



CS6013: Course Seminar, Fall 2026

Topic: Sparse Attention (Decode)

Instructor: Prof. Aditya Desai

Presenters: Ansh, Sumedh, Bhavesh, Baranee, Amit

Prefill Phase & KV Cache Initialization

Input Prompt Ingestion

Sequence:

Large amount of content as input

Tokens T_1, T_2 to T_N every token is given as input

Parallel Embedding & Projection

Input tokens converted to embedding $X (N \times d_{\text{emb}})$

Matrix multiplication computes Queries, Keys, and Values:

$$Q = XW_q, K = XW_k, V = XW_v (N \times d)$$

High arithmetic intensity (fully compute-bound execution).

KV Cache HBM Storage

Calculated Key K and Value V tensor written directly to High-Bandwidth Memory (HBM).

Establishes the initial KV Cache state for all N context tokens.

Prefill

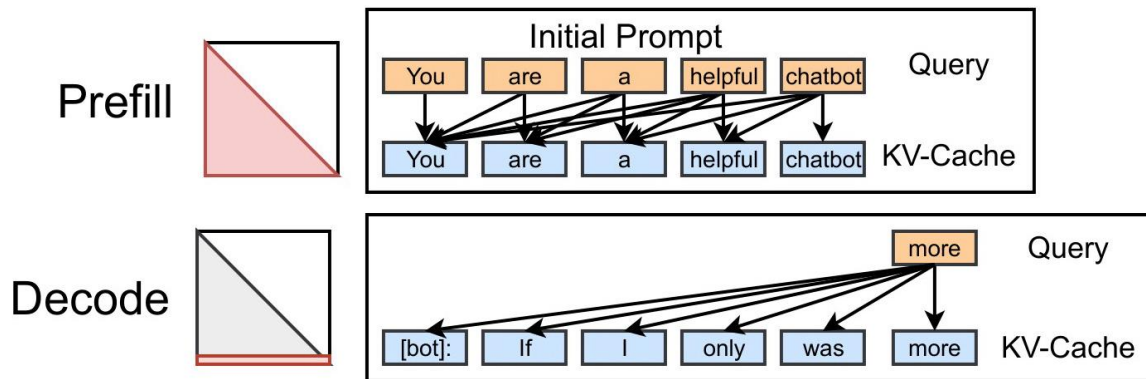
Parallely compute Query Key Value for each token

For each token uses its Query to find attention scores

Stores KV cache in HBM for further use

In Decode Phase

To generate token T_{N+1} , it does a forward pass computing current query's attention to past tokens and so on to predict next tokens



The Decode Phase is Memory-Bound

Sequential Token Generation:

- Generates tokens autoregressively: one token T_{N+1} at a time.
- Input is a single last generated Token.
- Token $T_n \rightarrow \text{embed } X \rightarrow \text{multiplied by } W_q, W_k \text{ and } W_v \text{ matrices} \rightarrow \text{completes the forward pass to generate token}$

Arithmetic Intensity:

- Prefill: **Matrix-Matrix** Multiplication $[X*W \text{ or } X*V] [(N*d)*(d*d)] \rightarrow \text{High FLOPs/Byte (Compute-Bound)}$.
- Decode: **Vector-Matrix** Multiplication $[q*K^T][(1*d)*(d*N)] \rightarrow \text{Low FLOPs/Byte (Memory-Bandwidth Bound)}$.
- Result: GPU Tensor Cores sit idle waiting for memory transfers.

Memory bound problem

To generate token T_{N+1} , the GPU must fetch the entire accumulated KV Cache from High-Bandwidth Memory (HBM) to SRAM and K and V are both $(N*d)$.

Roofline model

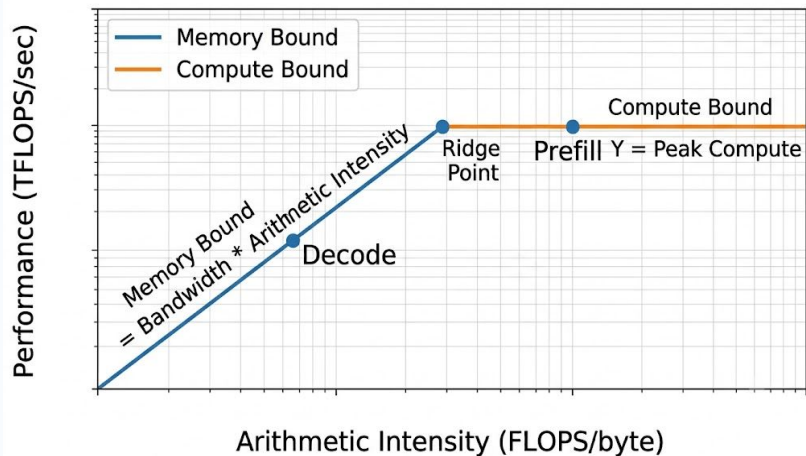
GEMV:

- total elements transferred for computation: $N+N^2=O(N^2)$
- Total operations = $N*(N \text{ multiply}+N \text{ add})=O(N^2)$

GEMM:

- total elements transferred for computation: $N^2+N^2=O(N^2)$
- Total operations = $N^2*(N \text{ multiply}+N \text{ add})=O(N^3)$

Let's look at ratio (Total operations/total elements transferred)



During Prefill we have lot more matrix multiplication thus leading to compute bound, whereas Decode is memory bound

Last 10 years growth in memory vs compute

2016 (Tesla P100): The ridge point was ~30 FLOPs/Byte

2026 (Blackwell B200 - FP16 Dense): The ridge point is ~281 FLOPs/Byte.

Specification	2016: NVIDIA Tesla P100	2026: NVIDIA B200 (Blackwell)	Growth Factor
Memory Bandwidth	732 GB/s	8,000 GB/s (8.0 TB/s)	~11x
FP16 Compute (Dense)	21.2 TFLOPS	2,250 TFLOPS	~106x

Last 10 years growth in memory vs compute

2016 (Tesla P100): The ridge point was ~30 FLOPs/Byte
90*90 matrix was enough to cross the ridge point and fully saturate the GPU's compute cores.

2026 (Blackwell B200 - FP16 Dense): The ridge point is ~281 FLOPs/Byte.

Today, if you multiply matrices smaller than 843*843 the B200's cores will starve. You are entirely memory-bound.

Specification	2016: NVIDIA Tesla P100	2026: NVIDIA B200 (Blackwell)	Growth Factor
Memory Bandwidth	732 GB/s	8,000 GB/s (8.0 TB/s)	~11x
FP16 Compute (Dense)	21.2 TFLOPS	2,250 TFLOPS	~106x

How could we solve it?

The Core Insight:

- Attention distributions a ($1 \times N$) are highly skewed.
- A small fraction of tokens drive the vast majority of the post-attention representation $o = aV$

The Sparsity Objective:

- Instead of fetching all N tokens from the KV Cache in HBM, fetch only k critical tokens ($k \ll N$).
- Reduces memory bandwidth demand from $O(N)$ to $O(k)$ per generation step.

Two Key Questions for Sparse Decode:

Can a sparse attention vector a' perfectly replicate dense attention aV ?

Computational Feasibility:
How do we efficiently find the best k tokens at runtime?

Theoretical Limits: The $(d+1)$ -Sparsity Guarantee

Theorem 1. *Let $V \in \mathbb{R}^{N \times d}$ be any value matrix, and let the dense attention output be $o = V^\top a$, where $a = (a_1, \dots, a_N)$ is an attention distribution over N context tokens. If $d < N - 1$, then this map is not injective on the attention simplex. In particular, there exist two distinct dense attention distributions a, a' such that $a \neq a'$, and $V^\top a = V^\top a'$. Thus, a d -dimensional post-attention embedding cannot preserve all dense attention-score variations over more than $d + 1$ context tokens.*

Intuition:

We are mapping N attention weights down into a d -dimensional output vector. Because V can have a maximum rank of d , the Rank-Nullity Theorem tells us there is a nullspace of dimension at least $(N - 1) - d$. Any shift along this nullspace produces zero change in the output proving that multiple distinct attention score distributions collapse into the exact same post-attention vector.

k-Sparse Attention Selection

Mathematical Optimization Objective:

Find a k-sparse attention vector $\mathbf{b}$ in $\mathbb{R}^N$ that minimizes post-attention reconstruction error:

$$\text{Arg min}_{\mathbf{b}} ||\mathbf{aV} - \mathbf{bV}||_2^2$$

such that $||\mathbf{b}||_0 \leq k$ {atmost k non zero entries}

The Sparsity Objective:

- Selecting the optimal k indices out of N candidate tokens is equivalent to the Sparse Subset Selection problem.
- Exact solution requires searching ${}^NC_k$ combinations which is NP-Hard.
- Exact k-sparse token selection is intractable during real-time autoregressive decoding.

