

Sparse Attention (Prefill)

CS6013 Seminar presentation

Indian Institute of Technology
Bombay

Kislay	Roll No.: [25M2111]
Advay	Roll No.: [23B1225]
Deeptanshu	Roll No.: [22B1256]
Siddharth	Roll No.: [24B1064]



Roadmap

1

Kislay

Foundations: LLMs, diffusion transformers, sparse attention

2

Deeptanshu

MIInference 1.0: dynamic sparse attention for prefill

3

Siddharth

SparseAttention 1 & 2: training-free and trainable sparsity

4

Kislay

Sparse VideoGen2: semantic-aware permutation

5

Advay

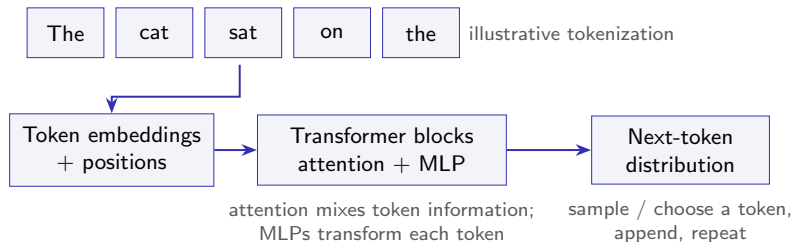
DeepSeek attention: efficient long-context models

Presentation order. Papers: MIInference 1.0; SparseAttention; SparseAttention2; Sparse VideoGen2; DeepSeek-V3.2; DeepSeek-V4 attention.



An LLM predicts the next token

Prompt: The cat sat on the



$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t \mid x_{<t})$$

Here, x_t is the token at position t and $x_{<t}$ is everything before it. A token is a vocabulary unit, often a word fragment rather than a whole word.

Original decoder-only schematic. Block foundations: Vaswani et al., Attention Is All You Need (2017), Secs. 3.1–3.5. Autoregressive inference: MInference, Sec. 2.



Attention is a learned lookup

Every token starts with a representation $x_i \in \mathbb{R}^D$. The model projects it into three roles:

Query q_i

What information
does token i need?

Key k_j

What kind of
information can token
 j offer?

Value v_j

What content will
token j contribute?

$$q_i = x_i W_Q, \quad k_j = x_j W_K, \quad v_j = x_j W_V$$

Vaswani et al. (2017), Secs. 3.2.1–3.2.3, Eq. (1). Simplified to one head; multi-head attention uses several learned projection sets in parallel.



Attention is a learned lookup

Every token starts with a representation $x_i \in \mathbb{R}^D$. The model projects it into three roles:

Query q_i

What information
does token i need?

Key k_j

What kind of
information can token
 j offer?

Value v_j

What content will
token j contribute?

$$q_i = x_i W_Q, \quad k_j = x_j W_K, \quad v_j = x_j W_V$$

For one query

Compare q_i with every allowed key k_j ; normalize the scores into weights; use those weights to mix the values v_j .

$$s_{ij} = \frac{q_i k_j^\top}{\sqrt{d}} \longrightarrow a_{ij} = \frac{e^{s_{ij}}}{\sum_{\ell \in \mathcal{A}_i} e^{s_{i\ell}}} \longrightarrow o_i = \sum_{j \in \mathcal{A}_i} a_{ij} v_j$$

d is the head vector width; $\mathcal{A}_i$ is the set of keys allowed by the mask.

Vaswani et al. (2017), Secs. 3.2.1–3.2.3, Eq. (1). Simplified to one head; multi-head attention uses several learned projection sets in parallel.



All queries are computed together as a matrix

Stack N token projections into $Q, K, V \in \mathbb{R}^{N \times d}$.

$$\text{Attention}(Q, K, V) = \underbrace{\text{softmax}_{\text{row}} \left(\frac{QK^{\top}}{\sqrt{d}} + M \right)}_{A \in \mathbb{R}^{N \times N}} V$$

Q $N \times d$ one query per token

K, V $N \times d$ one key and value per token

$QK^{\top}$ $N \times N$ one score for every query–key pair



All queries are computed together as a matrix

Stack N token projections into $Q, K, V \in \mathbb{R}^{N \times d}$.

$$\text{Attention}(Q, K, V) = \underbrace{\text{softmax}_{\text{row}} \left(\frac{QK^{\top}}{\sqrt{d}} + M \right)}_{A \in \mathbb{R}^{N \times N}} V$$

Q $N \times d$ one query per token

K, V $N \times d$ one key and value per token

$QK^{\top}$ $N \times N$ one score for every query-key pair

The mask M defines allowed interactions

$M_{ij} = 0$ for an allowed pair and $-\infty$ for a forbidden pair. A decoder-only LLM masks future tokens; a DiT usually permits all pairs.

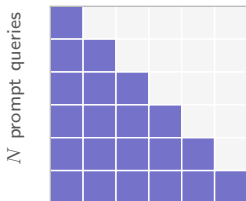
Vaswani et al. (2017), Secs. 3.2.1 and 3.2.3. DiT uses transformer blocks on latent patch tokens: Peebles & Xie (2023), Sec. 3.



LLM prefill and decode do different work

Prefill

N prompt keys



Future positions are masked.
Store per-layer keys and values.

$$O(N^2d)$$

attention for the whole prompt

One decode step

cached keys + new key



Only the newest token is processed.
Append its key and value to the cache.

$$O(Nd)$$

attention for one new query

Original schematic; costs per head/layer, excluding projections/MLPs. MInference (2024), Sec. 2; Cho et al., KV-Runahead (2024). Causal masking changes the constant, not quadratic scaling.



Diffusion learns to reverse a noise process

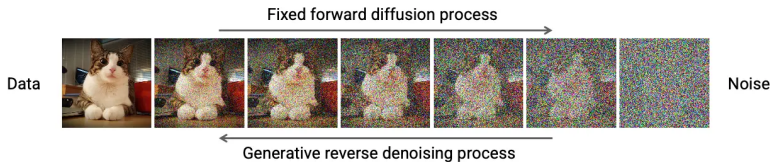
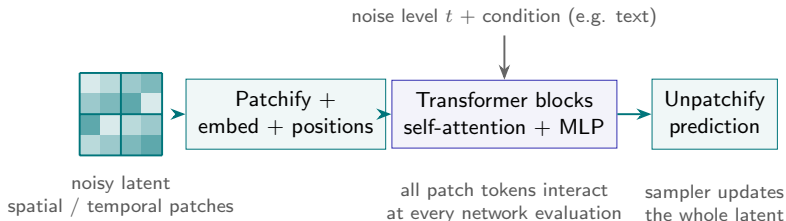


Image via Nathan Bailey, *A look at Diffusion Models*, Medium (2024).



A DiT processes the noisy latent as patch tokens



$$N = \underbrace{T'}_{\text{time}} \times \underbrace{H'}_{\text{rows}} \times \underbrace{W'}_{\text{columns}}$$

- Patchify the latent, add position and conditioning information.
- Let transformer blocks update all patch tokens at noise level t .
- Unpatchify the prediction and let the sampler take the next step.

Original schematic, extended to video. Peebles & Xie, DiT (2023), Sec. 3; Sparse VideoGen2, Sec. 5.1. T' , H' , W' count patches after latent compression and patchification, not raw video dimensions.

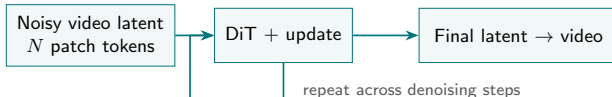


The same interaction appears in two schedules

LLM



Video diffusion transformer



LLM: full prompt attention happens during prefill; decode adds one query at a time.

Video DiT: full patch-token attention is revisited across denoising evaluations.

The common question

Can we compute fewer token pairs while preserving the interactions that matter?

Original schematic. Sources: MInference (2024), Sec. 2; DiT (2023), Sec. 3; Sparse VideoGen2 (2025), Sec. 3.1.



Double the tokens, quadruple the pairwise work

$$\underbrace{QK^T}_{N \times N} \quad \text{and} \quad \underbrace{AV}_{N \times d} \quad \Rightarrow \quad O(N^2 d)$$

Tokens N	Dense pairs N^2	One explicit FP16 matrix
32,768	1.07 billion	2 GiB
65,536	4.29 billion	8 GiB

Arithmetic illustration: $2N^2$ bytes for one FP16 matrix, one head, batch size one. Dense cost: Vaswani et al., Table 1. Repeated video attention: Sparse VideoGen2, Sec. 3.1. FlashAttention avoids this explicit matrix allocation.



Double the tokens, quadruple the pairwise work

$$\underbrace{QK^T}_{N \times N} \quad \text{and} \quad \underbrace{AV}_{N \times d} \quad \Rightarrow \quad O(N^2 d)$$

Tokens N	Dense pairs N^2	One explicit FP16 matrix
32,768	1.07 billion	2 GiB
65,536	4.29 billion	8 GiB

Repetition compounds the cost

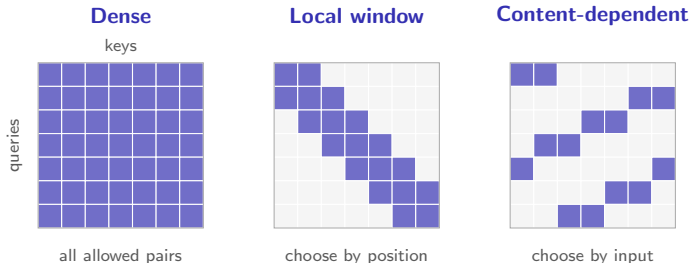
For L layers, LLM prefill attention is $O(LN^2d)$ per head. With S DiT evaluations, video generation is roughly $O(SLN^2d)$.

Projections, MLPs, conditioning and decoding add further work.

Arithmetic illustration: $2N^2$ bytes for one FP16 matrix, one head, batch size one. Dense cost: Vaswani et al., Table 1. Repeated video attention: Sparse VideoGen2, Sec. 3.1. FlashAttention avoids this explicit matrix allocation.



Sparse attention skips selected query–key pairs



Position-based support

Decide from token positions, such as a local window.

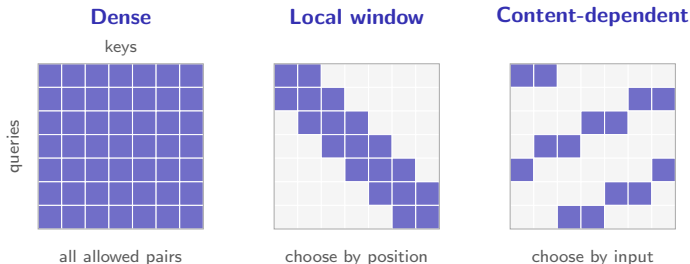
Content-dependent support

Decide from the current input's activations.

Original illustrative supports (blue = computed pair). Restricted attention foundation: Vaswani et al., Sec. 4. Dynamic sparse prefill: MInference, Sec. 3. Existing causal masks still apply.



Sparse attention skips selected query–key pairs



Position-based support

Decide from token positions, such as a local window.

Content-dependent support

Decide from the current input's activations.

Shared vocabulary

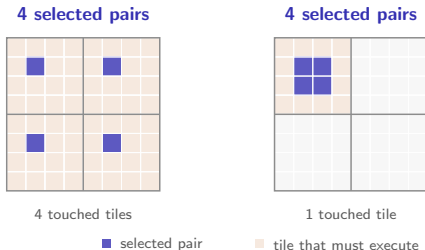
Density = fraction of otherwise allowed pairs that are computed.

Sparsity = $1 - \text{density}$.

Original illustrative supports (blue = computed pair). Restricted attention foundation: Vaswani et al., Sec. 4. Dynamic sparse prefill: MInference, Sec. 3. Existing causal masks still apply.



Fewer pairs must translate into faster kernels



Three questions for every method

1. How are important pairs found cheaply?
2. How much quality is preserved?
3. Can the GPU skip enough work?

Equal pair counts can touch different numbers of GPU tiles.

Report both **attention** speedup and **end-to-end** speedup.

Next: MInference 1.0

Deeptanshu: FlashAttention kernels, then per-head dynamic sparse patterns for long-context prefill.

Original toy tile illustration. Hardware motivation: FlashAttention, Secs. 2, 3.3; MInference 1.0, Sec. 3; Sparse VideoGen2, Secs. 3–4. Timing decomposition is schematic.



FlashAttention



Standard attention writes the score matrix twice

$Q, K, V \in \mathbb{R}^{N \times d}$: N tokens, head dimension d . HBM: the GPU's large off-chip memory, far slower than on-chip SRAM.

Operation	HBM traffic
1. Load Q, K ; compute $S = QK^T$; write S	read $2Nd$, write N^2
2. Read S ; compute $P = \text{softmax}(S)$; write P	read N^2 , write N^2
3. Load P, V ; compute $O = PV$; write O	read $N^2 + Nd$, write Nd

Dao et al., FlashAttention (2022). Algorithm 0; Theorem 2, Sec. 3.2.



Standard attention writes the score matrix twice

$Q, K, V \in \mathbb{R}^{N \times d}$: N tokens, head dimension d . HBM: the GPU's large off-chip memory, far slower than on-chip SRAM.

Operation	HBM traffic
1. Load Q, K ; compute $S = QK^T$; write S	read $2Nd$, write N^2
2. Read S ; compute $P = \text{softmax}(S)$; write P	read N^2 , write N^2
3. Load P, V ; compute $O = PV$; write O	read $N^2 + Nd$, write Nd

$$\Theta\left(\underbrace{Nd}_{Q, K, V, O} + \underbrace{N^2}_{S, P}\right)$$

At $N = 1024$, $d = 64$: S, P move $16\times$ the data of Q, K, V, O .

Dao et al., FlashAttention (2022). Algorithm 0; Theorem 2, Sec. 3.2.



Attention is limited by memory, not arithmetic

- ▶ Standard attention writes $N \times N$ to HBM, then reads it back
- ▶ Traffic, not arithmetic, sets the runtime
- ▶ **FlashAttention**: same output, $N \times N$ never written

Dao et al., FlashAttention (2022). Algorithm 0, Sec. 3.1; Fig. 2, Sec. 4.2.



Attention is limited by memory, not arithmetic

- ▶ Standard attention writes $N \times N$ to HBM, then reads it back
- ▶ Traffic, not arithmetic, sets the runtime
- ▶ **FlashAttention**: same output, $N \times N$ never written

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM read/write	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

GPT-2 medium, $N = 1024$, $d = 64$, 16 heads, batch 64, A100, **forward + backward**.



Attention is limited by memory, not arithmetic

- ▶ Standard attention writes $N \times N$ to HBM, then reads it back
- ▶ Traffic, not arithmetic, sets the runtime
- ▶ **FlashAttention**: same output, $N \times N$ never written

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM read/write	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

GPT-2 medium, $N = 1024$, $d = 64$, 16 heads, batch 64, A100, **forward + backward**.

Backward pass *recomputes* S , P , not re-reads them: extra FLOPs, still faster.

13% more arithmetic, 9× less traffic, 5.7× faster.



Attention is limited by memory, not arithmetic

- ▶ Standard attention writes $N \times N$ to HBM, then reads it back
- ▶ Traffic, not arithmetic, sets the runtime
- ▶ **FlashAttention**: same output, $N \times N$ never written

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM read/write	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

GPT-2 medium, $N = 1024$, $d = 64$, 16 heads, batch 64, A100, **forward + backward**.

Backward pass *recomputes* S , P , not re-reads them: extra FLOPs, still faster.

13% more arithmetic, 9× less traffic, 5.7× faster.

Softmax needs the *whole row* before it can normalize.

Softmax without the whole row?

Dao et al., FlashAttention (2022). Algorithm 0, Sec. 3.1; Fig. 2, Sec. 4.2.



Attention is limited by memory, not arithmetic

- ▶ Standard attention writes $N \times N$ to HBM, then reads it back
- ▶ Traffic, not arithmetic, sets the runtime
- ▶ **FlashAttention**: same output, $N \times N$ never written

Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM read/write	40.3 GB	4.4 GB
Runtime	41.7 ms	7.3 ms

GPT-2 medium, $N = 1024$, $d = 64$, 16 heads, batch 64, A100, **forward + backward**.

Backward pass *recomputes* S , P , not re-reads them: extra FLOPs, still faster.

13% more arithmetic, 9 $\times$ less traffic, 5.7 $\times$ faster.

Softmax needs the *whole row* before it can normalize.

Softmax without the whole row? Yes: the online softmax.

Dao et al., FlashAttention (2022). Algorithm 0, Sec. 3.1; Fig. 2, Sec. 4.2.



Online softmax: one query's state after j keys

One key at a time (block size 1). The $1/\sqrt{d}$ factor is dropped throughout — it only rescales the scores and changes nothing in the derivation. L_i and O_i are the paper's ℓ_i and O_i .

$$L_i = \sum_{t=1}^j e^{q_i k_t^\top}$$

Dao et al., FlashAttention (2022). Sec. 3.1.



Online softmax: one query's state after j keys

One key at a time (block size 1). The $1/\sqrt{d}$ factor is dropped throughout — it only rescales the scores and changes nothing in the derivation. L_i and O_i are the paper's ℓ_i and O_i .

$$L_i = \sum_{t=1}^j e^{q_i k_t^\top}$$

$$O_i = \frac{1}{L_i} \left(e^{q_i k_1^\top}, \dots, e^{q_i k_j^\top} \right) V_{1:j} \quad \text{equivalently} \quad O_i = \frac{\sum_{t \leq j} e^{q_i k_t^\top} v_t}{L_i}$$

$V_{1:j} \in \mathbb{R}^{j \times d}$ stacks $v_1, \dots, v_j$.

Dao et al., FlashAttention (2022). Sec. 3.1.



Online softmax: one query's state after j keys

One key at a time (block size 1). The $1/\sqrt{d}$ factor is dropped throughout — it only rescales the scores and changes nothing in the derivation. L_i and O_i are the paper's ℓ_i and O_i .

$$L_i = \sum_{t=1}^j e^{q_i k_t^\top}$$

$$O_i = \frac{1}{L_i} \left(e^{q_i k_1^\top}, \dots, e^{q_i k_j^\top} \right) V_{1:j} \quad \text{equivalently} \quad O_i = \frac{\sum_{t \leq j} e^{q_i k_t^\top} v_t}{L_i}$$

$V_{1:j} \in \mathbb{R}^{j \times d}$ stacks $v_1, \dots, v_j$.

Both are running sums over keys that have already arrived. Nothing here needs a key we have not seen.

Dao et al., FlashAttention (2022). Sec. 3.1.



Online softmax: un-normalize, add, renormalize

The $(j + 1)$ -th key arrives with score $q_i k_{j+1}^\top$.

$$L_i^{\text{new}} = L_i + e^{q_i k_{j+1}^\top}$$

Dao et al., FlashAttention (2022). Algorithm 1; Theorem 1.



Online softmax: un-normalize, add, renormalize

The $(j + 1)$ -th key arrives with score $q_i k_{j+1}^T$.

$$L_i^{\text{new}} = L_i + e^{q_i k_{j+1}^T}$$

$$O_i^{\text{new}} = \frac{\overbrace{O_i L_i}^{\text{un-normalize}} + \overbrace{e^{q_i k_{j+1}^T} v_{j+1}}^{\text{add}}}{\underbrace{L_i^{\text{new}}}_{\text{renormalize}}}$$

Dao et al., FlashAttention (2022). Algorithm 1; Theorem 1.



Online softmax: un-normalize, add, renormalize

The $(j + 1)$ -th key arrives with score $q_i k_{j+1}^\top$.

$$L_i^{\text{new}} = L_i + e^{q_i k_{j+1}^\top}$$

$$O_i^{\text{new}} = \frac{\overbrace{O_i L_i}^{\text{un-normalize}} + \overbrace{e^{q_i k_{j+1}^\top} v_{j+1}}^{\text{add}}}{\underbrace{L_i^{\text{new}}}_{\text{renormalize}}}$$

The result is **exact**, not an approximation.

Dao et al., FlashAttention (2022). Algorithm 1; Theorem 1.



Online softmax, made safe: the running maximum

Why: e^x overflows FP16 (max 65504) once $x > 11.1$. Subtracting the max makes every exponent ≤ 0 without affecting the final result.

$$M_i = \max_{t \leq j} q_i k_t^T \text{ (paper: } m_i) \quad L_i = \sum_{t \leq j} e^{q_i k_t^T - M_i}$$
$$O_i = \frac{\sum_{t \leq j} e^{q_i k_t^T - M_i} v_t}{L_i}$$

Dao et al., FlashAttention (2022). Sec. 3.1.



