at its position and earlier.



Computing All Attention Scores

One causal pass: compute all T outputs in one layer.

Each query compares with the keys at its position and earlier.

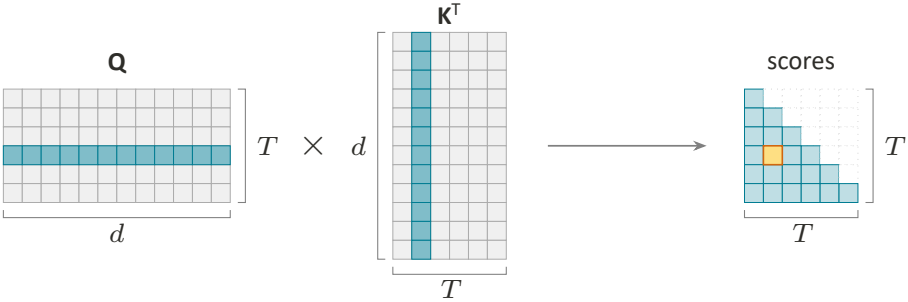


$T(T+1)/2$ dot products; about $2d$ FLOPs each.

Computing All Attention Scores

One causal pass: compute all T outputs in one layer.

Each query compares with the keys at its position and earlier.



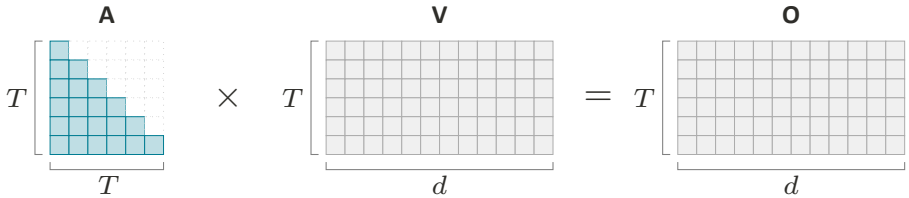
$T(T+1)/2$ dot products; about $2d$ FLOPs each.

All scores: $[T(T+1)/2](2d) \approx T^2 d$ FLOPs.

Computing All Attention Outputs

One causal pass: compute all T outputs in one layer.

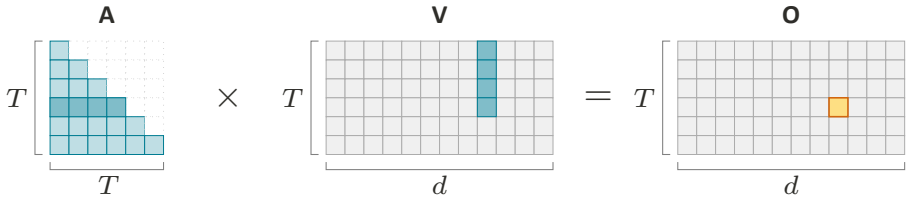
Apply softmax to each score row, then combine the values.



Computing All Attention Outputs

One causal pass: compute all T outputs in one layer.

Apply softmax to each score row, then combine the values.



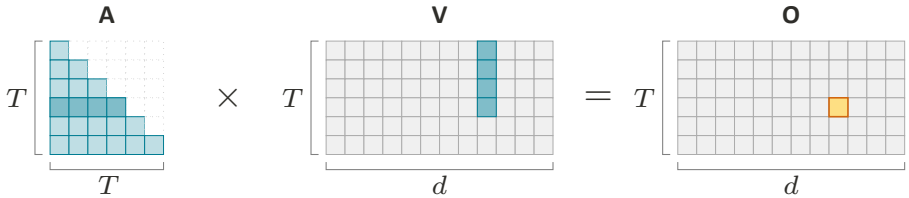
Row i: d weighted sums over i values.

All weighted sums: about $T^2 d$ FLOPs.

Computing All Attention Outputs

One causal pass: compute all T outputs in one layer.

Apply softmax to each score row, then combine the values.



Row i: d weighted sums over i values.

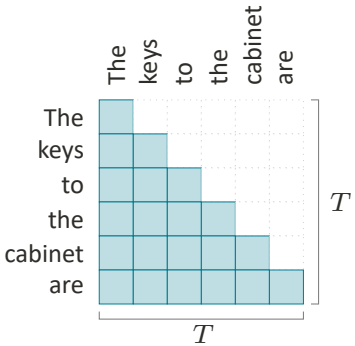
All weighted sums: about $T^2 d$ FLOPs.

Scores plus weighted sums: about $2T^2 d$ FLOPs.

One Full Pass Through a Layer

One layer has about $12d^2$ weights.

Applying them to T tokens costs about $24Td^2$ FLOPs.



$$T(T+1)/2 \approx T^2/2$$

Rows: queries. Columns: keys and values.

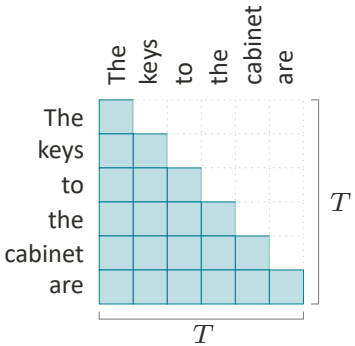
Operation	FLOPs
Projections	$8Td^2$
Scores	$\approx T^2 d$
Softmax	$\approx \frac{3}{2}HT^2$
Weighted sum	$\approx T^2 d$
MLP	$16Td^2$

H heads; d = total head width.
Two MLP matrices, width $4d$.

One Full Pass Through a Layer

One layer has about $12d^2$ weights.

Applying them to T tokens costs about $24Td^2$ FLOPs.



$$T(T+1)/2 \approx T^2/2$$

Rows: queries. Columns: keys and values.

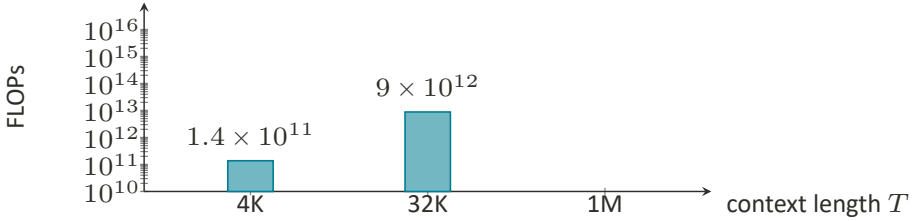
Operation	FLOPs
Projections	$8Td^2$
Scores	$\approx T^2 d$
Softmax	$\approx \frac{3}{2}HT^2$
Weighted sum	$\approx T^2 d$
MLP	$16Td^2$

H heads; d = total head width.
Two MLP matrices, width $4d$.

The Cost of a Full Pass

One causal pass through one layer; $d = 4{,}096$.

$$\text{FLOPs} \approx \underbrace{24Td^2}_{\text{projections and MLP}} + \underbrace{2T^2d}_{\text{scores and weighted sums}}$$

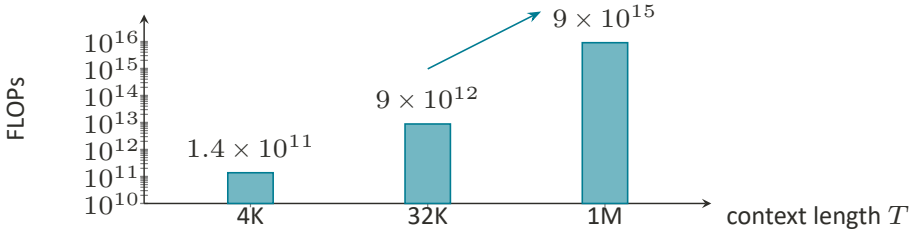


The Cost of a Full Pass

One causal pass through one layer; $d = 4{,}096$.

$$\text{FLOPs} \approx \underbrace{24Td^2}_{\text{projections and MLP}} + \underbrace{2T^2d}_{\text{scores and weighted sums}}$$

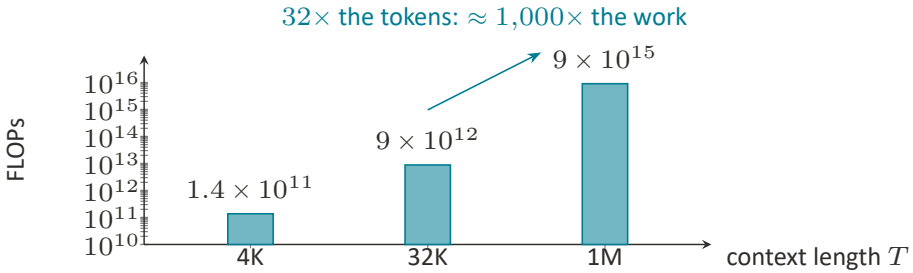
$32\times$ the tokens: $\approx 1{,}000\times$ the work



The Cost of a Full Pass

One causal pass through one layer; $d = 4{,}096$.