But How?

Randomly or heuristic based

Bypassing NP-Hardness: Exact k-sparse selection is NP-Hard we must use runtime heuristics/approximations.

Selection Strategies:

- Naïve/Baseline: Random sampling or static local context windows.
- Attention-Based (a): Greedily pick tokens with the highest attention scores (a_i).
- Value-Impact (aV): Pick tokens weighted by their representation magnitude $a_i V_i$.

The System Constraint

- Selection cost < Memory savings: If the search overhead to find k takes longer than fetching the full KV cache from HBM, the speedup collapses.

Start of Baranee's Part

Selecting top k tokens to maximise scores:

Lemma 4.1. *Consider n KV embeddings $\mathcal{K}, \mathcal{V}$ and query $\mathbf{q}$. Let $\mathbf{a}$ be the attention scores of tokens w.r.t query $\mathbf{q}$. Then the contribution of a token i , assuming no other token is used, towards the final output is proportional to*

$$a_i \|\mathbf{v}_i\|_2 \quad (3)$$

Proof for Lemma 4.1:

Using the following notations, $\mathbf{V} : n \times d$ value matrix, $\mathbf{a} : n \times 1$ attention scores. $\mathbf{S} : n \times n$ diagonal sampling matrix **with exactly one non-zero** in the diagonal (selection of a single token). Note that this ignores the cross token correlation in value vectors. Then under the sampling the final embedding is,

$$\mathbf{a}^\top \mathbf{S} \mathbf{V} \quad (5)$$

The residual is $\|\mathbf{a}^\top \mathbf{V} - \mathbf{a}^\top \mathbf{S} \mathbf{V}\|_2$. Then the best sampling is the one that minimizes the

$$\mathbf{S} = \operatorname{argmin} \|\mathbf{a}^\top \mathbf{V} - \mathbf{a}^\top \mathbf{S} \mathbf{V}\|_2^2 \quad (6)$$

$$\|\mathbf{a}^\top \mathbf{V} - \mathbf{a}^\top \mathbf{S} \mathbf{V}\|_2^2 \quad (7)$$

$$= (\mathbf{a}^\top (\mathbf{I} - \mathbf{S}) \mathbf{V}) (\mathbf{a}^\top (\mathbf{I} - \mathbf{S}) \mathbf{V})^\top \quad (8)$$

$$= \mathbf{a}^\top (\mathbf{I} - \mathbf{S}) \mathbf{V} \mathbf{V}^\top (\mathbf{I} - \mathbf{S}) \mathbf{a} \quad (9)$$

$$= \sum_{i=1}^n (1 - \mathbf{1}_{S_i}) a_i^2 \|V_{i,:}\|^2 \quad (10)$$

Converting top k tokens problems into MIPS:

Lemma 4.2. *Consider a KV Cache $\mathcal{K}, \mathcal{V}$, and query vector $\mathbf{q}$. The ordering imposed on scores $a_i \|\mathbf{v}_i\|_2$ for i^{th} token is same as the ordering imposed by inner product $\langle [\mathbf{q}, 1], [\mathbf{k}, \log(\|\mathbf{v}\|_2)] \rangle$ where $[]$ denotes concatenation.*

Proof for Lemma 4.2:

$$\begin{aligned} a_i \|v_i\|_2 &= \frac{e^{\langle \mathbf{q}, \mathbf{k} \rangle} \|v_i\|_2}{\sum e^{\langle \mathbf{q}, \mathbf{k} \rangle}} \\ &= \frac{e^{\langle \mathbf{q}, \mathbf{k} \rangle + \log(\|v_i\|)}}{\sum e^{\langle \mathbf{q}, \mathbf{k} \rangle}} \\ &\propto \langle \mathbf{q}, \mathbf{k} \rangle + \log(\|v_i\|) \\ &= \langle [\mathbf{q}, 1], [\mathbf{k}, \log(\|v_i\|)] \rangle \end{aligned}$$

End of Baranee's part

The MIPS Problem

Maximum Inner Product Search over n key vectors

- Given a query $q \in \mathbb{R}^d$ and keys $K \in \mathbb{R}^{n \times d}$, find $\operatorname{argmax} \langle q, k_i \rangle$.
- Exact scan is $O(n \cdot d)$ per query — too slow at long context.
- In LLM decoding, q arrives every token; K is the growing KV cache $\rightarrow$ this scan runs once per generated token.
- Approximate MIPS trades small quality loss for large speedups; families differ in how they prune the candidate set.
- **Deciding factor in LLMs: is the index structure GPU-vectorizable?**

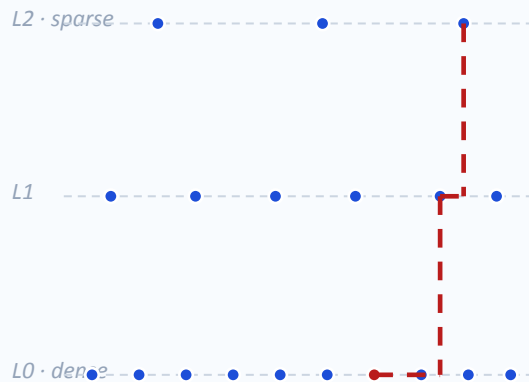
Four families we will cover

1. **Graph-based**
HNSW++
2. **Clustering**
IVF
3. **Quantization**
PQ
4. **Hashing**
LSH / Hamming

Graph-Based: HNSW++

- Multi-layer proximity graph over the keys; top layer sparse, bottom layer dense.
- Query starts at a coarse entry node, greedily moves toward higher-inner-product neighbors at each layer.
- Descent shrinks the candidate set as layers get denser; final layer returns the approximate top-k.
- Complexity $\sim O(\log n)$ per query for moderate dimensions.
- **Best for: high-d, medium n, recall-critical workloads.**
- **Limitation: pointer-chasing access pattern is hard to vectorize on GPU.**

Greedy descent through layers

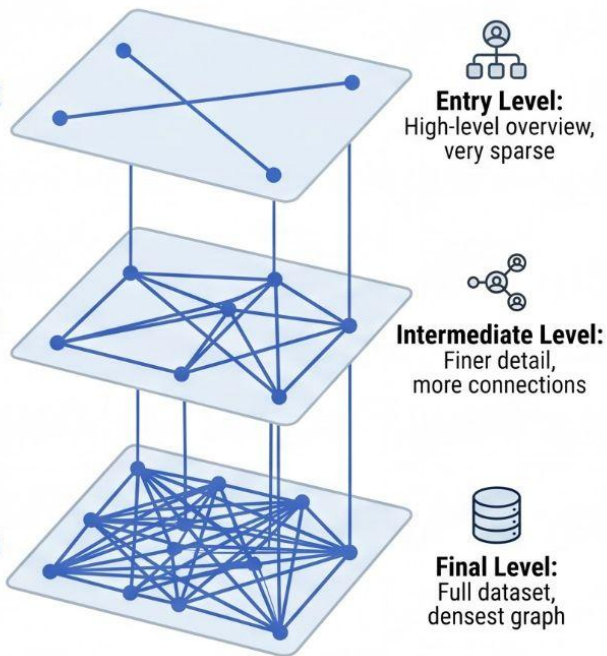


Entry → **winner**

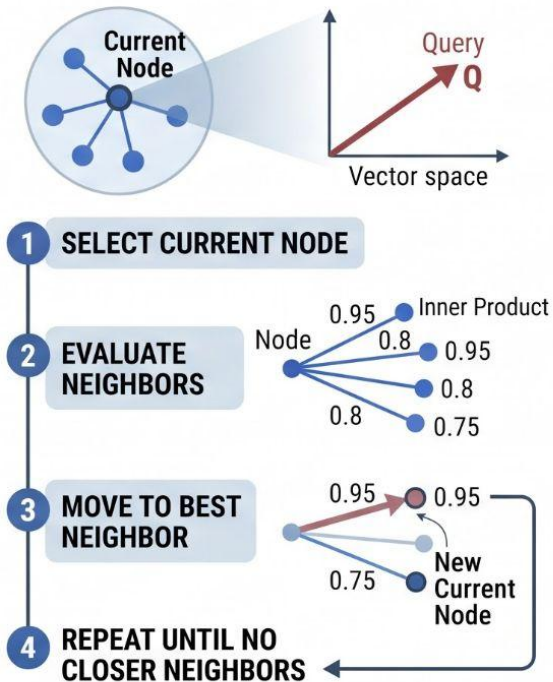
NSW++ WORKING PROCESS:

