Sparse Attentions and KV Cache Optimizations

Yuxiang Huang

huang-yx21@mails.tsinghua.edu.cn

04/28/2026

Recap: Attention Mechanism

- Hidden states, Attention weights:

$$\boldsymbol{H} \in \mathbb{R}^{n \times d}, \quad \boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^O \in \mathbb{R}^{d \times d}$$

- Linear Projection:

$$\boldsymbol{Q} = \boldsymbol{H}\boldsymbol{W}^Q, \quad \boldsymbol{K} = \boldsymbol{H}\boldsymbol{W}^K, \quad \boldsymbol{V} = \boldsymbol{H}\boldsymbol{W}^V$$

- Scale Dot-Product Attention:

$$\boldsymbol{S} = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}, \quad \boldsymbol{P} = \mathrm{softmax}(\boldsymbol{S} + \boldsymbol{M}), \quad \boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$$

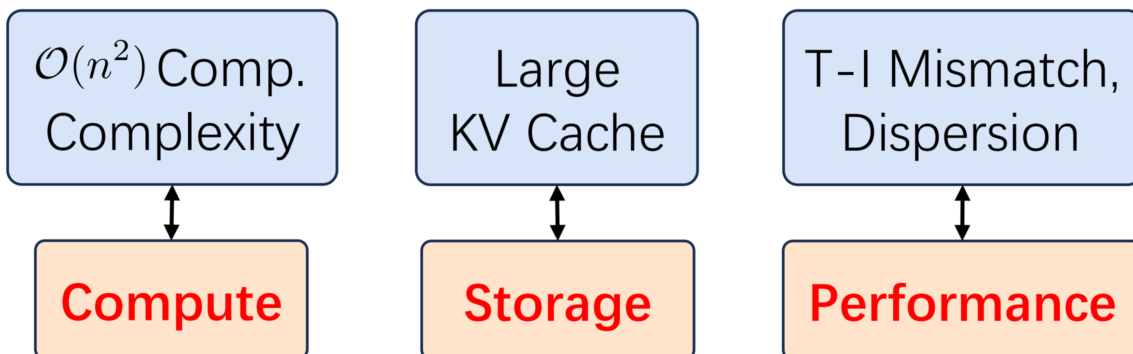
- Attention masks: $\boldsymbol{M} \in \{0, -\infty\}^{n \times n}$
    - Encoder: $m_{ij} = 0, \ \forall (i, j)$ Decoder: $m_{ij} = 0, \ i \geq j; \quad m_{ij} = -\infty, \ i < j$

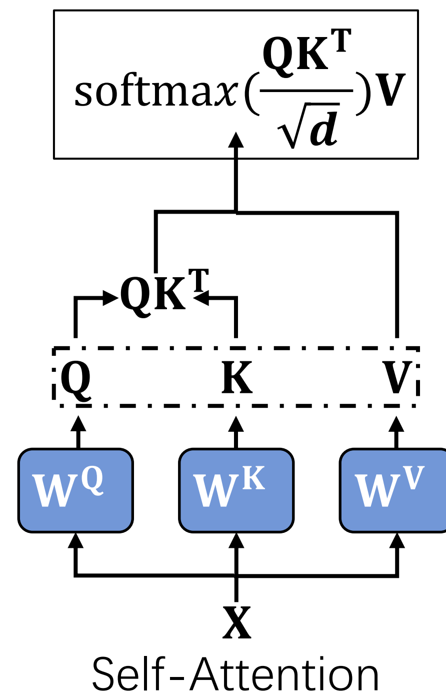
On Scaling Context Length: HW-SW Co-Design

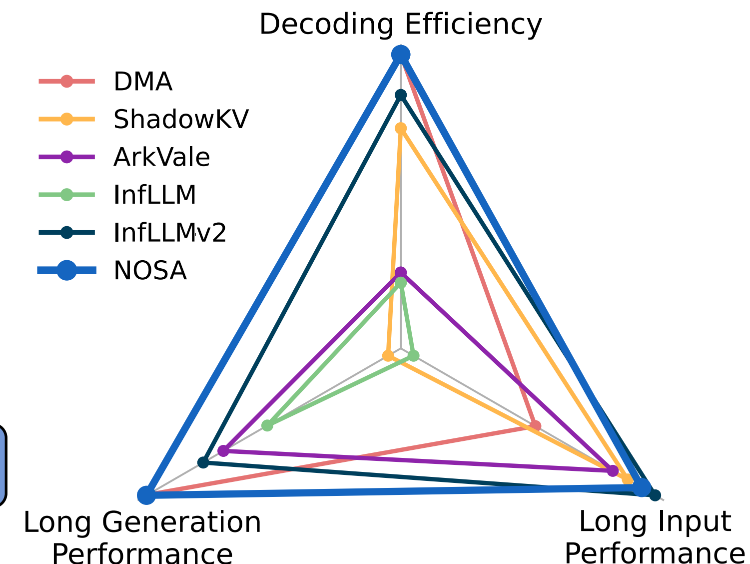
- **Self-Attention is the major bottleneck**

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

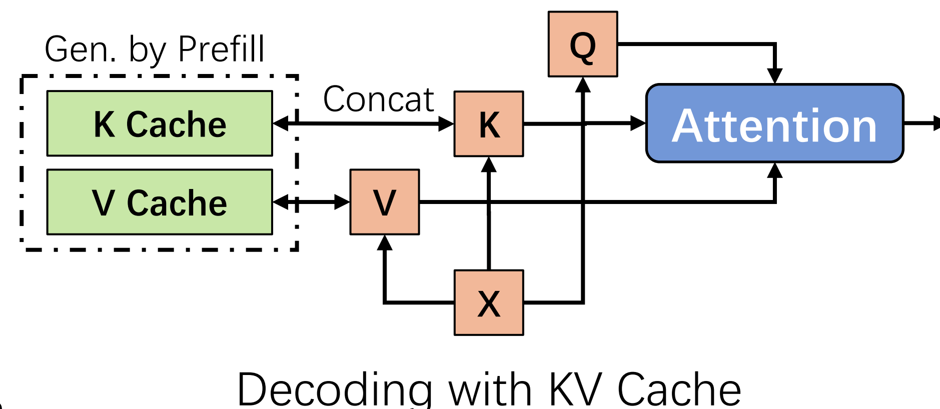
- When scaling the context length n







- [HW] Limited Resources: Compute, Memory, Storage
  - Compute & Mem: Sparse Attentions
  - Storage: KV cache optimizations
- [SW] **Resolving performance degradation**
  - Optimization-aware Training to alleviate T-I mismatch



Alleviating Compute & Mem. Access Burden

Training–Free Sparse Attention Mechanisms

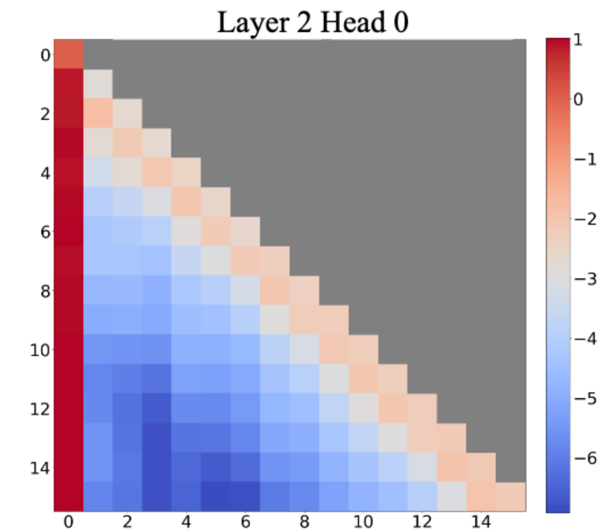
Static Pattern

Dynamic Pattern

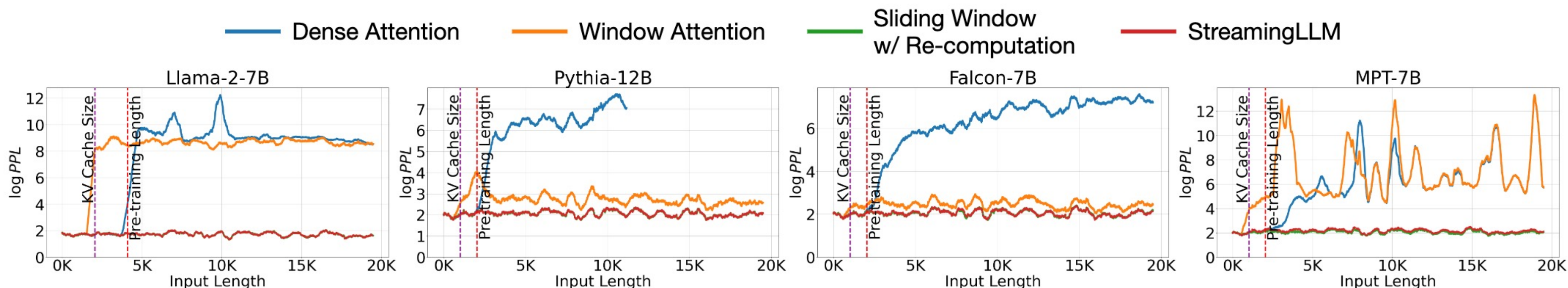
Ensemble Pattern

Static Sparse Patterns: Sink, SWA

- **Hand-crafted by certain observations**
- **StreamingLLM: attention sink + sliding window**
  - Observation: Initial tokens draw huge attention
  - Attention sink: the first (4-128) tokens
  - Sliding window: under Markov Assumption, local effect
  - How to do sparse attention: simply discarding everything except sink + sliding window
  - Complexity reduction: $\mathcal{O}(n^2) \rightarrow \mathcal{O}(|\text{sink} + \text{window}|n) = \mathcal{O}(n)$
- **Can do language modeling & context extrapolation! But how to retrieve from the middle?**



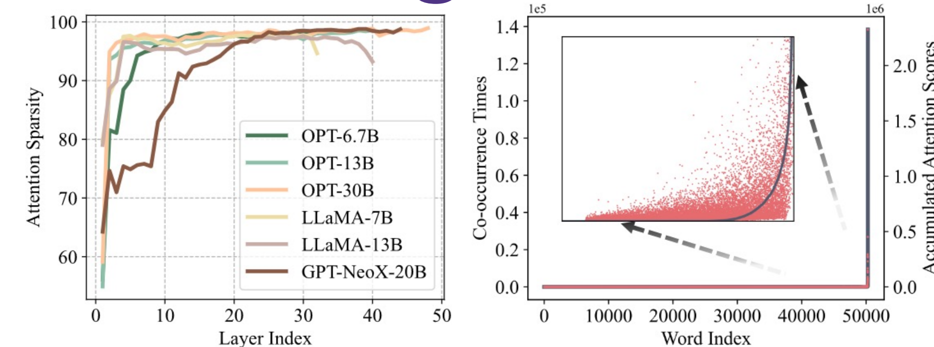
Attn Sink + SWA



Efficient Streaming Language Models with Attention Sinks (ICLR 2024)