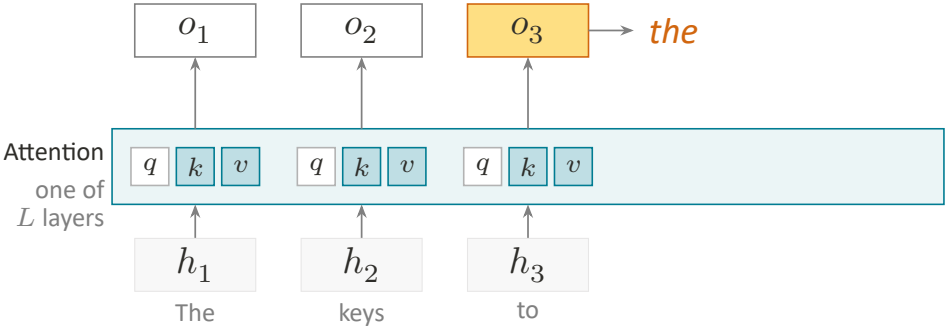
$$\text{FLOPs} \approx \underbrace{24Td^2}_{\text{projections and MLP}} + \underbrace{2T^2d}_{\text{scores and weighted sums}}$$



Much of this work repeats for each token and can be avoided.

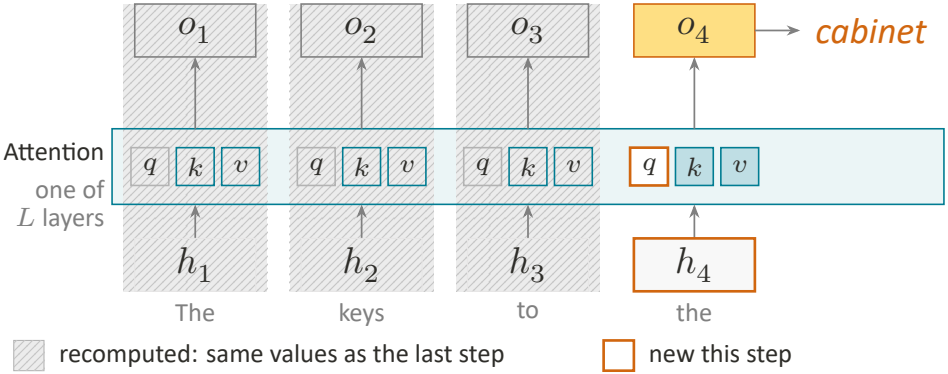
Generating One Token at a Time

Step 1: run the model on *The keys to*; predict *the*.



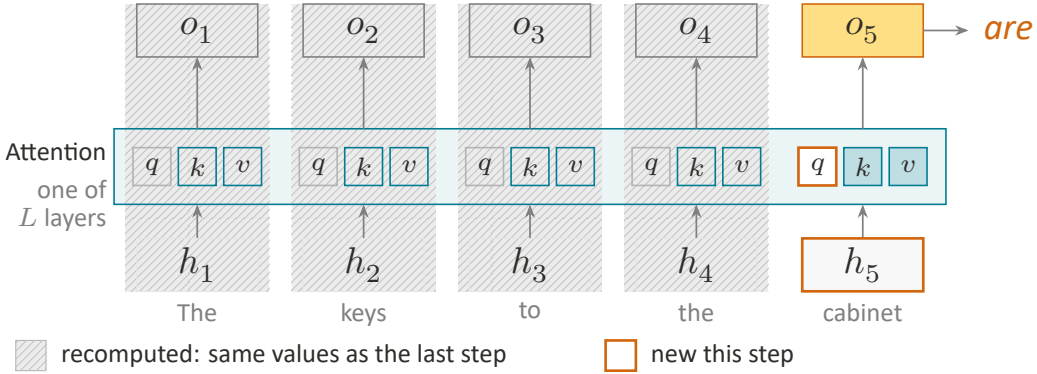
Generating One Token at a Time

Step 2: run it again on all four tokens; predict *cabinet*.



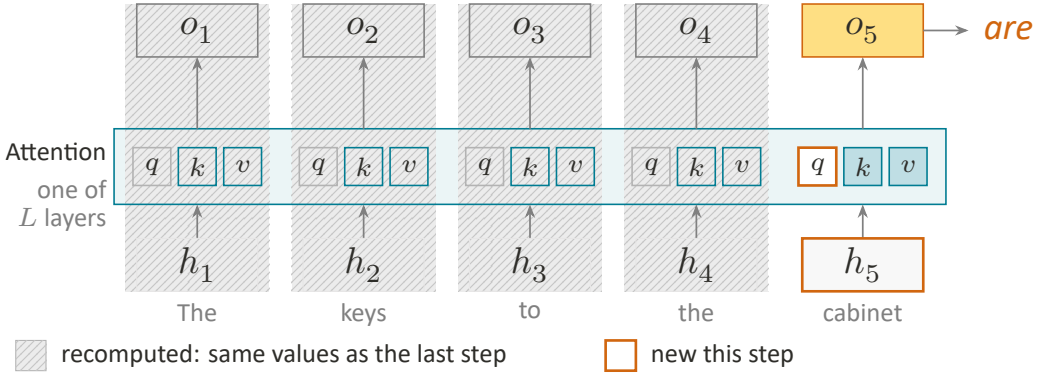
Generating One Token at a Time

Step 3: run it again on all five tokens; predict *are*.



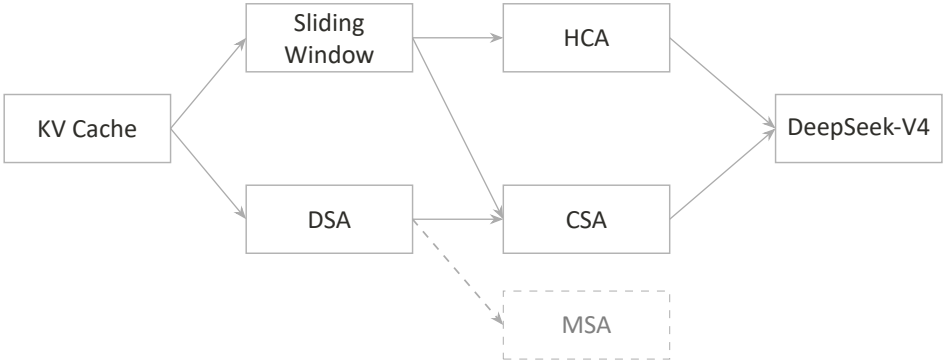
Generating One Token at a Time

Step 3: run it again on all five tokens; predict *are*.

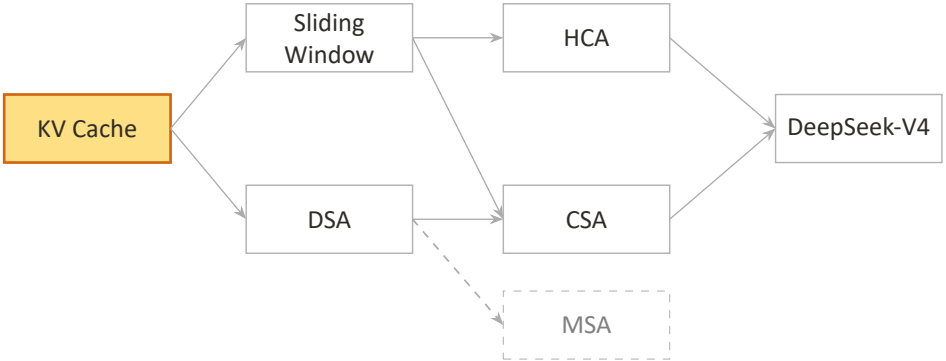


How can we avoid this repeated work?

Efficient Long-Context Attention

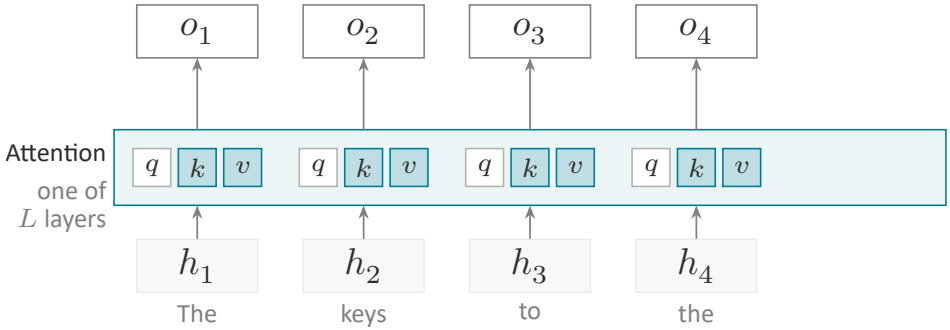


Efficient Long-Context Attention



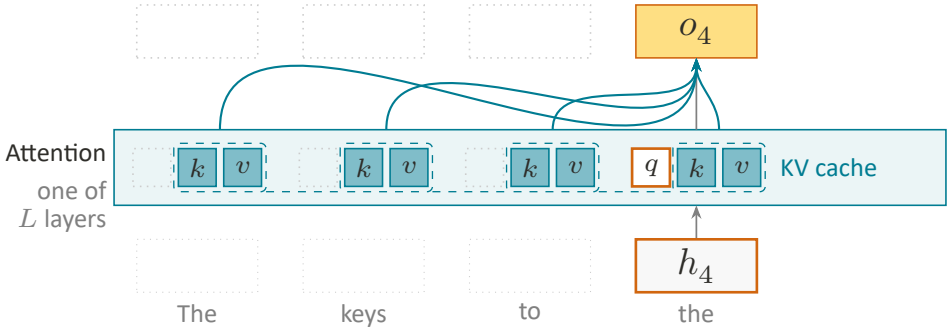
KV Caching

When token 4 arrives, nothing at positions 1–3 changes.