HIERARCHICAL APPROACH TO APPROXIMATE MIPS

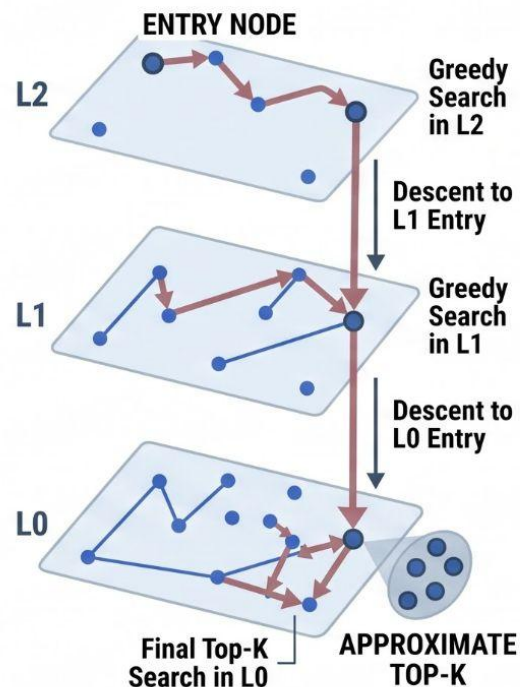
LAYERED GRAPH ARCHITECTURE



INTERNAL GREEDY SEARCH (Per Layer)



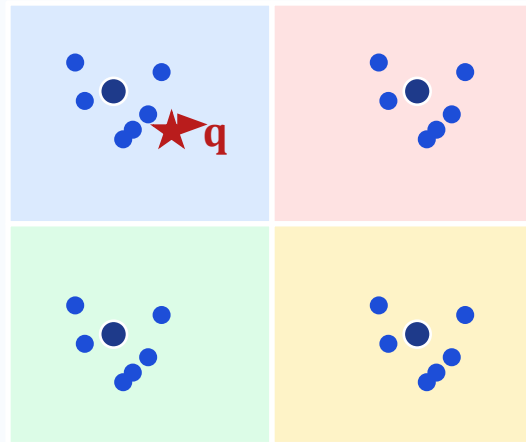
HIERARCHICAL DESCENT



Clustering:

- Run k-means on the keys $\rightarrow$ partition into k clusters; store centroid + member list per cluster.
- At query time, compute $\langle q, \text{centroid} \rangle$ for all k centroids, probe the top- p nearest clusters, scan only their members.
- Complexity $\sim O(n/k)$ when probing one cluster; tune p to trade recall for speed.
- **Best for: very large n where a full graph is too expensive to build or store.**
- **Limitation: imbalanced clusters and queries near boundaries degrade recall; retrain when keys drift.**

Partition + probe nearest cluster

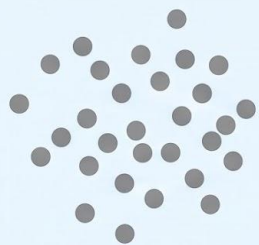


probe this cluster $\rightarrow$ scan members

Training & Index Construction (Pre-computation)

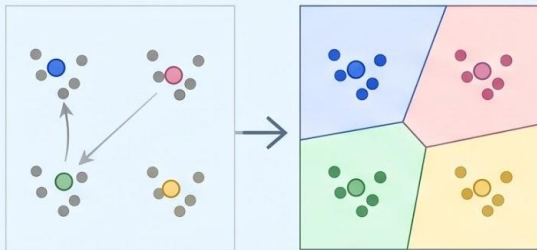
Pre-computation

Input Keys



Input Keys

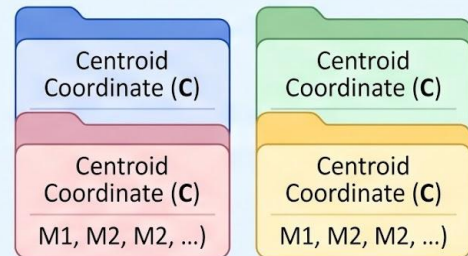
Run Clustering (K-Means)



Run **Clustering** (K-Means)

Find natural groups in your data.

Build Search Index



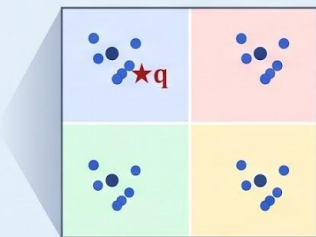
Build Search Index

Store cluster centroids and member lists.

Search & Retrieval (Query Time)

Search & retrieval

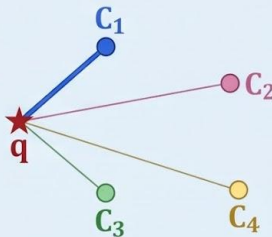
Query Point (q) Arrives



Query Point (q) Arrives

Find nearest neighbors for 'q'.

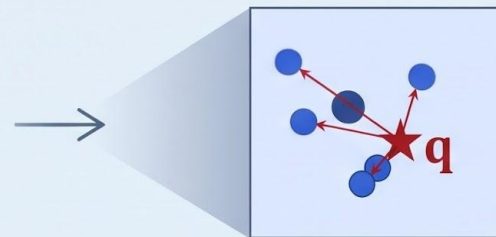
Centroid Probing



Centroid Probing

Compare 'q' with centroids only.
Fast and cheap.

Cluster Scan (Final Search)



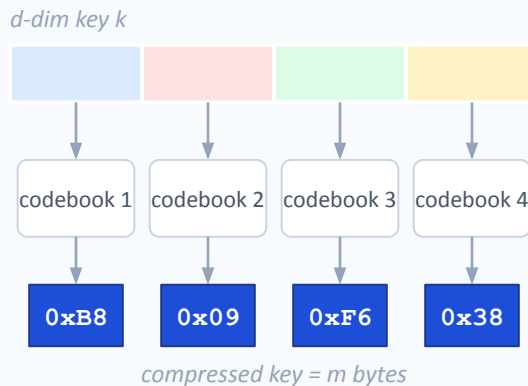
Cluster Scan (Final Search)

Scan members of that single sub-group.
Accurate results.