Dynamic Sparse Patterns: Token Voting

- **Attention is naturally sparse**
- Only few tokens receives large softmax probability
- Retain these tokens and discard others

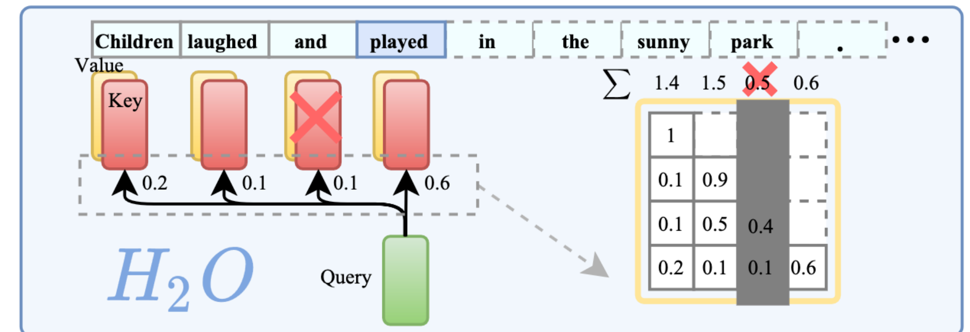


Attention Sparsity

- **H2O: Voting by aggregated probabilities from subsequent tokens**
  - First, prefill by SDPA $\boldsymbol{S} = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}, \quad \boldsymbol{P} = \mathrm{softmax}(\boldsymbol{S} + \boldsymbol{M}), \quad \boldsymbol{O} = \boldsymbol{P}\boldsymbol{V}$
  - Then, vote tokens and identify **Heavy Hitters**

$$\boldsymbol{h} \in \mathbb{R}^n, \quad h_j = \sum_{i=j}^{n} p_{ij}, \quad \mathrm{HH}_k = \{j \mid j \in \text{top-}k(\boldsymbol{h})\}$$

  - Discard other tokens' KV cache and start decoding



- **Faster Decode, Can do retrieval tasks!**
- Need to materialize $\boldsymbol{P} \in \mathbb{R}^{n \times n}$, **not compatible with Flash Attention** → Slow Prefill, Large Mem Util.

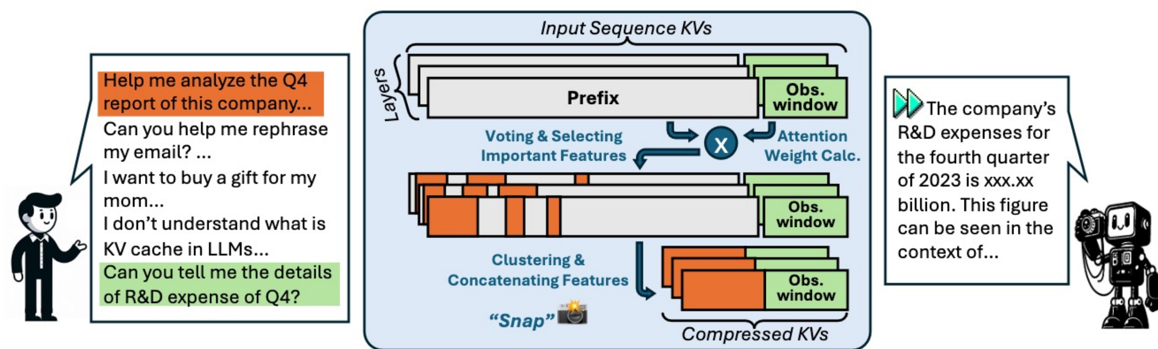
H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models (NeurIPS 2023)

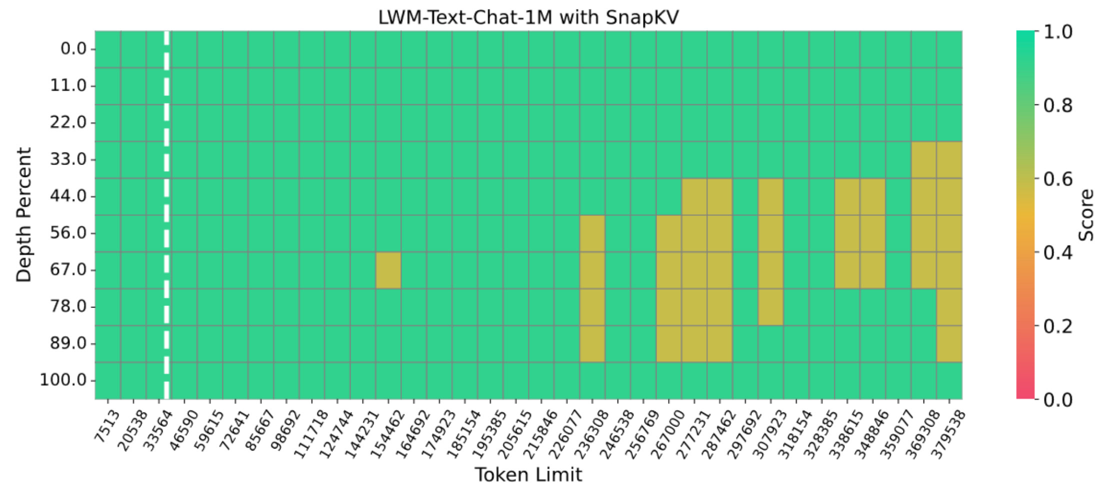
Dynamic Sparse Patterns: Voting From Query

- **Attn-score based voting is generally good. Just find a FA-compatible way.**
- **SnapKV: only use query tokens to vote**
  - Query tokens are at the last, and generally few
  - Document prefill can still use Flash Attention
  - Vote tokens by queries, smoothed by a pooling

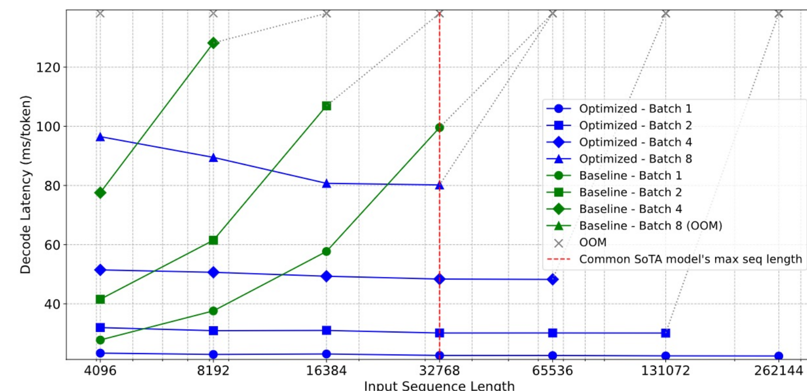
$$\boldsymbol{h}' \in \mathbb{R}^n, \quad h'_j = \sum_{i=n-W}^{n} p_{ij},$$

$$\boldsymbol{h} = \text{pooling}(\boldsymbol{h}'), \quad \text{HH}_k = \{j \mid j \in \text{top-}k(\boldsymbol{h})\}$$





Excellent Retrieval Performance

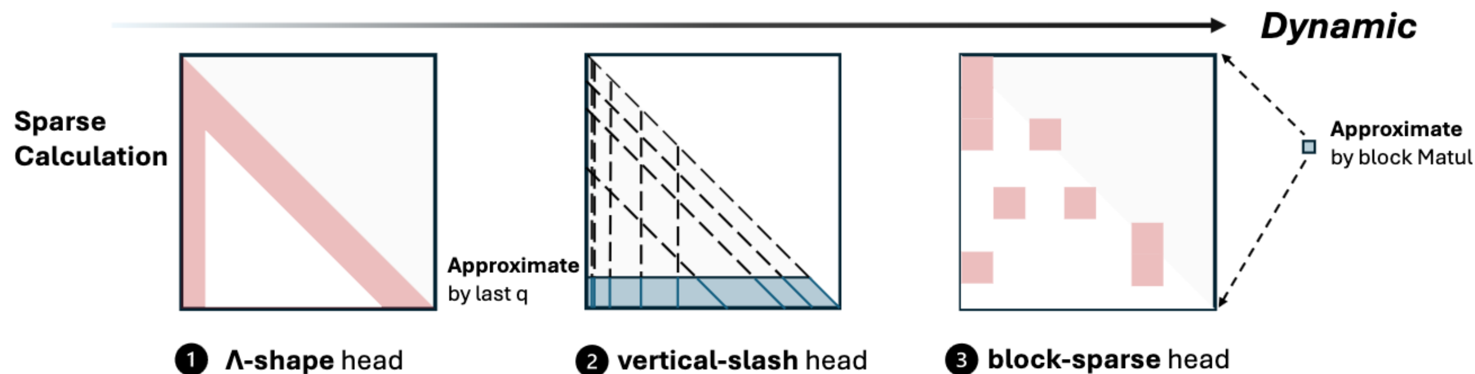


Faster Generation Speed

SnapKV: LLM Knows What You are Looking for Before Generation (NeurIPS 2024)

Ensemble of Sparse Patterns

- **Assign each head its most relevant pattern**
- **Minference: a mixture of 3 typical sparse patterns**

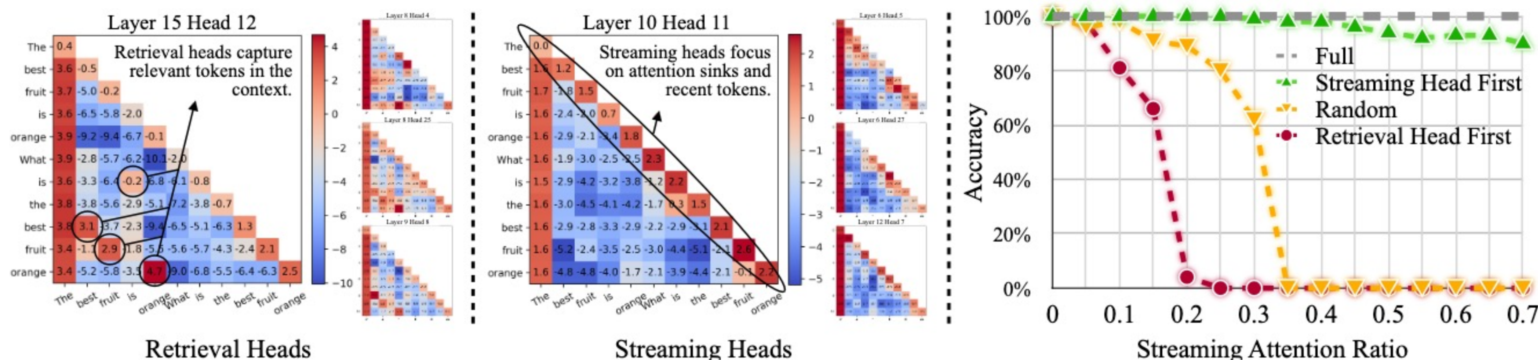


- Lambda-shape head: same as StreamingLLM, attn sink + SWA
- Vertical-slash head: vertical lines and slash lines observed in some heads
- Block-sparse head: dynamically determined by query tokens
- Implementation: all 3 patterns are implemented in block sparse kernels for efficiency
- Pattern searching for each attn head: search on a calibration dataset to minimize losses