KV Caching

When token 4 arrives, nothing at positions 1–3 changes.

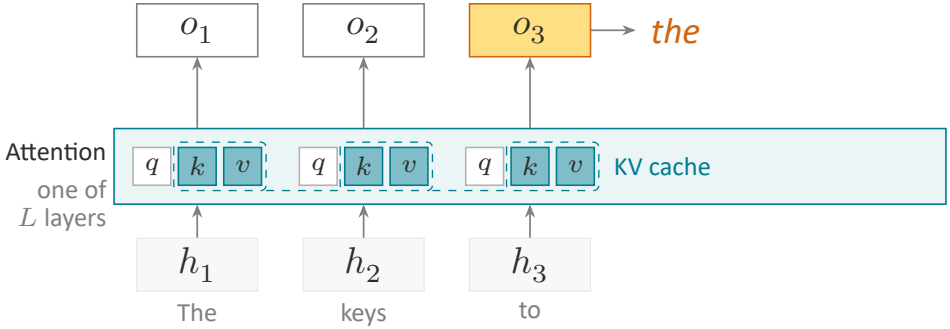


Keep every k_j, v_j, in every layer: this is the KV cache.

KV Caching: How to Use the Cache?

Shazeer (2019)

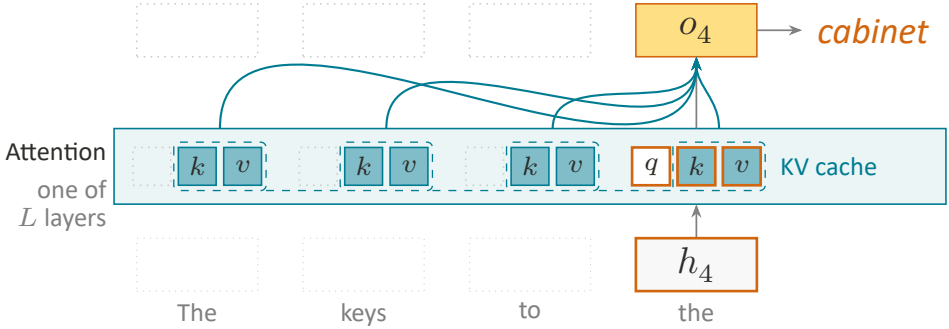
Prefill: run the prompt's positions together; store each k_j, v_j.



KV Caching: How to Use the Cache?

Shazeer (2019)

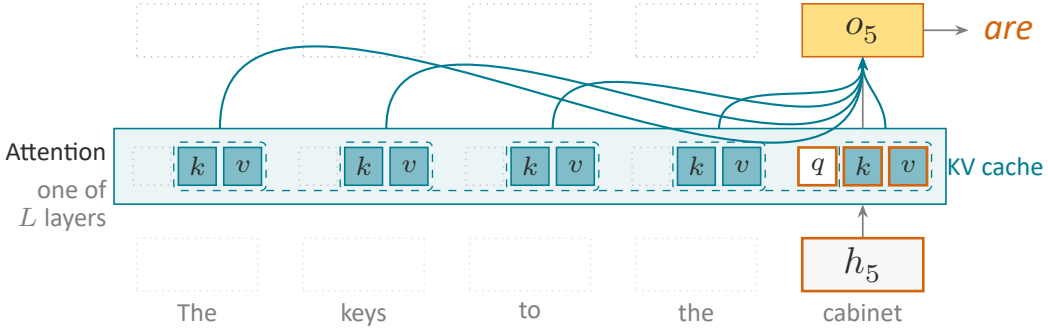
Decode: compute only position 4; append (k_4, v_4); read all four.



KV Caching: How to Use the Cache?

Shazeer (2019)

Decode: compute only position 5; append (k_5, v_5); read all five.



KV Caching

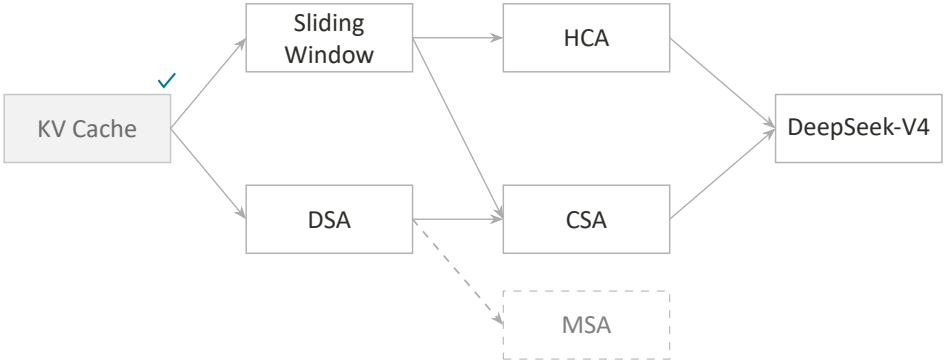
Operation	FLOPs without KV cache
Projections	$8td^2$
Scores	$\approx t^2 d$
Softmax	$\approx \frac{3}{2} H t^2$
Weighted sum	$\approx t^2 d$
MLP	$16td^2$
Total	$\approx 24td^2 + 2t^2 d$

KV Caching

Operation	FLOPs without KV cache	FLOPs with KV cache
Projections	$8td^2$	$8d^2$
Scores	$\approx t^2 d$	$\approx 2td$
Softmax	$\approx \frac{3}{2}Ht^2$	$\approx 3Ht$
Weighted sum	$\approx t^2 d$	$\approx 2td$
MLP	$16td^2$	$16d^2$
Total	$\approx 24td^2 + 2t^2 d$	$\approx 24d^2 + 4td$

With caching: $O(d^2 + td)$ per token, not $O(d^2 t + t^2 d)$.

Efficient Long-Context Attention

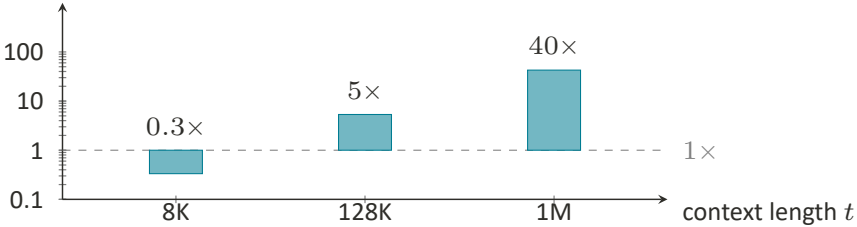


The Cost of Cached Decoding

With a KV cache: one new token, in one layer.

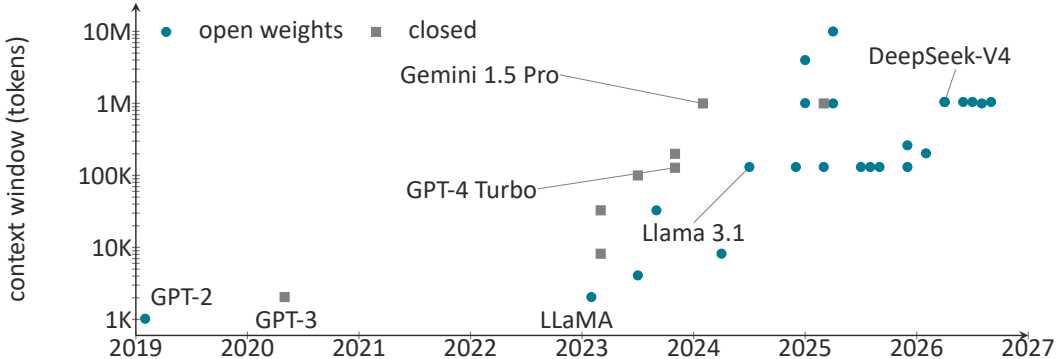
$$\text{FLOPs per token} \approx \underbrace{24d^2}_{2\times\text{ parameters}} + \underbrace{4td}_{\text{attention}} = 24d^2\left(1 + \frac{t}{6d}\right)$$

Cost of attention relative to $2\times$ parameters, with $d = 4{,}096$



Growth in Context-Window Size

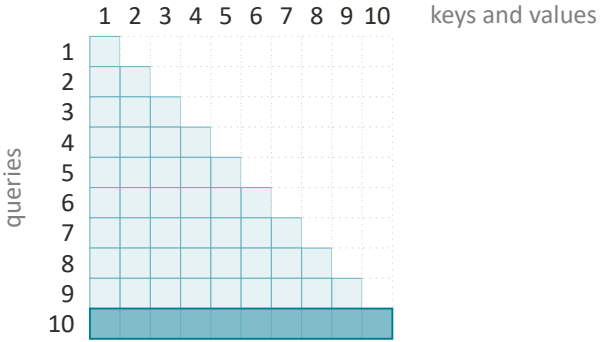
Context window: maximum supported number of context tokens.



Sources: several official reports and model cards.