Online softmax, made safe: the running maximum

Why: e^x overflows FP16 (max 65504) once $x > 11.1$. Subtracting the max makes every exponent ≤ 0 without affecting the final result.

$$M_i = \max_{t \leq j} q_i k_t^T \text{ (paper: } m_i) \quad L_i = \sum_{t \leq j} e^{q_i k_t^T - M_i}$$

$$O_i = \frac{\sum_{t \leq j} e^{q_i k_t^T - M_i} v_t}{L_i}$$

$$M_i^{\text{new}} = \max(M_i, q_i k_{j+1}^T)$$

$$L_i^{\text{new}} = L_i e^{M_i - M_i^{\text{new}}} + e^{q_i k_{j+1}^T - M_i^{\text{new}}}$$

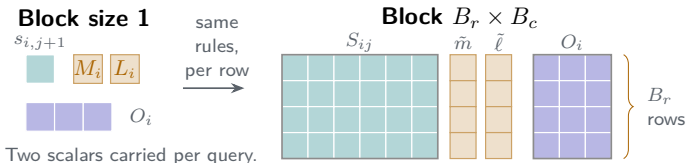
$$O_i^{\text{new}} = \frac{O_i L_i e^{M_i - M_i^{\text{new}}} + e^{q_i k_{j+1}^T - M_i^{\text{new}}} v_{j+1}}{L_i^{\text{new}}}$$

Dao et al., FlashAttention (2022). Sec. 3.1.



From one key to a $B_r \times B_c$ block

$s_{i,j+1} = q_i k_{j+1}^T$, one score. $S_{ij} = Q_i K_j^T$: query block Q_i (B_r rows) against key block K_j (B_c rows).



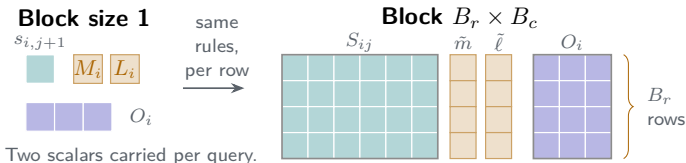
Two scalars carried per query.

M and L stop being scalars: one entry per query row in the block. B_c keys are scored at once, so the maximum and the sum are taken along each row. Illustrative tile: $B_r = 4$, $B_c = 6$; the head width is drawn as 3 columns.



From one key to a $B_r \times B_c$ block

$s_{i,j+1} = q_i k_{j+1}^T$, one score. $S_{ij} = Q_i K_j^T$: query block Q_i (B_r rows) against key block K_j (B_c rows).



M and L stop being scalars: one entry per query row in the block. B_c keys are scored at once, so the maximum and the sum are taken along each row. Illustrative tile: $B_r = 4$, $B_c = 6$; the head width is drawn as 3 columns.

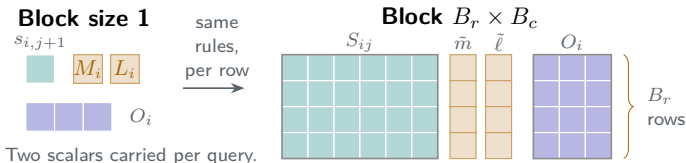
$$\tilde{m}_{ij} = \text{rowmax}(S_{ij}) \quad \tilde{P}_{ij} = \exp(S_{ij} - \tilde{m}_{ij}) \quad \tilde{l}_{ij} = \text{rowsum}(\tilde{P}_{ij})$$

Then the previous two frames' rules, applied row-wise, with $\text{diag}(\cdot)$ doing the per-row rescaling.



From one key to a $B_r \times B_c$ block

$s_{i,j+1} = q_i k_{j+1}^T$, one score. $S_{ij} = Q_i K_j^T$: query block Q_i (B_r rows) against key block K_j (B_c rows).



M and L stop being scalars: one entry per query row in the block. B_c keys are scored at once, so the maximum and the sum are taken along each row. Illustrative tile: $B_r = 4$, $B_c = 6$; the head width is drawn as 3 columns.

$$\tilde{m}_{ij} = \text{rowmax}(S_{ij}) \quad \tilde{P}_{ij} = \exp(S_{ij} - \tilde{m}_{ij}) \quad \tilde{l}_{ij} = \text{rowsum}(\tilde{P}_{ij})$$

Then the previous two frames' rules, applied row-wise, with $\text{diag}(\cdot)$ doing the per-row rescaling.

Tile size 1 was the whole idea; $B_r \times B_c$ is that idea sized to the hardware.

Dao et al., FlashAttention (2022). Algorithm 1.



Now the traffic: $\Theta(N^2 d^2 / M)$

Here M is the **on-chip SRAM capacity** in entries (about 100 KB per thread block), not the running max M_i .

Each Q block ($\Theta(M)$ entries) must see *all* of K, V :

$$\text{traffic} = \underbrace{\Theta\left(\frac{Nd}{M}\right)}_{\# Q \text{ blocks}} \times \underbrace{\Theta(Nd)}_{K, V \text{ per block}} + \underbrace{\Theta(Nd)}_{Q \text{ in, } O \text{ out, once}}$$

Dao et al., FlashAttention (2022). Theorem 2, Proposition 3, Sec. 3.2. The bound is stated there for FlashAttention-1's loop order; counted for FlashAttention-2's, as above, it comes out the same.



Now the traffic: $\Theta(N^2 d^2 / M)$

Here M is the **on-chip SRAM capacity** in entries (about 100 KB per thread block), not the running max M_i .

Each Q block ($\Theta(M)$ entries) must see *all* of K, V :

$$\begin{aligned} \text{traffic} &= \underbrace{\Theta\left(\frac{Nd}{M}\right)}_{\# Q \text{ blocks}} \times \underbrace{\Theta(Nd)}_{K, V \text{ per block}} + \underbrace{\Theta(Nd)}_{Q \text{ in, } O \text{ out, once}} \\ &= \Theta\left(\frac{N^2 d^2}{M} + Nd\right) = \Theta\left(\frac{N^2 d^2}{M}\right) \quad \text{since } M \leq Nd \end{aligned}$$

Dao et al., FlashAttention (2022). Theorem 2, Proposition 3, Sec. 3.2. The bound is stated there for FlashAttention-1's loop order; counted for FlashAttention-2's, as above, it comes out the same.



Now the traffic: $\Theta(N^2 d^2 / M)$

Here M is the **on-chip SRAM capacity** in entries (about 100 KB per thread block), not the running max M_i .

Each Q block ($\Theta(M)$ entries) must see *all* of K, V :

$$\begin{aligned} \text{traffic} &= \underbrace{\Theta\left(\frac{Nd}{M}\right)}_{\# Q \text{ blocks}} \times \underbrace{\Theta(Nd)}_{K, V \text{ per block}} + \underbrace{\Theta(Nd)}_{Q \text{ in, } O \text{ out, once}} \\ &= \Theta\left(\frac{N^2 d^2}{M} + Nd\right) = \Theta\left(\frac{N^2 d^2}{M}\right) \quad \text{since } M \leq Nd \end{aligned}$$

Against standard attention's $\Theta(Nd + N^2)$: ratio $\approx d^2/M$, and $d^2 \ll M$ for $d = 64\text{--}128$ — the $40.3 \rightarrow 4.4$ GB we saw earlier.

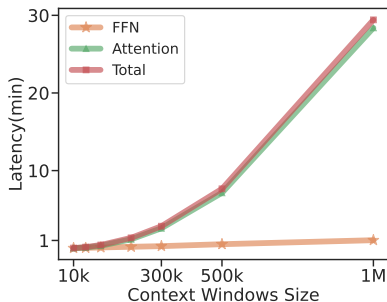
Dao et al., FlashAttention (2022). Theorem 2, Proposition 3, Sec. 3.2. The bound is stated there for FlashAttention-1's loop order; counted for FlashAttention-2's, as above, it comes out the same.



MInference 1.0



Attention is most of the prefill latency



1M tokens: **~30 min** to pre-fill on one A100.

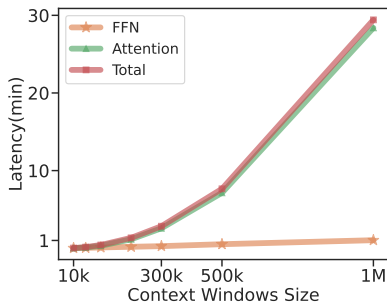
Attention is almost all of it; the FFN stays flat.

(a) Attention incurs heavy cost.

Jiang et al., MInference 1.0 (2024). Fig. 2a (cropped), Sec. 2.1; abstract.



Attention is most of the prefill latency



1M tokens: **~30 min** to pre-fill on one A100.

Attention is almost all of it; the FFN stays flat.

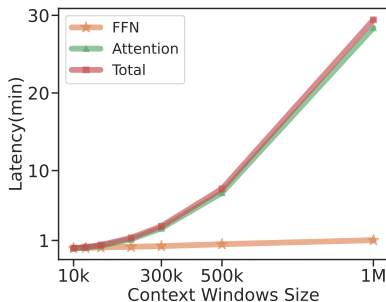
Even with FlashAttention, still $\Theta(N^2)$ work for a prompt of N tokens.

(a) Attention incurs heavy cost.

Jiang et al., MInference 1.0 (2024). Fig. 2a (cropped), Sec. 2.1; abstract.



Attention is most of the prefill latency



1M tokens: **~30 min** to pre-fill on one A100.

Attention is almost all of it; the FFN stays flat.

Even with FlashAttention, still $\Theta(N^2)$ work for a prompt of N tokens.

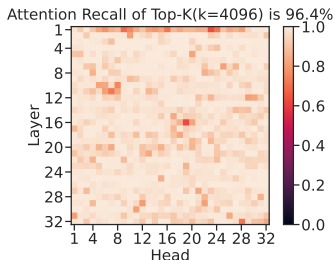
(a) Attention incurs heavy cost.

The only way out is to stop computing most of the matrix.

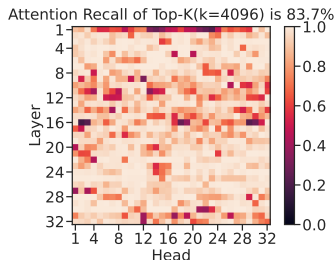
Jiang et al., MInference 1.0 (2024). Fig. 2a (cropped), Sec. 2.1; abstract.



Attention is sparse — but not in a fixed place



(b) Attention is sparse.



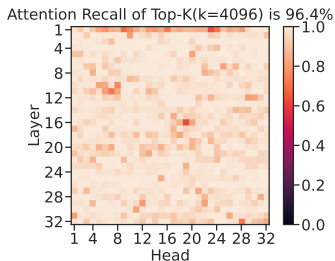
(c) Sparsity of attention is dynamic.

128k context: the **top 4k columns** hold **96.4%** of the attention mass

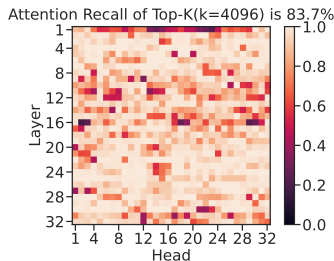
Jiang et al., MInference 1.0 (2024). Figs. 2b and 2c (cropped), Sec. 2.1.



Attention is sparse — but not in a fixed place



(b) Attention is sparse.



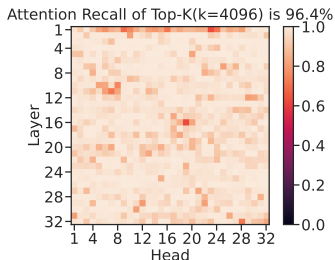
(c) Sparsity of attention is dynamic.

128k context: the **top 4k columns** hold **96.4%** of the attention mass, but only **83.7%** on a different prompt.

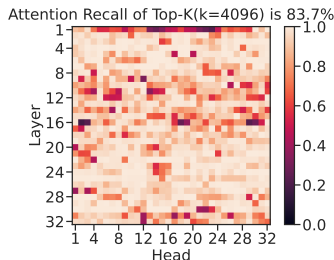
Jiang et al., MInference 1.0 (2024). Figs. 2b and 2c (cropped), Sec. 2.1.



Attention is sparse — but not in a fixed place



(b) Attention is sparse.



(c) Sparsity of attention is dynamic.

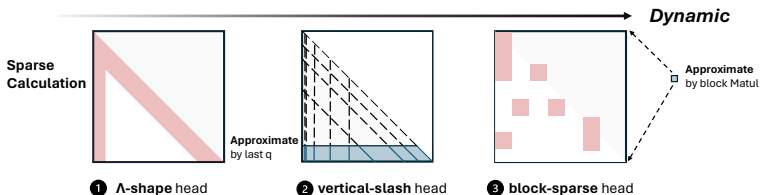
128k context: the **top 4k columns** hold **96.4%** of the attention mass, but only **83.7%** on a different prompt.

Sparsity exists but it is dynamic.

Jiang et al., MInference 1.0 (2024). Figs. 2b and 2c (cropped), Sec. 2.1.



Three shapes keep coming back



A-shape (Λ -shape in the figure) First tokens plus a local window. *Static*.

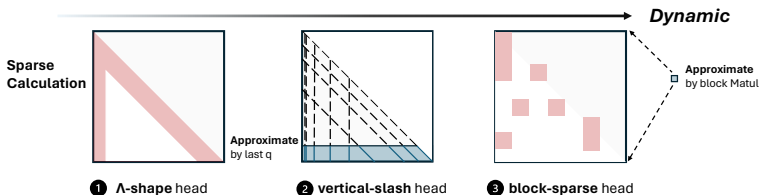
Vertical-Slash
A few whole columns, plus a few fixed diagonal offsets.

Block-Sparse
Scattered, but clustered into blocks.

Jiang et al., MInference 1.0 (2024). Fig. 4 (cropped), Sec. 2.2.



Three shapes keep coming back



A-shape (Λ -shape in the figure) First tokens plus a local window. *Static*.

Vertical-Slash
A few whole columns, plus a few fixed diagonal offsets.

Block-Sparse
Scattered, but clustered into blocks.

Dynamic does not mean arbitrary. The surviving mass keeps falling into one of **three spatial patterns**.

Jiang et al., MInference 1.0 (2024). Fig. 4 (cropped), Sec. 2.2.



Pattern per head, indices per prompt

Which pattern? stable per head, decided once, offline.
Which lines? change with the prompt, rebuilt every time.

Jiang et al., MInference 1.0 (2024). Fig. 3a, Sec. 2.2; Table 1, Sec. 2.2.



Pattern per head, indices per prompt

Which pattern? stable per head, decided once, offline.

Which lines? change with the prompt, rebuilt every time.

	A-shape	Vertical-Slash	Block-Sparse	Top- k
Spatial distribution	static	dynamic	dynamic	dynamic
	structured	structured	structured	fine-grained
Latency on GPU	low	<i>medium</i>	low	high
Time to build the index	zero	small	small	high

Top- k : every row keeps its own k largest scores.

Jiang et al., MInference 1.0 (2024). Fig. 3a, Sec. 2.2; Table 1, Sec. 2.2.



Pattern per head, indices per prompt

Which pattern? stable per head, decided once, offline.

Which lines? change with the prompt, rebuilt every time.

	A-shape	Vertical-Slash	Block-Sparse	Top-k
Spatial distribution	static	dynamic	dynamic	dynamic
	structured	structured	structured	fine-grained
Latency on GPU	low	<i>medium</i>	low	high
Time to build the index	zero	small	small	high

Top-k: every row keeps its own k largest scores.

Top-k is what we would want, and it is unusable: scattered single entries cannot fill a tile.

The three usable patterns have **sparsity a kernel can load**.

Jiang et al., MInference 1.0 (2024). Fig. 3a, Sec. 2.2; Table 1, Sec. 2.2.



What has to be found, and what finding it may cost

OFFLINE once

1. Search

one pattern and one
setting per head

Offline. Once per
model, about 15 min
on one A100.

Jiang et al., MInference 1.0 (2024). Sec. 3.2.



What has to be found, and what finding it may cost

OFFLINE once

1. Search

one pattern and one
setting per head

Offline. Once per
model, about 15 min
on one A100.



ONLINE every prompt

2. Index

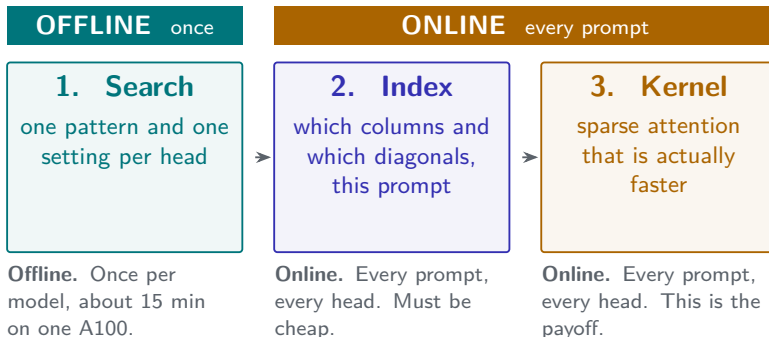
which columns and
which diagonals,
this prompt

Online. Every prompt,
every head. Must be
cheap.

Jiang et al., MInference 1.0 (2024). Sec. 3.2.



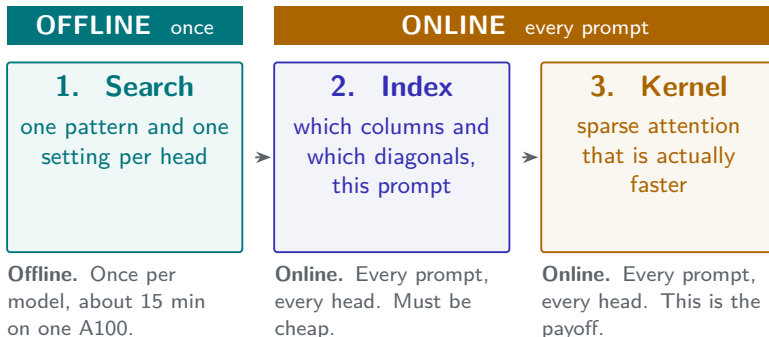
What has to be found, and what finding it may cost



Jiang et al., MInference 1.0 (2024). Sec. 3.2.



What has to be found, and what finding it may cost



Step 1 is paid once. **Only Steps 2 and 3 cost time per prompt.**

Jiang et al., MInference 1.0 (2024). Sec. 3.2.



MIInference, Step 1 of 3

1. Search · 2. Index · 3. Kernel

Step 1

Search

one pattern and one setting per head

OFFLINE



Step 1: pick one pattern per head, offline

1. Search · 2. Index · 3. Kernel

1. **Grid.** A hand-picked grid of settings per pattern: A-shape window sizes; Vertical-Slash (k_v, k_s), how many vertical and slash lines to keep; Block-Sparse k_b , how many blocks each query block keeps.



Step 1: pick one pattern per head, offline

1. Search · 2. Index · 3. Kernel

1. **Grid.** A hand-picked grid of settings per pattern: A-shape window sizes; Vertical-Slash (k_v, k_s), how many vertical and slash lines to keep; Block-Sparse k_b , how many blocks each query block keeps.
2. **Match the cost.** Adjust each setting i until its FLOPs t_i , measured *inside the GPU kernel*, reach a target t :

while $|t_i - t| > \epsilon$: change setting i .



Step 1: pick one pattern per head, offline

1. Search · 2. Index · 3. Kernel

1. **Grid.** A hand-picked grid of settings per pattern: A-shape window sizes; Vertical-Slash (k_v, k_s), how many vertical and slash lines to keep; Block-Sparse k_b , how many blocks each query block keeps.
2. **Match the cost.** Adjust each setting i until its FLOPs t_i , measured *inside the GPU kernel*, reach a target t :

while $|t_i - t| > \epsilon$: change setting i .

3. **Pick by quality.** On one 30k-token sample, compare each candidate's output y_i with dense attention's y and keep the closest:

$$y = \text{softmax}(QK^T/\sqrt{d})V, \quad p_{\text{best}} = \arg \min_i \|y_i - y\|.$$



Step 1: pick one pattern per head, offline

1. Search · 2. Index · 3. Kernel

1. **Grid.** A hand-picked grid of settings per pattern: A-shape window sizes; Vertical-Slash (k_v, k_s), how many vertical and slash lines to keep; Block-Sparse k_b , how many blocks each query block keeps.
2. **Match the cost.** Adjust each setting i until its FLOPs t_i , measured *inside the GPU kernel*, reach a target t :

while $|t_i - t| > \epsilon$: change setting i .

3. **Pick by quality.** On one 30k-token sample, compare each candidate's output y_i with dense attention's y and keep the closest:

$$y = \text{softmax}(QK^T/\sqrt{d})V, \quad p_{\text{best}} = \arg \min_i \|y_i - y\|.$$

~15 min on one A100, then frozen.

Jiang et al., MInference 1.0 (2024). Algorithm 1, Sec. 3.2; Table 7, Appendix C.2.