MInference 1.0: Accelerating Pre-filling for Long-Context LLMs via Dynamic Sparse Attention (NeurIPS 2024)

More on Head Heterogeneity

- **Generally, attn heads can be classified into**
  - Streaming Heads: not retrieving information from the middle
  - Retrieval Heads: always retrieving information from the middle



- DuoAttention： introduce a lightweight training stage to identify all retrieval heads
  - The number of retrieval heads can be small → sparsity coming from head heterogeneity
  - For Streaming Heads, nearly no KV cache storage and no attention compute

$$\texttt{attn}_{i,j} = \alpha_{i,j} \cdot \texttt{full_attn} + (1 - \alpha_{i,j}) \cdot \texttt{streaming_attn}$$

DuoAttention: Efficient Long-Context LLM Inference with Retrieval and Streaming Heads (ICLR 2025)

Summary

- **Training free sparse attentions**
  - Static Sparse Patterns: Attention Sink, SWA, punctuations, special tokens, etc.
  - Dynamic Sparse Patterns: token voting, vertical and slash, block sparse, etc.
- **Ensemble sparse patterns**
  - Assign different heads different sparse patterns
  - Training-free ensemble: search a pattern arrangement on a calibration dataset
    - Minference
  - Lightweight training ensemble: selecting heads between {Retrieval, Streaming} heads
    - DuoAttention
- **Core assumption: attention is naturally sparse; patterns come from observations**
- 😁 **Sparse and fast;** 😭 **Training-inference mismatch, not end-to-end trainable**

Reducing KV Storage Pressure

KV Cache Optimizations (Engineering side)

KV Eviction

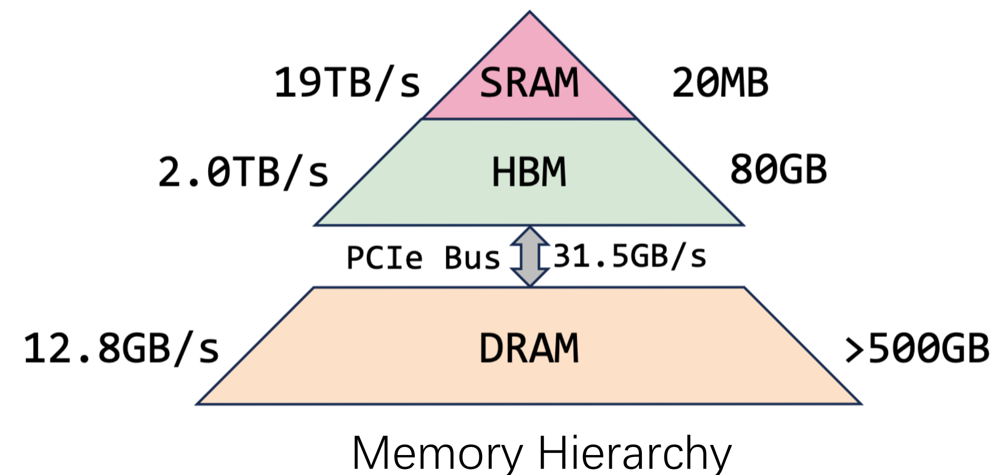
KV Offloading

KV Quantization

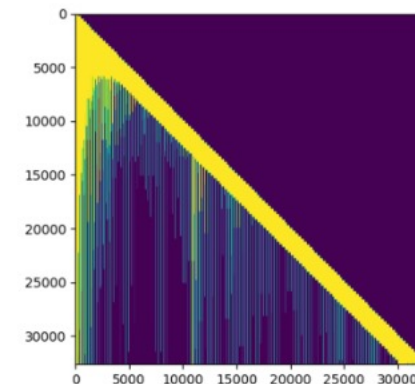
Won't be talking into LLM architectural improvement, e.g. MLA

KV Cache Eviction

- **GPU Memory is limited**
  - 80GB for A100/H100
- **Directly discard unimportant KV pairs**
- Lot's of Sparse Attention methods can originally evict KV
  - StreamingLLM: only keep attn sink + window, discard other KV pairs
  - H2O, SnapKV: evict KV pairs that are not in the topk set
  - DuoAttention: evict nearly all KVs for streaming heads
- Eviction does not only restricted to KV cache eviction
  - Tokens can be directly evicted before feeding into the LLM
    - LLM-Lingua2: Rate the importance of each token via a small LM
  - Can also be conducted at context level: agent skills can be viewd as eviction, too



Memory Hierarchy



Eviction Pattern

LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression (ACL 2024 Findings)

The core in KV Eviction: Importance Score

- **Rating the importance of KV pairs is challenging. Why?**
- Recall H2O's important score:

$$\boldsymbol{h} \in \mathbb{R}^n, \quad h_j = \sum_{i=j}^{n} p_{ij}, \quad \mathrm{HH}_k = \{j \mid j \in \text{top-}k(\boldsymbol{h})\}$$

- H2O: the importance of a token is defined by the probability score voted by **subsequent** tokens
- We don't always have access to subsequent tokens
  - E.g. When sequence too long, we do chunked prefill, i.e., prefill the input chunk by chunk
- Find a way to **"estimate"** the important score
- One way: post train a retaining head to predict the important score via a regression task **(Locret)**

$$\tilde{\mathbf{S}} = \mathbf{R}([\mathbf{Q}, \mathbf{K}, \mathbf{V}]) = \sigma([\mathbf{Q}, \mathbf{K}, \mathbf{V}]\mathbf{W}_1)\mathbf{W}_2.$$

$$\underset{\mathbf{W}_1^{(i)}, \mathbf{W}_2^{(i)}, i=1,2\cdots,L}{\operatorname{argmin}} \mathbb{E}_{d \in \mathcal{D}} \left[\sum_{i=1}^{L} \sum_{j=1}^{h} \sum_{k=1}^{n_q(d)} \mathcal{L}\left(\tilde{\mathbf{S}}[k]_j^{(i)}, \mathbf{S}[k]_j^{(i)}\right) \right]$$

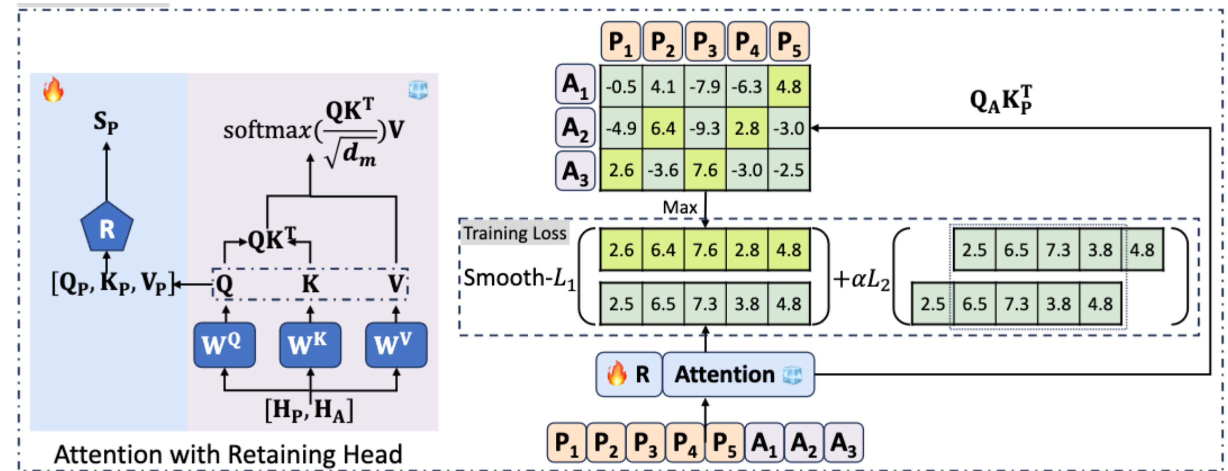
The core in KV Eviction: Importance Score

- **Locret:** post train a retaining head to predict the important score via a regression task
- Another way: put the importance score predictor into LLM architecture and pretrain the model **(DMA)**
- **DMA: Dynamic mask attention**

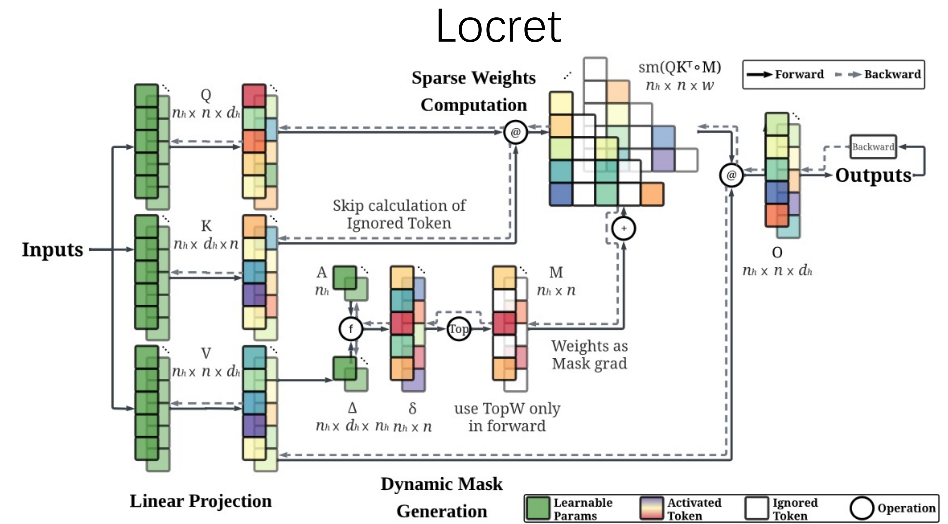
$$b_j = \exp(\tau(\mathbf{v_j W_1}) \odot \mathbf{W_2})$$

$$\mathbf{o}_i = \sum_j \frac{b_j \exp(\mathbf{q}_i \mathbf{k}_j^\top + m_{ij})\mathbf{v}_j}{\sum_l b_l \exp(\mathbf{q}_i \mathbf{k}_l^\top + m_{il})}$$