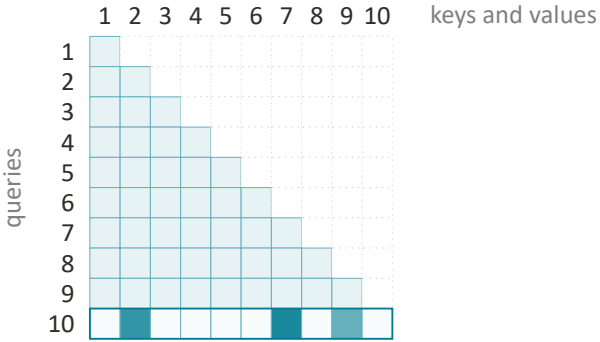
Full Attention Reads Every Earlier Token

Each new token reads the keys and values of all earlier tokens.



Full Attention Reads Every Earlier Token

Each new token reads the keys and values of all earlier tokens.

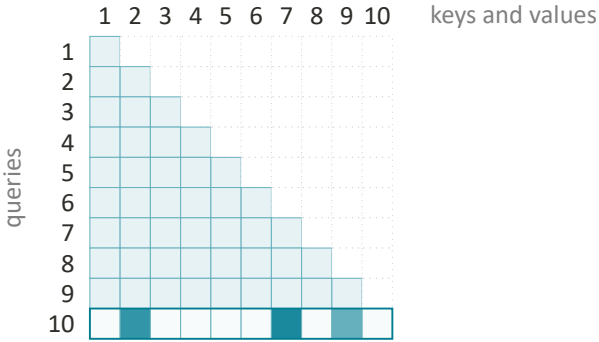


Yet at 128K, 3% of positions receive about 96% of the attention.

Source: Jiang et al. (2024), MInference, Fig. 2(b); LLaMA-3-8B.

Full Attention Reads Every Earlier Token

Each new token reads the keys and values of all earlier tokens.

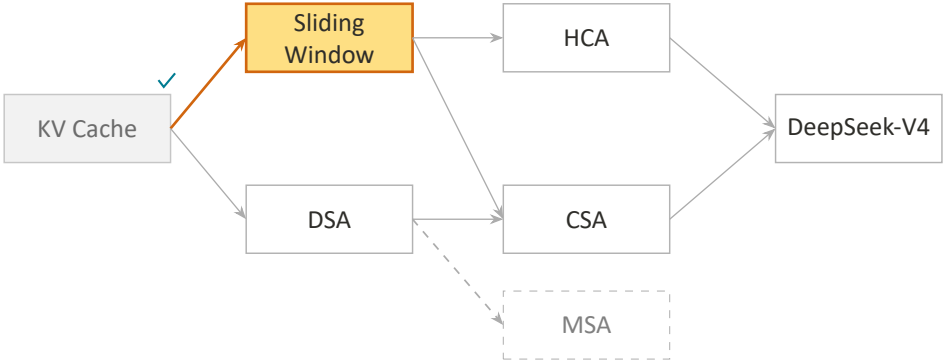


Yet at 128K, 3% of positions receive about 96% of the attention.

Source: Jiang et al. (2024), MInference, Fig. 2(b); LLaMA-3-8B.

Can each new token read only the entries it needs?

Efficient Long-Context Attention

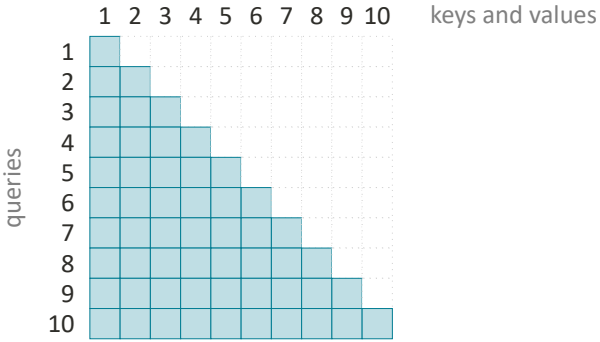


What if each layer forgot everything but its last few tokens?

Sliding-Window Attention

Child et al. (2019)

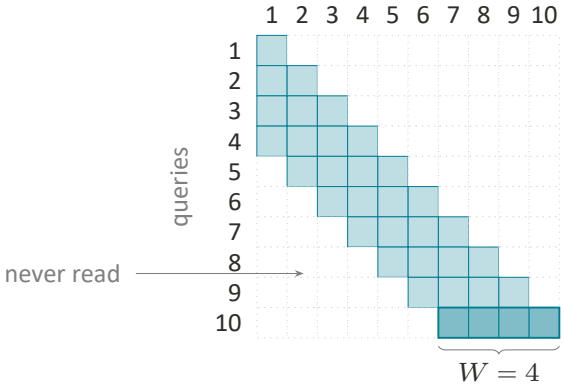
Each query reads only the last W positions, including its own.



Sliding-Window Attention

Child et al. (2019)

Each query reads only the last W positions, including its own.



pairs per layer:

about TW, not $T^2/2$

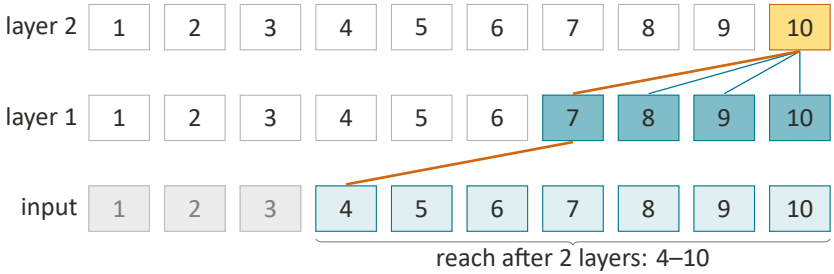
Older entries are never read again, so each layer keeps only W.

Sliding-Window Attention

Tokens outside the window still reach t through earlier layers.

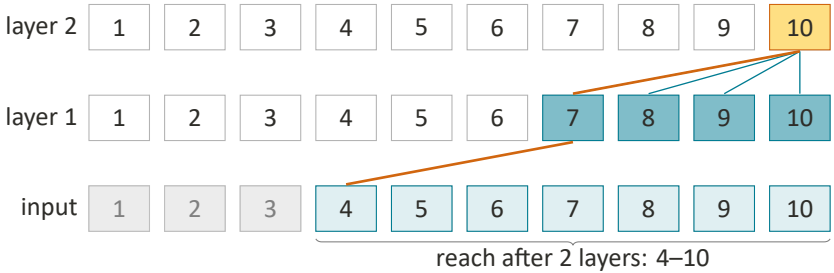
Sliding-Window Attention

Tokens outside the window still reach t through earlier layers.



Sliding-Window Attention

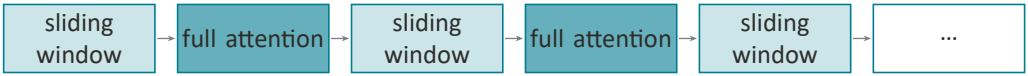
Tokens outside the window still reach t through earlier layers.



Mistral 7B: $W = 4{,}096$, 32 layers; about 131K tokens of reach.

Sliding-Window Variants

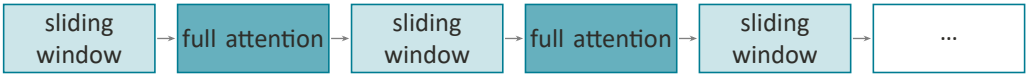
Alternating attention styles: gpt-oss (2025).



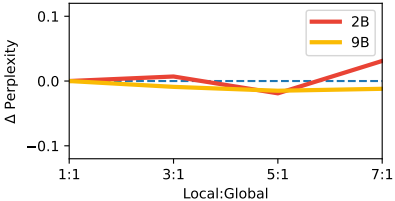
Source: Gemma Team (2025), Gemma 3, Fig. 3; 2B and 9B models.

Sliding-Window Variants

Alternating attention styles: gpt-oss (2025).



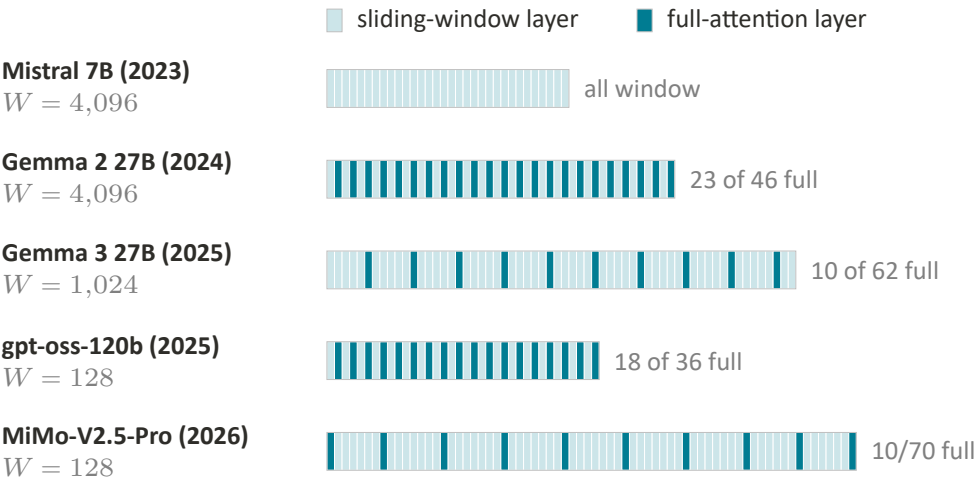
How many full layers are needed?