Minference, Step 2 of 3

1. Search · **2. Index** · 3. Kernel

Step 2

Index

which columns and diagonals, for this prompt

ONLINE



Step 2, Block-Sparse: pool, score, keep the top k_b

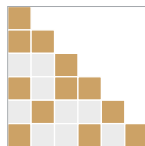
1. Search · **2. Index** · 3. Kernel

1. Pool each block of B rows 2. Score blocks, keep the top k_b



$$Q: N \times d \quad \hat{Q}: (N/B) \times d$$

$K \rightarrow \hat{K}$ the same way



$$\hat{A} = \hat{Q} \hat{K}^T: (N/B)^2 \text{ scores}$$

top k_b the kept-block list,
which the Step 3
kernel walks

Illustrative: $B = 4$, and the kept blocks are the same set the kernel slide uses later.
The real kernel has `block_size = 64` and $k_b = 100$.

Jiang et al., MInference 1.0 (2024). Algorithm 3, Sec. 3.3; Appendix D.2.



Step 2, Block-Sparse: pool, score, keep the top k_b

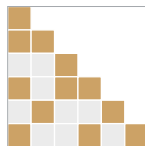
1. Search · **2. Index** · 3. Kernel

1. Pool each block of B rows 2. Score blocks, keep the top k_b



$$Q: N \times d \quad \hat{Q}: (N/B) \times d$$

$K \rightarrow \hat{K}$ the same way



$$\hat{A} = \hat{Q} \hat{K}^T: (N/B)^2 \text{ scores}$$

Illustrative: $B = 4$, and the kept blocks are the same set the kernel slide uses later.

The real kernel has `block_size = 64` and $k_b = 100$.

Pool *first*, then choose the top- k_b blocks

Jiang et al., MInference 1.0 (2024). Algorithm 3, Sec. 3.3; Appendix D.2.



Step 2, Block-Sparse: pool, score, keep the top k_b

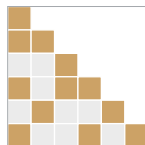
1. Search · **2. Index** · 3. Kernel

1. Pool each block of B rows 2. Score blocks, keep the top k_b



$$Q: N \times d \quad \hat{Q}: (N/B) \times d$$

$K \rightarrow \hat{K}$ the same way



$$\hat{A} = \hat{Q} \hat{K}^T: (N/B)^2 \text{ scores}$$

top k_b the kept-block list,
which the Step 3
kernel walks

Illustrative: $B = 4$, and the kept blocks are the same set the kernel slide uses later.
The real kernel has `block_size = 64` and $k_b = 100$.

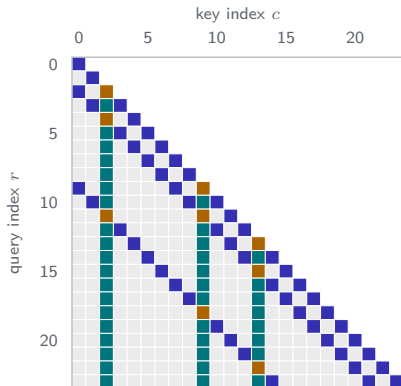
Pool *first*, then choose the top- k_b blocks — and not the reverse, as scoring all of QK^T first gives the same block scores at $B^2 \times$ **the cost**.

Jiang et al., MInference 1.0 (2024). Algorithm 3, Sec. 3.3; Appendix D.2.



Step 2, Vertical-Slash: the example we will carry

1. Search · **2. Index** · 3. Kernel



Colours

- vertical line
columns 2, 9, 13
- slash line
offsets $d = r - c$: 0, 2, 9
- on both
- causal, but skipped

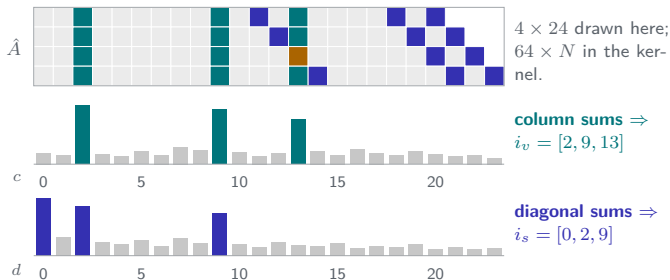
Jiang et al., MInference 1.0 (2024). Sec. 2.2. Toy example.



Step 2, Vertical-Slash: the lines, from 64 rows not N

1. Search · **2. Index** · 3. Kernel

Score the last `last_q` rows only



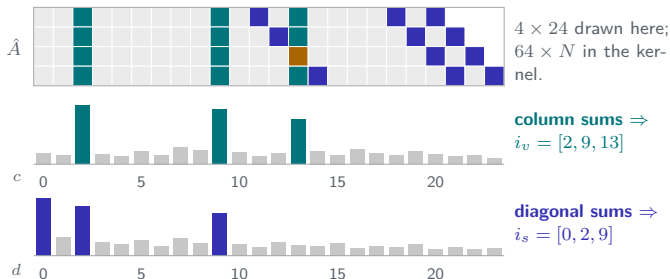
Bar heights are illustrative: only which three bars win is meant to be read off.



Step 2, Vertical-Slash: the lines, from 64 rows not N

1. Search · **2. Index** · 3. Kernel

Score the last `last_q` rows only



Bar heights are illustrative: only which three bars win is meant to be read off.

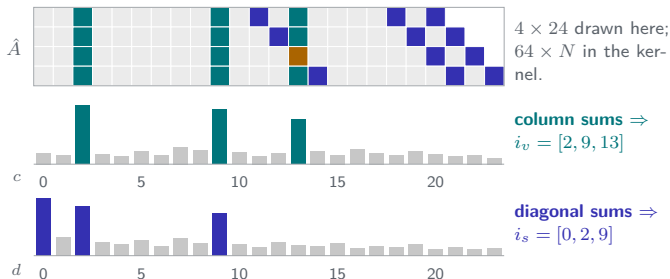
$\hat{A} = \text{softmax}(Q[-64:] K^T / \sqrt{d} + m_{\text{causal}})$, with m_{causal} the causal mask.
Cost $O(\text{last_q} \cdot N \cdot d)$ for `last_q` = 64: **linear in N** .



Step 2, Vertical-Slash: the lines, from 64 rows not N

1. Search · **2. Index** · 3. Kernel

Score the last `last_q` rows only



Bar heights are illustrative: only which three bars win is meant to be read off.

$\hat{A} = \text{softmax}(Q[-64:]K^T/\sqrt{d} + m_{\text{causal}})$, with m_{causal} the causal mask.
Cost $O(\text{last_q} \cdot N \cdot d)$ for `last_q` = 64: **linear in N** .

Measured overhead **5–15%** of attention time, and the index stays under 160 MB for LLaMA-3-8B at 1M context.

Jiang et al., MInference 1.0 (2024). Algorithm 2, Sec. 3.3; Appendix D.2.



Step 2, A-shape: no index to build

1. Search · **2. Index** · 3. Kernel

Keep the *first few tokens*, plus a *local window* around each query.

Jiang et al., MInference 1.0 (2024). Sec. 3.2; Table 1, Sec. 2.2; Appendix D.2.



Step 2, A-shape: no index to build

1. Search · **2. Index** · 3. Kernel

Keep the *first few tokens*, plus a *local window* around each query.

That rule depends only on *where* a token sits, never on what the prompt says, so the kept cells are the same for every prompt.

Jiang et al., MInference 1.0 (2024). Sec. 3.2; Table 1, Sec. 2.2; Appendix D.2.



Step 2, A-shape: no index to build

1. Search · **2. Index** · 3. Kernel

Keep the *first few tokens*, plus a *local window* around each query.

That rule depends only on *where* a token sits, never on what the prompt says, so the kept cells are the same for every prompt.

Nothing to estimate: **no index cost**, fixed once offline.

Jiang et al., MInference 1.0 (2024). Sec. 3.2; Table 1, Sec. 2.2; Appendix D.2.



MIInference, Step 3 of 3

1. Search · 2. Index · **3. Kernel**

Step 3

Kernel

sparse attention that is actually faster

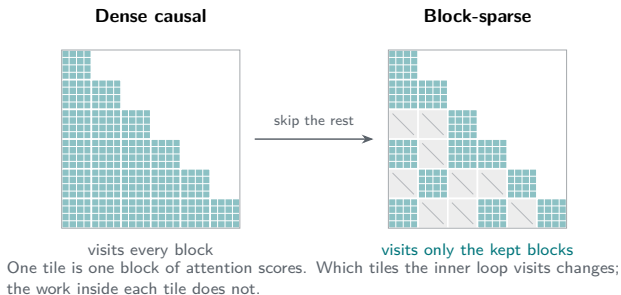
ONLINE



Step 3, Block-Sparse: a one-line change to FlashAttention

1. Search · 2. Index · **3. Kernel**

Steps 1 and 2 chose *what* to skip. Nothing is faster yet.



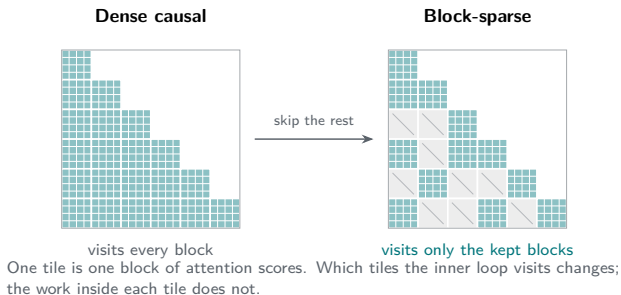
Jiang et al., MInference 1.0 (2024). Appendix C.4.1; FlashAttention-2, Algorithm 1.



Step 3, Block-Sparse: a one-line change to FlashAttention

1. Search · 2. Index · **3. Kernel**

Steps 1 and 2 chose *what* to skip. Nothing is faster yet.



The inner loop visits only the **blocks Step 2 kept**. The online softmax mechanism is **unchanged**.

Jiang et al., MInference 1.0 (2024). Appendix C.4.1; FlashAttention-2, Algorithm 1.



Step 3, Vertical-Slash: the example, again

1. Search · 2. Index · **3. Kernel**

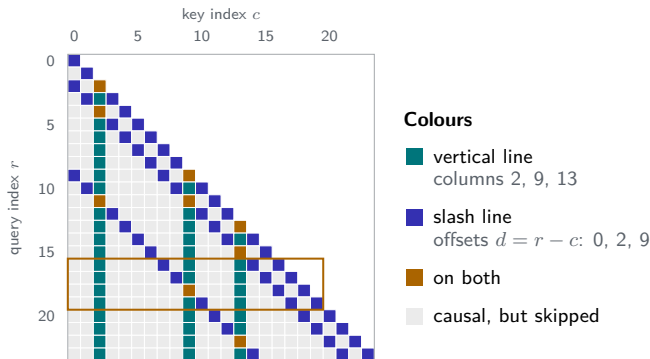
Same kernel idea as Block-Sparse — walk a **list** of tiles, never the whole grid. What is hard here is only **which tiles**: this mask does not line up with them.



Step 3, Vertical-Slash: the example, again

1. Search · 2. Index · **3. Kernel**

Same kernel idea as Block-Sparse — walk a **list** of tiles, never the whole grid. What is hard here is only **which tiles**: this mask does not line up with them.



The amber outline is **row block 5**, rows 16–19. Every later picture zooms into it.

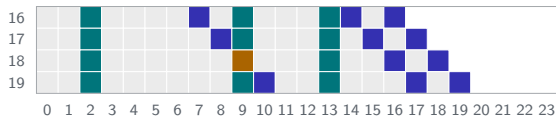
Jiang et al., MInference 1.0 (2024). Sec. 2.2. Toy example.



Step 3, Vertical-Slash: split the row block in two

1. Search · 2. Index · **3. Kernel**

Row block 5 (rows 16–19): all 24 key columns



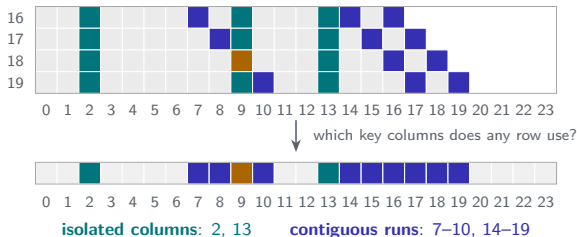
Jiang et al., MInference 1.0 (2024). Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: split the row block in two

1. Search · 2. Index · **3. Kernel**

Row block 5 (rows 16–19): all 24 key columns



Column 9 is amber, **both**: the run 7–10 already covers it.

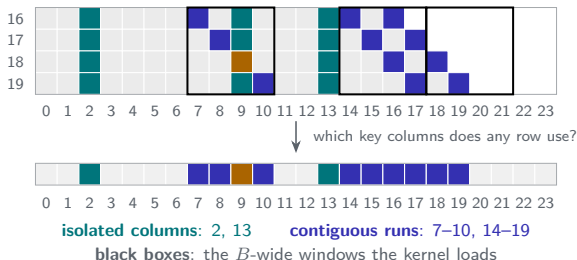
Jiang et al., MInference 1.0 (2024). Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: split the row block in two

1. Search · 2. Index · **3. Kernel**

Row block 5 (rows 16–19): all 24 key columns



Column 9 is amber, **both**: the run 7–10 already covers it.

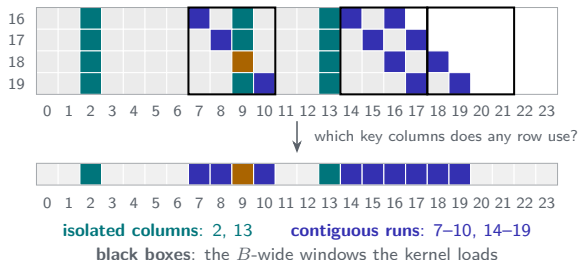
Jiang et al., MInference 1.0 (2024). Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: split the row block in two

1. Search · 2. Index · **3. Kernel**

Row block 5 (rows 16–19): all 24 key columns



Column 9 is amber, **both**: the run 7–10 already covers it.

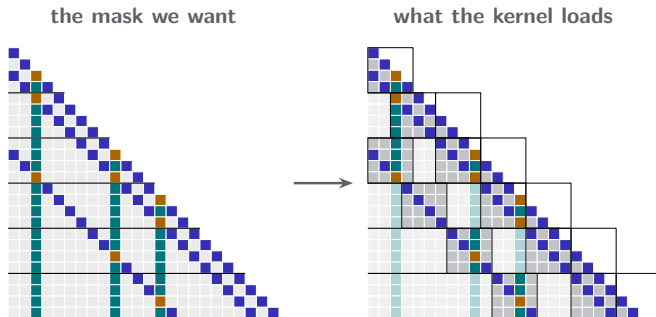
Every row block splits the same way. Now all six:

Jiang et al., MInference 1.0 (2024). Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: what the kernel actually loads

1. Search · 2. Index · **3. Kernel**



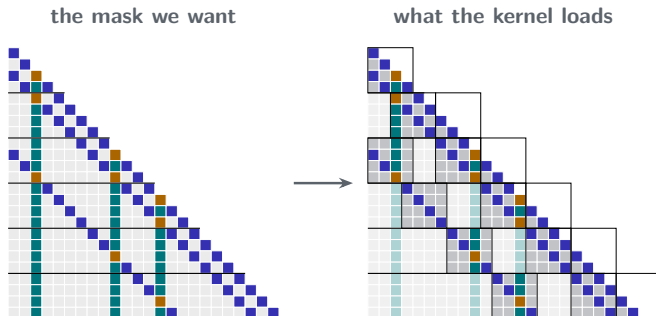
Black box: a loaded window. Grey: loaded, then masked. Pale green: a vertical no window reaches.

Jiang et al., MInference 1.0 (2024). Algorithm 5, Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: what the kernel actually loads

1. Search · 2. Index · **3. Kernel**



Black box: a loaded window. Grey: loaded, then masked. Pale green: a vertical no window reaches.

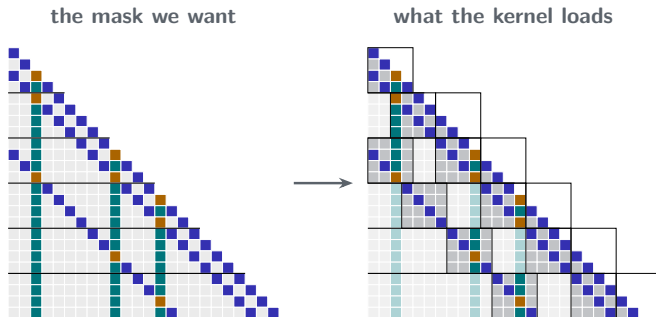
Black boxes: **ordinary block-sparse tiles**. Pale green: left over.

Jiang et al., MInference 1.0 (2024). Algorithm 5, Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: what the kernel actually loads

1. Search · 2. Index · **3. Kernel**



Black box: a loaded window. Grey: loaded, then masked. Pale green: a vertical no window reaches.

Black boxes: **ordinary block-sparse tiles**. Pale green: left over.
Not exact, a superset: **all high-mass attention covered**.

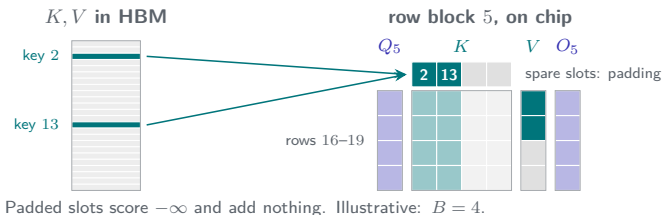
Jiang et al., MInference 1.0 (2024). Algorithm 5, Appendix C.4.2. Toy example.



Step 3, Vertical-Slash: the gather

1. Search · 2. Index · **3. Kernel**

Row block 5: columns 2 and 13 left. **PIT** (permutation-invariant transformation) packs them into one tile.



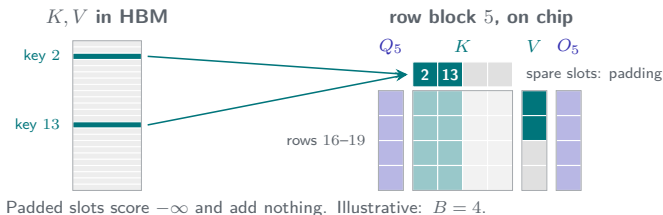
Jiang et al., MInference 1.0 (2024). Algorithm 5, Appendix C.4.2; PIT (Zheng et al., 2023). Toy example.



Step 3, Vertical-Slash: the gather

1. Search · 2. Index · **3. Kernel**

Row block 5: columns 2 and 13 left. **PIT** (permutation-invariant transformation) packs them into one tile.



Position-blind inner loop: **same kernel, no new code.**

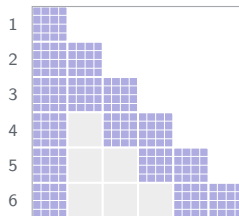
Jiang et al., MInference 1.0 (2024). Algorithm 5, Appendix C.4.2; PIT (Zheng et al., 2023). Toy example.



Step 3, A-shape: either kernel could run it

1. Search · 2. Index · **3. Kernel**

The block list, for every row block i



First g tokens plus a local band of width w , computed on the fly: no index buffer.
Illustrative: $g = w = 4$ (paper: 1024 and 4096).

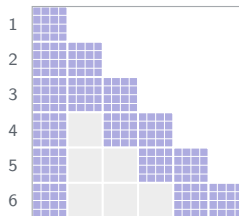
Jiang et al., MInference 1.0 (2024). Sec. 3.3; Appendix C.4.1–C.4.2.



Step 3, A-shape: either kernel could run it

1. Search · 2. Index · **3. Kernel**

The block list, for every row block i



First g tokens plus a local band of width w , computed on the fly: no index buffer.
Illustrative: $g = w = 4$ (paper: 1024 and 4096).

Either kernel style runs this mask.

Jiang et al., MInference 1.0 (2024). Sec. 3.3; Appendix C.4.1–C.4.2.



The quality survives — and the dynamism earns its keep

InfiniteBench: 10 real tasks at $\sim 214k$ tokens (QA, summaries, code, retrieval).

RULER: 13 synthetic tasks, rebuilt at *each* length.

InfLLM: keeps a local window, and retrieves distant context blocks from a memory.

	InfiniteBench avg.	RULER avg., % at length		
		32K	64K	128K
LLaMA-3-8B-262K				
Full attention	38.2	80.8	77.4	72.2
StreamingLLM (= A-shape)	23.8	17.2	14.2	9.4
InfLLM	34.8	55.6	43.0	39.5
MIInference	38.8	85.0	82.3	77.6

Jiang et al., MIInference 1.0 (2024). Tables 2 and 3, Sec. 4.2.



The quality survives — and the dynamism earns its keep

InfiniteBench: 10 real tasks at $\sim 214k$ tokens (QA, summaries, code, retrieval).

RULER: 13 synthetic tasks, rebuilt at *each* length.

InfLLM: keeps a local window, and retrieves distant context blocks from a memory.