Product Quantization (PQ)-->add ex fig (Will remove)

- Split each d -dim key into m sub-vectors of length d/m .
- Train a small codebook (e.g. 256 centroids) per sub-vector position.
- Each key $\rightarrow m$ codebook indices (m bytes for 256-size codebooks) — dramatically shrinks the cache.
- Asymmetric Distance Computation (ADC): q stays full-precision; per-subspace distances come from a precomputed LUT, summed to approximate $\langle q, k \rangle$.
- **Best for: memory-bound settings; $\sim O(n)$ but tiny constants and excellent cache locality.**
- **Limitation: quantization error caps achievable recall.**

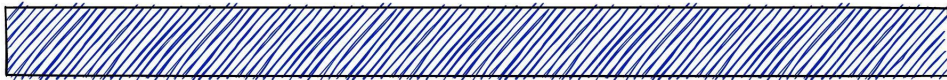
Split $\rightarrow$ quantize $\rightarrow$ lookup



$LUT(q): \langle q, \text{sub-centroid} \rangle$ for all 256 codes $\times$
 m subspaces $\rightarrow \Sigma \approx \langle q, k \rangle$

Product Quantization (PQ)

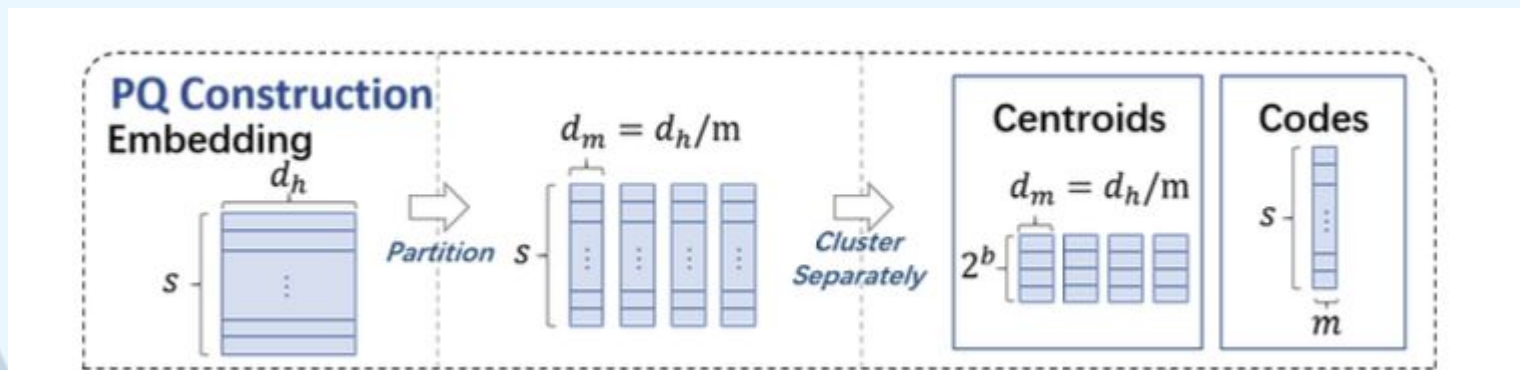
- Take a big, high-dimensional vector.
- Split it into equally sized chunks or subvectors
- Assigning each of these subvectors to its nearest *centroid* (also called reproduction/reconstruction values),
- Replacing these centroid values with unique IDs — each ID represents a centroid



<https://www.pinecone.io/learn/series/faiss/product-quantization/>

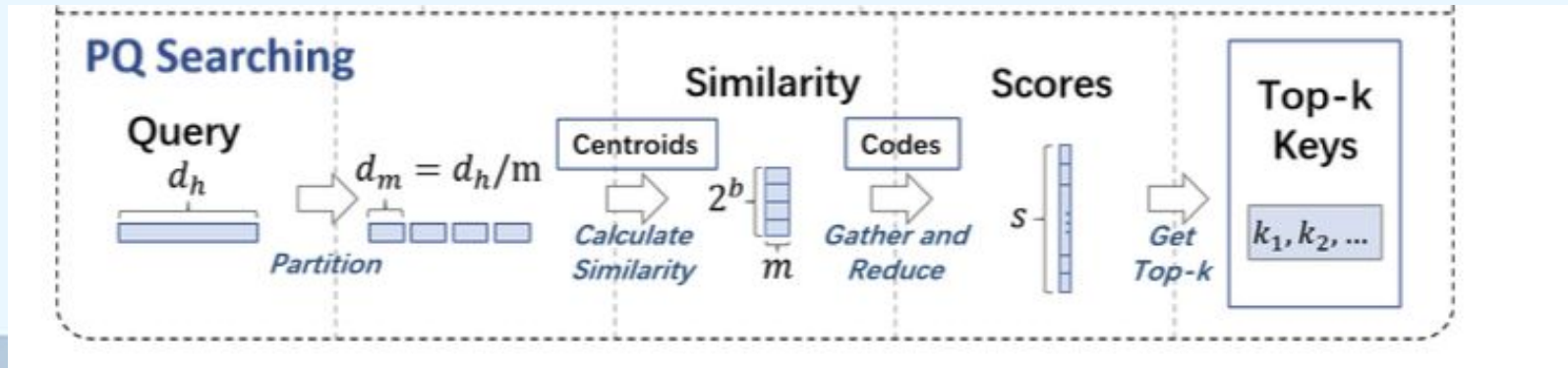
PQ Cache: PQ Construction

- Define $K = 2^b$, this represents the number of centroids uniquely represented by the bit level code b
- Partition d_h into m subparts and cluster each partition into 1 of K centroids across sequence length
- Off load FP16 KV to CPU and keep codes on GPU. Memory reduction by a factor $mb/(16d_h)$; $m=2, K=64 \Rightarrow b=6, d_h=128$, factor reduction ~ 170

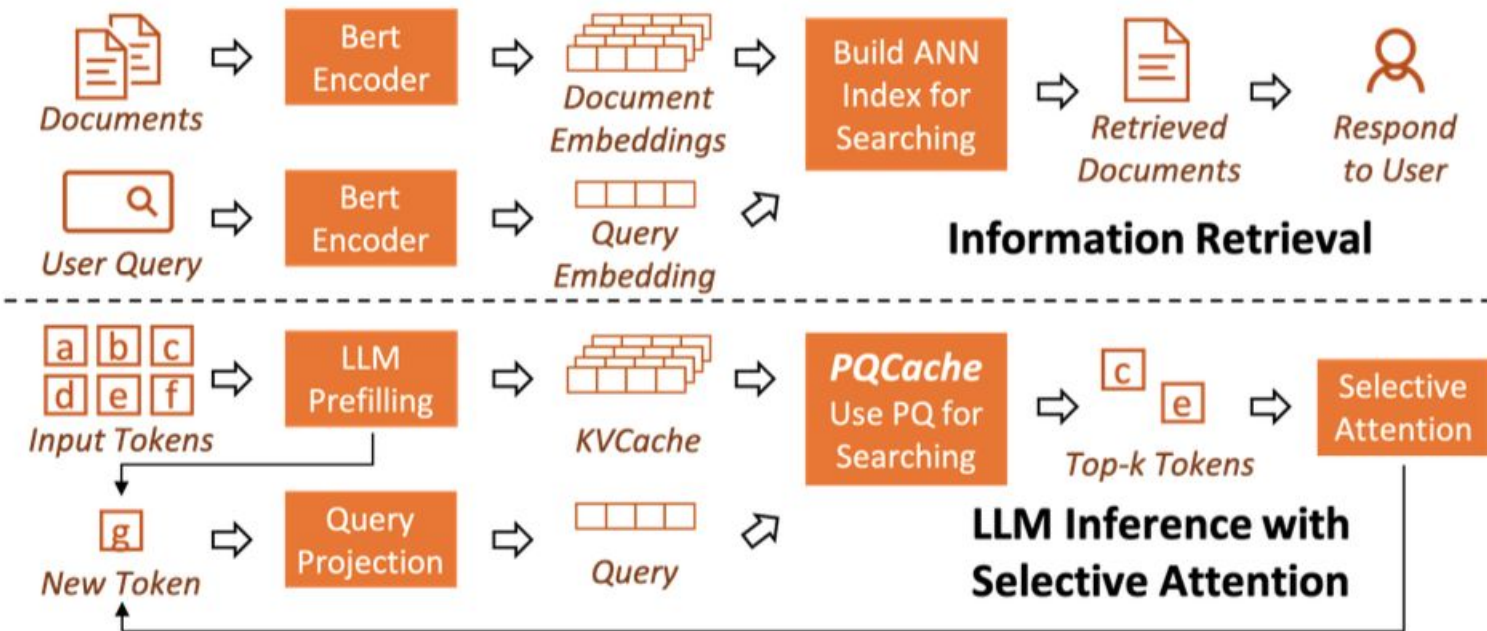


PQ Cache: PQ Searching

- For every new Query, partition it into m partitions as before.
- Compute similarity and construct LUT. (on GPU)
- Get approximate attention scores
- Choose top- k keys based on these scores (fetched from CPU)
- Retrieve top- k KV pairs
- AllKV = initial KV(GPU) + top- k KV (Fetched from CPU) + Local KV(GPU)