Validation perplexity barely changes in this test.

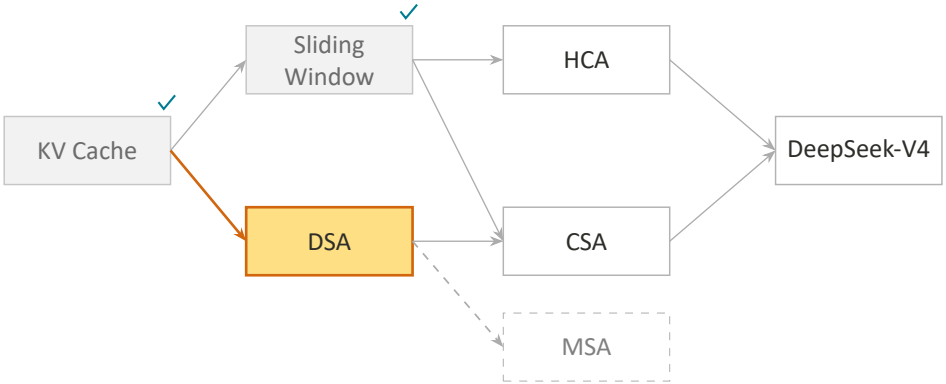
Source: Gemma Team (2025), Gemma 3, Fig. 3; 2B and 9B models.

Released Models Use Windows



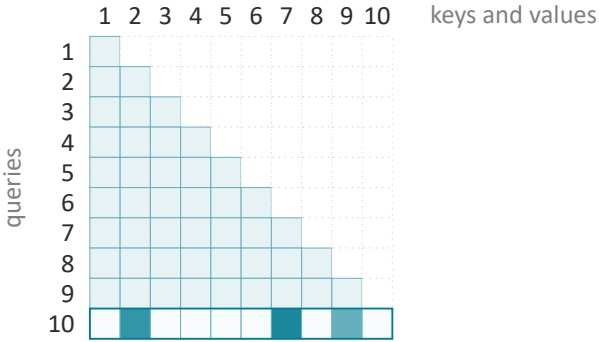
Sources: model reports and configurations.

Efficient Long-Context Attention



Full Attention Reads Every Earlier Token

Each new token reads the keys and values of all earlier tokens.



Yet at 128K, 3% of positions receive about 96% of the attention.

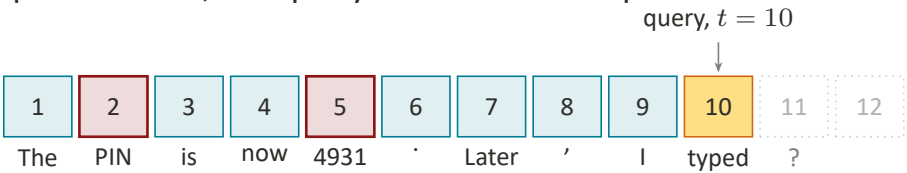
Source: Jiang et al. (2024), MInference, Fig. 2(b); LLaMA-3-8B.

Can each new token read only the entries it needs?

DeepSeek Sparse Attention (DSA)

DeepSeek-AI (2025)

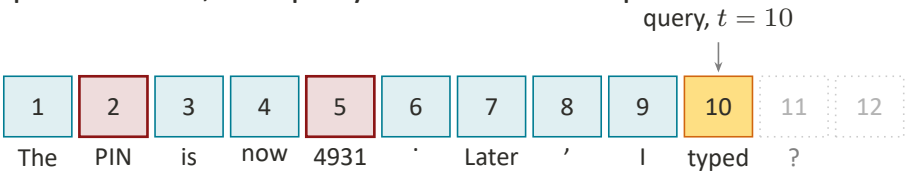
To predict *4931*, the query at $t = 10$ needs positions 2 and 5.



DeepSeek Sparse Attention (DSA)

DeepSeek-AI (2025)

To predict *4931*, the query at $t = 10$ needs positions 2 and 5.



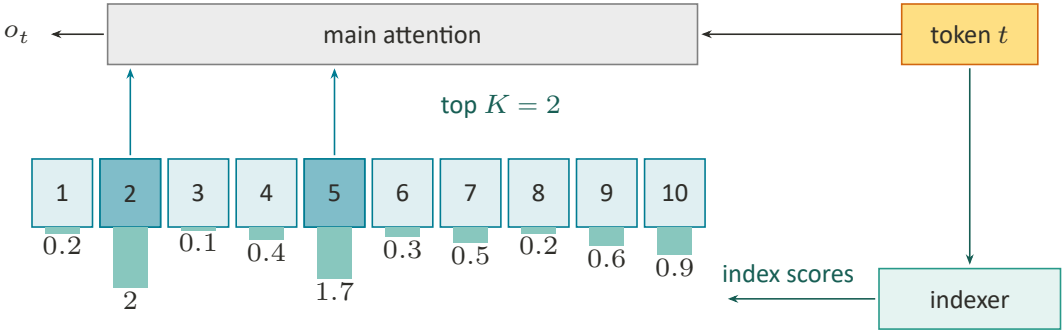
Real tests hide such a number at a random depth in a long prompt.

Source: Mohtashami and Jaggi (2023), passkey retrieval, §4.2.

DeepSeek Sparse Attention (DSA)

1. A small **indexer** scores every stored entry for query t.

DeepSeek Sparse Attention (DSA)



1. A small **indexer** scores every stored entry for query t.
2. **Main attention** reads the K highest-scoring entries.

V3.2: $K = 2{,}048$, all 61 layers. Earlier: NSA (Yuan et al., 2025) selects blocks.

DSA: Lightning Indexer

64 small heads
(width 128), in FP8

$$I_{tj} = \sum_{h=1}^{H_I} w_{t,h} \, \mathrm{ReLU}(q^I_{t,h} \cdot k^I_j)$$

one small search key
per position

DSA: Lightning Indexer

64 small heads
(width 128), in FP8

$$I_{tj} = \sum_{h=1}^{H_I} w_{t,h} \, \mathrm{ReLU}(q^I_{t,h} \cdot k^I_j)$$

one small search key
per position

$$\mathcal{S}_t := \mathrm{TopK}_K(I_{t1}, \dots, I_{tt})$$

the K largest scores, like Lecture 4's router

DSA: Lightning Indexer

64 small heads
(width 128), in FP8

$$I_{tj} = \sum_{h=1}^{H_I} w_{t,h} \, \mathrm{ReLU}(q^I_{t,h} \cdot k^I_j)$$

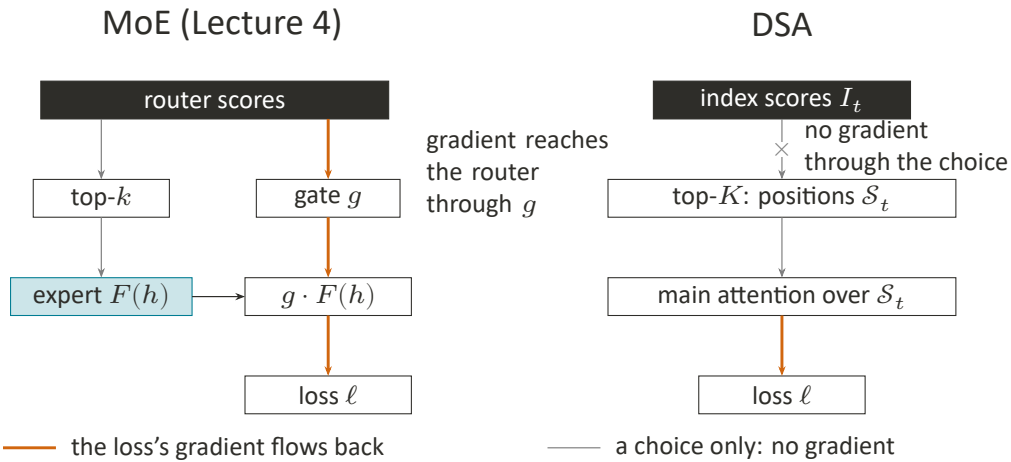
one small search key
per position

$$\mathcal{S}_t := \mathrm{TopK}_K(I_{t1}, \dots, I_{tt})$$

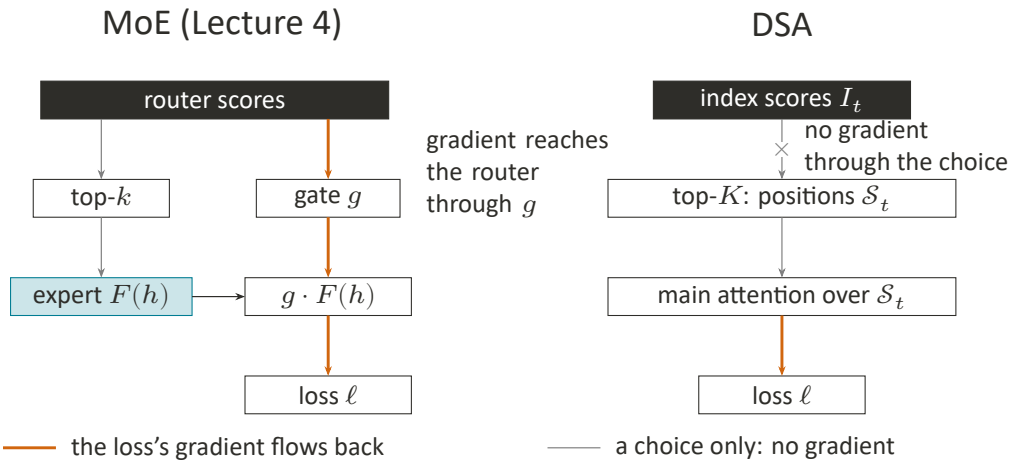
the K largest scores, like Lecture 4's router

Top-K selection is not differentiable. How do we train the indexer?