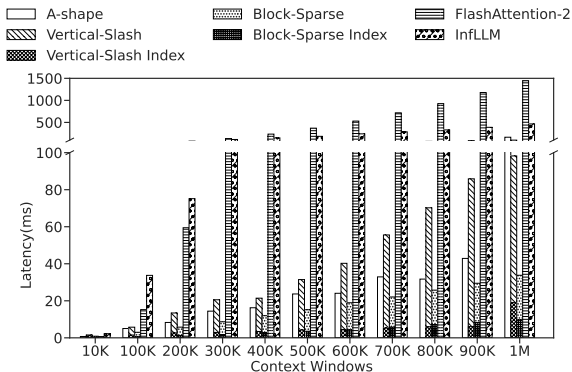
	InfiniteBench	RULER avg., % at length		
	avg.	32K	64K	128K
LLaMA-3-8B-262K				
Full attention	38.2	80.8	77.4	72.2
StreamingLLM (= A-shape)	23.8	17.2	14.2	9.4
InfLLM	34.8	55.6	43.0	39.5
MIInference	38.8	85.0	82.3	77.6

Above dense at every length, and the **gap widens** with length.

Jiang et al., MIInference 1.0 (2024). Tables 2 and 3, Sec. 4.2.



The kernel speedup is real, index cost included

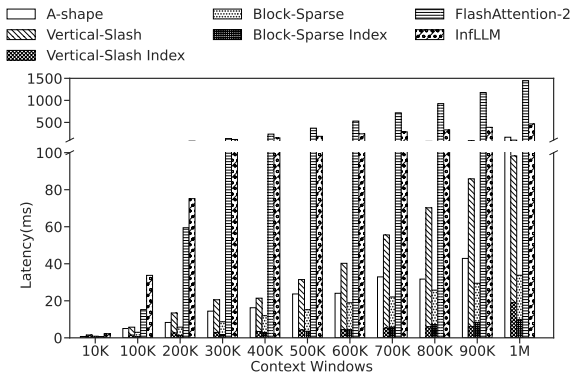


One attention kernel, one A100; note the **broken** y-axis above 100 ms.
FlashAttention-2: the dense baseline, part 1's algorithm better parallelised.

Jiang et al., MInference 1.0 (2024). Fig. 10 (cropped) and Fig. 12, Appendix D.2.



The kernel speedup is real, index cost included



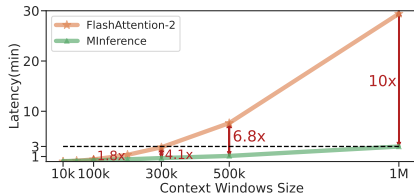
One attention kernel, one A100; note the **broken** y-axis above 100 ms.
FlashAttention-2: the dense baseline, part 1's algorithm better parallelised.

Every bar includes its own index build — **5–20%** of the total across patterns.

Jiang et al., MInference 1.0 (2024). Fig. 10 (cropped) and Fig. 12, Appendix D.2.



Thirty minutes to three



(b) Latency Speedup

End-to-end pre-fill, LLaMA-3-8B, one A100.

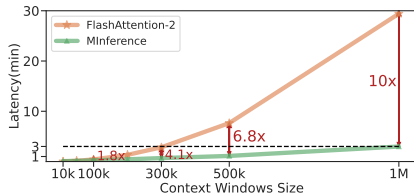
Context	Speedup
100k	1.8×
300k	4.1×
500k	6.8×
1M	10×

30 min → 3 min for a 1M-token prompt, with no fine-tuning and no change to the model.

Jiang et al., MInference 1.0 (2024). Fig. 1b (cropped), Sec. 4.2.



Thirty minutes to three



(b) Latency Speedup

End-to-end pre-fill, LLaMA-3-8B, one A100.

Next: Siddharth on SparseAttention.

Context	Speedup
100k	1.8×
300k	4.1×
500k	6.8×
1M	10×

30 min → 3 min for a 1M-token prompt, with no fine-tuning and no change to the model.

Jiang et al., MInference 1.0 (2024). Fig. 1b (cropped), Sec. 4.2.



Why SparseAttention after MInference?

MInference shows that structured sparsity can make long-context prefill practical. Its results also expose the central trade-off: the mask must be cheap while preserving the interactions that matter.

Model (seq_len)	Attention (Sparsity)	Speed (1/t)↑	WikiText (Ppl.) ↓	Longbench ↑	InfiniteBench ↑	NIAH ↑
Llama3.1 (128K)	Full-Attention	156.9	6.013	38.682	0.6594	0.907
	MInference (0.5)	140.1	10.631	28.860	0.5152	0.832
	FlexPrefill (0.5)	240.6	6.476	38.334	0.6460	0.858
	MInference (0.3)	115.7	6.705	34.074	0.6532	0.870
	FlexPrefill (0.42)	206.9	6.067	38.334	0.6581	0.878
	SparseAttn (0.54)	708.1	6.020	39.058	0.6638	0.909

At 50% sparsity, WikiText perplexity rises from 6.013 to 10.631; the less aggressive setting reduces that gap but also reduces speed.

Comparison reproduced from Zhang et al., SparseAttention (ICML 2025), Table 1. Motivation: Zhang et al., SparseAttention (ICML 2025), Secs. 1–2.



Why SparseAttention after MInference?

MInference shows that structured sparsity can make long-context prefill practical. Its results also expose the central trade-off: the mask must be cheap while preserving the interactions that matter.

Model (seq_len)	Attention (Sparsity)	Speed (1/t)↑	WikiText (Ppl.) ↓	Longbench ↑	InfiniteBench ↑	NIAH ↑
Llama3.1 (128K)	Full-Attention	156.9	6.013	38.682	0.6594	0.907
	Minference (0.5)	140.1	10.631	28.860	0.5152	0.832
	FlexPrefill (0.5)	240.6	6.476	38.334	0.6460	0.858
	Minference (0.3)	115.7	6.705	34.074	0.6532	0.870
	FlexPrefill (0.42)	206.9	6.067	38.334	0.6581	0.878
	SparseAttn (0.54)	708.1	6.020	39.058	0.6638	0.909

At 50% sparsity, WikiText perplexity rises from 6.013 to 10.631; the less aggressive setting reduces that gap but also reduces speed. Patterns also differ across language, image, and video models. SparseAttention therefore targets a **training-free selector that adapts to each input** with little prediction overhead.

Comparison reproduced from Zhang et al., SparseAttention (ICML 2025), Table 1. Motivation: Zhang et al., SparseAttention (ICML 2025), Secs. 1–2.



Sparse attention needs a useful mask at low overhead

Dense attention forms

$$P = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right), \quad O = PV.$$

Softmax often drives many entries of P close to zero. Sparse attention exploits this in two steps:

predict important entries



compute only those entries

The mask must cost much less than the work it removes.

Existing masks fall into three broad families:

Pattern based: predefined structures such as windows or sinks.

Dynamic: construct the mask from the current input.

Training based: train with native sparse attention.

Universality. Task-specific patterns do not transfer reliably across models and modalities.

Usability. Accurate prediction can be expensive; cheap prediction can miss important regions. Some dynamic methods become worthwhile only at very long sequences.

Sparse-attention categories and limitations: Zhang et al., SpargeAttention (ICML 2025), Secs. 1–2.



SparseAttention1 uses three complementary techniques

1 Selective block prediction

Compress only internally similar blocks to one representative, build a coarse attention map, and retain the block pairs with high predicted mass.

Goal: universality and accuracy

2 Sparse warp online softmax

During the tiled kernel, compare local and running maxima to omit value updates whose softmax weights are already negligible.

Goal: more skipped work

3 Quantized implementation

Apply the same sparse decisions inside the 8-bit SageAttention kernel. Quantization supplies an additional implementation speedup.

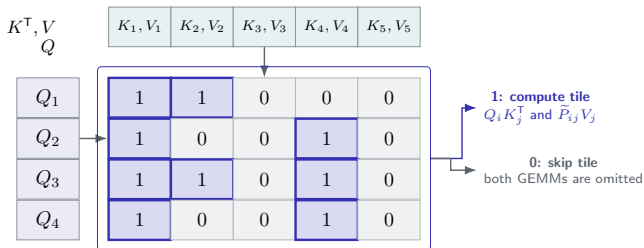
Supporting optimization

Method overview: Zhang et al., SparseAttention (ICML 2025), Sec. 1.



FlashAttention tiles define the unit of sparsity

FlashAttention streams one (Q_i, K_j, V_j) block tuple through each inner-loop iteration. SparseAttention keeps that layout and predicts one bit per tile.



$$M_g[i, j] = 0 \implies \text{skip } Q_i K_j^T \text{ and } \tilde{P}_{ij} V_j.$$

Tiled formulation, selective blocks, and fix blocks: Zhang et al., SparseAttention (ICML 2025), Secs. 3.1–3.2 and Figs. 3–4.



FlashAttention tiles define the unit of sparsity

Neighboring tokens often form coherent bands inside query and key blocks. A high-similarity block can therefore use one representative token; a heterogeneous block is always computed.

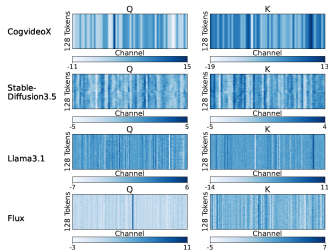
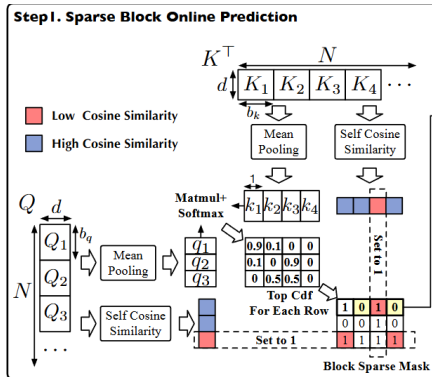


Figure 4. Exemplary patterns of the query and key in the attention of various models.



Compress selective blocks; protect non-self-similar fix blocks.

Tiled formulation, selective blocks, and fix blocks: Zhang et al., SparseAttention (ICML 2025), Secs. 3.1–3.2 and Figs. 3–4.



Mean pooling gives a cheap block-level proxy

Split Q and K into blocks Q_i and K_j .
Replace each candidate block by its
mean:

$$q_i = \text{mean}(Q_i, \text{axis} = 0),$$

$$k_j = \text{mean}(K_j, \text{axis} = 0).$$

Compressed block prediction: Zhang et al., SpargeAttention (ICML 2025), Sec. 3.2.



Mean pooling gives a cheap block-level proxy

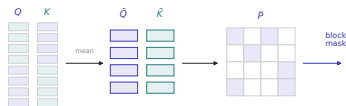
Split Q and K into blocks Q_i and K_j .
Replace each candidate block by its mean:

$$q_i = \text{mean}(Q_i, \text{axis} = 0),$$

$$k_j = \text{mean}(K_j, \text{axis} = 0).$$

Then compute one score per block pair:

$$\hat{S} = qk^T, \quad \hat{P}_i = \text{softmax}(\hat{S}_i).$$



Compressed block prediction: Zhang et al., SpargeAttention (ICML 2025), Sec. 3.2.



Mean pooling gives a cheap block-level proxy

Split Q and K into blocks Q_i and K_j .
Replace each candidate block by its mean:

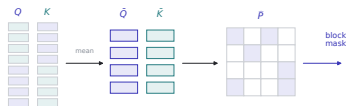
$$q_i = \text{mean}(Q_i, \text{axis} = 0),$$

$$k_j = \text{mean}(K_j, \text{axis} = 0).$$

Then compute one score per block pair:

$$\hat{S} = qk^T, \quad \hat{P}_i = \text{softmax}(\hat{S}_i).$$

The proxy only **ranks candidate tiles**. The selected tiles still use exact token-level attention.



Compressed block prediction: Zhang et al., SpargeAttention (ICML 2025), Sec. 3.2.



Self-similarity decides whether the mean is trustworthy

A block mean is reliable only when its tokens are internally coherent.

SparseAttention measures

$$\text{CosSim}(X) = \text{mean}\left(\frac{XX^T}{|\max(XX^T)|}\right).$$

Fix blocks bypass the approximation and are always computed.

Self-similarity judge and fix blocks: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2 and Algorithm 1.



Self-similarity decides whether the mean is trustworthy

A block mean is reliable only when its tokens are internally coherent.

SparseAttention measures

$$\text{CosSim}(X) = \text{mean} \left(\frac{XX^T}{|\max(XX^T)|} \right).$$

Using threshold θ :

$$s_i^Q = \text{CosSim}(Q_i), \quad s_j^K = \text{CosSim}(K_j).$$

Fix blocks bypass the approximation and are always computed.

Self-similarity judge and fix blocks: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2 and Algorithm 1.

high $c(B)$



low $c(B)$



Self-similarity decides whether the mean is trustworthy

A block mean is reliable only when its tokens are internally coherent.

SparseAttention measures

$$\text{CosSim}(X) = \text{mean} \left(\frac{XX^T}{|\max(XX^T)|} \right).$$

Using threshold θ :

$$s_i^Q = \text{CosSim}(Q_i), \quad s_j^K = \text{CosSim}(K_j).$$

$s \geq \theta$: **selective block**, $s < \theta$: **fix block**.

Fix blocks bypass the approximation and are always computed.

high $c(B)$



low $c(B)$



Self-similarity judge and fix blocks: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2 and Algorithm 1.



TopCDF: keep enough predicted mass

For each proxy row $\hat{P}_i$, sort the values and retain the smallest descending prefix whose cumulative mass reaches τ :

$$M_{\text{CDF}}[i, :] = \text{TopCdf}(\hat{P}_i, \tau).$$

TopCDF mask construction: Zhang et al., SpargeAttention (ICML 2025), Sec. 3.2.



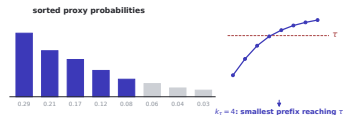
TopCDF: keep enough predicted mass

For each proxy row $\hat{P}_i$, sort the values and retain the smallest descending prefix whose cumulative mass reaches τ :

$$M_{\text{CDF}}[i, :] = \text{TopCdf}(\hat{P}_i, \tau).$$

Equivalently,

$$\min_{S \subseteq [n]} |S| \quad \text{s.t.} \quad \sum_{j \in S} \hat{p}_j \geq \tau.$$



TopCDF mask construction: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2.



TopCDF: keep enough predicted mass

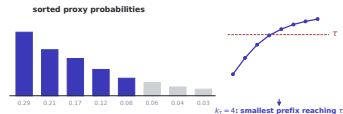
For each proxy row $\hat{P}_i$, sort the values and retain the smallest descending prefix whose cumulative mass reaches τ :

$$M_{\text{CDF}}[i, :] = \text{TopCdf}(\hat{P}_i, \tau).$$

Equivalently,

$$\min_{S \subseteq [n]} |S| \quad \text{s.t.} \quad \sum_{j \in S} \hat{p}_j \geq \tau.$$

τ directly controls the predicted mass that the block mask preserves.



TopCDF mask construction: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2.



Stage 1 constructs a conservative block mask

1. Compress and judge

$$q_i = \text{mean}(Q_i, \text{axis} = 0), \quad s_i^Q = \text{CosSim}(Q_i),$$

$$k_j = \text{mean}(K_j, \text{axis} = 0), \quad s_j^K = \text{CosSim}(K_j).$$

2. Predict and select

$$\hat{S}_{ij} = q_i k_j^T, \quad \hat{S}[:, j] = -\infty \quad \text{if } s_j^K < \theta,$$

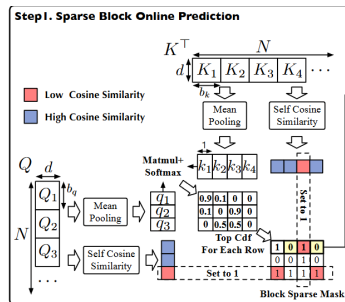
$$\hat{P}[i] = \text{softmax}(\hat{S}[i]),$$

$$M_g[i, :] = \text{TopCdf}(\hat{P}[i], \tau).$$

3. Protect every fix block

$$M_g[i, :] = 1 \text{ if } s_i^Q < \theta, \quad M_g[:, j] = 1 \text{ if } s_j^K < \theta.$$

Only selective-selective block pairs may be set to zero.

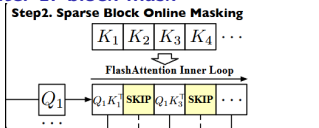


Prediction equations and fix-block rules: Zhang et al., SparseAttention (ICML 2025), Sec. 3.2 and Algorithm 1.



Two online filters remove different work

Filter 1: block mask

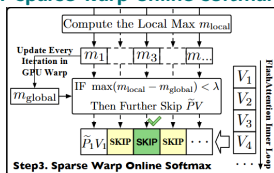


Apply M_g directly in the FlashAttention inner loop:

$$M_g[i, j] = 0 \Rightarrow \begin{array}{l} \text{skip } Q_i K_j^T, \\ \text{skip } \tilde{P}_{ij} V_j. \end{array}$$

A rejected tile avoids both GEMMs.

Filter 2: sparse warp online softmax



$$\begin{aligned} m_{\text{local}} &= \text{rowmax}(S_{ij}), \\ m_{ij} &= \max(m_{i,j-1}, m_{\text{local}}), \\ \tilde{P}_{ij} &= e^{S_{ij} - m_{ij}}. \end{aligned}$$

$$\max(m_{\text{local}} - m_{ij}) < \lambda < 0 \implies \tilde{P}_{ij} \approx 0.$$

Hence $O_{ij} \approx O_{i,j-1}$; the warp skips $\tilde{P}_{ij} V_j$ **only**.

First-stage masking and sparse warp online softmax: Zhang et al., SpargeAttention (ICML 2025), Secs. 3.3–3.4 and Fig. 3.



Quantization and sparsity compose inside one kernel

SageAttention changes the arithmetic used by each attention tile; SpargeAttention changes which tile operations execute. The implementation combines them by adding two checks to the existing inner loop:

1. **Before the tile computation:** if $M_g[i, j] = 0$, skip the entire inner-loop iteration.
2. **Before the output update:** if the warp-level max gap is below λ , skip only $\tilde{P}_{ij} V_j$ for that warp.

quantized tile arithmetic

+

two sparse decisions

The mask construction and online-softmax logic remain the conceptual contribution; the quantized kernel provides an additional implementation speedup.

SageAttention integration and kernel checks: Zhang et al., SpargeAttention (ICML 2025), Sec. 3.5 and Algorithm 1.



Full SparseAttention1 algorithm

Algorithm 1 Implementation of SparseAttn.

```
1: Input: Matrices  $Q(\text{FP16}), K(\text{FP16}), V(\text{FP16}) \in \mathbb{R}^{N \times d}$ , block size  $b_q, b_{kv}$ , count of GPU Warps  $c_w$ , hyper-parameters  $\tau, \theta$ , and  $\lambda$ .
2: Divide  $Q$  to  $T_m = N/b_q$  blocks  $\{Q_i\}$ ; divide  $K, V$  to  $T_n = N/b_{kv}$  blocks  $\{K_i\}$  and  $\{V_i\}$ .
3:  $\hat{Q}_i, \hat{K}_j, \delta_Q, \delta_K = \text{Quant}(Q_i, K_j)$ ; // per-block quantization in SageAttention.
4:  $q = \{q_i\} = \{\text{mean}(Q_i, \text{axis} = 0)\}$ ;  $k = \{k_j\} = \{\text{mean}(K_j, \text{axis} = 0)\}$ ;
5:  $\hat{S} = qk^\top$ ;  $s_{qi} = \text{CosSim}(Q_i)$ ;  $s_{kj} = \text{CosSim}(K_j)$ ;  $\hat{S}[:, j] = -\infty$ , If  $s_{kj} < \theta$ ;
6:  $\hat{P}[i] = \text{Softmax}(\hat{S}[i])$ ;  $M[i, :] = \text{TopCdf}(\hat{P}[i], \tau)$ ;  $M[i, :] = 1$ , If  $s_{qi} < \theta$ ;  $M[:, j] = 1$ , If  $s_{kj} < \theta$ ;
7: for  $i = 1$  to  $T_m$  do
8:   Load  $\hat{Q}_i$  and  $\delta_Q[i]$  into a SM;
9:   for  $j$  in  $[1, T_n]$  do
10:    if  $M[i, j]! = 0$  then
11:      Load  $\hat{K}_j, \hat{V}_j$ , and  $\delta_K[j]$  into the SM;
12:       $S_{ij} = \text{Matmul}(\hat{Q}_i, \hat{K}_j^\top) \times \delta_Q \times \delta_K$ ; // dequantization of SageAttention.
13:       $m_{\text{local}} = \text{rowmax}(S_{ij})$ ;  $m_{ij} = \max(m_{i, j-1}, m_{\text{local}})$ ;  $\tilde{P}_{ij} = \exp(S_{ij} - m_{ij})$ ;  $l_{ij} = e^{m_{i, j-1} - m_{ij}} l_{i, j-1} + \text{rowsum}(\tilde{P}_{ij})$ ;
14:       $i_w = \text{range}(c_w)$ ;  $I_w = \lceil \frac{i_w * b_q}{c_w} : \frac{(i_w + 1) * b_q}{c_w} \rceil$ ;
15:      if  $\max(m_{\text{local}}[I_w] - m_{ij}[I_w]) > \lambda$  then
16:         $O_{ij}[I_w] = \text{diag}(e^{m_{i, j-1}[I_w] - m_{ij}[I_w]}) O_{i, j-1}[I_w] + \text{Matmul}(\tilde{P}_{ij}[I_w], V_j)$ ; // Paralleled by  $c_w$  warps.
17:      end if
18:    end if
19:  end for
20:   $O_i = \text{diag}(l_{i, T_n})^{-1} O_{i, T_n}$ ;
21:  Write  $O_i$ ;
22: end for
23: return  $O = \{O_i\}$ ;
```

Blue lines construct and apply the two sparse filters. Quantized arithmetic remains inside the retained tiles.