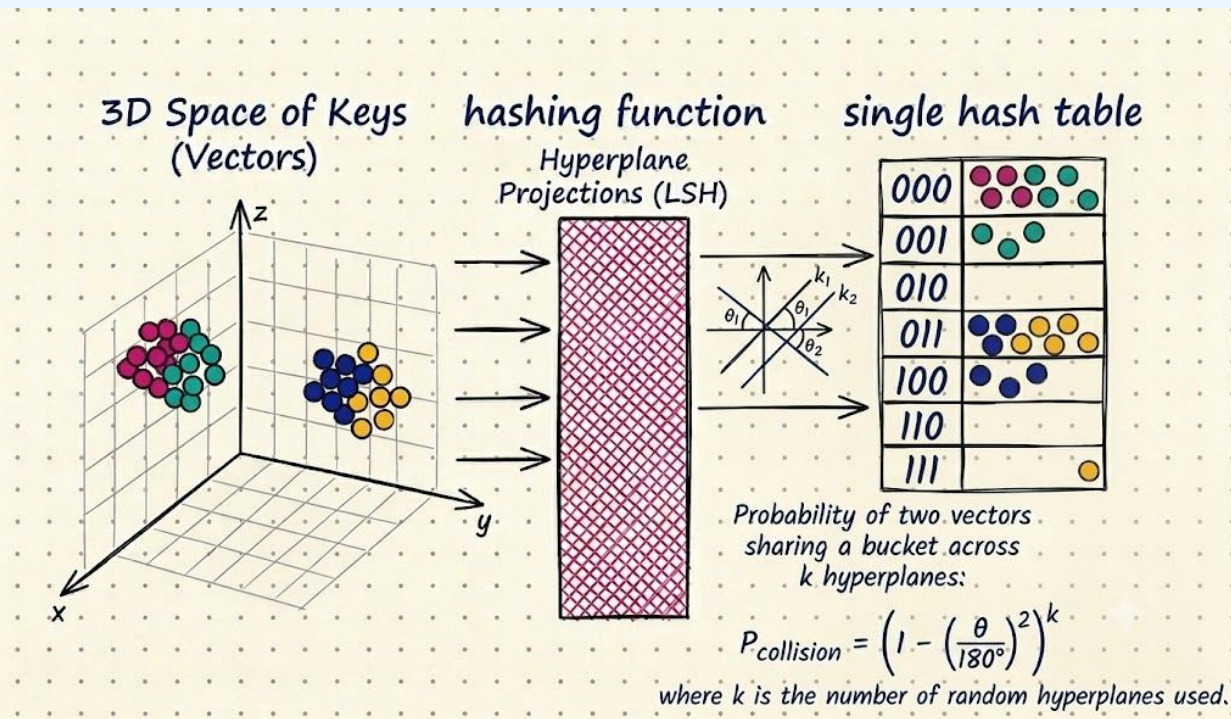
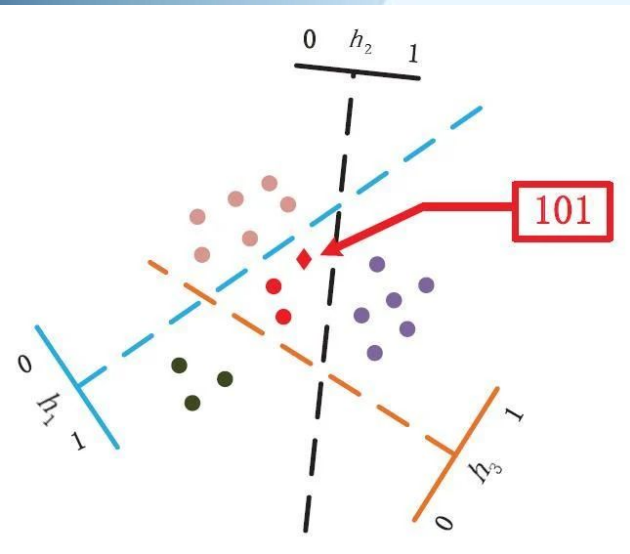


PQ Cache: Parallels between IR and LLM Inference



Locality-Sensitive Hashing

- Sample L random hyperplanes; for each vector, record on which side it lies $\rightarrow$ an L -bit signature.



MIPS Methods at a Glance

Method	Speed	Accuracy	Memory	GPU-friendly
HNSW++ (graph)	$\sim O(\log n)$	High recall	Moderate (graph)	Poor (pointer-chasing)
IVF (clustering)	$\sim O(n/k)$	Moderate	Low (centroids)	Moderate $\rightarrow$ more GPU friendly
Product Quantization	$\sim O(n)$, tiny const	Capped by quant err	Very low (codes)	Excellent (LUT)
LSH / Hamming	$\sim O(n / 2^L)$	Moderate (tables)	Low (bit codes)	Excellent (popcount) $\rightarrow$ not GPU friendly

No free lunch — speed and recall trade off; GPU-friendliness often decides what is deployable in LLM decoding.

From MIPS to GPU-Aware Sparse Attention

- During autoregressive decoding, per-token cost is dominated by reading the KV cache from GPU HBM (memory-bound, not compute-bound).
- Sparse attention reduces KV reads by attending only to a subset of tokens — i.e. a MIPS query per head per token.
- Index structure must be GPU-vectorizable: avoid pointer-chasing; prefer popcount, LUT lookups, batched matmul.
- Three methods exemplify this design point — covered next:

HashAttention · PQCache · SqueezeAttention

Bottleneck =

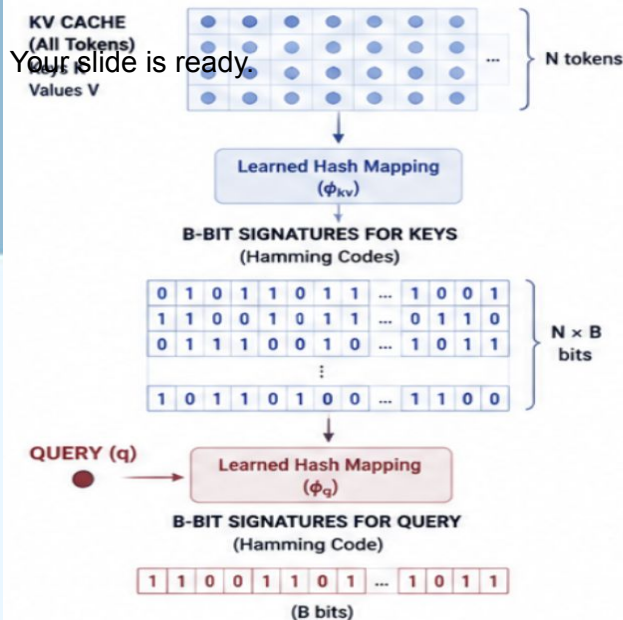
HBM bandwidth, not FLOPs

Index op must be

a single kernel launch

HashAttention:

LAYERED ARCHITECTURE (PURE HASHATTENTION)



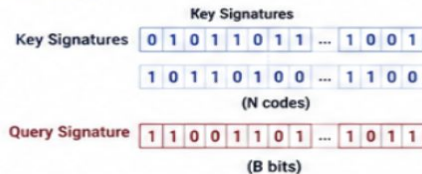
● Key / Value Token ● Query □ Binary Bit (0/1)

N : Total number of tokens B : Hash code length (e.g., 32)

PURE HASHATTENTION: GREEDY SEARCH IN HAMMING SPACE (PER LAYER)

1 PROJECT KEYS & QUERY

Map all keys and the query to B-bit Hamming space using learned mapping functions.



2 COMPUTE HAMMING DISTANCE

For each key code, compute Hamming distance to the query code.

Key Index	Key Code	Hamming Distance
0	0 1 0 1 1 0 1 1 ... 1 0 0 1	3
1	1 1 0 0 1 0 1 1 ... 0 1 1 0	1
2	0 1 1 1 0 0 1 0 ... 1 0 1 1	2
...	...	...
N-1	1 0 1 1 0 1 0 0 ... 1 1 0 0	4

3 RANK KEYS

Rank all keys in ascending order of Hamming distance.