- Importance predictor is trained with LLM backbone from the pretrain stage



Locret

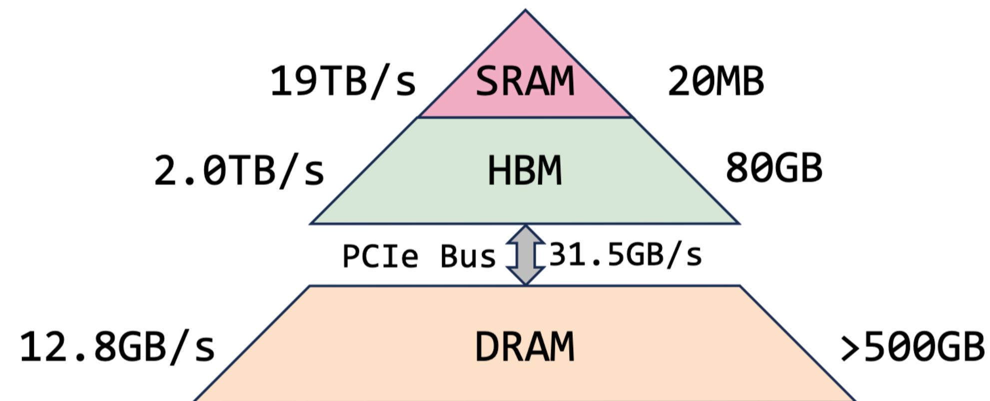


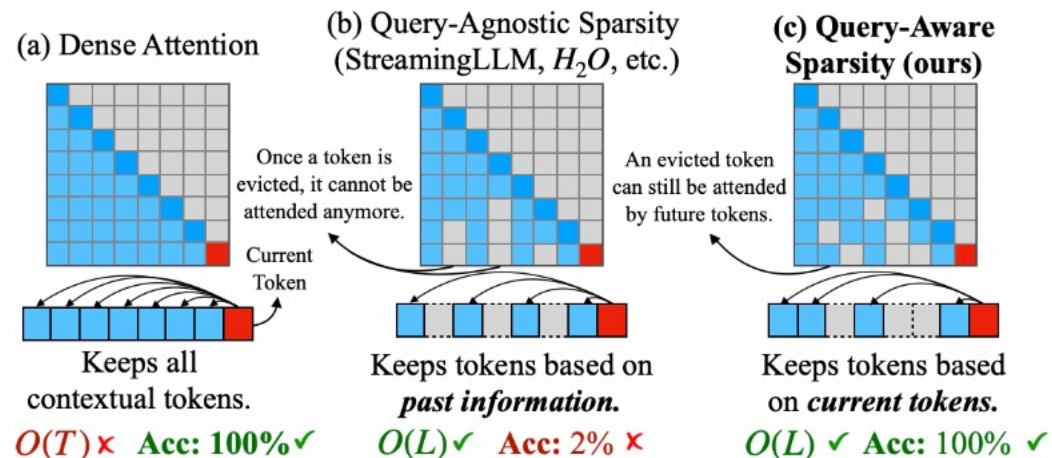
DMA

Locret: Enhancing Eviction in Long-Context LLM Inference with Trained Retaining Heads on Consumer-Grade Devices (TMLR 2025)
Trainable Dynamic Mask Sparse Attention (Arxiv:2508.02124)

KV Cache Offloading

- **Eviction's essential flaw: cannot tackle hard retrieval (recall) tasks**
- Eviction degrade LLMs to RNNs. Recall is still needed.
- Put most KVs on CPU mem. Only recall relevant parts to GPU mem at each decoding steps.
- **Quest:**
  - Organize KV into blocks
  - Have a representation vector for each block
- **InfLLM:**
  - Only recall selected blocks to GPU via PCIe.
- Offloading for (1) ctx len. extrapolation (2) higher decoding throughput



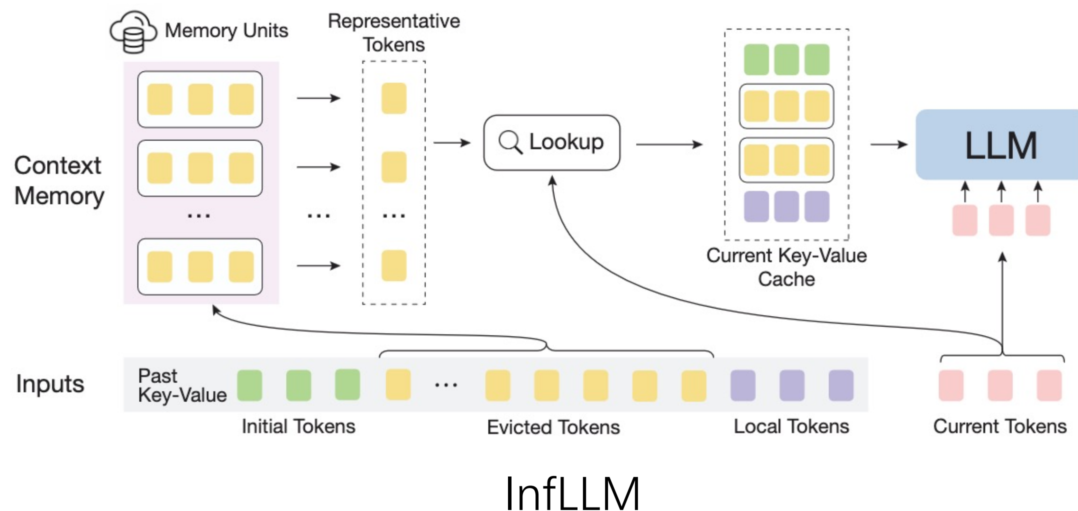


Quest

Quest: Query-Aware Sparsity for Efficient Long-Context LLM Inference (ICML 2024)
InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory (NeurIPS 2024)

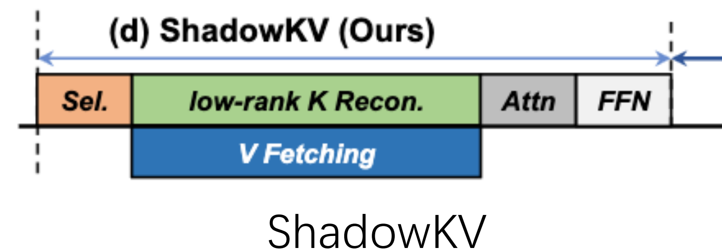
KV Cache Offloading

- **InfLLM**: inference inputs longer than trained context length via offloading
  - Offload both K and V to CPU
  - Choose an outstanding token in each block to represent the Ks



InfLLM

- **Problem: Massive PCIe data transfer.** HBM bandwidth: 2TB/s; PCIe bandwidth: 31.5GB/s
- **ShadowKV**: reduce PCIe data transfer by only offloading Vs
  - Low rank structure of K: SVD K, store low rank Ks
  - High rank structure of V: cannot compress, so offload Vs.
  - Overlap reconstructing K with loading V
  - Higher batch size, larger decoding throughputs

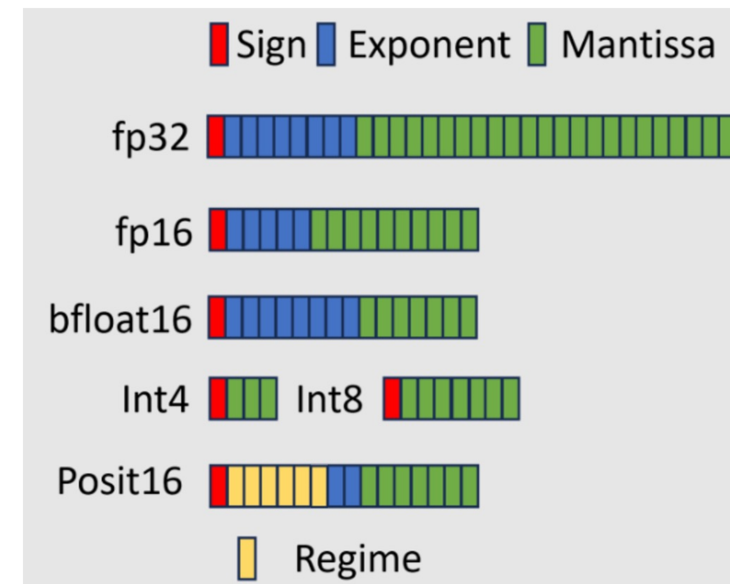


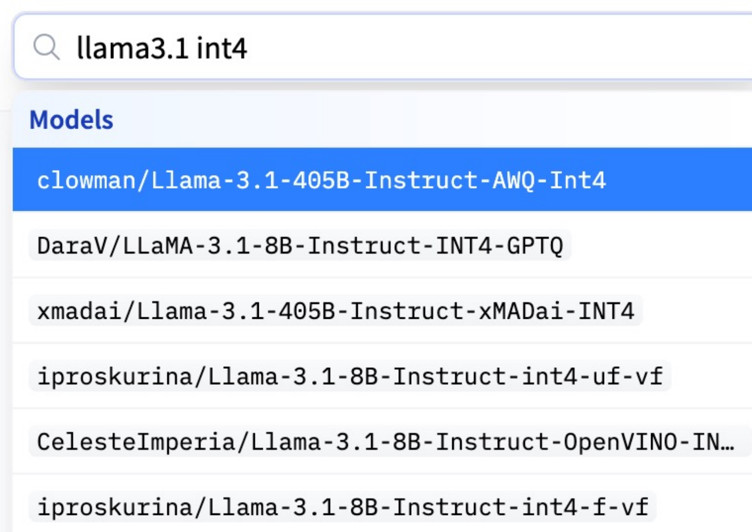
ShadowKV

InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory (NeurIPS 2024)
ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference (ICML 2025)

KV Quantization

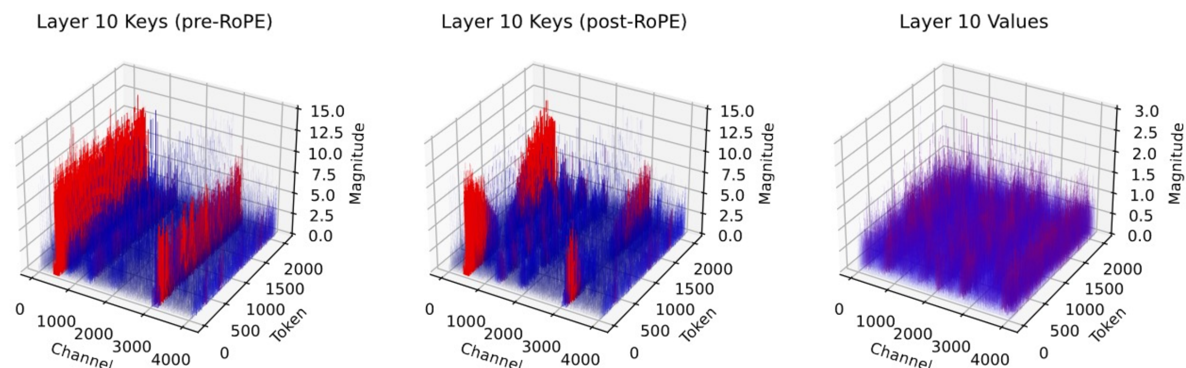
- **Quantization**: Store Mat. and calculate Mat. Mul. In low-bit representations, e.g. FP8 ,FP4, INT8, INT4, INT2
- LLM weights are commonly quantized
- Why quantization?
  - Storage saving (both disk and GPU memory)
  - Faster decoding: less memory access
- **KV cache can also be quantized**
  - Store KV cache in INT4/INT2, Dequant for inference
- Tiling the tensor for quantization
  - For a better precision, we do $x_q = \mathrm{round}(c(x-b))$
  - Store (x_q, c, b) as the quantized representation
  - Each tile shares a (c, b) to save space