Algorithm 1 reproduced from Zhang et al., SparseAttention (ICML 2025), Sec. 3.



SpargAttention1 results

Model (seq_len)	Attention (Sparsity)	Speed (1/t)↑	WikiText (Ppl.) ↓	Longbench ↑	InfiniteBench ↑	NIAH ↑
Llama3.1 (128K)	Full-Attention	156.9	6.013	38.682	0.6594	0.907
	Minference (0.5)	140.1	10.631	28.860	0.5152	0.832
	FlexPrefill (0.5)	240.6	6.476	38.334	0.6460	0.858
	Minference (0.3)	115.7	6.705	34.074	0.6532	0.870
	FlexPrefill (0.42)	206.9	6.067	38.334	0.6581	0.878
	SpargeAttn (0.54)	708.1	6.020	39.058	0.6638	0.909

Model (seq_len)	Attention (Sparsity)	Speed (1/t)↑	CLIPSIM ↑	CLIP-T ↑	VQA-a ↑	VQA-t ↑	FScore ↑
CogvideoX (17K)	Full-Attention	166.0	0.1819	0.9976	80.384	75.946	5.342
	Minference (0.5)	264.6	0.1728	0.9959	70.486	62.410	2.808
	FlexPrefill (0.6)	175.3	0.1523	0.9926	1.5171	4.5034	1.652
	Minference (0.3)	196.9	0.1754	0.9964	77.326	63.525	3.742
	FlexPrefill (0.45)	142.0	0.1564	0.9917	7.7259	8.8426	2.089
	SpargeAttn (0.46)	507.9	0.1798	0.9974	78.276	74.846	5.030
Mochi (22K)	Full-Attention	164.2	0.1725	0.9990	56.472	67.663	1.681
	Minference (0.5)	202.4	0.1629	0.9891	6.668	50.839	0.653
	FlexPrefill (0.48)	191.3	0.1667	0.9898	0.582	0.0043	X
	Minference (0.3)	147.7	0.1682	0.9889	14.541	42.956	0.833
	FlexPrefill (0.4)	171.7	0.1677	0.9909	2.941	0.7413	X
	SpargeAttn (0.47)	582.4	0.1720	0.9990	54.179	67.219	1.807

Llama3.1: 708.1 speed at 0.54 sparsity

CogVideoX: 507.9 speed with CLIP-T 0.9974

Original Table 1 excerpt: Zhang et al., SpargAttention (ICML 2025), Sec. 4.2. Arrows indicate metric direction.



Why SparseAttention2?

SparseAttention1 keeps the pretrained model fixed, so its sparse mask must remain conservative. At more aggressive sparsity, the same training-free strategy can lose important attention mass.

Method	IQ↑	OC↑	AQ↑	VR↑	VQA-a↑	VQA-t↑	Sparsity↑	Attn Time↓	E2E Time↓
Full Attention	63.67	20.27	64.41	0.1084	81.28	85.80	0%	97s	159s
SparseAttn	35.28	13.41	40.53	-0.1398	3.258	0.608	89%	12.6s	74.6s

Mask limitation

The 89% sparse result is faster, but the reported generation metrics collapse.

SparseAttention2

Adapt the model so a much smaller support can preserve the dense model's behavior.

Motivation and full-attention comparison: Zhang et al., SparseAttention2 (2026), Sec. 1 and Fig. 1.



Sparse-attention error has two components

For one dense attention row p and binary mask m , define the retained mass

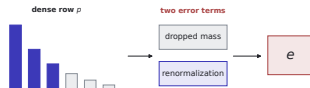
$$r = (p \odot m) \mathbf{1}^T, \quad o = pV,$$
$$o_s = \frac{p \odot m}{r} V.$$

Add and subtract $p \odot m$:

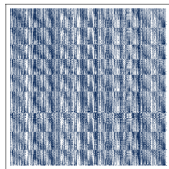
$$o - o_s = \left[\underbrace{p \odot (1 - m)}_{\text{dropped contribution}} + \underbrace{\left(1 - \frac{1}{r}\right) (p \odot m)}_{\text{renormalization shift}} \right] V.$$

A safe mask must retain enough mass and avoid an extreme renormalization.

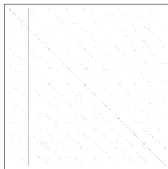
Sparse-attention error decomposition, Eq. (8): Zhang et al., SpargeAttention2 (2026), Sec. 3.1.



Top- k and Top- p fail in complementary regimes



(a) A uniform P . We keep the largest probabilities whose sum reaches 60% in each row.



(b) A skewed P , where we keep the largest probabilities whose sum reaches 60% in each row.

Uniform attention

A fixed k can retain too little total mass when probability is spread across many entries.

Skewed attention

Top- p may stop after an attention sink reaches the threshold, leaving too little support diversity.

Combined rule

Top- p protects retained mass, while Top- k protects support.

P distribution	Top- k	Top- p	Hybrid
Uniform	0.4150	0.3726	0.3707
Skewed	0.1664	0.2160	0.1671

Uniform and skewed attention analysis: Zhang et al., SparseAttention2 (2026), Sec. 3.2, Fig. 2 and Table 1.



Hybrid Top- k +Top- p protects mass and support

First form the pooled block-level proxy

$$\bar{P} = \text{softmax} \left(\frac{\text{pool}(Q) \text{pool}(K)^T}{\sqrt{d}} \right).$$

For every row, retain the union of the two selections:

$$\bar{M}_{ij} = 1 \iff j \in \text{Top-}k(\bar{P}_{i,:}) \cup \text{Top-}p(\bar{P}_{i,:}).$$

Therefore

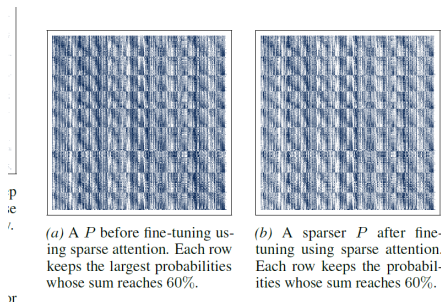
$$|S_{\text{hybrid}}| \geq k_{\min}, \quad \sum_{j \in S_{\text{hybrid}}} \bar{P}_{ij} \geq p_0, \quad k_{\text{hybrid}} = \max(k_{\min}, k_p).$$

Top- k supplies a support floor; Top- p supplies a retained-mass floor.

Hybrid masking rule, Eq. (9): Zhang et al., SparseAttention2 (2026), Sec. 4.1.



Sparse fine-tuning concentrates attention



Fine-tuning	Sparsity	L1 Error
(a) Before	51.3%	0.4901
(b) After	56.9%	0.4119

Observed effect

Fine-tuning concentrates probability on fewer entries. The reported sparsity increases while the sparse-attention L1 error decreases.

From Eq. (8), more retained mass reduces both the dropped contribution and the renormalization shift.

Fine-tuning concentrates attention and reduces sparse error: Zhang et al., SpargeAttention2 (2026), Sec. 3.2 and Table 2.



Velocity distillation preserves the dense model's function

Wan-style video models use flow matching.
The model predicts the velocity that moves a noisy latent toward the clean video latent:

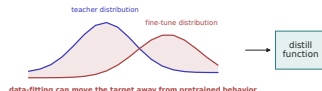
$$x_t = tx_1 + (1 - t)x_0, \quad v_t = x_1 - x_0.$$

Ordinary training fits the sampled video target:

$$\mathcal{L}_{\text{FM}} = \mathbb{E} \|u_{\theta}(x_t, c, t) - v_t\|^2.$$

A small fine-tuning distribution can move the model away from its pretrained behavior. SpargeAttention2 freezes the dense teacher and uses

$$\mathcal{L}_{\text{VD}} = \mathbb{E} \|u_{\text{sparse}}(x_t, c, t) - u_{\text{full}}(x_t, c, t)\|^2.$$



Flow matching and velocity distillation: Zhang et al., SpargeAttention2 (2026), Secs. 2.3 and 4.2.



Velocity distillation preserves the dense model's function

Wan-style video models use flow matching. The model predicts the velocity that moves a noisy latent toward the clean video latent:

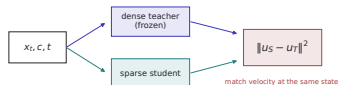
$$x_t = tx_1 + (1 - t)x_0, \quad v_t = x_1 - x_0.$$

Ordinary training fits the sampled video target:

$$\mathcal{L}_{\text{FM}} = \mathbb{E} \|u_\theta(x_t, c, t) - v_t\|^2.$$

A small fine-tuning distribution can move the model away from its pretrained behavior. SpargeAttention2 freezes the dense teacher and uses

$$\mathcal{L}_{\text{VD}} = \mathbb{E} \|u_{\text{sparse}}(x_t, c, t) - u_{\text{full}}(x_t, c, t)\|^2.$$



Flow matching and velocity distillation: Zhang et al., SpargeAttention2 (2026), Secs. 2.3 and 4.2.



SpargAttention2 algorithm

A. Sparse attention operator

1. Tile Q into $\{Q_i\}$ and K, V into $\{K_j, V_j\}$.
2. Pool every block and compute

$$\bar{P} = \text{softmax} \left(\frac{\text{pool}(Q) \text{pool}(K)^T}{\sqrt{d}} \right).$$

3. Build the hybrid mask

$$\bar{M} = \text{TopK}(\bar{P}) \vee \text{TopP}(\bar{P}).$$

4. In the FlashAttention loop, compute only tiles with $\bar{M}[i, j] = 1$.

B. Adaptation by velocity distillation

1. Copy the dense weights to the sparse student and freeze the dense teacher.
2. Sample (x_1, c) , noise x_0 , and t , then construct $x_t = tx_1 + (1 - t)x_0$.
3. Evaluate $u_{\text{full}}(x_t, c, t)$ and $u_{\text{sparse}}(x_t, c, t)$ on the same input.
4. Update only the sparse student by minimizing

$$\mathcal{L}_{\text{VD}} = \|u_{\text{sparse}} - u_{\text{full}}\|^2.$$

Hybrid masking controls inference; velocity distillation controls adaptation.



SpargAttention2 results

Method	IQ↑	OC↑	AQ↑	VR↑	VQA-a↑	VQA-t↑	Sparsity↑	Attn Time↓	E2E Time↓
Full Attention	68.01	22.44	64.66	0.1238	86.15	87.00	0%	2550s	3043s
SpargAttn	38.46	16.26	42.16	-0.1306	9.926	4.451	86%	415s	908s
VSA	64.03	21.27	63.37	0.1074	77.63	85.60	90%	651s	1144s
VMoBA	67.18	20.85	63.64	0.1117	81.66	83.21	90%	832s	1325s
SLA	64.43	20.89	61.89	0.1078	76.90	82.60	95%	285s	778s
SpargAttn2	69.08	21.62	64.92	0.1149	85.21	87.48	95%	157s	650s

95% sparsity and 157
s attention time

650 s end-to-end,
versus 3043 s dense

IQ 69.08 and VQA-t
87.48

Original Table 5, Wan2.1-14B at 720p, reproduced verbatim: Zhang et al., SpargAttention2 (2026), Sec. 5.2.



Sparse1 adapts computation; Sparse2 adapts the model



SparseAttention 1

fixed model $\Rightarrow$ **conservative sparse computation**

- ▶ compress only trusted blocks;
- ▶ TopCDF preserves predicted mass;
- ▶ uncertain blocks are computed exactly;
- ▶ online-softmax gap removes negligible *PV*.

SparseAttention 2

aggressive sparsity $\Rightarrow$ **train model to preserve behavior**

- ▶ hybrid Top- k +Top- p protects mass and support;
- ▶ training concentrates attention onto sparse support;
- ▶ velocity distillation preserves dense model dynamics.



Sparse VideoGen2

A training-free method for accelerating video diffusion transformers.



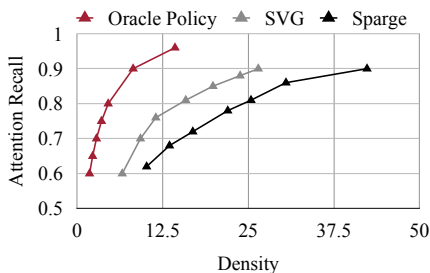
Compare the corresponding dense and SVG2 frames in each pair.

Can we preserve the video while computing fewer interactions?

Yang et al., Sparse VideoGen2 (2025). Appendix A, Fig. 9 (two examples, cropped). CC BY 4.0.



An oracle reveals the opportunity



13% work
can retain about
95% mass

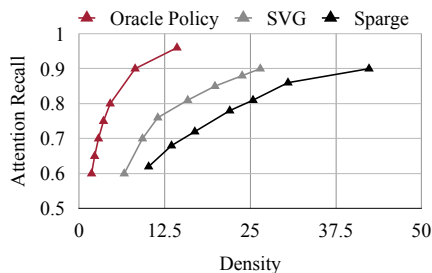
under the paper's oracle
selection.

Wan 2.1 image-to-video attention maps.
Left: less attention work. Up: more mass
retained.

Yang et al., Sparse VideoGen2 (2025). Sec. 3.1, Fig. 3 (cropped). Recall = retained attention mass. CC BY 4.0.



An oracle reveals the opportunity



13% work
can retain about
95% mass

under the paper's oracle
selection.

The obstacle

Finding that oracle mask
already requires full attention.

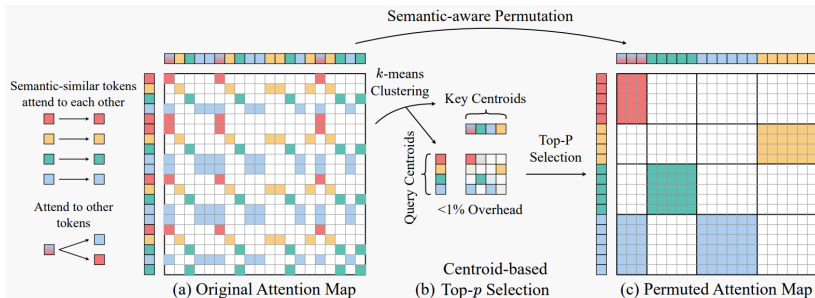
Wan 2.1 image-to-video attention maps.
Left: less attention work. Up: more mass
retained.

SVG2 needs a cheap estimate of which interactions matter.

Yang et al., Sparse VideoGen2 (2025). Sec. 3.1, Fig. 3 (cropped). Recall = retained attention mass. CC BY 4.0.



Semantic-aware permutation



Yang et al., Sparse VideoGen2 (2025). Secs. 4.1–4.2. CC BY 4.0.



Centroids make selection inexpensive

One centroid represents each cluster:

$$\bar{q}_i = \frac{1}{|Q_i|} \sum_{q \in Q_i} q, \quad \bar{k}_j = \frac{1}{|K_j|} \sum_{k \in K_j} k.$$

Full scores

$$QK^T$$

$N_Q \times N_K$ token pairs

Centroid scores

$$s_{ij} = \bar{q}_i \bar{k}_j^T / \sqrt{d}$$

$C_Q \times C_K$ cluster pairs

Yang et al., Sparse VideoGen2 (2025). Secs. 4.1–4.2, 5.1. Centroids approximate scores for selecting cluster pairs.



Centroids make selection inexpensive

One centroid represents each cluster:

$$\bar{q}_i = \frac{1}{|Q_i|} \sum_{q \in Q_i} q, \quad \bar{k}_j = \frac{1}{|K_j|} \sum_{k \in K_j} k.$$

Full scores

$$QK^T$$

$N_Q \times N_K$ token pairs

Centroid scores

$$s_{ij} = \bar{q}_i \bar{k}_j^T / \sqrt{d}$$

$C_Q \times C_K$ cluster pairs

The paper uses 200 query clusters and 1000 key clusters:

Only 200,000 proxy scores per head, instead of all token pairs (14000x14000).

Yang et al., Sparse VideoGen2 (2025). Secs. 4.1–4.2, 5.1. Centroids approximate scores for selecting cluster pairs.



Cluster size changes attention mass

All 8 keys have logit 0, so each gets mass $1/8$.

Cluster A: 2 keys

$$\boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}}$$

$$2 \times \frac{1}{8} = \frac{1}{4}$$

A: 25%

Cluster B: 6 keys

$$\boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}}$$

$$6 \times \frac{1}{8} = \frac{3}{4}$$

B: 75%

Equal centroid logits do not imply equal total cluster mass.

Yang et al., Sparse VideoGen2 (2025). Sec. 4.2, Eq. 2. Equal-logit toy example. s_{ij} is the centroid score.



Cluster size changes attention mass

All 8 keys have logit 0, so each gets mass $1/8$.

Cluster A: 2 keys

$$\boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}}$$

$$2 \times \frac{1}{8} = \frac{1}{4}$$

Cluster B: 6 keys

$$\boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}} \quad \boxed{\frac{1}{8}}$$

$$6 \times \frac{1}{8} = \frac{3}{4}$$

A: 25%

B: 75%

Equal centroid logits do not imply equal total cluster mass.

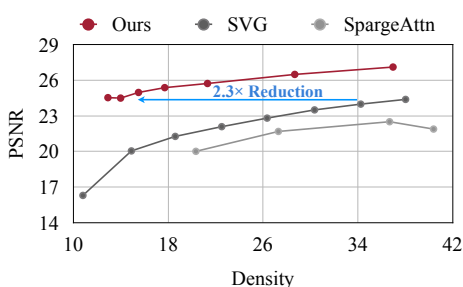
The proxy accounts for the number of keys in each cluster:

$$\hat{P}_{ij} = \frac{|K_j| e^{s_{ij}}}{\sum_{\ell} |K_{\ell}| e^{s_{i\ell}}}.$$

Yang et al., Sparse VideoGen2 (2025). Sec. 4.2, Eq. 2. Equal-logit toy example. s_{ij} is the centroid score.



Better grouping improves the quality–density trade-off



Density

Sparse attention work divided by dense work.

PSNR

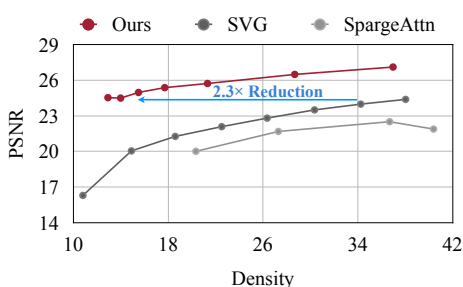
Similarity to the dense output, in dB.

Wan 2.1 image-to-video.

Yang et al., Sparse VideoGen2 (2025). Fig. 2 (cropped), Sec. 5.4. The 2.3× annotation compares density. CC BY 4.0.



Better grouping improves the quality–density trade-off



Density

Sparse attention work divided by dense work.

PSNR

Similarity to the dense output, in dB.

At comparable density, SVG2 better preserves the dense output.

Wan 2.1 image-to-video.

PSNR measures agreement with dense output. Higher is closer.

Yang et al., Sparse VideoGen2 (2025). Fig. 2 (cropped), Sec. 5.4. The 2.3× annotation compares density. CC BY 4.0.



Measured end-to-end generation speed

720p generation on a single H100. Standard SVG2 settings.

Model / task	Density	PSNR (dB)	Speedup
HunyuanVideo 13B T2V	25.45%	30.452	2.30×
Wan 2.1 14B T2V	29.51%	25.808	1.60×
Wan 2.1 14B I2V	31.28%	26.562	1.58×

T2V: text-to-video. I2V: image-to-video.

VBench (dense / SVG2): Hunyuan T2V 0.850 / 0.852; Wan T2V 0.846 / 0.842.

Yang et al., Sparse VideoGen2 (2025). Table 1, Sec. 5.1. Speedup is total generation latency relative to the dense baseline.



Measured end-to-end generation speed

720p generation on a single H100. Standard SVG2 settings.

Model / task	Density	PSNR (dB)	Speedup
HunyuanVideo 13B T2V	25.45%	30.452	2.30×
Wan 2.1 14B T2V	29.51%	25.808	1.60×
Wan 2.1 14B I2V	31.28%	26.562	1.58×

T2V: text-to-video. I2V: image-to-video.