4 SELECT TOP-K KEYS

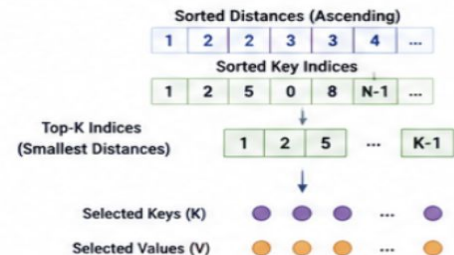
Select the Top-K keys with the smallest Hamming distances.

5 GATHER K & V

Gather keys and values corresponding to the selected indices.

6 ATTENTION COMPUTATION

Compute scaled dot-product attention using only the selected Top-K tokens.



$$\text{softmax}\left(\frac{QK_{\text{topK}}^T}{\sqrt{d_k}}\right)V_{\text{topK}}$$

Repeat at every layer during inference.

PQCache → algo can be discussed + pgnum on all slides

- Store the KV cache in PQ-compressed form (m codebook indices per key) — fits much longer contexts in HBM.
- Per query: build an $m \times 256$ lookup table of $\langle q, \text{sub-centroid} \rangle$ values; sum across subspaces → approximate $\langle q, K[i] \rangle$ for every i in one pass.
- Take top- k of the approximate scores; fetch full-precision K, V rows only for those k tokens.
- **Strength: scan is a tight LUT + reduce kernel — extremely GPU-friendly.**
- **Limitation: ADC error caps recall; codebook must be retrained when key distribution drifts.**

Flow

1 · PQ codes
 m bytes per key



2 · LUT(q)
 $m \times 256 \langle q, \text{sub-centroid} \rangle$



3 · top- k IDs
from approximate $\langle q, K \rangle$

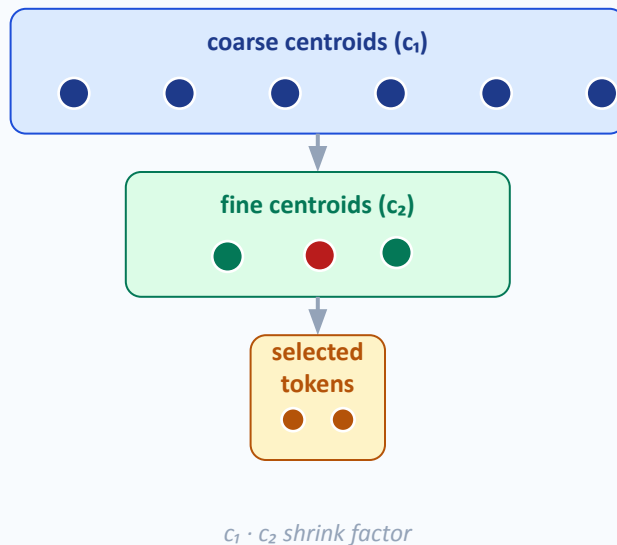


4 · fetch full K, V → attention
read-back only k rows

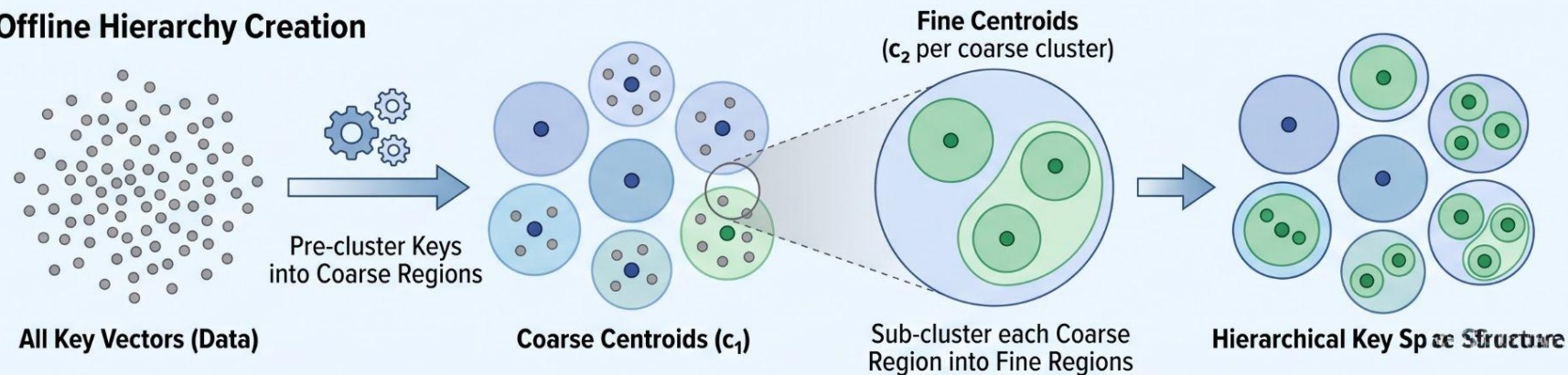
SqueezeAttention (Hierarchical)

- Offline: run 2-tier k-means on the keys — first into c_1 coarse clusters, then each coarse cluster into c_2 fine clusters.
- Online per query:
 - compute $\langle q, \text{coarse_centroid} \rangle$ for all c_1 ; keep top few
 - for each kept cluster, compute $\langle q, \text{fine_centroid} \rangle$ for its c_2 children; keep top few
 - finally scan member tokens of selected fine clusters
- Effective candidate set shrinks by $\sim c_1 \cdot c_2$ vs full scan.
- **Strength: pure matmul at every tier $\rightarrow$ fully GPU-vectorized.**
- **Limitation: cluster boundaries cause recall loss; suffers when attention is flat.**

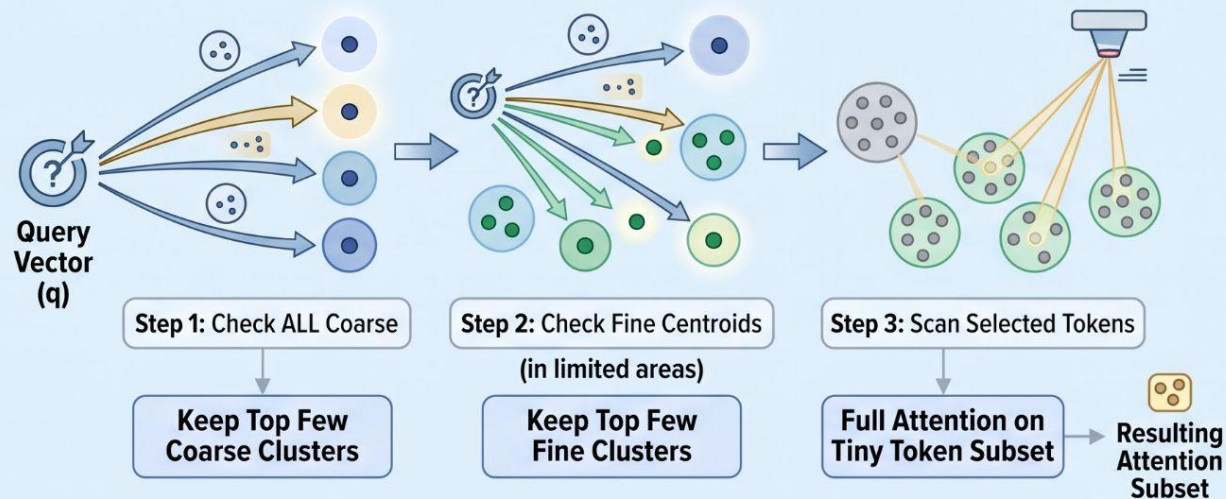
3-tier narrowing



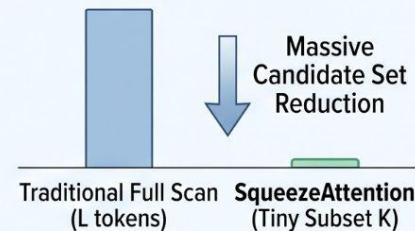
Offline Hierarchy Creation



Online Query Retrieval



ADVANTAGES & EFFICIENCY



- Pure Matmul:** Fully GPU-Vectorized for max speed.
- Dramatically Faster:** Queries are processed much quicker.
- Candidate Set Shrinks by $\sim c_1 \cdot c_2$.**

Fixed K vs Dynamic K (Y Dynamic K?)