KV Quantization

- **Finding the right way to tile KV cache for quantization** $x_q = \mathrm{round}(c(x-b))$
- We wish a tile shares similar data distribution, so they can agree to similar (c, b)
- **KVQuant: K and V has different features**
  - K: each channel (dim) shares similar data distribution, but not each token
  - V: each token shares similar distribution
  - Pre-RoPE Ks are easier to quantize
  - Tile K along each channel, tile V alone each token, keep attn sink unquantized



Config	Avg. bit	Niah1	Niah2	Niah3	MKey1	MKey2	MKey3	MValue	MQuery	VT	CWE	FWE	QA1	QA2	Avg.
fp16 Baseline	16	100	99.8	98.6	94	68.2	11	55.95	64.5	37.88	9.64	30.4	31.6	31.6	56.40
KIVI-2-gs32-r128	3.05	76	85.6	59.6	72.6	11.4	0	34.7	46.45	39.6	8.26	**30.53**	24.8	27.6	39.78
KVQuant-3bit-1%	3.33	**99.8**	**98.8**	**95.2**	**92.8**	**61.6**	**6.4**	**47.5**	**54.45**	**41.04**	**8.52**	29.33	**31.0**	**31.0**	**53.65**
KVQuant-2bit-1%	2.33	95.4	86.8	49.8	73.6	23.4	0	16.65	22.95	22.52	5.14	24.0	26.4	28.4	36.54

KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization (NeurIPS 2024)

Summary

- **KV cache optimization (engineering side): Eviction, Offloading, Quantization**
- **KV Eviction**: discard unimportant KV cache units by some importance score
  - 😁 Efficient; 😭 cannot tackle recall tasks
- **KV Offloading**: offload unimportant KV cache units to CPU memory and recall them if needed
  - 😁 can tackle recall tasks, also efficient; 😭 hard to implement, lots of system concerns
- **KV Quantization**: store KV cache in low bit representation
  - 😁 (usually) plug-and-play, simple and effective; 😭 need hardware support, otherwise slow

- More on KV cache optimizations: LLM architectural improvement (not going to delve into)
  - Utilize low-rank: DeepSeek MLA – derive K and V from a compressed latent vector
  - KV cache sharing: Google Gemma-4 – share the KV caches across the last layers
  - Hybrid model: SWA/RNN layers do not have KV cache

Enhancing Long-Context Performance

Trainable Sparse Attentions & Non-softmax Attentions

Trainable Sparse Attentions

- **What can't be achieved by training-free sparse attentions?**
- **Training-Inference (T-I) mismatch:**
  - Sparse Attentions are not trained with the LLM → OOD Inference
  - Some patterns cannot be trained (H2O, SnapKV, etc.)
  - Sparse patterns are designed based on observations: too much human effort involved

Training-free Offloading (ShadowKV)

Step 1: To start, we need to understand the relationship between the number of dogs each person is responsible for and their <skip lines>
\r\n Old\r\n \r\n Until Next\r\n
\r\n \r\n \r\n \r\n /\r\n

- **Prefill-Decoding (P-D) mismatch:**
  - Training free sparse attentions commonly optimize one stage, but not two
  - Prefill: Minference; Decode: H2O, SnapKV
  - Hard to unify them: inaccurate prefill already perturbs the KV cache
- **Also, cannot do long decoding and reasoning**

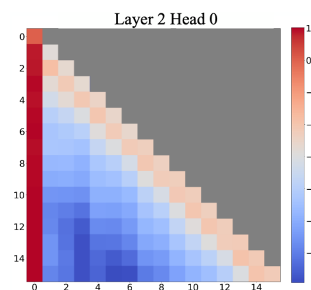
NOSA
ShadowKV
Perplexity
6
4
2
0
25
50
75
100
Generation Progress (%)

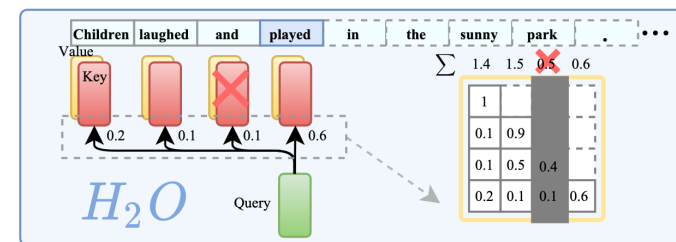
Native Sparse Attention (NSA)

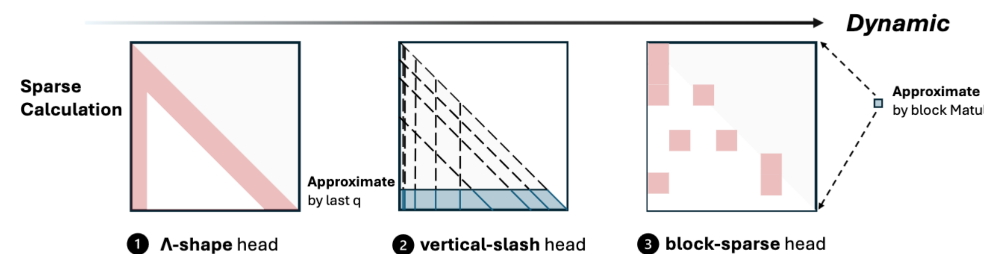
- **Solution: pretrain the LLM with sparse attention**
- What features are compulsory for Trainable Sparse Attention?
  - **1. Differentiable:** gradient must be able to flow back
  - **2. Efficient:** sparse patterns should be supported by hardware features
  - **3. Support Causal LM:** each token should select KVs separately
- Take a second look at the sparse patterns:
  - StreamingLLM (😁: 1, 2, 3; 😭: None)
  - H2O, SnapKV (😁: 1; 😭: 2, 3)
    - Not universal for prefill and decode
  - Vertical and Slash (😁: 1, 3; 😭: 2)
    - Hard to reduce mem. acc.
  - Block Sparse (😁: 1, 2, 3; 😭: None)

T-I Mismatch

P-D Mismatch





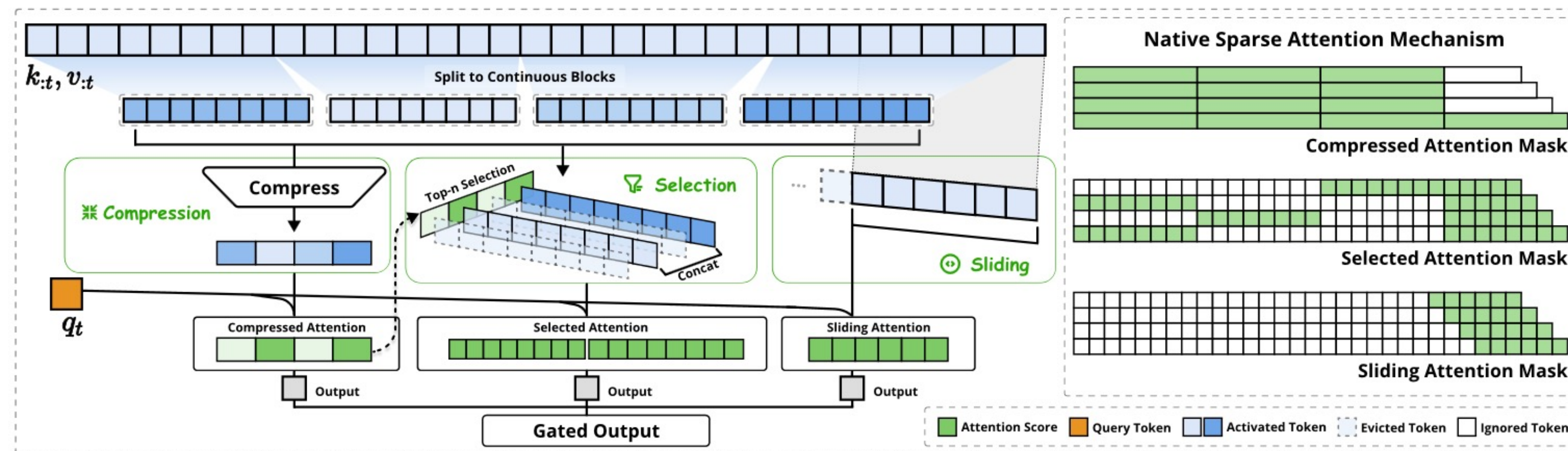


Native Sparse Attention (NSA)

- **“Good” patterns: attn sink, sliding window, block sparse attn**
  - **No T-I mismatch, no P-D mismatch at all, also efficient**
- **NSA:** 3 basic patterns – sliding window, compressed attn, block sparse attn

T-I Mismatch

P-D Mismatch



- **Comp Attn:** compress B=64 KV pairs to a single (kc, vc), using a small MLP
  - Then, an SDPA is conducted over the compressed (kc, vc)s
- **Block Sparse Attention:** use comp attn score to identify top-k most relevant blocks

Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention (ACL 2025)

Native Sparse Attention (NSA)

- **No T-I mismatch:** model pretrained with sparse attention
- **No P-D mismatch: each q selects KV separately**, consistent in decoding
- **Efficiency: an excellent practice of using Tensor Core GEMM**
- **GEMM on GPUs:** `mma.sync.aligned.m16n8k16.row.col.f32.bf16.bf16.f32`
  - This operation describes $\boldsymbol{D} = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{C}, \quad \boldsymbol{A} \in \mathbb{R}^{16\times16}, \boldsymbol{B} \in \mathbb{R}^{16\times8}, \boldsymbol{C} \in \mathbb{R}^{16\times8}$
  - In Flash Attention: $\boldsymbol{S}_B = \boldsymbol{Q}_B \boldsymbol{K}_B^\top$ `(BLOCK_Q, BLOCK_D) * (BLOCK_D, BLOCK_KV)`
  - Each q selects KV separately: each q does not attention to the same set of $\boldsymbol{K}_B$
  - So we can't do GEMM with `(BLOCK_Q, BLOCK_D) * (BLOCK_D, BLOCK_KV)`
- One Alternative: set `BLOCK_Q=1`
  - Can't use Tensor Core anymore: we do not have `mma.sync.aligned.m1n8k16`…
  - Can only use CUDA core: significantly slower than Tensor core

Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention (ACL 2025)

Recall: FlashAttn (Triton Impl.)