VBench (dense / SVG2): Hunyuan T2V 0.850 / 0.852; Wan T2V 0.846 / 0.842.

What still takes time

The first 30% of denoising steps use dense attention.

Clustering, reordering and the rest of the model still take time.

Yang et al., Sparse VideoGen2 (2025). Table 1, Sec. 5.1. Speedup is total generation latency relative to the dense baseline.



SVG2: accurate selection needs efficient execution

Group Find similar Q vectors and similar K vectors.



Select Estimate cluster mass and keep a top- p prefix.



Compute Use original Q/K/V on the selected cluster pairs.



Restore Return each output to its original token position.

Centroid reuse and a custom kernel make the saved work worthwhile.

Next: Advay on attention in DeepSeek language models.

Yang et al., Sparse VideoGen2 (2025). Secs. 4–6. Training-free, with a model-dependent speed / quality trade-off.



Where we are going

We now try to see how sparse attention can be incorporated into the pre-training phase itself. Specifically, how it was done by Deepseek

1. **Preliminaries.** MHA, MQA, GQA, RoPE. We build a *unified cost model* and fix notation for the rest of the talk.
2. **DeepSeek-V2: MLA.** Low-rank joint KV compression, weight absorption, decoupled RoPE. Two execution modes (MHA-mode / MQA-mode).
3. **DeepSeek-V3.2: DSA.** Lightning indexer + fine-grained top- k selection.
4. **DeepSeek-V4: CSA + HCA.** Compression *along the sequence*, then sparsity; hybrid interleaving.

After each stage we add a row to a running **scoreboard**: cache, FLOPs & arithmetic intensity



Row-vector / batched convention. Hidden states stack in rows, $H \in \mathbb{R}^{n \times d}$ and a projection is a *right* multiplication HW .

Shapes			
n	sequence length (context)	d_c	KV / query latent dimension
d	model hidden size	d_h^R	decoupled RoPE head dim
l	number of layers	k	# selected KV entries
n_h	number of query heads	m, m'	sequence compression rates
$d_h (=c)$	per-head dimension	n_{win}	sliding-window size
n_g	number of KV groups	b	bytes per stored element

- ▶ One multiply-accumulate = 2 FLOPs; a matmul $(n \times k)(k \times m)$ costs $2nkm$.
- ▶ In general $n_h d_h \neq d$ (V2: $d = 5120$, $n_h d_h = 16384$)



Three phases, three different bottlenecks

Every trick helps in *some* of these phases and not others.

	Training	Prefill	Decode
Input	n tokens	n tokens	one token, cache of n
Attn shape	$n \times n$ causal	$n \times n$ causal	$1 \times n$
Bottleneck	FLOPs + activation memory	FLOPs	memory bandwidth
Scaling	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$ /token, $\Theta(n^2)$ total
Hurts	recompute / check-pointing	time-to-first-token	tokens/s, max batch

- ▶ **Prefill is compute-bound**
- ▶ **Decode is memory-bound**
- ▶ We will see MQA/GQA/MLA for decode; DSA/CSA/HCA for both.



The five numbers we will track

For **one attention layer**, and per token where stated:

1. **KV cache** for $\mathcal{M}$ stored elements per token per layer. Total bytes $= \mathcal{M} \cdot l \cdot n \cdot b$ per sequence.
2. **Prefill FLOPs** split into a *projection* term (linear in n) and an *attention* term (quadratic in n).
3. **Decode FLOPs** per generated token.
4. **Arithmetic intensity** $\mathcal{I}$ of the decode attention: FLOPs per byte moved. Compare against the hardware ridge point.
5. **Quality** from the papers' own ablations. A saving that costs accuracy is not a saving.

Hardware anchor (H800, the machine all three papers use):

BF16 peak ≈ 990 TFLOP/s, HBM BW ≈ 3.35 TB/s



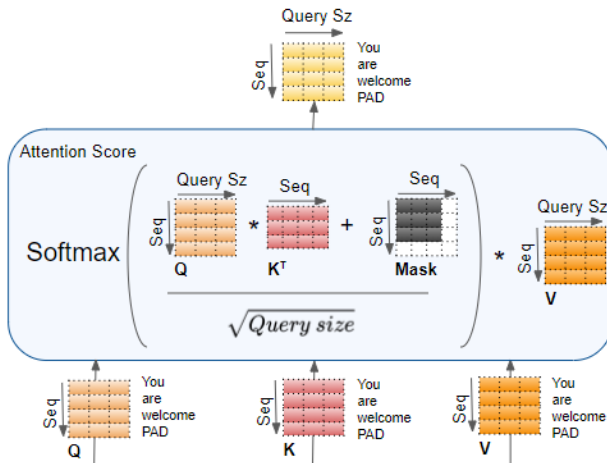
$$AI_{\text{ridge}} \approx \frac{990}{3.35} \approx 300 \frac{\text{FLOP}}{\text{byte}}$$



Preliminaries



MHA dataflow



Repeated independently for each of the n_h heads, and for each of the l layers.



Standard Multi-Head Attention

Let $H = [\mathbf{h}_1; \dots; \mathbf{h}_n] \in \mathbb{R}^{n \times d}$ be the layer input.

$$Q = HW_Q, \quad K = HW_K, \quad V = HW_V \text{ where } W_Q, W_K, W_V \in \mathbb{R}^{d \times n_h d_h}$$

$$Q = [Q_1, \dots, Q_{n_h}], K = [K_1, \dots, K_{n_h}], V = [V_1, \dots, V_{n_h}]$$

$$\text{where } Q_i, K_i, V_i \in \mathbb{R}^{n \times d_h}$$

$$O_i = \text{Softmax} \left(\frac{Q_i K_i^\top}{\sqrt{d_h}} + C \right) V_i \text{ where } C_{ts} = \begin{cases} 0 & \text{if } s \leq t \\ -\infty & \text{otherwise} \end{cases}$$

$$U = [O_1, \dots, O_{n_h}] W_O \text{ where } W_O \in \mathbb{R}^{n_h d_h \times d}$$

At decode step t we need $K_{1:t}$ and $V_{1:t}$, and they do not depend on t . So we cache them rather than recompute:

$$\mathcal{M}_{\text{MHA}} = \underbrace{2}_{K \text{ and } V} \cdot n_h d_h \text{ elements per token per layer.}$$



Derivation 1 — prefill FLOPs

Per layer, sequence length n , counting 2 FLOPs per MAC.

Projections. Four matmuls of shape $(n \times d)(d \times n_h d_h)$ or its transpose:

$$F_{\text{proj}} = 4 \cdot 2 n d n_h d_h = 8 n d n_h d_h.$$

Attention. Causality means only $\approx n^2/2$ query-key pairs are live. Scores $Q_i K_i^\top$: $2 \cdot \frac{n^2}{2} \cdot d_h$ per head = $n^2 n_h d_h$ over all heads. The Softmax $\cdot V_i$ product costs the same. Hence

$$F_{\text{attn}} = 2 n^2 n_h d_h.$$

Crossover. Attention overtakes the projections when

$$2 n^2 n_h d_h > 8 n d n_h d_h \iff n > 4d$$

This is independent of the head configuration. For DeepSeek-V2 ($d = 5120$): $n \approx 20\text{K}$. For DeepSeek-V4-Pro ($d = 7168$): $n \approx 29\text{K}$.



Derivation 2 — decode: FLOPs, bytes, and intensity

One new token, cache already holding n entries. Per layer:

$$F_{\text{proj}}^{\text{dec}} = 8 d n_h d_h,$$

$$F_{\text{attn}}^{\text{dec}} = \underbrace{2n n_h d_h}_{\text{qK}^\top} + \underbrace{2n n_h d_h}_{\text{sV}} = 4n n_h d_h.$$

Bytes moved to compute the attention part (the whole KV cache):

$$B_{\text{attn}}^{\text{dec}} = b \cdot 2n n_h d_h \stackrel{b=2}{=} 4n n_h d_h \text{ bytes.}$$

Arithmetic intensity of decode attention:

$$\mathcal{I}_{\text{MHA}} = \frac{4n n_h d_h}{4n n_h d_h} = \boxed{1 \frac{\text{FLOP}}{\text{byte}}}$$



Lets Batch

Run B sequences at once. Compare the two things we read from HBM:

	FLOPs	Bytes read	Intensity
Weights (W^Q, W^K, W^V, W^O)	$B \cdot 8d n_h d_h$	$8d n_h d_h$ (<i>shared!</i>)	$\propto B$
KV cache	$B \cdot 4n n_h d_h$	$4Bn n_h d_h$ (<i>private</i>)	1, always

- ▶ Weight reads amortise across the batch
- ▶ KV reads do not: every sequence owns its cache
- ▶ The cache also *caps* the batch. With memory budget M ,

$$B_{\max} = \frac{M}{\mathcal{M} \cdot l \cdot n \cdot b}, \quad \text{throughput} \propto B_{\max} \propto \frac{1}{\mathcal{M}}.$$



[Appendix] How bad is MHA?

DeepSeek-V2's configuration ($n_h=128$, $d_h=128$, $l=60$), if it had used MHA, in BF16:

$$\mathcal{M}l = 2 \cdot 16384 \cdot 60 = 1,966,080 \text{ elements} = 3.75 \text{ MiB per token.}$$

Context	KV cache, one sequence	fits on an 8×H800 node (640 GB)?
4K	15 GiB	yes, batch ≈ 40
32K	120 GiB	batch ≈ 5
128K	480 GiB	<i>barely one sequence</i>

A 236B-parameter model in FP8 is ≈ 236 GB of weights. At 128K context the cache for a *single* user would exceed the weights by $2\times$. Serving is impossible.

Reading the number the other way: decoding one token requires streaming $3.75 \text{ MiB} \times n$ bytes. At 3.35 TB/s and $n=128\text{K}$ that is ≈ 147 ms per token, *per sequence* — about 7 tokens/s, before any compute.



First fix: share the keys and values across heads

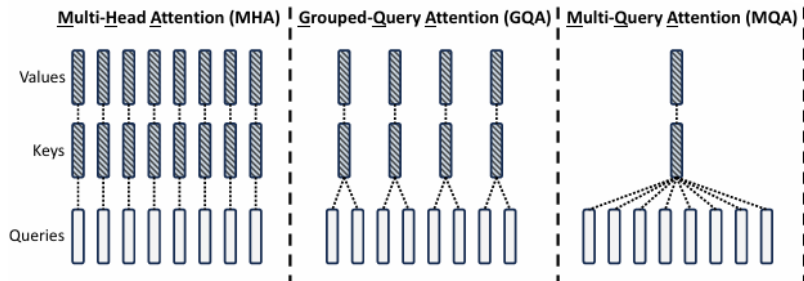


Figure 1: Different models to reduce KV cache sizes

n_h query heads, but only n_g distinct key/value heads; head i reads group $\lceil in_g/n_h \rceil$. $W^K, W^V \in \mathbb{R}^{d \times n_g d_h}$.

$$\mathcal{M}_{\text{GQA}} = 2 n_g d_h$$

$$\mathcal{M}_{\text{MQA}} = 2 d_h \quad (n_g = 1).$$



Impact of MQA and GQA

Substitute $n_h d_h \rightarrow n_g d_h$ *only in the K, V projections and in the cache.*
The query heads are untouched, and keys are *broadcast* to them.

Phase	Effect	Why
Training	almost none	score matrix still $n_h \times n \times n$
Prefill	none on attention small saving on projections	$F_{\text{attn}} = 2n^2 n_h d_h$, unchanged W^K, W^V shrink by n_h/n_g
Decode	$n_h/n_g \times$ less cache and less traffic	fewer distinct K, V to store/read

For the intensity perspective FLOPs stay at $4n n_h d_h$, bytes fall to $b \cdot 2n n_g d_h$:

$$\mathcal{I}_{\text{GQA}} = \frac{4n n_h d_h}{4n n_g d_h} = \frac{n_h}{n_g}.$$



[Appendix] What MQA and GQA take away

Sharing K , V removes capacity: heads in a group attend over the same subspace and differ only through their queries.

DeepSeek's controlled ablation — three $\approx 7B$ dense models, 1.33T tokens, identical but for the attention mechanism (layers adjusted to match parameters):

Benchmark	MQA (7.1B)	GQA-8 (6.9B)	MHA (6.9B)
BBH (EM, 3-shot)	33.2	35.6	37.0
MMLU (Acc, 5-shot)	37.9	41.2	45.2
C-Eval (Acc, 5-shot)	30.0	37.7	42.9
CMMLU (Acc, 5-shot)	34.6	38.4	43.5

MQA loses 7.3 MMLU points against MHA — not a rounding error, roughly a model-size generation.

This sets up the entire talk: the axis *cache size* vs *quality* appeared to be a hard trade-off. MLA's claim in the next section is that it is not — that you can have MQA-level cache *and* better-than-MHA quality.¹

¹DeepSeek-AI, *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*, arXiv:2405.04434, 2024, Table 8 (Appendix D.1).



Rotary Positional Encoding

Self-attention is permutation-equivariant: $\text{Attn}(PH) = P \text{Attn}(H)$ for any permutation P . Without help it cannot tell “dog bites man” from “man bites dog”.

Design goals for a positional scheme:

1. The score between positions t and s should depend on $t - s$ (*relative*), not on t and s separately
2. No extra parameters, no extra memory, negligible FLOPs
3. Should be compatible with caching for every future query.

Formally, we want $f(\mathbf{x}, t)$ with

$$\langle f(\mathbf{q}, t), f(\mathbf{k}, s) \rangle = g(\mathbf{q}, \mathbf{k}, t - s).$$

Additive learned embeddings fail goal 1.



Two dimensions first. Identify $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$ with $z = x_1 + ix_2$ and try a pure rotation, $f(\mathbf{x}, t) = z e^{it\theta}$. Then

$$\langle f(\mathbf{q}, t), f(\mathbf{k}, s) \rangle = \operatorname{Re} \left[q e^{it\theta} \overline{k e^{is\theta}} \right] = \operatorname{Re} \left[q \bar{k} e^{i(t-s)\theta} \right],$$

which depends on t, s only through $t - s$. ✓ In real form, with $\mathcal{R}_t = \begin{pmatrix} \cos t\theta & -\sin t\theta \\ \sin t\theta & \cos t\theta \end{pmatrix}$, orthogonality gives the same statement in one line:

$$(\mathbf{q} \mathcal{R}_t)(\mathbf{k} \mathcal{R}_s)^\top = \mathbf{q} \mathcal{R}_t \mathcal{R}_s^\top \mathbf{k}^\top = \mathbf{q} \mathcal{R}_{t-s} \mathbf{k}^\top.$$

Full head. Split the d_h dimensions into $d_h/2$ planes and rotate each at its own frequency:

$$\mathcal{R}_t = \bigoplus_{j=1}^{d_h/2} \begin{pmatrix} \cos t\theta_j & -\sin t\theta_j \\ \sin t\theta_j & \cos t\theta_j \end{pmatrix}, \quad \theta_j = \beta^{-2(j-1)/d_h}, \quad \beta = 10^4.$$

Low j rotates fast (local detail), high j slowly (long-range).

Cost. Implemented as two elementwise products, $\Theta(n n_h d_h)$ FLOPs



Scoreboard, after Section 1

Per layer; concrete column uses $n_h=128$, $d_h=128$, $l=60$, $d=5120$ (DeepSeek-V2 shape), BF16, $n = 128K$. Quality from the 7B model.

Mechanism	$\mathcal{M}$ (elem/tok/layer)	Cache @128K	Prefill attn FLOPs	$\mathcal{I}_{\text{decode}}$	MMLU
MHA	$2n_h d_h = 32768$	480 GiB	$2n^2 n_h d_h$	1	45.2
GQA-8	$2n_g d_h = 2048$	30 GiB	$2n^2 n_h d_h$	16	41.2
MQA	$2d_h = 256$	3.8 GiB	$2n^2 n_h d_h$	128	37.9
MLA (V2)	?	?	?	?	?
DSA (V3.2)	?	?	?	?	?
CSA/HCA (V4)	?	?	?	?	?



Multi-head Latent Attention

DeepSeek V2



Multi-head Latent Attention

MLA's premise: the K and V of a token are derived from the same d dimensional space and are not $2n_h d_h$ independent numbers. Empirically they are still lower-rank.

So: do not throw away heads. Instead

- ▶ Cache a single low-rank **latent** $\mathbf{c}_t^{KV} \in \mathbb{R}^{d_c}$ from which every head's K and V can be reconstructed

Every head keeps its own W_i^{UK}, W_i^{UV} in training, so heads still see *different* subspaces unlike GQA and MQA.²

²DeepSeek-AI, *DeepSeek-V2*, arXiv:2405.04434, 2024, §2.1.



MLA dataflow

Multi-Head Latent Attention (MLA)

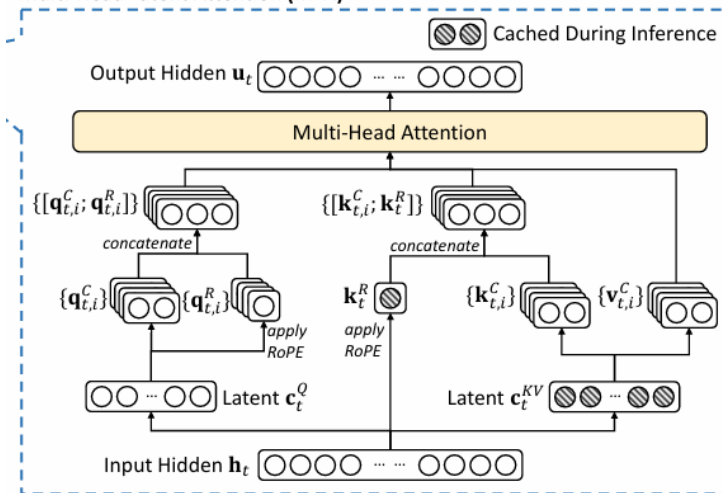


Figure 2: Subpart of DeepseekV2 transformer showing MLA



Low-rank joint KV compression

Keys and values. One down-projection, two per-head up-projections:

$$\mathbf{c}_t^{KV} = \mathbf{h}_t W^{DKV}, \quad W^{DKV} \in \mathbb{R}^{d \times d_c}, \quad d_c \ll n_h d_h$$
$$\mathbf{k}_t^C = \mathbf{c}_t^{KV} W^{UK}, \quad \mathbf{v}_t^C = \mathbf{c}_t^{KV} W^{UV}, \quad W^{UK}, W^{UV} \in \mathbb{R}^{d_c \times n_h d_h}$$

Queries. Compressed too

$$\mathbf{c}_t^Q = \mathbf{h}_t W^{DQ}, \quad \mathbf{q}_t^C = \mathbf{c}_t^Q W^{UQ}, \quad W^{DQ} \in \mathbb{R}^{d \times d'_c}, \quad W^{UQ} \in \mathbb{R}^{d'_c \times n_h d_h}$$

What must be cached? Only $\mathbf{c}_t^{KV}$: d_c elements per token per layer, in place of $2n_h d_h$. For V2, 512 instead of 32768 — a $64\times$ reduction

V2 settings: $d_c = 512 = 4d_h$, $d'_c = 1536$, $n_h = 128$, $d_h = 128$, $d = 5120$, $l = 60$.



The naive version is not actually faster

Suppose at decode we cache $\mathbf{c}_s^{KV}$ and, at each step, reconstruct $\mathbf{k}_s^C = \mathbf{c}_s^{KV} W^{UK}$ and $\mathbf{v}_s^C = \mathbf{c}_s^{KV} W^{UV}$ for all $s \leq n$, then run ordinary attention.

Cost of the reconstruction alone, per decoded token, per layer:

$$2 \cdot 2 n d_c n_h d_h = 4 n d_c n_h d_h \quad \text{vs. MHA's total attention cost } 4 n n_h d_h.$$

That is a factor of $d_c = 512$ *more* arithmetic.

And the reconstructed $\mathbf{k}^C, \mathbf{v}^C$ must be written and read, so we have saved storage but reintroduced the traffic.



Derivation 3 — weight absorption

Write W_i^{UQ} , W_i^{UK} , W_i^{UV} for the slices of the up-projections belonging to head i (each in $\mathbb{R}^{d_c \times d_h}$ or $\mathbb{R}^{d'_c \times d_h}$), and W_i^O for the corresponding block of W^O .

(a) Query side. The content part of the score is

$$\mathbf{q}_{t,i}^C (\mathbf{k}_{s,i}^C)^\top = (\mathbf{c}_t^Q W_i^{UQ}) (\mathbf{c}_s^{KV} W_i^{UK})^\top = \mathbf{c}_t^Q \underbrace{(W_i^{UQ} W_i^{UK\top})}_{\widetilde{W}_i \in \mathbb{R}^{d'_c \times d_c}} (\mathbf{c}_s^{KV})^\top.$$

$\widetilde{W}_i$ is *constant*: fold it into the weights once, at load time. Then $\widetilde{\mathbf{q}}_{t,i} = \mathbf{c}_t^Q \widetilde{W}_i \in \mathbb{R}^{d_c}$ is computed once per token, and the score is a plain dot product against the *cached latent*. $\mathbf{k}^C$ is never formed.