DSA: Lightning Indexer



DSA: Lightning Indexer



Idea: train the indexer to imitate dense attention.

DSA: Lightning Indexer

Pretrained dense model ✓ → **Indexer warm-up** (model frozen; train the indexer) → **Joint sparse training** (top-K on; train everything)

DSA: Lightning Indexer

Pretrained dense model ✓ → **Indexer warm-up** (model frozen; train the indexer) → Joint sparse training (top-K on; train everything)

dense attention, summed over heads, normalized

$$\mathcal{L}_I = \sum_t D_{\mathrm{KL}}\left(p_t \,\middle\|\, \mathrm{softmax}(I_{t1}, \dots, I_{tt})\right)$$

zero exactly when the two distributions agree

DSA: Lightning Indexer

Pretrained dense model ✓	→	**Indexer warm-up** model frozen; train the indexer	→	Joint sparse training top-K on; train everything

dense attention, summed over heads, normalized

$$\mathcal{L}_I = \sum_t D_{\mathrm{KL}}\left(p_t \,\middle\|\, \mathrm{softmax}(I_{t1}, \ldots, I_{tt})\right)$$

zero exactly when the two distributions agree

The indexer learns to score entries like dense attention does.

V3.2: from V3.1-Terminus, 1,000 steps on 2.1B tokens at 128K context.

DSA: Lightning Indexer

Pretrained dense model ✓ → Indexer warm-up ✓ (model frozen; train the indexer) → **Joint sparse training** (top-K on; train everything)

Turn on top-K ($K = 2{,}048$) and keep training the whole model.

DSA: Lightning Indexer

Pretrained dense model ✓ → Indexer warm-up ✓ (model frozen; train the indexer) → **Joint sparse training** (top-K on; train everything)

Turn on top-K ($K = 2{,}048$) and keep training the whole model.

The indexer keeps imitating dense attention, on the selected entries.

V3.2: 15,000 steps on 944B tokens at 128K context.

Models Use Learned Selection

Windows

Mistral 7B (2023)
$W = 4{,}096$;
all 32 layers

Gemma 3 27B (2025)
$W = 1{,}024$;
52 of 62 layers

gpt-oss-120b (2025)
$W = 128$;
18 of 36 layers

MiMo-V2.5-Pro (2026)
$W = 128$;
60 of 70 layers

Learned selection

DeepSeek-V3.2 (2025)
top 2,048 tokens

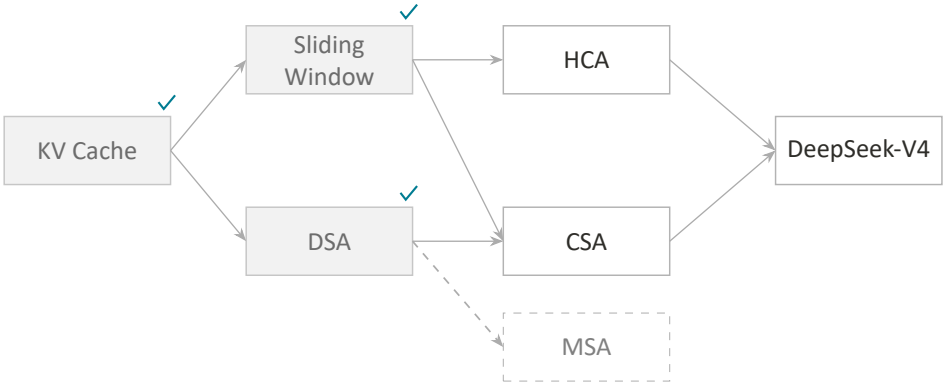
GLM-5 (2026)
top 2,048 tokens

GLM-5.1 (2026)
top 2,048 tokens

MiniMax-M3 (2026)
16 blocks of 128 tokens

Sources: model reports and configurations.

Efficient Long-Context Attention

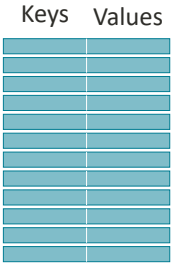


DSA still stores every entry. Can we store fewer?

Can We Merge Cache Entries?

So far, we have reduced cache storage or reads by selecting entries. Can we merge entries so every token can contribute?

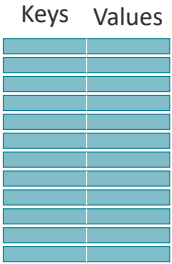
Full attention



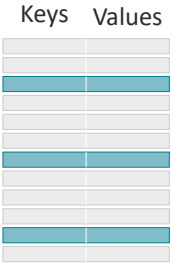
Can We Merge Cache Entries?

So far, we have reduced cache storage or reads by selecting entries.
Can we merge entries so every token can contribute?

Full attention



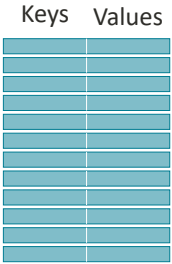
Select entries
(DSA, sliding window)



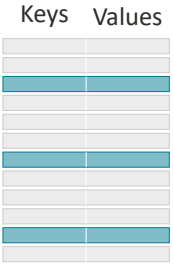
Can We Merge Cache Entries?

So far, we have reduced cache storage or reads by selecting entries.
Can we merge entries so every token can contribute?

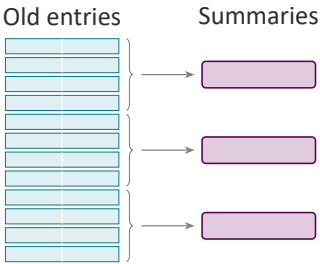
Full attention



Select entries
(DSA, sliding window)



Next: merge entries

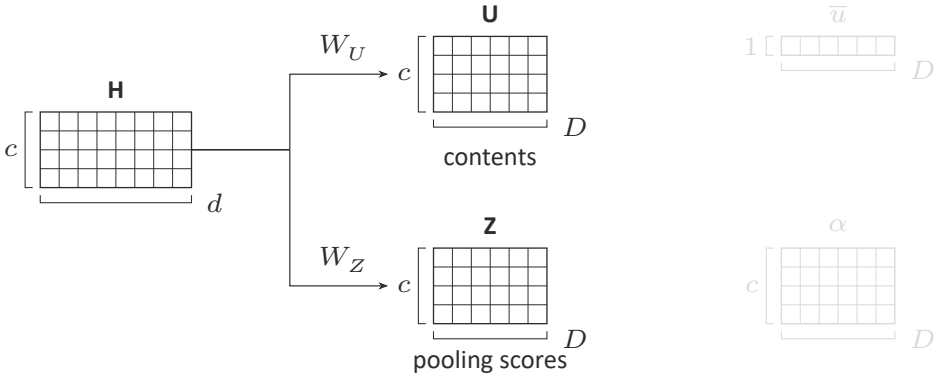


Learned Summaries: What Goes In

Take one block of c consecutive positions in one layer (here $c = 4$).