Note: The figure is from FA1, but the algorithm is from FA2, as the FA2 is the de facto implementation.

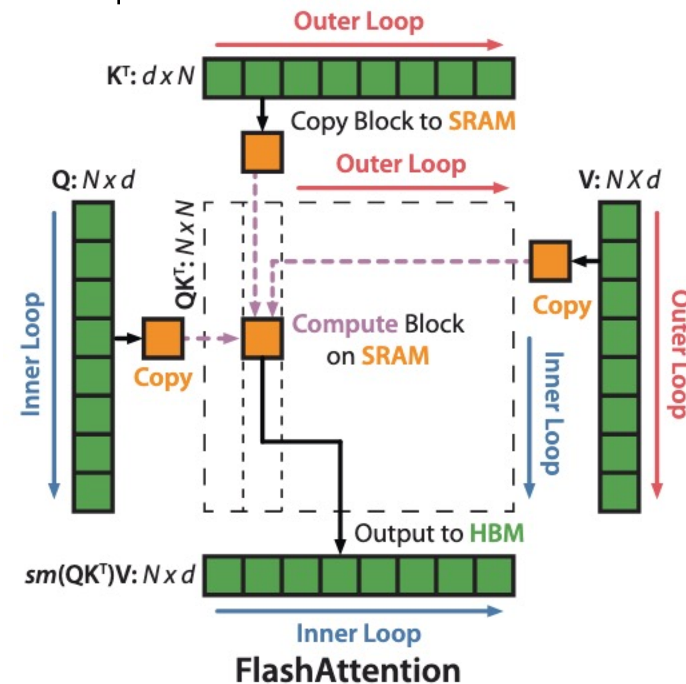
```
# Inner loop, in kernel, sequential
q = tl.load(q_ptr)                          # load q
for start_n in range(0, N, BLOCK_N):        # inner loop sequential
    k_blk = tl.load(K + start_n)            # load k
    v_blk = tl.load(V + start_n)            # load v
    qk = tl.dot(q, k_blk.t)                 # calc q*k

    m_ij = tl.max(qk) * scale               # block max
    p = tl.exp((qk * scale) - m_ij)         # stable exp
    l_ij = tl.sum(p)                        # sum exp

    acc_o = acc_o * tl.exp(m_i - m_ij) + tl.dot(p, v_blk) # new a (numerator)
    lse_i = tl.log(exp(lse_i - m_ij) + l_ij)              # log sum exp (denominator)
    m_i = m_ij                                            # update block mask

out = acc_o / exp(lse_i) # numerator / denominator

# Outer loop, parallel
result = flash_attn_kernel[(B, H, NUM_Q_BLOCKS)](...)
```



$$\frac{\exp s_{ij}}{\sum_{j=1}^{n} \exp s_{ij}} = \frac{\exp\left(s_{ij} - c\right)}{\sum_{j=1}^{n} \exp\left(s_{ij} - c\right)}$$

Native Sparse Attention (NSA)

- Satisfying `mma.sync.aligned.m16n8k16.row.col.f32.bf16.bf16.f32`
  - Flash Attention: `(BLOCK_Q, BLOCK_D) * (BLOCK_D * BLOCK_KV)`
- GQA is commonly used in modern LLMs
  - Just ask each query group shares a sparse pattern
  - With in a query group, all qs attend to the same set of KV
  - `(GROUP_SIZE, BLOCK_D) * (BLOCK_D * BLOCK_KV)`, **`GROUP_SIZE>=16`**

```
q = tl.load(q_ptr + tl.arange(G))          # load q [G, D]
for start_n in range(0, K):                # inner loop sequential
    k_blk = tl.load(K + ...)               # load k [BKV, D]
    qk = tl.dot(q, k_blk.t)                # GEMM q*k [G, D]*[D, BKV] = [G, BKV]
    ...
out = acc_o / exp(lse_i)                   # numerator / denominator

# Outer loop, parallel
result = flash_attn_kernel[(B, HKV, N)](...)
```

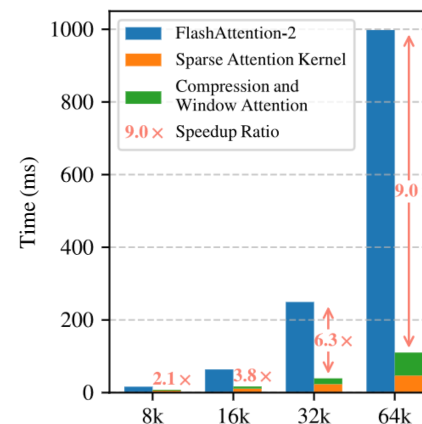
T-I Mismatch

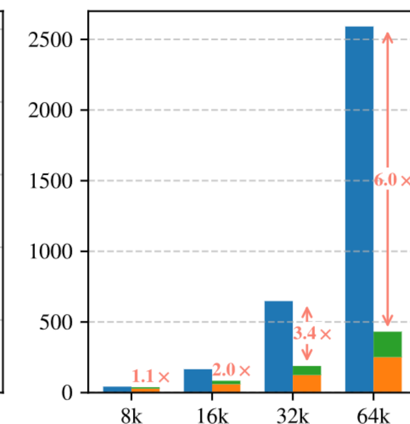
P-D Mismatch

Native Sparse Attention (NSA)

- Good performance, Fast long-context inference speed

Model	SQA			MQA				Synthetic		Code	Avg.
	MFQA-en	MFQA-zh	Qasper	HPQ	2Wiki	GovRpt	Dur	PassR-en	PassR-zh	LCC	
H2O	0.428	0.429	0.308	0.112	0.101	0.231	0.208	0.704	0.421	0.092	0.303
InfLLM	0.474	0.517	0.356	0.306	0.250	0.277	0.257	0.766	0.486	0.143	0.383
Quest	0.495	0.561	0.365	0.295	0.245	0.293	0.257	0.792	0.478	0.135	0.392
Exact-Top	0.502	0.605	0.397	0.321	0.288	0.316	0.291	0.810	0.548	0.156	0.423
Full Attn	**0.512**	0.623	0.409	0.350	0.305	**0.324**	0.294	0.830	**0.560**	0.163	0.437
NSA	0.503	**0.624**	**0.432**	**0.437**	**0.356**	0.307	**0.341**	**0.905**	0.550	**0.232**	**0.469**





- **Remaining Problem: slower for short context (dense inference)**

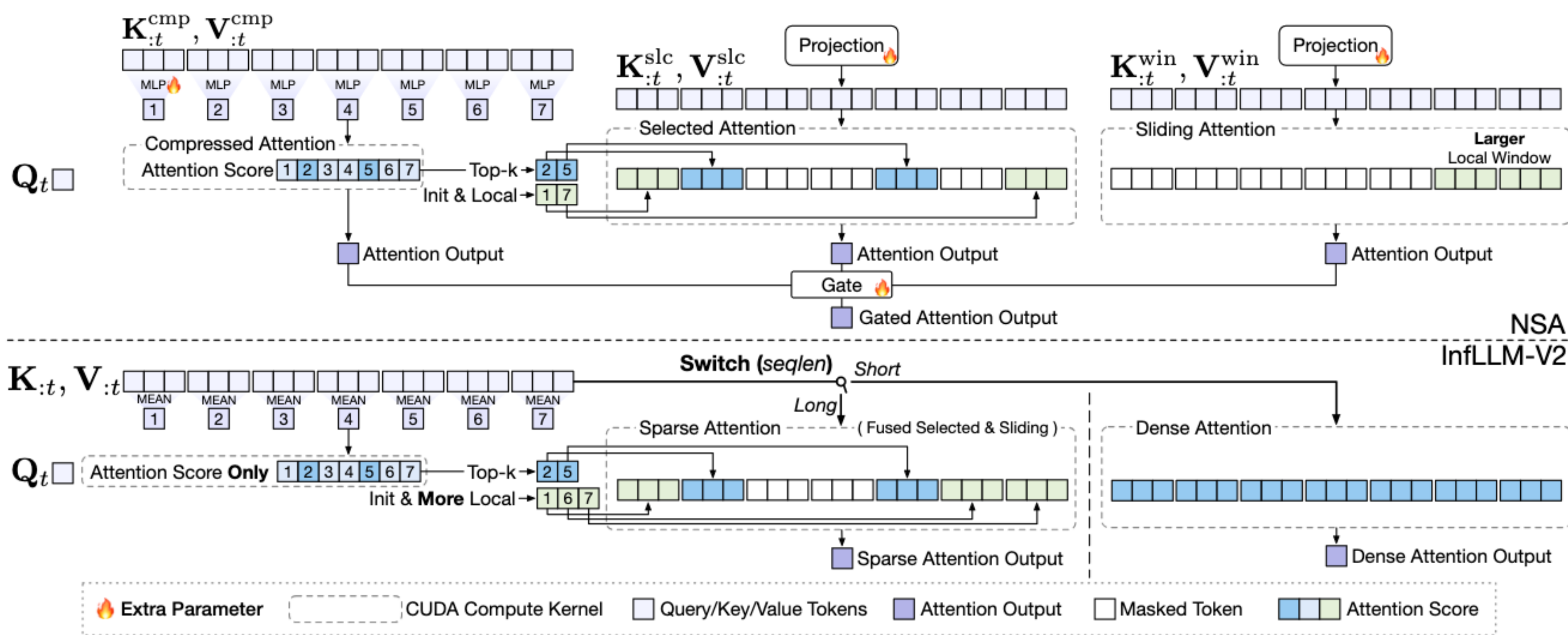
$$\boldsymbol{o}_i = g_{win}\boldsymbol{o}_i^{win} + g_{comp}\boldsymbol{o}_i^{comp} + g_{blk}\boldsymbol{o}_i^{blk}$$

- For short context n < k, still need to calculate $\boldsymbol{o}_i^{comp}$, otherwise T-I mismatch
- **Solution 1:** get rid of $\boldsymbol{o}_i^{comp}$ completely
- **Solution 2:** plug an external indexer to predict token importance w.r.t q (DSA, skipped today)

Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention (ACL 2025)

Dense-Sparse Switchable SA (InfLLMv2)

- **Solution 1:** get rid of o_i^{comp} completely $\quad o_i = g_{win} o_i^{win} + g_{comp} o_i^{comp} + g_{blk} o_i^{blk}$