(b) Output side. With attention weights $s_{ts,i}$,

$$\begin{aligned}\mathbf{u}_t &= \sum_i \mathbf{o}_{t,i} W_i^O = \sum_i \left(\sum_s s_{ts,i} \mathbf{c}_s^{KV} W_i^{UV} \right) W_i^O \\ &= \sum_i \underbrace{\left(\sum_s s_{ts,i} \mathbf{c}_s^{KV} \right)}_{\text{in } \mathbb{R}^{d_c}} \underbrace{\left(W_i^{UV} W_i^O \right)}_{\mathbb{R}^{d_c \times d}}.\end{aligned}$$

Associativity of matrix multiplication, twice. That is the whole trick.



What absorption turned MLA into

After absorption, *every head reads the same cached vector*. Structurally this is **MQA with a head dimension of d_c** : one shared KV “head”, n_h query heads.

	MHA	MQA	MLA (absorbed)
distinct KV per token	$2n_h d_h$	$2d_h$	d_c
do heads see different subspaces?	yes	no	yes , via $\widetilde{W}_i$
where the head-specific work lives	in K, V	nowhere	in the <i>query-side</i> matrix

MLA gets a small cache while keeping n_h distinct bilinear forms $\widetilde{W}_i$. The head-specific structure is *moved to the query side*, where it costs computation but not storage.

Except that we have quietly assumed there is no RoPE.



Derivation 4 — why RoPE destroys the trick

Recall from Section 1 that RoPE inserts a position-dependent rotation *between* the query and key projections. Applying it to $\mathbf{k}^C$ as usual:

$$\mathbf{q}_{t,i}^C (\mathbf{k}_{s,i}^C)^\top = \mathbf{c}_t^Q W_i^{UQ} \boxed{\mathcal{R}_{t-s}} W_i^{UK^\top} (\mathbf{c}_s^{KV})^\top.$$

The matrix in the middle now depends on *both* t and s , so

$$\widetilde{W}_i(t, s) = W_i^{UQ} \mathcal{R}_{t-s} W_i^{UK^\top}$$

is a different matrix for every query–key pair. It cannot be precomputed



Decoupled RoPE — the fix

Idea: stop asking one vector to be both compressible and position-carrying.

$$\begin{aligned}\mathbf{q}_t^R &= \text{RoPE}(\mathbf{c}_t^Q W^{QR}), W^{QR} \in \mathbb{R}^{d'_c \times n_h d_h^R} \text{ (per head)} \\ \mathbf{k}_t^R &= \text{RoPE}(\mathbf{h}_t W^{KR}), W^{KR} \in \mathbb{R}^{d \times d_h^R} \text{ (**one**, shared by all heads)} \\ \mathbf{q}_{t,i} &= [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R], \quad \mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R]\end{aligned}$$

The score splits additively — concatenation makes the dot product a sum:

$$\mathbf{q}_{t,i} \mathbf{k}_{s,i}^\top = \underbrace{\mathbf{c}_t^Q \widetilde{W}_i (\mathbf{c}_s^{KV})^\top}_{\text{content: absorbable, no RoPE}} + \underbrace{\mathbf{q}_{t,i}^R (\mathbf{k}_s^R)^\top}_{\text{position: RoPE, tiny, uncompressed}}$$

- ▶ $\mathbf{k}_s^R$ is **shared across heads** and only $d_h^R = 64$ wide
- ▶ **New cache:** $\mathcal{M}_{\text{MLA}} = d_c + d_h^R$ per token per layer.



MLA in full

Boxed quantities are the *only* things stored per token.

$$\begin{aligned} \mathbf{c}_t^Q &= \mathbf{h}_t W^{DQ} & \mathbf{c}_t^{KV} &= \boxed{\mathbf{h}_t W^{DKV}} \\ [\mathbf{q}_{t,1}^C; \dots; \mathbf{q}_{t,n_h}^C] &= \mathbf{c}_t^Q W^{UQ} & [\mathbf{k}_{t,1}^C; \dots; \mathbf{k}_{t,n_h}^C] &= \mathbf{c}_t^{KV} W^{UK} \\ [\mathbf{q}_{t,1}^R; \dots; \mathbf{q}_{t,n_h}^R] &= \text{RoPE}(\mathbf{c}_t^Q W^{QR}) & \mathbf{k}_t^R &= \boxed{\text{RoPE}(\mathbf{h}_t W^{KR})} \\ \mathbf{q}_{t,i} &= [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R] & \mathbf{k}_{t,i} &= [\mathbf{k}_{t,i}^C; \mathbf{k}_{t,i}^R] \\ [\mathbf{v}_{t,1}^C; \dots; \mathbf{v}_{t,n_h}^C] &= \mathbf{c}_t^{KV} W^{UV} \\ \mathbf{o}_{t,i} &= \sum_{s \leq t} \text{Softmax}_s \left(\frac{\mathbf{q}_{t,i} \mathbf{k}_{s,i}^\top}{\sqrt{d_h + d_h^R}} \right) \mathbf{v}_{s,i}, & \mathbf{u}_t &= [\mathbf{o}_{t,1}; \dots; \mathbf{o}_{t,n_h}] W^O. \end{aligned}$$

Two details the paper adds for stability: an **RMSNorm** after each compressed latent ($\mathbf{c}^Q$, $\mathbf{c}^{KV}$), and extra **scaling factors** at the width bottlenecks — low-rank projections change the output scale of the layer. DeepSeek-V2-Lite (16B) drops the *query* compression entirely.



Two execution modes for the same weights

Absorption is a *runtime* choice, not an architectural one. The same checkpoint runs in two algebraically identical ways.

	MHA mode	MQA mode
Used for W^{UK}, W^{UV} $\mathbf{k}^C, \mathbf{v}^C$	training, prefill applied explicitly materialised, n_h heads of width d_h	decoding absorbed into W^{UQ}, W^O never formed
Attention over	$d_h + d_h^R$ per head	$d_c + d_h^R$, one shared entry
Reads per step	the full $n \times n_h d_h$ tensors	only $n(d_c + d_h^R)$

Why not use MQA mode everywhere? Absorption *increases* FLOPs: each head works in $d_c = 512$ dimensions instead of $d_h = 128$.

- ▶ **Prefill** is compute-bound
- ▶ **Decode** is bandwidth-bound at $\mathcal{I} = 1$



The two modes, side by side

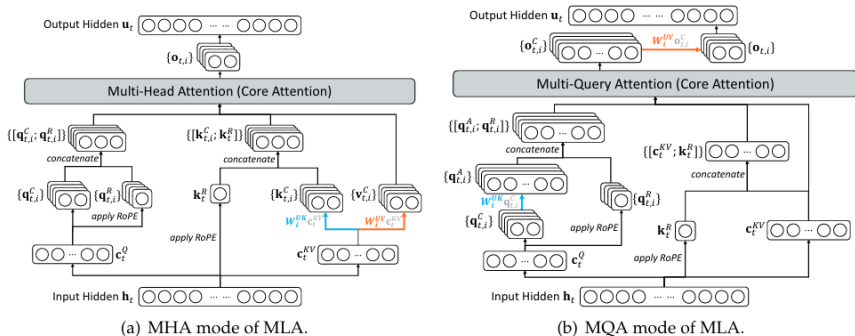


Figure 3: MHA and MQA mode of MLA



Derivation 5 — MLA cache and decode intensity

Cache. Per token per layer,

$$\mathcal{M}_{\text{MLA}} = d_c + d_h^R = 512 + 64 = 576 \quad \text{vs} \quad \mathcal{M}_{\text{MHA}} = 2n_h d_h = 32768.$$

Decode FLOPs (MQA mode), per token per layer: scores run over $d_c + d_h^R$, the value accumulation over d_c :

$$F_{\text{attn}}^{\text{dec}} = 2n n_h (d_c + d_h^R) + 2n n_h d_c = 2n n_h (2d_c + d_h^R).$$

Bytes read: $b n (d_c + d_h^R)$. Hence

$$\mathcal{I}_{\text{MLA}} = \frac{2n n_h (2d_c + d_h^R)}{b (d_c + d_h^R)} \stackrel{\text{v2}}{=} \frac{2 \cdot 128 \cdot 1088}{2 \cdot 576} \approx \boxed{242 \frac{\text{FLOP}}{\text{byte}}}$$

Against a ridge point of ≈ 300



Derivation 6 — what MLA costs in prefill and training

Prefill attention (MHA mode). Scores over $d_h + d_h^R$, values over d_h :

$$F_{\text{attn}} = \underbrace{n^2 n_h (d_h + d_h^R)}_{\mathbf{q}K^\top} + \underbrace{n^2 n_h d_h}_{\mathbf{s}V} = n^2 n_h (2d_h + d_h^R)$$

against MHA's $2n^2 n_h d_h$. Ratio = $\frac{2d_h + d_h^R}{2d_h} = \frac{320}{256} = 1.25$. **MLA pays 25% more prefill attention FLOPs** the price of the decoupled RoPE channel.



Projections, per token per layer (V2 numbers, in MFLOPs):

$W^{DKV} (2dd_c)$	5.2	$W^{UQ} (2d'_c n_h d_h)$	50.3
$W^{DQ} (2dd'_c)$	15.7	$W^{QR} (2d'_c n_h d_h^R)$	25.2
$W^{KR} (2d d_h^R)$	0.7	$W^O (2n_h d_h d)$	167.8
$W^{UK}, W^{UV} (2 \cdot 2d_c n_h d_h)$	33.6	total	298.5

MHA's would be $8d n_h d_h = 671.1$ MFLOPs. **MLA uses 44% of it** — the low-rank bottlenecks are cheaper than dense $d \rightarrow n_h d_h$ projections.

Training. This is what the query compression is for: it saves no cache (queries are never cached) but shrinks the stored activation from $n_h d_h = 16384$ to $d'_c = 1536$ per token per layer



[Appendix] Does it actually work?

Two matched pairs, identical but for the attention mechanism.

Benchmark	Small MoE ($\approx 16B$, 1.33T tok)		Large MoE ($\approx 250B$, 420B tok)	
	MHA	MLA	MHA	MLA
KV cache / token	110.6K	15.6K	860.2K	34.6K
BBH (EM)	37.9	39.0	46.6	50.7
MMLU (Acc)	48.7	50.0	57.5	59.0
C-Eval (Acc)	51.6	50.9	57.9	59.2
CMMLU (Acc)	52.3	53.4	60.7	62.5

- ▶ 14% of the cache at small scale, 4% at large scale — and MLA *wins* on 7 of 8 comparisons.
- ▶ Contrast Section 1: MQA bought $128\times$ less cache for -7.3 MMLU. MLA buys $25\times$ less cache for $+1.5$ MMLU.
- ▶ The advantage *grows* with scale: at $n_h d_h = 860K$ per token, MHA is spending capacity on a cache that MLA shows is not needed.

Source: DeepSeek-V2, Table 9 (Appendix D.2).



Scoreboard, after MLA

Per layer; $n_h=128$, $d_h=128$, $l=60$, $d=5120$, $d_c=512$, $d_h^R=64$; BF16, $n = 128K$.

Mechanism	$\mathcal{M}$ (elem/tok/layer)	Cache @128K	Prefill attn FLOPs	$\mathcal{I}_{\text{decode}}$	MMLU
MHA	$2n_h d_h = 32768$	480 GiB	$2n^2 n_h d_h$	1	45.2
GQA-8	$2n_g d_h = 2048$	30 GiB	(same)	16	41.2
MQA	$2d_h = 256$	3.8 GiB	(same)	128	37.9
MLA (V2)	$d_c + d_h^R = 576$	8.4 GiB	$1.25 \times$ MHA	242	> MHA
DSA (V3.2)	?	?	?	?	?
CSA/HCA (V4)	?	?	?	?	?

MMLU column is not directly comparable.



DeepSeek Sparse Attention

DeepSeek V3.2



The remaining problem

MLA left the scoreboard in an asymmetric state:

- ▶ decode attention at $\mathcal{I} \approx 242$ against a ridge of 300 — effectively done;
- ▶ prefill attention still $\Theta(n^2)$, and 25% worse than MHA.

Every mechanism so far shrinks *what we store per token*. None of them changes *how many tokens each query looks at*.



Some Insights

- ▶ Trained attention distributions are extremely peaked: for a given query, almost all of the softmax mass sits on a small number of keys, and which keys those are varies per query.
- ▶ You cannot know which entries matter without computing the scores, which is the thing you are trying to avoid.



Three constraints any selector must satisfy

Each of DSA's odd-looking choices answers one of these.

1. **The selector must be cheaper than what it saves.** It must be *narrow* (few heads, small dimension) and ideally run in low precision.
2. **Selection must be shared across query heads.** At the kernel level a fetched KV entry has to be reused by many queries, or the gather is bandwidth-bound and the FLOP saving never materialises.
3. **Top- k is not differentiable.** The gradient of the main loss cannot teach the selector what to select. It needs its own training signal.

Constraint 2 is why DSA is built on **MLA in MQA mode**: the cached entry is *already* one vector shared by all query heads, so “select entries for this token” is the natural granularity.



The lightning indexer

A deliberately impoverished attention-like scorer. For query t and preceding token s :

$$l_{t,s} = \sum_{j=1}^{n_h^l} w_{t,j}^l \cdot \text{ReLU}(\mathbf{q}_{t,j}^l \cdot \mathbf{k}_s^l),$$

with $\mathbf{q}_{t,j}^l \in \mathbb{R}^{d_h^l}$ and scalars $w_{t,j}^l$ from $\mathbf{h}_t$, and $\mathbf{k}_s^l \in \mathbb{R}^{d_h^l}$ from $\mathbf{h}_s$.

Four choices, each answering constraint 1:

- ▶ **One key per token**, shared by all n_h^l indexer heads to save cache
- ▶ ReLU, **not softmax**. $l_{t,s}$ only has to *rank*; it is never a probability.
- ▶ **FP8** since the score feeds into an argsort
- ▶ **Learned per-head gates** $w_{t,j}^l$: cheap expressivity, letting the query choose which indexer subspace to trust.



Fine-grained token selection

Select Top-K and run exact attention on what is left:

$$\mathcal{S}_t = \{ s \mid l_{t,s} \in \text{Top-}k(l_{t,:}) \}, \quad \mathbf{u}_t = \text{Attn}(\mathbf{h}_t, \{\mathbf{c}_s\}_{s \in \mathcal{S}_t}).$$

- ▶ $\mathbf{c}_s = [\mathbf{c}_s^{KV}; \mathbf{k}_s^R]$ is exactly MLA's cached latent. **DSA changes nothing about what is stored** it only decides which past tokens to consider.
- ▶ Selection is *fine-grained*: per token, not per block. $\mathcal{S}_t$ is an arbitrary subset, different for every query.
- ▶ It is *per query token*, shared across all n_h query heads
- ▶ Core attention over $\mathcal{S}_t$ is ordinary MLA attention in MQA mode.

V3.2 setting: $k = 2048$. At 128K context each query sees 1.6% of the sequence; at 1M, 0.2%.



DSA under MLA

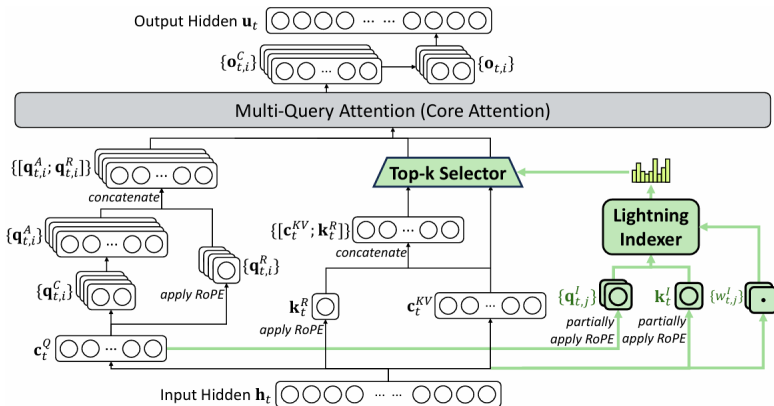


Figure 4: Additions because of DSA over the normal MLA flow



Derivation 7 — prefill FLOPs

Per layer. Two terms now, with different orders.

Indexer — still all-pairs, so still quadratic, but narrow:

$$F_{\text{idx}} = 2 \cdot \frac{n^2}{2} \cdot n_h^l d_h^l = n^2 n_h^l d_h^l$$

Core attention — each of n queries attends to exactly k entries, in MQA mode:

$$F_{\text{core}} = 2n k n_h (2d_c + d_h^R) \quad \text{— linear in } n$$



With V3.2's shape ($n_h=128$, $d_c=512$, $d_h^R=64$, $k=2048$; $n_h^I=64$, $d_h^I=128$):

per layer	dense MLA	DSA indexer	DSA core
formula	$n^2 n_h (2d_h + d_h^R)$	$n^2 n_h^I d_h^I$	$2nkn_h(2d_c + d_h^R)$
coefficient	$4.10 \times 10^4 n^2$	$8.19 \times 10^3 n^2$	$5.70 \times 10^8 n$
at $n = 128K$	$5.37 \times 10^9 n$	$1.07 \times 10^9 n$	$0.57 \times 10^9 n$

Speedup = $5.37 / (1.07 + 0.57) = 3.3\times$ at 128K, and **4.7** $\times$ at 1M.

But note where the cost now sits. The two DSA terms cross at $n^2 n_h^I d_h^I = 2nkn_h(2d_c + d_h^R)$, i.e. $n \approx 70K$. Beyond that the *indexer* is the dominant cost, and it is still $\Theta(n^2)$.



Derivation 8 — decode: the core becomes constant

Per generated token, per layer, with n tokens cached:

$$F_{\text{idx}}^{\text{dec}} = 2n n_h^l d_h^l (\Theta(n)) \quad F_{\text{core}}^{\text{dec}} = 2n_h k (2d_c + d_h^R) \Theta(1)$$

Core attention cost stops growing with context once $n > k$,

Bytes moved — the number that actually matters at decode:

	formula	at $n = 128\text{K}$, per layer
MLA (dense)	$b(d_c + d_h^R) n$	151.0 MB
DSA: indexer scan	$b_l d_h^l n$ (FP8, $b_l=1$)	16.8 MB
DSA: gathered latents	$b(d_c + d_h^R) k \Theta(1)$	2.4 MB

$151.0/19.2 = 7.9\times$ less traffic per decoded token.

Intensity. Core attention keeps MLA's $\mathcal{I} \approx 242$. The indexer scan runs at $\mathcal{I} = 2n_h^l d_h^l / (b_l d_h^l) = 2n_h^l = 128$



Derivation 9 — DSA makes the cache *bigger*

The indexer keys must be cached too. They are functions of $\mathbf{h}_s$, so recomputing them each step would defeat the purpose.

$$\mathcal{M}_{\text{DSA}} = \underbrace{d_c + d_h^R}_{576, \text{ MLA latent}} + \underbrace{d_h^I}_{128, \text{ indexer key}} = 704 \text{ elements/token/layer}$$

The trade that is being made:

- ▶ More storage, and $+\Theta(n)$ arithmetic that MLA did not have;
- ▶ in exchange, core attention drops from $\Theta(n)$ to $\Theta(k)$ per decoded token, and from $\Theta(n^2)$ to $\Theta(nk)$ in prefill.

It is a good trade only because $k \ll n$.



[Appendix] Checking the derivation against the paper

The V3.2 paper reports *measured* serving cost on H800s at \$2/GPU-hour (their Figure 3), which we can compare against the two numbers we just derived.

at 128K context	our derivation	paper's measurement
Prefill speedup	$3.3\times$	$\approx 3.5\times$
Decode speedup	$7.9\times$	$\approx 8.5\times$

Close enough to trust the model, and the small shortfall is in the direction we would expect — our count ignores kernel-level wins (fused FP8 indexer, better occupancy on the gathered attention) and ignores the projection FLOPs, which are unchanged and therefore dilute both ratios in our favour.

One implementation note. For *short*-sequence prefill, where k is a large fraction of n and the gather is pure overhead, V3.2 falls back to a masked MHA-mode kernel that simulates DSA's output exactly. Same mathematics, different kernel, chosen by sequence length — the same roofline reasoning as MLA's two modes.



Training, part 1: the dense warm-up

Top- k blocks the gradient, so the indexer cannot learn from the language-modelling loss. It is instead trained to **imitate true attention**.

Stage 1 — dense warm-up. Start from DeepSeek-V3.1-Terminus (already extended to 128K). Freeze *everything* except the indexer; keep attention fully dense.