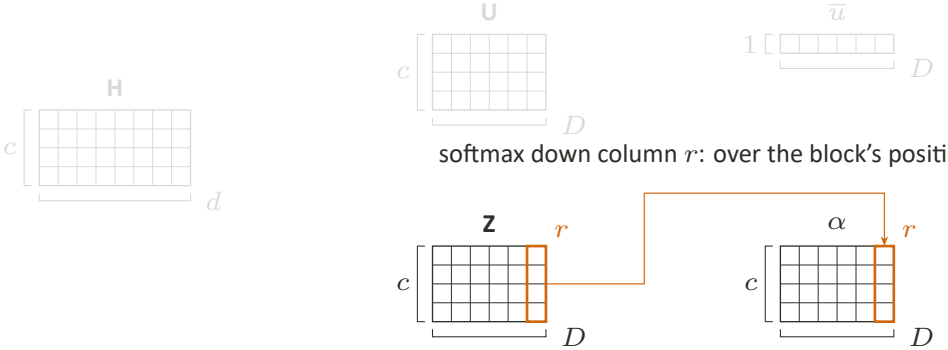
$$U = HW_U\,, \qquad Z = HW_Z + B$$

← learned bias for each position in the block



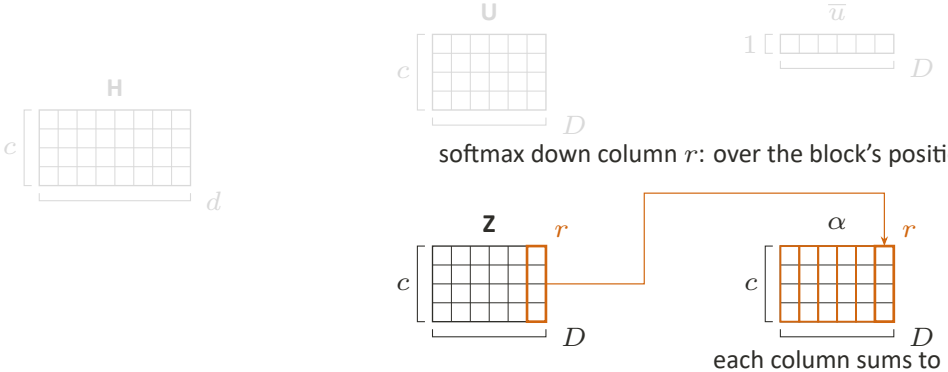
Learned Summaries: Pooling Weights

$$\alpha_{jr} = \frac{e^{Z_{jr}}}{\sum_{i \in \mathcal{B}} e^{Z_{ir}}}$$



Learned Summaries: Pooling Weights

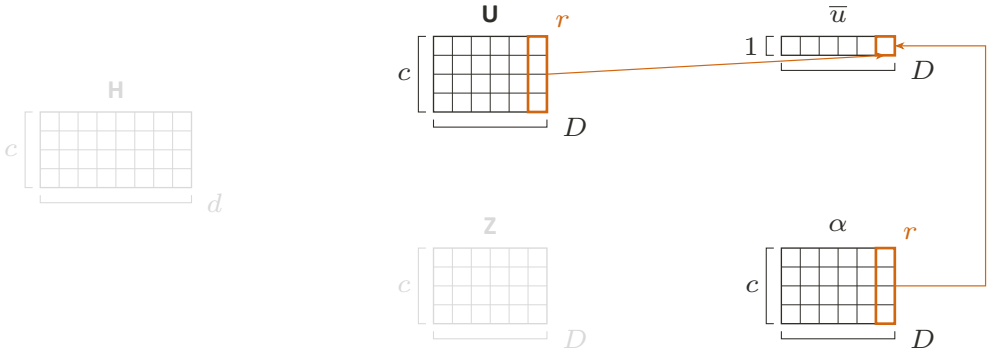
$$\alpha_{jr} = \frac{e^{Z_{jr}}}{\sum_{i \in \mathcal{B}} e^{Z_{ir}}}$$



Each coordinate has its own weights across the block.

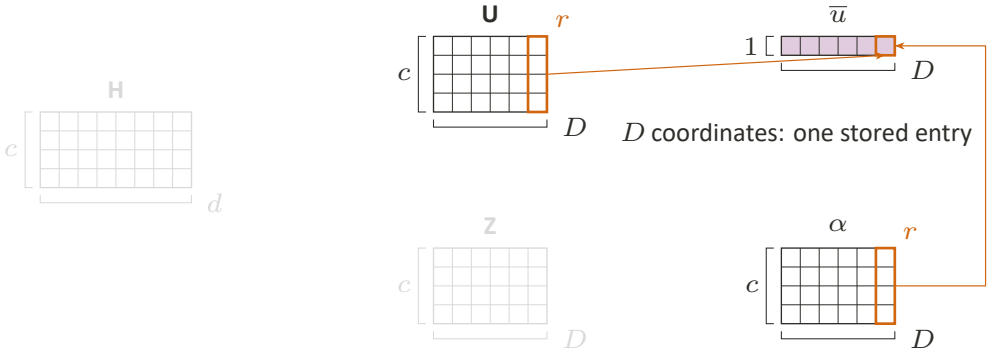
Learned Summaries: Weighted Sum

$$\overline{u}_r = \sum_{j \in \mathcal{B}} \alpha_{jr} U_{jr}$$



Learned Summaries: Weighted Sum

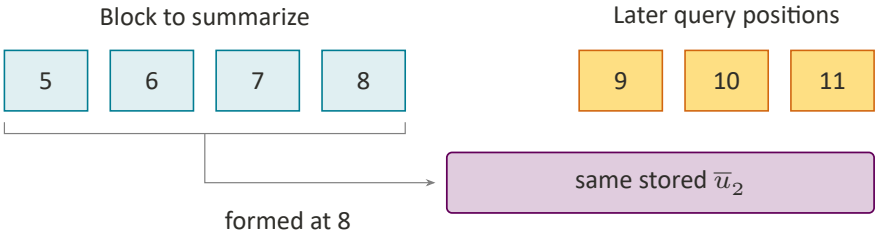
$$\overline{u}_r = \sum_{j \in \mathcal{B}} \alpha_{jr} U_{jr}$$



Several positions become one stored vector.

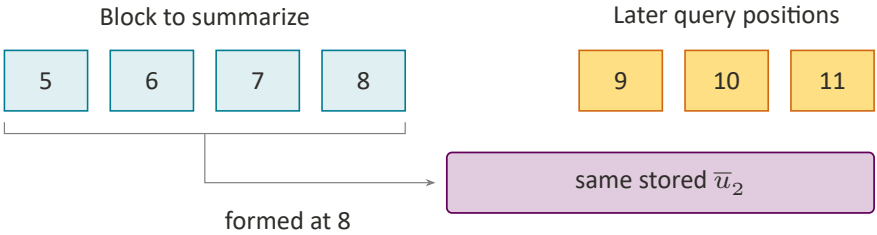
Learned Summaries: Build and Read

A summary is built once, when its block is complete, then stored.



Learned Summaries: Build and Read

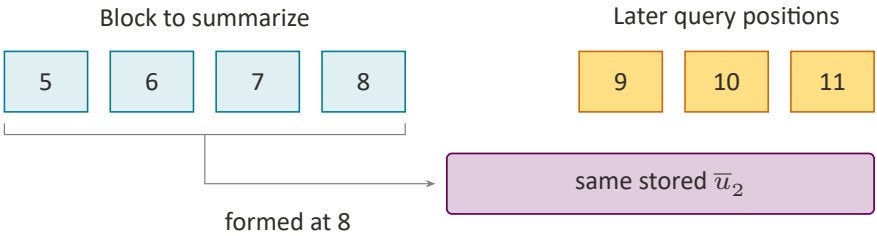
A summary is built once, when its block is complete, then stored.



A query reads a summary only after its block ends: no future tokens.

Learned Summaries: Build and Read

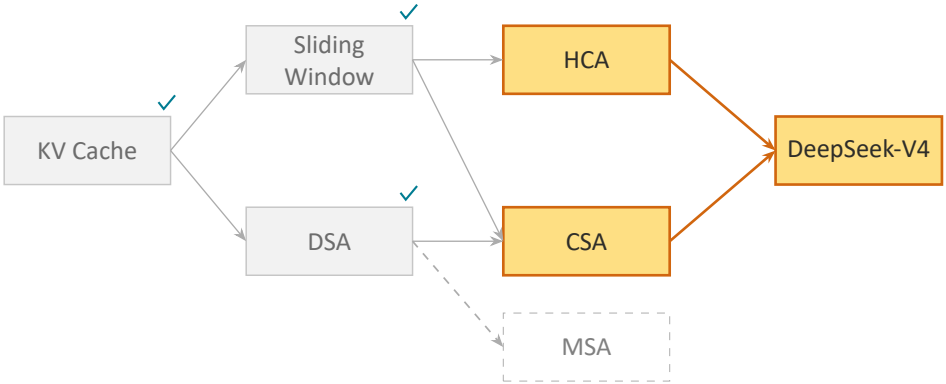
A summary is built once, when its block is complete, then stored.



A query reads a summary only after its block ends: no future tokens.

Pooling is differentiable, so the prediction loss trains W_U, W_Z, and B.

Efficient Long-Context Attention

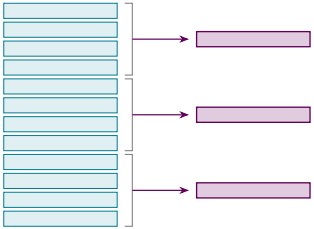


DeepSeek-V4: CSA + HCA

DeepSeek-AI (2026)

DeepSeek-V4 uses learned summaries in two settings.

Compressed Sparse Attention (CSA)



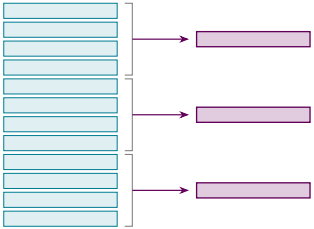
Merge every 4 positions: many fine summaries.

DeepSeek-V4: CSA + HCA

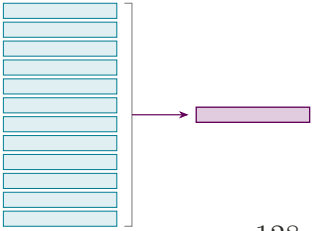
DeepSeek-AI (2026)

DeepSeek-V4 uses learned summaries in two settings.

Compressed Sparse Attention (CSA)



Heavily Compressed Attention (HCA)



$c = 128$

(fewer, coarser summaries)

Merge every 4 positions: many fine summaries. Merge every 128 positions: few coarse summaries.

The pooling we built matches HCA's; CSA pools overlapping blocks.