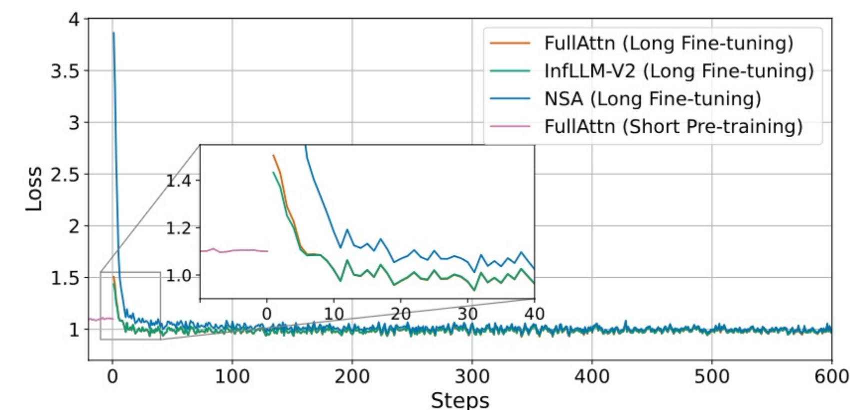


- No compressed attention, use mean pooling to obtain the K representation
- No additional parameters to train

Dense-Sparse Switchable SA (InfLLMv2)

- Obtaining K representations: NSA – MLP; InfLLMv2 – Mean Pooling
- Removing additional parameters: Easier to adapt in long-context continual pretraining (CPT)
- Significantly better than NSA if only trained in CPT
- Sparse Training → Dense Inference: even better than Full Attention
- **A smoother transition from softmax attention to sparse attention is the key to success**

Method	SG1	SG2	SG3	MK1	MK2	MK3	MV	MQ	VT	CWE	FWE	QA1	QA2	Avg.
FULLATTN	100.00	100.00	100.00	96.00	94.00	92.00	82.00	98.50	93.20	44.40	91.33	48.00	56.00	84.26
SHORT+YARN	98.00	68.00	50.00	46.00	6.00	0.00	32.00	31.50	36.00	21.40	87.33	26.00	26.00	40.63
INFLLM	98.00	6.00	4.00	10.00	10.00	10.00	9.00	7.50	70.00	16.00	80.67	18.00	24.00	27.94
MINFERENCE	**100.00**	**100.00**	**100.00**	76.00	36.00	46.00	79.50	93.50	88.00	**64.20**	**92.67**	32.00	**44.00**	73.22
NSA	**100.00**	88.00	82.00	54.00	38.00	30.00	59.00	61.50	56.00	34.40	86.00	56.00	34.00	59.92
INFLLM-V2 (SPARSE)														
w/ *LSE Approx*	**100.00**	**100.00**	**100.00**	**94.00**	**82.00**	62.00	**98.50**	94.50	**98.00**	50.40	82.67	**72.00**	40.00	**82.62**
w/o *LSE Approx*	**100.00**	**100.00**	**100.00**	92.00	80.00	**64.00**	**98.50**	**95.50**	**98.00**	47.80	81.33	70.00	40.00	82.09
INFLLM-V2 (DENSE)	100.00	100.00	100.00	94.00	98.00	98.00	99.00	98.00	98.40	52.80	90.00	76.00	44.00	88.32



Non-Softmax Attentions

- We were discussing SAs as heuristics to softmax attention. **But is softmax attention good?**
- Yes: common practice, softmax has been used for decays, SOTA models use softmax
- No: softmax is a dispersive function
- **Definition (Dispersion)**

Given a bounded sequence $\{z_i\}_{i=1}^{\infty} \subset \mathbb{R}^n$, $|z_i| \leq M < +\infty$, $\forall i \in \mathbb{N}_+$, and a mapping $f : \mathbb{R}^n \to \Delta^{n-1}$, where $\Delta^{n-1} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \mathbf{1}^\top \boldsymbol{x} = 1\}$.

Define $\boldsymbol{p}_n = f(\boldsymbol{z}_{1:n})$. If

$$\lim_{n\to\infty} \frac{H^S(\boldsymbol{p}_n)}{\log n} = 1,$$

then we say that f is a *dispersive* function. Otherwise, i.e.

$$\lim_{n\to\infty} \frac{H^S(\boldsymbol{p}_n)}{\log n} \leq c < 1,$$

f is a non-dispersive function.

Non-Softmax Attentions

- **Definition (Dispersion)**

  Given a bounded sequence $\{z_i\}_{i=1}^{\infty} \subset \mathbb{R}^n$, $|z_i| \leq M < +\infty$, $\forall i \in \mathbb{N}_+$, and a mapping $f : \mathbb{R}^n \to \Delta^{n-1}$, where $\Delta^{n-1} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \mathbf{1}^\top \boldsymbol{x} = 1\}$.

  Define $\boldsymbol{p}_n = f(\boldsymbol{z}_{1:n})$. If

$$\lim_{n \to \infty} \frac{H^S(\boldsymbol{p}_n)}{\log n} = 1,$$

  then we say that f is a *dispersive* function. Otherwise, i.e.

$$\lim_{n \to \infty} \frac{H^S(\boldsymbol{p}_n)}{\log n} \leq c < 1,$$

  f is a non-dispersive function.

- Intuition of Dispersion: when scaling the context length, the "chaos" of softmax score grows tremendously. Note that $H^S(\boldsymbol{u}_n) = \log n$, so dispersion -> nearly completely random
- **Theorem (Softmax Dispersion)**: softmax is dispersive on bounded sequences.

Non-Softmax Attentions

- **Theorem (Non-Dispersion of Sparse Attentions)**:

Given a bounded sequence $\{z_i\}_{i=1}^{\infty} \subset \mathbb{R}^n$, $|z_i| \leq M < +\infty$, $\forall i \in \mathbb{N}_+$, and a mapping $f : \mathbb{R}^n \to \Delta^{n-1}$, where $\Delta^{n-1} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \mathbf{1}^\top \boldsymbol{x} = 1\}$.

Denote the positions where $f(\boldsymbol{z}_{1:n})$ gives positive probability as $\mathcal{S}_n = \{j \mid f(\boldsymbol{z}_{1:n})_j > 0\}$.

If $|\mathcal{S}| = \mathcal{O}(n^\beta)$, $\beta \in [0, 1)$, then

$$\lim_{n \to \infty} \frac{H^S(f(\boldsymbol{z}_{1:n}))}{\log n} = \beta < 1,$$

so f is a non dispersive function.

- **Top-k sparse attentions** are non-dispersive: $\mathcal{O}(k) = \mathcal{O}(1), \ \beta = 0$
- Can also relax to $0 < \beta < 1$
- Vanilla softmax attention: dispersion; introducing some sparsity: can be non dispersion
- **Now, let's get rid of softmax!**

Non-Softmax Attentions

- **Variational inference's view of softmax**

$$\mathrm{softmax}(\boldsymbol{z}) = \arg\max_{\boldsymbol{p}\in\Delta^{n-1}} \{\boldsymbol{p}^\top \boldsymbol{z} + H^S(\boldsymbol{p})\}$$

- Similar to Fenchel-Young's Conjugate:

Given a regularizer Ω, the conjugate of Ω is

$$\Omega^*(\boldsymbol{y}) = \sup_{\boldsymbol{x}\in\mathrm{dom}\,\Omega} \{\boldsymbol{x}^\top \boldsymbol{y} - \Omega(\boldsymbol{p})\}$$

- For softmax, the regularizer is the negative Shannon's Entropy
- Replacing the Shannon's Entropy to Tsallis' Entropy

$$H^T(\boldsymbol{p}) = \frac{1}{\alpha(\alpha-1)}\sum_{i=1}^{n}(p_i^\alpha - p_j), \quad \mathrm{entmax}_\alpha(\boldsymbol{z};\tau) = \arg\max_{\boldsymbol{p}\in\Delta^{n-1}} \{\boldsymbol{p}^\top \boldsymbol{z} + H^T(\boldsymbol{p})\}$$

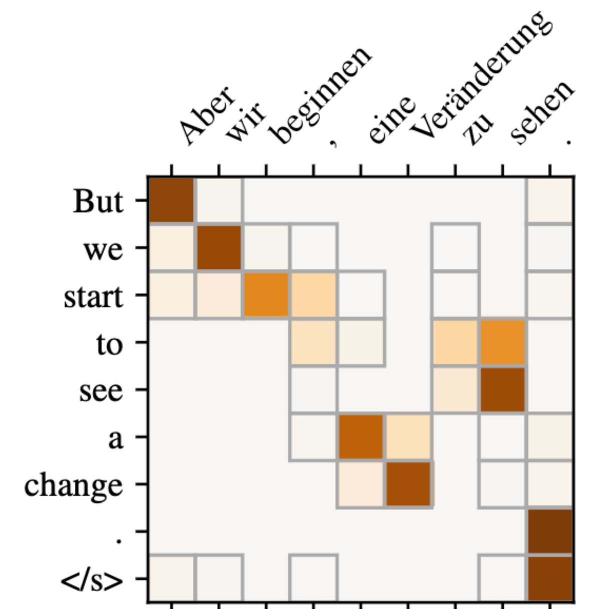
- **Entmax:** $\mathrm{entmax}_\alpha(\boldsymbol{z};\tau)_i = [(\alpha-1)z_i - \tau]_+^{\frac{1}{\alpha-1}}$, find a τ $s.t.$ $\mathbf{1}^\top \mathrm{entmax}_\alpha(\boldsymbol{z};\tau) = 1$

Sparse Sequence-to-Sequence Models (ACL 2019)

Non-Softmax Attentions

- **Entmax:** $\mathrm{entmax}_\alpha(\boldsymbol{z};\tau)_i = [(\alpha-1)z_i - \tau]_+^{\frac{1}{\alpha-1}}$, find a τ $s.t.$ $\mathbf{1}^\top \mathrm{entmax}_\alpha(\boldsymbol{z};\tau) = 1$
- Entmax can produce zero probability →Sparse
- No topk, tau is determined by the data geometry → Adaptively sparse
- If the activation follows $|\mathcal{S}_n| = \mathcal{O}(n^\beta), \beta < 1$, then entmax is a non-dispersive mapping
  - Empirically true, haven't been proved yet
- Great interpretability: irrelevant tokens gets 0 probabilities
- **Actually a very old method, nothing new**

method	DE→EN	EN→DE	JA→EN	EN→JA	RO→EN	EN→RO
softmax	25.70 ± 0.15	21.86 ± 0.09	20.22 ± 0.08	25.21 ± 0.29	29.12 ± 0.18	28.12 ± 0.18
1.5-entmax	**26.17** ± 0.13	**22.42** ± 0.08	**20.55** ± 0.30	**26.00** ± 0.31	**30.15** ± 0.06	**28.84** ± 0.10
sparsemax	24.69 ± 0.22	20.82 ± 0.19	18.54 ± 0.11	23.84 ± 0.37	29.20 ± 0.16	28.03 ± 0.16