FIXED K

- Same budget k per query, every head, every layer.
- Simple, predictable latency.
- Misses long-tail tokens when scores are flat; wastes compute when scores are sharp.
- **Examples:** HashAttention, PQCache (with fixed top- k).

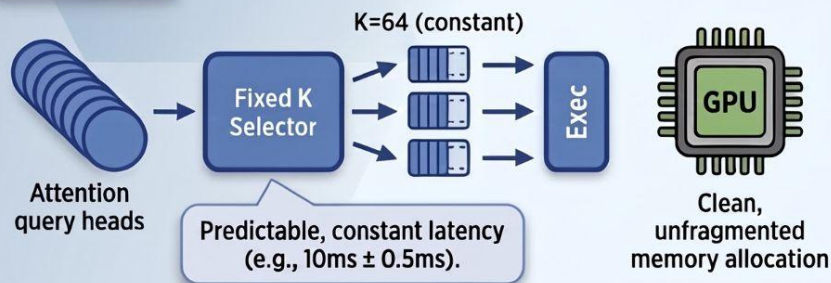
DYNAMIC K

- Budget adapts per head per query — include token i iff $\langle q, k_i \rangle$ exceeds threshold τ , or budget chosen so cumulative attention mass $\geq p$ (top- p).
- Better quality-latency fit to per-query score distribution.
- Harder to predict latency; needs extra pass to estimate τ or p .
- **Examples:** top- p variants, vAttention (budget set by (ϵ, δ) guarantee).

vAttention = Dynamic K, but with a *verified* accuracy guarantee.

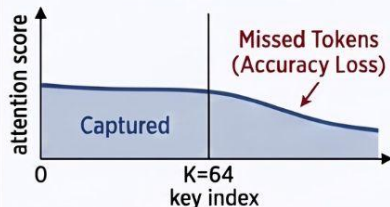
FIXED K

MECHANISM FLOWCHART



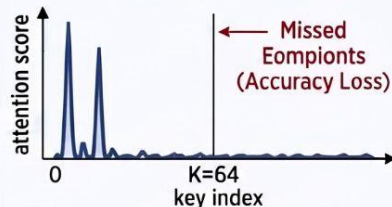
SCORES VISUALIZATION

FLAT SCORES



MISSES ACCURACY ON FLAT DISTRIBUTION.

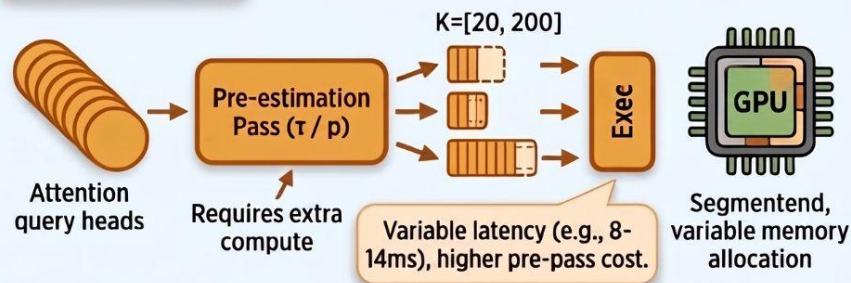
HEAVY HITTERS



WASTES COMPUTE ON ZERO SCORES.

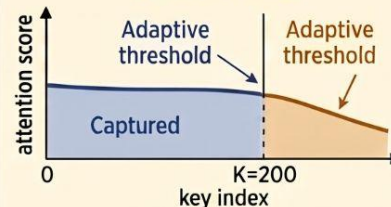
DYNAMIC K

MECHANISM FLOWCHART



SCORES VISUALIZATION

FLAT SCORES



ADAPTS TO CAPTURE ACCURACY.

HEAVY HITTERS



SAVES COMPUTE ON SHARP DISTRIBUTIONS.

Missed Emponents

Is Top-k enough?

LIMITATIONS OF TOP-K

- Top-K methods assume attention scores are concentrated, but they fail on aggregation tasks requiring full-context integration.
- Top-K keeps rank, throws away mass.

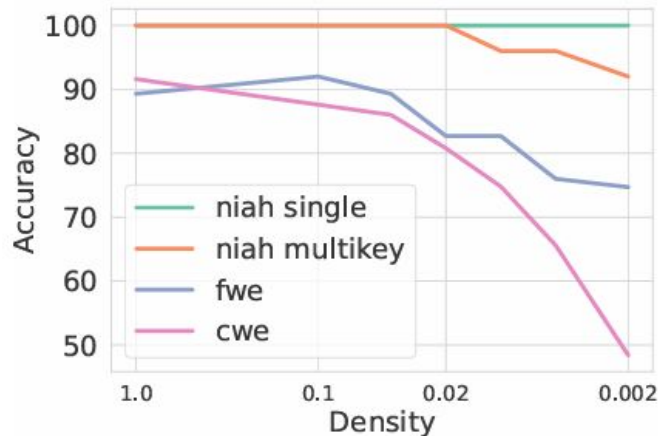
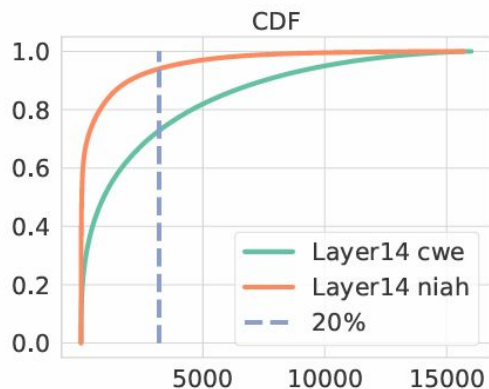


Figure 1 While TopK attention performs well on retrieval tasks (niah) where the useful information reduces to a few words, it degrades severely in aggregated tasks like word extraction (cwe, fwe). x-axis: proportion of attention keys used for TopK attention.

Why is Top-k not enough?

1. Tail Truncation

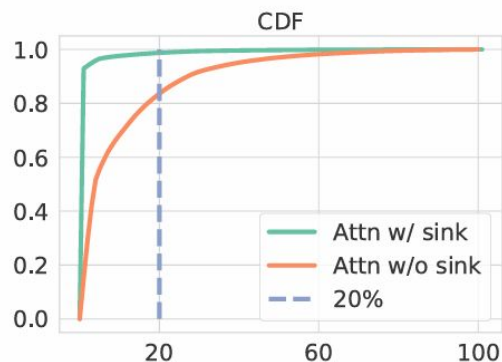
Top-K truncates the tail with a large tail of tokens collectively carrying substantial attention mass.



(a) Long tailed phenomena

2. Sparsity Illusion

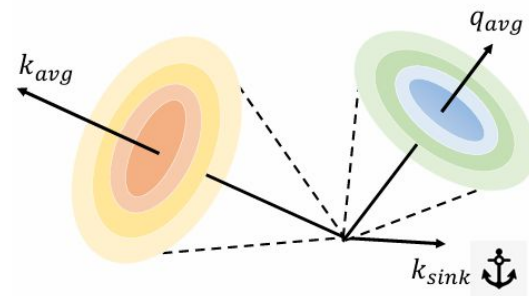
The **attention sink** creates the **illusion of sparsity**, while removing it reveals much more distributed attention.



(b) Attention sink reshapes sparsity

3. Geometric Mismatch

- Keys and queries lie within narrow cones of opposite orientations, except for the attention sink.
- This mismatch degrades nearest-neighbor search performance.