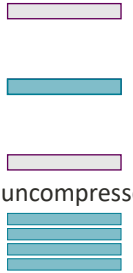
DeepSeek-V4: Token Selection

DeepSeek-AI (2026)

CSA ($c = 4$)

Compressed Sparse Attention

Select the top K summaries, as in DSA.



recent tokens, uncompressed: $W = 128$

plus recent uncompressed tokens

DeepSeek-V4: Token Selection

DeepSeek-AI (2026)

CSA ($c = 4$)

Compressed Sparse Attention

Select the top K summaries, as in DSA.

recent tokens, uncompressed: $W = 128$

plus recent uncompressed tokens

HCA ($c = 128$)

Heavily Compressed Attention

Read every ready summary.

recent tokens, uncompressed: $W = 128$

plus recent uncompressed tokens

DeepSeek-V4: Token Selection

DeepSeek-AI (2026)

CSA ($c = 4$)

Compressed Sparse Attention

Select the top K summaries, as in DSA.

recent tokens, uncompressed: $W = 128$

plus recent uncompressed tokens

HCA ($c = 128$)

Heavily Compressed Attention

Read every ready summary.

recent tokens, uncompressed: $W = 128$

plus recent uncompressed tokens

CSA: compress the history, then select among the summaries.

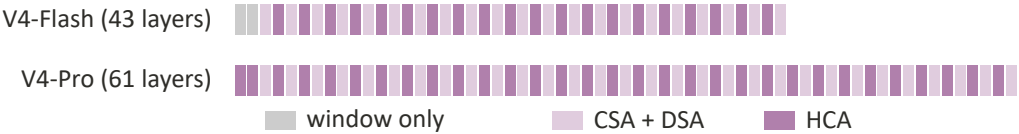
CSA's indexer scores compressed entries; its design and training may return in Lecture 7.

DeepSeek-V4: Putting It Together

DeepSeek-AI (2026)

Most layers interleave CSA and HCA.

Every CSA and HCA layer also attends to recent uncompressed tokens.

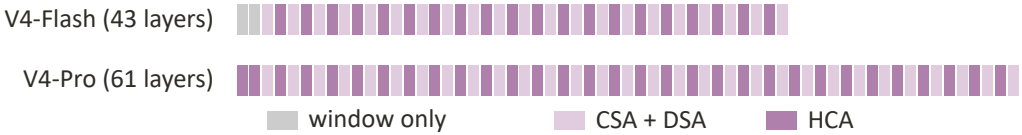


DeepSeek-V4: Putting It Together

DeepSeek-AI (2026)

Most layers interleave CSA and HCA.

Every CSA and HCA layer also attends to recent uncompressed tokens.



	V4-Flash	V4-Pro
CSA: one summary per	4 positions	4 positions
HCA: one summary per	128 positions	128 positions
Sliding window	128 tokens	128 tokens
CSA top K	512	1,024
Parameters (total / active)	284B / 13B	1.6T / 49B

Other Variants: Compressed Indexing

Qwen3.8-Flash-Next

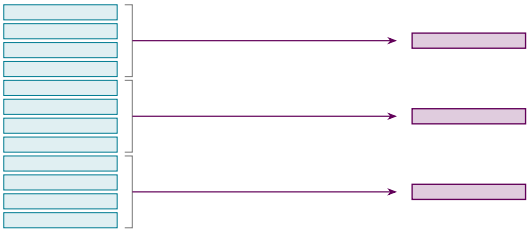
QSA: average pooling

GLM-5.3-Flash

IndexPool: learned weighted pooling

Pool indexer keys

Token keys



Blocks of 4

Compressed search keys

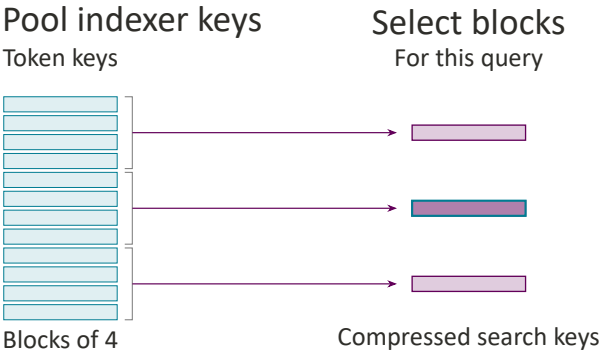
Other Variants: Compressed Indexing

Qwen3.8-Flash-Next

QSA: average pooling

GLM-5.3-Flash

IndexPool: learned weighted pooling



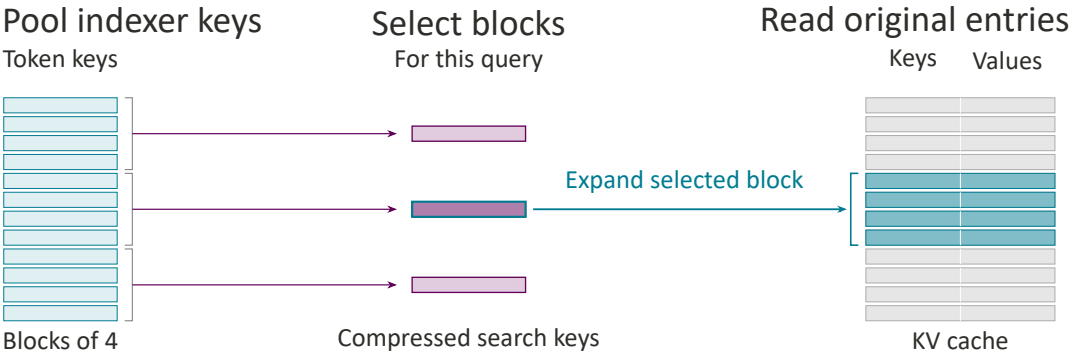
Other Variants: Compressed Indexing

Qwen3.8-Flash-Next

QSA: average pooling

GLM-5.3-Flash

IndexPool: learned weighted pooling

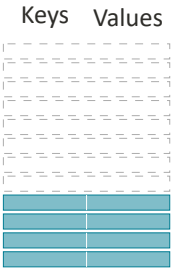


Selected blocks expand into original token entries.

The indexer searches fewer keys. Per-token KV entries remain stored.

Trade-offs of Different Methods

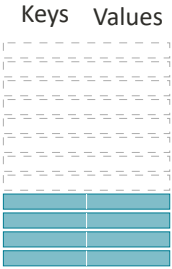
Sliding window



Old entries are removed.
Later queries cannot read them.

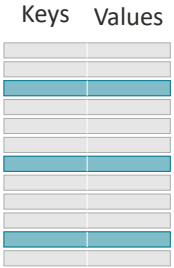
Trade-offs of Different Methods

Sliding window



Old entries are removed.
Later queries cannot read them.

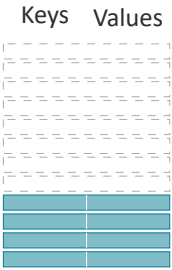
DSA



Unselected entries remain.
A later query may pick them.

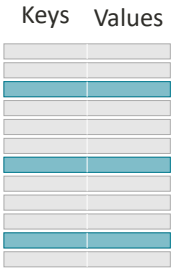
Trade-offs of Different Methods

Sliding window



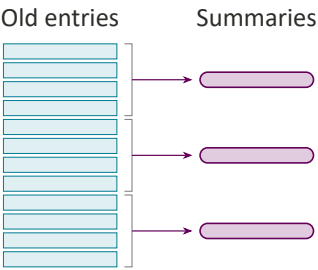
Old entries are removed.
Later queries cannot read them.

DSA



Unselected entries remain.
A later query may pick them.

Compression



Summaries replace old entries.
Lost detail cannot be recovered.

Summary

The computation needed for full attention grows with context length.

Supporting larger context windows requires more efficient attention.

Summary

The computation needed for full attention grows with context length.

Supporting larger context windows requires more efficient attention.

Several ideas have made attention more efficient:

1. **KV caching:** reuse keys and values already computed.
2. **Select fewer tokens:** sliding windows, DSA, …
3. **Compress tokens into summaries:** CSA, HCA, …

Questions?

Next: Lecture 6, Compact Representations.

The NanoGPT Speedrun

Time to reach 3.28 validation loss on FineWeb with 8 H100s

Date	Time	Main change
May 2024	45 min	GPT-2 Small baseline (llm.c)
Oct 2024	24.9 min	Muon for hidden-layer matrices
Oct 2024	12.0 min	ReLU^2, QK-norm, faster Muon
Nov 2024	5.0 min	FlexAttention: sliding windows
Dec 2025	1.9 min	FP8, Polar Express
Aug 2026	68 s	Fused kernels, more FP8
Aug 2026	39.9 s	ANVIL2, n-gram tables, and more

Source: Jordan et al., modded-nanogpt record history; PR #360 merged.

The NanoGPT Speedrun

Time to reach 3.28 validation loss on FineWeb with 8 H100s

Window warm-up
Nov 2024

Block windows
Dec 2024

Long/short windows
Jan 2025

Date	Time	Main change
May 2024	45 min	GPT-2 Small baseline (llm.c)
Oct 2024	24.9 min	Muon for hidden-layer matrices
Oct 2024	12.0 min	ReLU^2, QK-norm, faster Muon
Nov 2024	5.0 min	FlexAttention: sliding windows
Dec 2025	1.9 min	FP8, Polar Express
Aug 2026	68 s	Fused kernels, more FP8
Aug 2026	39.9 s	ANVIL2, n-gram tables, and more

Source: Jordan et al., modded-nanogpt record history; PR #360 merged.