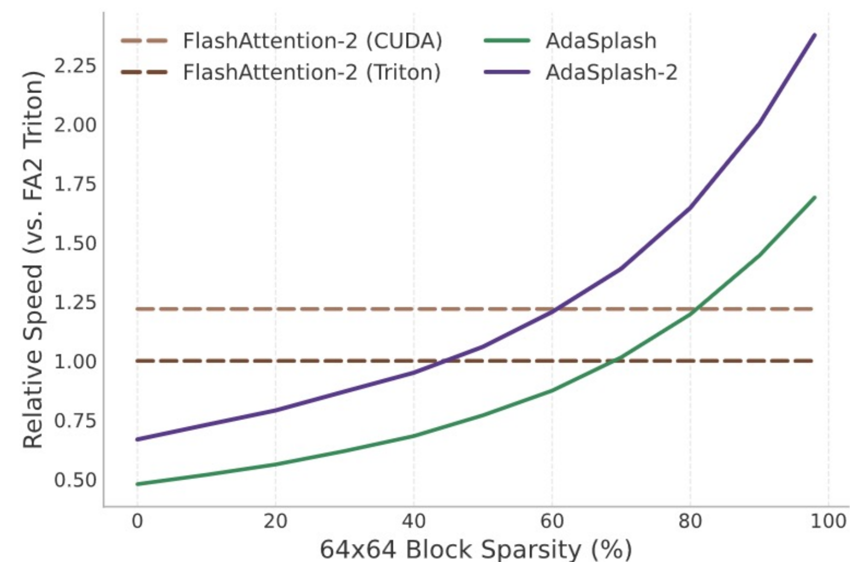


Non-Softmax Attentions

- **Scaling entmax models: needs better kernel to find the tau**
- AdaSplash/AdaSplash-2 kernels: similar to FlashAttention to softmax attention
- Finally scaling entmax models in 2025/2026
- **Significantly better than Softmax+RoPE due to adaptiveness**

Table 1. RULER benchmark results for 1B parameter models trained up to 32K context-length. Best average results are in **bold**.

Model	Len.	MK-1	MK-2	MK-3	MQ	MV	NIAH-1	NIAH-2	NIAH-3	CWE	FWE	VT	QA-H	QA-S	Avg.
Softmax (RoPE)	4K	80.0	62.2	38.2	47.6	43.4	100.0	100.0	98.8	5.1	24.9	8.0	24.8	35.0	51.4
	8K	78.4	45.0	30.2	36.7	31.0	100.0	99.2	98.2	0.1	24.3	2.8	22.4	20.2	45.3
	16K	67.6	10.0	14.8	22.1	22.7	100.0	99.2	99.2	0.0	19.0	0.2	17.0	18.7	37.7
	32K	41.4	8.2	0.8	17.4	20.8	96.2	61.6	75.6	0.0	6.3	0.0	16.8	10.1	27.3
Entmax (RoPE)	4K	75.4	26.0	23.6	29.9	52.3	100.0	100.0	98.8	36.3	32.3	3.9	25.8	36.7	49.3
	8K	71.8	19.4	8.6	22.5	42.5	100.0	99.8	94.6	8.9	23.7	3.6	23.2	19.6	41.4
	16K	53.2	8.6	1.0	12.9	22.0	100.0	81.4	79.4	0.1	18.6	1.2	18.2	20.8	32.1
	32K	33.2	4.4	0.6	7.5	12.9	97.0	40.8	64.2	0.0	15.0	0.2	18.6	9.2	23.4
Softmax (NAPE)	4K	88.2	29.8	13.0	28.9	27.9	100.0	100.0	100.0	35.3	27.4	14.8	21.8	37.9	48.1
	8K	78.4	38.6	12.6	33.2	34.1	100.0	100.0	99.0	27.0	26.3	12.3	19.2	22.9	46.4
	16K	85.6	19.6	7.4	26.8	30.6	100.0	100.0	98.8	18.0	25.6	9.8	20.0	21.3	43.3
	32K	66.6	4.2	2.0	26.4	26.5	100.0	97.0	98.0	8.3	3.3	7.7	19.0	19.5	36.8
Entmax (NAPE)	4K	81.2	12.2	27.2	51.9	55.6	100.0	100.0	79.8	48.9	58.1	31.4	30.2	34.5	**54.7**
	8K	73.4	4.6	17.6	62.3	31.5	100.0	100.0	84.0	33.7	48.3	38.4	29.0	26.0	**49.9**
	16K	79.2	1.4	9.8	38.7	24.0	100.0	100.0	74.2	22.9	51.1	34.5	27.2	27.5	**45.4**
	32K	63.0	1.8	4.8	25.8	16.8	100.0	92.0	76.0	11.7	45.8	27.4	25.0	22.1	**39.4**



AdaSplash: Adaptive Sparse Flash Attention **(ICML 2025)**
AdaSplash-2: Faster Differentiable Sparse Attention **(Arxiv:2604.15180)**

Non-Softmax Attentions

- **Non-dispersion property of entmax**
  - Promising in handling very long context length
- **ASEntmax:** Proved that entmax is good for context extrapolation

	S-NIAH-1 (ID)		S-NIAH-1 (OOD)			S-NIAH-2 (ID)		S-NIAH-2 (OOD)	
Model	1K	2K	4K	8K	16K	1K	2K	4K	8K
Softmax	**100.0**	99.4	94.2	11.4	0.8	**100.0**	**100.0**	4.8	0.0
SSMax	**100.0**	99.8	99.2	92.0	75.2	99.4	99.2	64.4	14.8
Entmax	99.8	99.8	89.0	21.6	1.2	99.6	99.4	64.8	7.2
ASEntmax	99.6	**100.0**	**100.0**	**99.8**	**97.4**	99.4	99.4	**83.2**	**25.4**

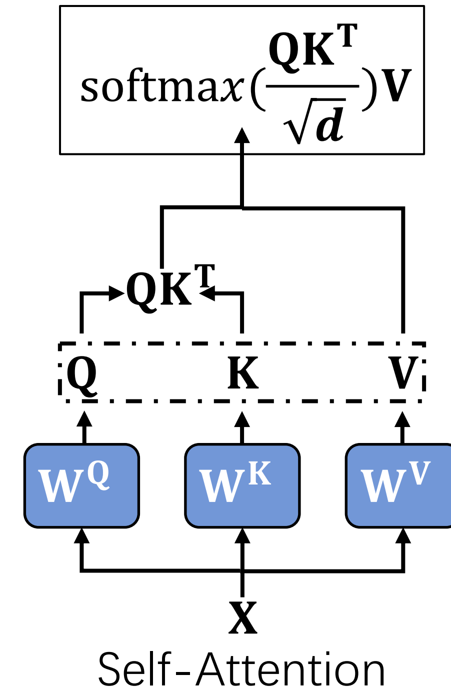
		S-NIAH-1					S-NIAH-2			
		ID		OOD			ID		OOD	
	Model	1K	2K	4K	8K	16K	1K	2K	4K	8K
RoPE	Softmax	99.8	79.0	0.0	0.0	0.0	99.6	11.4	0.0	0.0
RoPE	SSMax	99.0	83.0	0.0	0.0	0.0	99.6	53.6	0.0	0.0
RoPE	Entmax	100.0	79.6	0.0	0.0	0.0	99.6	53.0	0.0	0.0
RoPE	ASEntmax	99.8	87.2	0.0	0.0	0.0	99.0	83.6	0.0	0.0
RoPE	+ ABF, Softmax	99.2	97.2	93.4	75.4	75.6	0.0	0.0	0.0	0.0
RoPE	+ ABF, SSMax	98.2	98.0	97.6	97.4	98.0	30.8	37.4	4.4	0.2
RoPE	+ ABF, Entmax	98.4	98.2	99.4	100.0	100.0	98.8	89.0	64.8	32.4
RoPE	+ ABF, ASEntmax	100.0	99.6	99.6	99.6	94.0	98.6	83.6	30.8	7.2
NAPE	Softmax	100.0	99.4	94.2	11.4	0.8	100.0	100.0	4.8	0.0
NAPE	SSMax	100.0	99.8	99.2	92.0	75.2	99.4	99.2	64.4	14.8
NAPE	Entmax	99.8	99.8	89.0	21.6	1.2	99.6	99.4	64.8	7.2
NAPE	ASEntmax	99.6	100.0	100.0	99.8	97.4	99.4	99.4	83.2	25.4

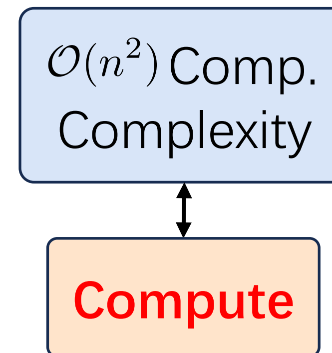
Conclusion

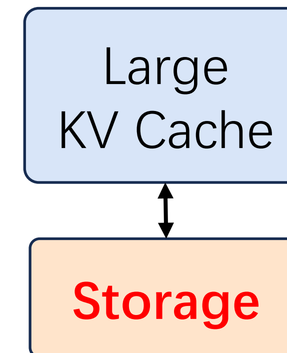
- **Frontiers of Sparse Attentions**
  - **Trainable Sparse Attentions:** ask the model to get “used to” sparse attns from pretraining
  - **Non-Softmax Attentions:** softmax might be a bad choice for efficiency and performance
- These are only selected topics of frontier research
- **What’s missing:**
  - **Explaining Sparse Attention:** Why are attentions sparse? Where does attention sink come from? Why are there vertical lines and slash lines (Minference)?
  - **Explaining Sparse Training:**
    - Why LLM trained with sparse attentions can converge? How’s their generalizability?
    - A phenomenon: sparse train → dense inference is better than dense train → dense inference. Why does this happens?
  - **Learning theories:** Can we develop provable learnable sparse attentions?

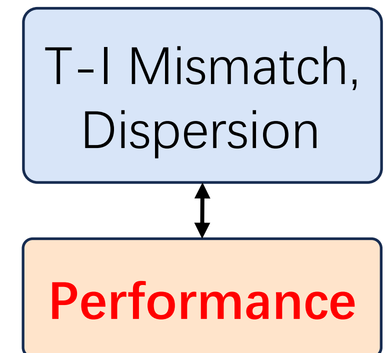
General Conclusion

- **Training free SAs:**
  - Observe some sparse pattern then use them to accelerate inference
  - Attn Sink, sliding window, block sparse, etc.
- **KV cache optimizations:**
  - Eviction, Offloading, Quantization
  - Target: shrink the KV cache size
- **Improving SA's performance:**
  - Natively train LLMs with sparse attentions
  - Replacing softmax with better mappings









Sparsity is the key to context length scaling!

Thanks!