- ▶ Build the target: sum the main attention scores across all n_h heads, then L_1 -normalise along the sequence dimension to get $p_{t,:} \in \mathbb{R}^t$.
- ▶ Train the indexer to match it:

$$\mathcal{L}^I = \sum_t \mathbb{D}_{\text{KL}}(p_{t,:} \parallel \text{Softmax}(I_{t,:})).$$

Cost: $\text{lr } 10^{-3}$, 1000 steps, 16 sequences of 128K = **2.1B tokens**. Trivial next to pre-training.



Training, part 2: the sparse stage

Stage 2. Switch on top- k selection and unfreeze everything, so the main model can adapt to being fed a sparse view.

The KL target is now restricted to the tokens that were actually selected, $\mathcal{S}_t = \{s \mid l_{t,s} \in \text{Top-}k(l_{t,:})\}$:

$$\mathcal{L}^I = \sum_t \mathbb{D}_{\text{KL}}(p_{t,\mathcal{S}_t} \parallel \text{Softmax}(l_{t,\mathcal{S}_t})).$$

The two loss paths are kept strictly separate:

- ▶ the indexer is trained by $\mathcal{L}^I$ *only*;
- ▶ the main model is trained by the language-modelling loss *only*.

Cost: $\text{lr } 7.3 \times 10^{-6}$, $k = 2048$, 15,000 steps of 480 sequences of 128K = **943.7B tokens** — a continued-pre-training run, not a fine-tune, but still a small fraction of the original budget.



Does sparsifying hurt? — the parity evidence

V3.2-Exp differs from V3.1-Terminus by DSA and its continued pre-training

- ▶ **Standard benchmarks:** no substantial degradation, short- or long-context.
- ▶ **Human preference:** ChatBot Arena Elo closely matched (10 Nov 2025).
- ▶ **Held-out long context:** on AA-LCR, V3.2-Exp scores *four points higher* in reasoning mode; on Fiction.liveBench it consistently beats Terminus. Both were unseen test sets released after the model.



Scoreboard, after Section 3

Rows 1–4 at V2's shape ($l=60$); DSA at V3.2's ($l=61$, $d=7168$, $k=2048$, $n_h^l=64$, $d_h^l=128$). Prefill column is relative to dense MLA at $n = 128K$.

Mechanism	$\mathcal{M}$ (elem/tok/layer)	Cache @128K	Prefill attn	$\mathcal{I}_{\text{decode}}$	Quality
MHA	$2n_h d_h = 32768$	480 GiB	$\Theta(n^2)$	1	45.2 MMLU
GQA-8	$2n_g d_h = 2048$	30 GiB	(same)	16	41.2
MQA	$2d_h = 256$	3.8 GiB	(same)	128	37.9
MLA (V2)	$d_c + d_h^R = 576$	8.4 GiB	$1.25\times$ MHA	242	$>$ MHA
DSA (V3.2)	$d_c + d_h^R + d_h^l = 704$	9.5 GiB	$0.31\times$ MLA	142	$\approx$ MLA
CSA/HCA (V4)	?	?	?	?	?

The quadratic term finally moved. But:

- ▶ the cache went *up*
- ▶ decode core attention is now $\Theta(1)$, so what remains is the $\Theta(n)$ indexer scan and the $\Theta(n^2)$ indexer in prefill.
- ▶ **At 1M tokens the indexer is the bottleneck.**



Compressed Sparse Attention & Heavily Compressed Attention DeepSeek V4



The bottleneck DSA left behind

At 128K, DSA's indexer and core attention were comparable. At 1M they are not:

$$\underbrace{n^2 n_h^l d_h^l}_{\text{indexer, } \Theta(n^2)} \quad \text{vs} \quad \underbrace{2nk n_h(2d_c + d_h^R)}_{\text{core, } \Theta(n)} \quad \implies \quad \text{crossover at } n \approx 70\text{K}.$$

At 1M tokens the indexer is 94% of DSA's attention FLOPs. **We have made the quadratic term cheap, not absent.**



- ▶ Every mechanism so far has been an operation *on* the sequence.
- ▶ V4 changes the sequence itself: compress m consecutive tokens into one KV entry, so that everything downstream sees a sequence of length n/m .
- ▶ Quadratic terms fall by m , the cache falls by m , and DSA still applies *on top*, on the compressed sequence.



Two mechanisms, interleaved

V4 does not pick one compression rate. It builds two attention layers with very different operating points and alternates them.

	CSA (Compressed Sparse)	HCA (Heavily Compressed)
Compression rate	$m = 4$	$m' = 128 (\gg m)$
Compression style	overlapped, dual-stream	non-overlapped, single stream
Then what?	DSA: top- k selection	nothing — dense over all entries
Attends to	k entries ($k=1024$)	all n/m' entries
Cache/token/layer	$(d_h + d_h^l)/m = 160$	$d_h/m' = 4$
Role	fine-grained retrieval	cheap global context

V4-Pro: 61 layers, the first two HCA, the rest alternating. V4-Flash: 43 layers, the first two pure sliding-window, the rest alternating.



CSA architecture

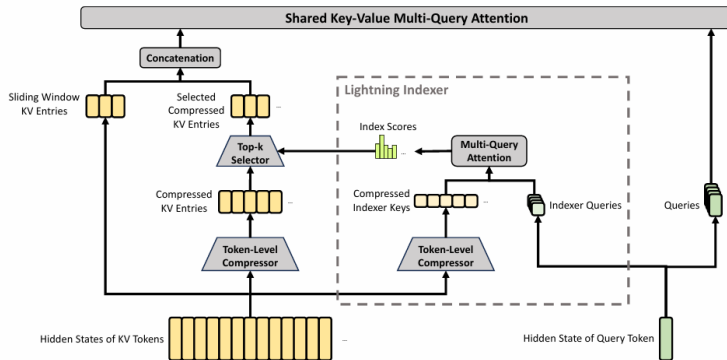


Figure 5: CSA Architecture on top of DSA and MLA



CSA: building a compressed KV entry

Two parallel KV streams and two weight streams from the same hidden states ($d_h = c = 512$):

$$C^a = HW^{aKV}, \quad C^b = HW^{bKV}, \quad Z^a = HW^{aZ}, \quad Z^b = HW^{bZ} \in \mathbb{R}^{n \times d_h}.$$

Compressed entry i pools stream a over block i and stream b over block $i - 1$, with learnable positional biases $B^a, B^b \in \mathbb{R}^{m \times d_h}$:

$$\begin{aligned} [S_{mi:m(i+1)-1}^a; S_{m(i-1):mi-1}^b] &= \text{Softmax}([Z_{mi:m(i+1)-1}^a + B^a; Z_{m(i-1):mi-1}^b + B^b]) \\ C_i^{\text{Comp}} &= \sum_{j=mi}^{m(i+1)-1} S_j^a \odot C_j^a + \sum_{j=m(i-1)}^{mi-1} S_j^b \odot C_j^b \end{aligned}$$

- ▶ Softmax normalises over all $2m$ contributing positions, **per channel**
- ▶ C_i^{Comp} draws on $2m$ tokens, but there are n/m entries: **compression rate** $1/m$, with consecutive entries overlapping.



Why it is built that way

- 1. Learned, content-dependent pooling, not mean-pooling.** The weights S come from $Z = HW^Z$, so *the token decides how much of itself survives compression*.
- 2. Learnable positional biases B^a, B^b .** The pooling is a sum, hence permutation-invariant, so position *within* a block is otherwise unrecoverable.
- 3. The overlapped dual stream.** A hard block boundary is a hard information cut: tokens at $mi - 1$ and mi land in different summaries and can never be compared. The b -stream shifts the window by one block, so every entry summarises a $2m$ -token span and consecutive spans overlap.



CSA: sparsity on top of compression

Having shortened the sequence to n/m , apply DSA to it unchanged.

Indexer keys come from the *same* compression operator, giving

$K^{I\text{Comp}} \in \mathbb{R}^{(n/m) \times d_h^I}$; queries come off the shared latent:

$$\mathbf{c}_t^Q = \mathbf{h}_t W^{DQ}, [\mathbf{q}_{t,1}^I; \dots; \mathbf{q}_{t,n_h^I}^I] = \mathbf{c}_t^Q W^{IUQ}, \quad \mathbf{w}_t^I = \mathbf{h}_t W^w$$

$$l_{t,s} = \sum_{h=1}^{n_h^I} w_{t,h}^I \text{ReLU}(\mathbf{q}_{t,h}^I \cdot K_s^{I\text{Comp}}), \quad s < \lfloor t/m \rfloor$$

and $\mathcal{C}_t^{\text{SprsComp}} = \{C_s^{\text{Comp}} \mid l_{t,s} \in \text{Top-}k(l_{t,:})\}$.

- ▶ The indexer runs over n/m candidates, so its quadratic term drops by m directly
- ▶ $\mathbf{c}_t^Q$ is **shared** by the indexer and attention queries
- ▶ $s < \lfloor t/m \rfloor$ enforces strict causality: a query may not see its *own* block, which also summarises tokens after t . That blind spot is what the sliding window repairs.



[Appendix] Shared key-value MQA, and the grouped output projection

Shared KV. Each selected C_s^{Comp} serves as *both* key and value:

$$\mathbf{q}_t = \mathbf{c}_t^Q W^{UQ}, \quad \mathbf{o}_{t,i} = \text{CoreAttn}(\mathbf{q}_{t,i}, \mathbf{c}_t^{\text{SprsComp}}, \mathbf{c}_t^{\text{SprsComp}}).$$

This halves what compression must produce and store, and makes the entry a single MQA-style vector shared by all n_h query heads — the kernel constraint from Section 3, now built into the architecture rather than inherited from MLA.

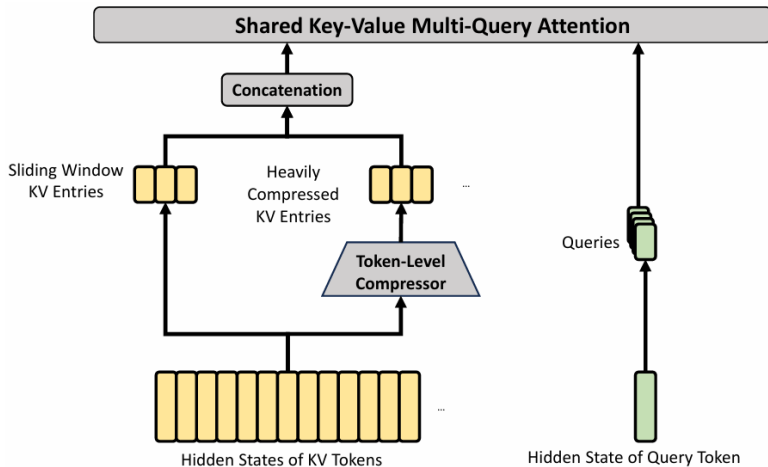
Grouped output projection. With $d_h = 512$ and $n_h = 128$, the concatenated output is $d_h n_h = 65536$ wide, and projecting that to d dominates the layer. So split the heads into g groups, project each to d_g , then project the concatenation to d :

$$\underbrace{2 d_h n_h d}_{\text{naive}} = 9.4 \times 10^8 \longrightarrow \underbrace{2 d_h n_h d_g + 2 d_g g d}_{\text{grouped}} = 3.7 \times 10^8 \quad (2.5 \times \text{cheaper})$$

for V4-Pro ($g=16$, $d_g=1024$, $d=7168$). A low-rank factorisation of W^O in the same spirit as MLA's bottlenecks.



HCA Architecture



HCA: compression without selection

One stream, no overlap, far larger rate $m' = 128$:

$$C = HW^{KV}, \quad Z = HW^Z, \quad S_{m'i:m'(i+1)-1} = \text{Softmax}_{\text{row}}(Z_{m'i:m'(i+1)-1} + B),$$

$$C_i^{\text{Comp}} = \sum_{j=m'i}^{m'(i+1)-1} S_j \odot C_j \quad \implies \quad \frac{n}{m'} \text{ entries.}$$

Then **dense** attention over all of them

Why no indexer? Nothing is left to select from: at 1M tokens HCA sees only 8192 entries

Complexity is still $\Theta(n^2/m')$ — quadratic, with the constant divided by 128: a small quadratic term is acceptable if the coefficient is small enough.



[Appendix] Detail 1: partial RoPE, and undoing it on the output

Shared KV creates a positional problem that does not arise anywhere else in this talk.

RoPE is applied *partially* — to the last 64 dimensions of each query and each KV entry. Since the entry is also the *value*, the attention output inherits it:

$$\mathbf{o}_{t,i} = \sum_s s_{ts,i} (\mathbf{c}_s \mathcal{R}_s)$$

which carries **absolute** positions s — exactly what RoPE exists to avoid.

The fix is one line: apply RoPE at position $-t$ to the last 64 dimensions of the output. Then

$$\mathbf{o}_{t,i} \mathcal{R}_{-t} = \sum_s s_{ts,i} \mathbf{c}_s \mathcal{R}_s \mathcal{R}_{-t} = \sum_s s_{ts,i} \mathbf{c}_s \mathcal{R}_{s-t},$$

so each entry's *contribution to the output* is modulated by its distance from the query, not by where it sits in the document. Relativity is restored on the value path as well as the score path.



[Appendix] Details 2–4: window, sink, normalisation

Sliding-window branch. $n_{\text{win}} = 128$ *uncompressed* KV entries for the most recent tokens, concatenated with the compressed ones. It repairs the causal blind spot (strict causality stopped a query seeing its own block, i.e. its immediate predecessors), and local dependencies are both what compression damages most and what language modelling needs most sharply. Cost $\Theta(n_{\text{win}})$ per token, constant in n .

Attention sink. Learnable per-head logits z'_h added to the denominator only:

$$s_{h,i,j} = \frac{\exp(z_{h,i,j})}{\sum_k \exp(z_{h,i,k}) + \exp(z'_h)}.$$

Total attention mass may now be less than 1, even near 0. This matters *because* of compression: when no compressed entry holds anything relevant, forcing a probability distribution would inject an average of irrelevant summaries. The sink lets a head abstain.

RMSNorm on each query head and on the single KV-entry head, immediately before core attention — prevents exploding logits, and is why V4 needs no QK-Clip in its Muon setup.



[Appendix] Configurations

	DeepSeek-V4-Flash	DeepSeek-V4-Pro
Total / activated params	284B / 13B	1.6T / 49B
Layers l , hidden d	43, 4096	61, 7168
First layers	$2 \times$ pure sliding window	$2 \times$ HCA
Remainder	CSA / HCA interleaved	CSA / HCA interleaved
CSA rate m	4	4
CSA top- k	512	1024
HCA rate m'	128	128
Indexer n_h^l, d_h^l	64, 128	64, 128
Query heads n_h	64	128
Head dim $d_h (= c)$	512	512
Query latent d_c	1024	1536
Output groups g, d_g	8, 1024	16, 1024
Window n_{win}	128	128
Context	1M	1M
Pre-training tokens	32T	33T

Note $d_h = 512$ — four times the usual 128. Compression makes wide heads affordable: there are only n/m of them.



Derivation 10 — the cache

Compressed entries are stored, not tokens. Per token per layer (V4-Pro):

$$\mathcal{M}_{\text{CSA}} = \frac{d_h + d_h^l}{m} = \frac{512 + 128}{4} = 160, \quad \mathcal{M}_{\text{HCA}} = \frac{d_h}{m'} = \frac{512}{128} = 4.$$

(The sliding window costs $\Theta(n_{\text{win}})$ total, not per token — negligible at long context.)

Averaging over 30 CSA and 31 HCA layers:

$$\mathcal{M}_{\text{V4-Pro}} \approx \frac{30(160) + 31(4)}{61} \approx 81 \quad \text{per token per layer, vs DSA's 704.}$$

Precision, on top of the structural saving. KV entries are stored mixed: BF16 for the 64 RoPE dimensions, FP8 for the other 448



Derivation 11 — prefill FLOPs

Per layer, dropping constants in n . Compression divides the candidate sequence by m :

CSA indexer: $\frac{n^2 n_h^l d_h^l}{m} = 2.05 \times 10^3 n^2$

CSA core: $4n(k + n_{\text{win}}) n_h d_h = 3.02 \times 10^8 n$

HCA: $\frac{2n^2 n_h d_h}{m'} = 1.02 \times 10^3 n^2$

Averaged over the interleave: $\approx 1.54 \times 10^3 n^2 + 1.7 \times 10^8 n$ per layer, against DSA's $8.19 \times 10^3 n^2 + 5.70 \times 10^8 n$.

per layer, coefficient of n	DSA (V3.2)	V4-Pro
at $n = 128\text{K}$	1.64×10^9	3.7×10^8
at $n = 1\text{M}$	8.76×10^9	1.70×10^9



Derivation 12 — decode traffic

The number that decides tokens/second. Per decoded token, per layer, at $n = 1\text{M}$:

	bytes read	at 1M
DSA: indexer scan (FP8)	$b_l d_h^l n = 128n$	128.0 MB
DSA: gathered latents	$b(d_c + d_h^R)k$, constant	2.4 MB
CSA: indexer scan (FP4)	$b_l d_h^l n/m = 16n$	16.0 MB
CSA: gathered entries	$\approx 1.125 d_h(k + n_{\text{win}})$, constant	0.7 MB
HCA: full scan	$1.125 d_h n/m' = 4.5n$	4.5 MB
V4-Pro, layer average	$\approx 10.3n + \text{const}$	10.6 MB

Two compounding effects, worth separating:

- ▶ **Structural:** the scan is over n/m or n/m' entries, not n .
- ▶ **Numerical:** the indexer runs in FP4 rather than FP8, halving the scan again. Legitimate because the score is only ever used to sort.

Note what has *not* gone away: the $\Theta(n)$ scan. Decode is still linear in context.



[Appendix] Why interleave rather than choose?

The hybrid is the least-discussed and most consequential choice in the paper, so it is worth pricing directly. Per layer, at 1M:

	all-CSA	all-HCA	interleaved (actual)
Cache / token / layer	160	4	81
Prefill coefficient (n^2)	2.05×10^3	1.02×10^3	1.54×10^3
Decode scan bytes / token	$16n$	$4.5n$	$10.3n$
Token-level retrieval?	every layer	never	every other layer

The argument: an all-HCA model is cheaper on every axis, but at $m' = 128$ no layer can ever address an individual token — a needle-in-a-haystack retrieval has no mechanism to succeed. An all-CSA model retains that ability everywhere and pays $40\times$ the cache for it. Interleaving buys token-level addressing in half the layers at roughly half the cost, on the hypothesis that once *some* layer has retrieved the needle, the residual stream carries it and later layers need only coarse context.

This hypothesis is not tested in the paper. There is no ablation on the interleave ratio, no all-CSA or all-HCA baseline. The reported long-context results are consistent with it, but do not isolate it.



Results, and how much they can tell us

Efficiency (Figure 1, right), at 1M context vs DeepSeek-V3.2:

	single-token FLOPs	KV cache
DeepSeek-V4-Pro (49B active)	27%	10%
DeepSeek-V4-Flash (13B active)	10%	7%

Quality, in the paper's own framing:

- ▶ *Long-context*: V4-Pro-Max surpasses even Gemini-3.1-Pro on academic 1M-token benchmarks
- ▶ *Reasoning*: beats GPT-5.2 and Gemini-3.0-Pro, short of GPT-5.4 and Gemini-3.1-Pro
- ▶ *Knowledge*: behind Gemini-3.1-Pro, attributed to total training FLOPs.



Scoreboard, complete

Per layer. Rows 1–4 at V2's shape, DSA at V3.2's, V4 at V4-Pro's. Prefill column relative to the row above it.

Mechanism	$\mathcal{M}$ (elem/tok/layer)	Cache @128K	Prefill attn	$\mathcal{I}_{\text{decode}}$	Quality
MHA	$2n_h d_h = 32768$	480 GiB	$\Theta(n^2)$	1	45.2 MMLU
GQA-8	$2n_g d_h = 2048$	30 GiB	(same)	16	41.2
MQA	$2d_h = 256$	3.8 GiB	(same)	128	37.9
MLA (V2)	$d_c + d_h^R = 576$	8.4 GiB	$1.25\times$ MHA	242	$>$ MHA
DSA (V3.2)	$d_c + d_h^R + d_h^I = 704$	9.5 GiB	$0.31\times$ MLA	142	parity
V4-Pro	≈ 81 (CSA 160 / HCA 4)	1.1 GiB	$0.23\times$ DSA	—	parity*

* no isolating ablation exists; see previous slide



What the four generations actually did

	lever	phase	what it cost
GQA / MQA	delete KV heads	decode	quality (-7.3 MMLU)
MLA (V2)	low-rank joint compression + absorption	decode	+25% prefill FLOPs; a bolted-on RoPE chan- nel
DSA (V3.2)	learned top- k se- lection	prefill + decode	+11% cache; a second training stage
CSA/HCA (V4)	compress the se- quence; hybridise	prefill + decode + cache	lossy by construction; many moving parts