(c) Geometry of attention

MagicPIG : Reframe Attention as an Expectation

CLASSICAL SOFTMAX ATTENTION

$$o = \text{Softmax}\left(\frac{qK^T}{\sqrt{d}}\right)V = wV \quad q \in \mathbb{R}^{1 \times d} \quad K, V \in \mathbb{R}^{n \times d} \quad w \in \mathbb{R}^{1 \times n}$$

BECOMES

1. FORMULATION AS AN EXPECTATION

$$o = \mathbb{E}_{i \sim w} [v_i]$$

2. MONTE CARLO APPROXIMATION

$$i_1, \dots, i_B \stackrel{\text{iid}}{\sim} w \quad \bar{o} = \frac{1}{B} \sum_{j=1}^B v_{i_j}$$

Oracle Sampling needs “w”

— Importance sampling removes that requirement

Estimator:

$$X_{IS} = \frac{1}{\tilde{Z}} \sum_{j=1}^{\mathcal{B}} \frac{\tilde{w}_{ij}}{u_{ij}} v_{ij},$$

Unnormalized Weight:

$$\tilde{w}_i = \exp\left(\frac{qk_i^T}{\sqrt{d}}\right)$$

Normalizing Constant $\tilde{Z}$:

$$\tilde{Z} = \sum_{j=1}^{\mathcal{B}} \frac{\tilde{w}_{ij}}{u_{ij}}$$

Asymptotic Consistency:

$$\Pr\left[\lim_{\mathcal{B} \rightarrow \infty} X_{IS} = o\right] = 1$$

Estimator Variance (eq. 7):

$$\text{Var}(\tilde{X}_{IS}) = \frac{1}{\mathcal{B}\tilde{Z}^2} \mathbb{E}_{i \sim u} \left[\frac{\tilde{w}_i^2}{u_i^2} (v_i - o)^2 \right]$$

Optimal Proposal Dist.:

$$u_i^* \propto \tilde{w}_i |v_i - o|$$

Locality-Sensitive Hashing

Hyperplane Partitioning

Uses $K \times L$ random projection hyperplanes W to segment high-dimensional space into hash buckets.

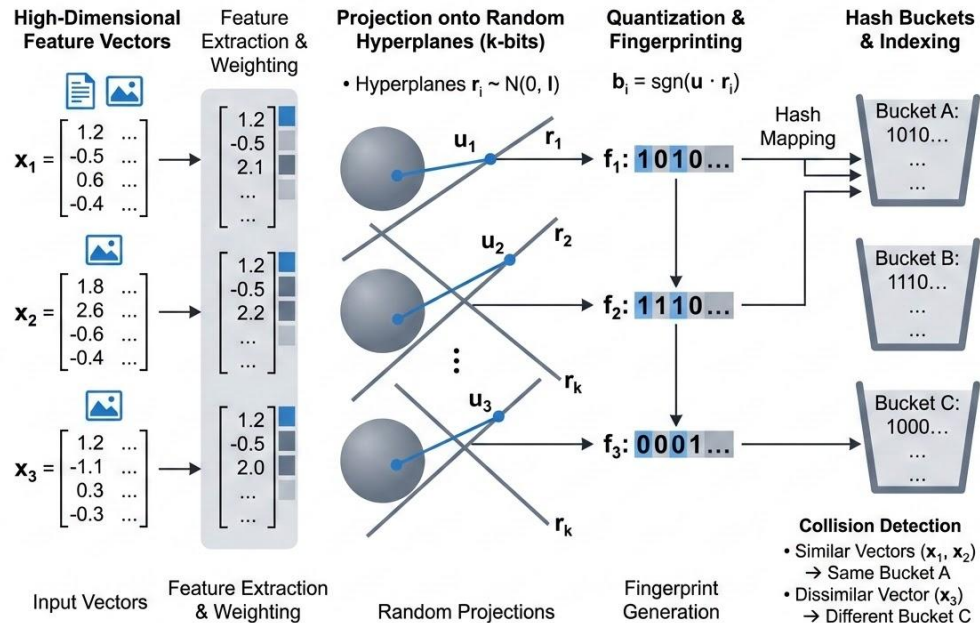
Code Calculation

Computes hash codes mapping high-dimensional key vectors to a K-bit integer fingerprint.

Our proposal “u”

Pick the keys which share the hash with the query in atleast 2 tables

SIMHASH LOCALITY SENSITIVE HASHING DIAGRAM

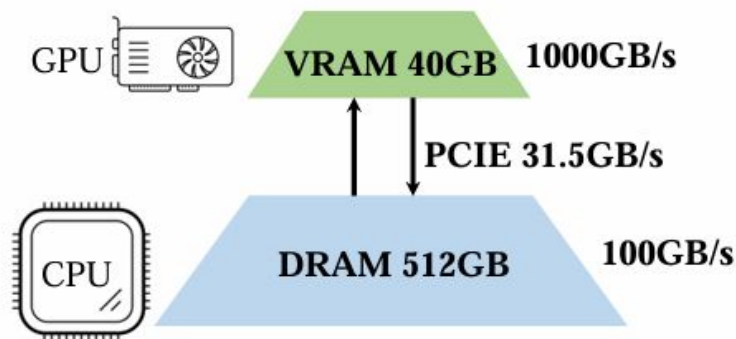


$$p_i = 1 - \frac{1}{\pi} \arccos \frac{qk_i^T}{\|q\| \|k_i\|},$$

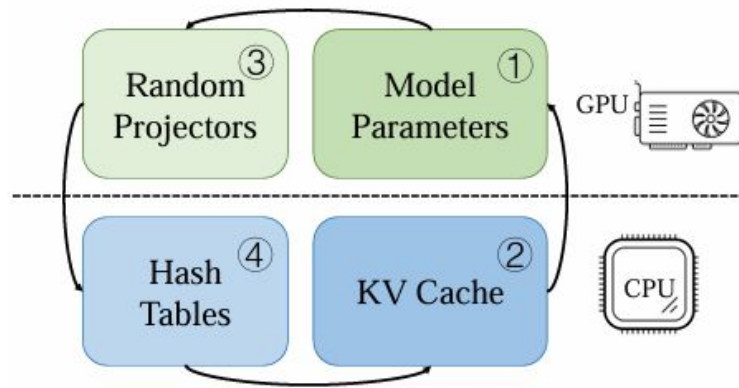
$$u_i = 1 - (1 - p_i^K)^L - Lp_i^K(1 - p_i^K)^{L-1}$$

inclusion prob. - sample iff ≥ 2 of L tables collide

System Co-Design: splitting work across CPU, GPU



(a) Memory Hierarchy



(b) Workload Partition

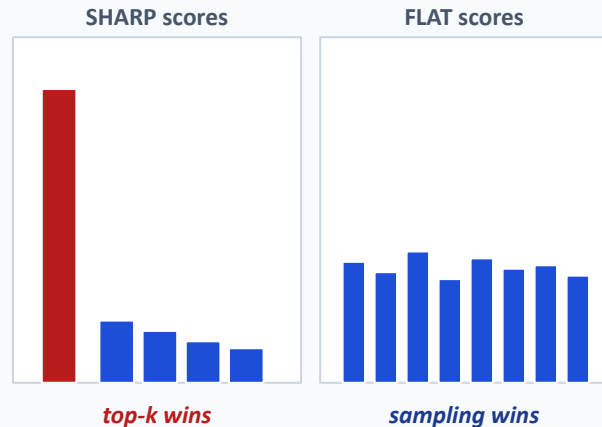
Figure 7 Left: Memory hierarchy of hardware. GPU VRAM has high bandwidth but is limited. CPU DRAM is sufficient but is relatively slow. The limited bandwidth of PCIe forbids large-scale data transfer. **Right:** Workload partition of MAGICPIG. Linear projections and hash function computation (by random projection) are done on GPU while sampling with hash tables and attention are done on CPU. The execution order is ①③④② at decoding time.

Ansh

vAttention: Why Verified?(emphasize need for random sampling)

- Top-k assumes a few heavy hitters dominate — it fails when attention scores are flat (long tail matters).
- Random sampling handles flat distributions well — it fails when a few outliers dominate (wastes samples on noise).
- Insight: top-k and random sampling are complementary — combine them.
- vAttention = sinks + sliding window + approximate top-k (heavy hitters) + adaptive uniform sample (long tail), with sample size chosen to guarantee (ϵ, δ) accuracy.
- **First practical sparse attention with user-specified, provable approximation guarantee.**

When does each strategy win?



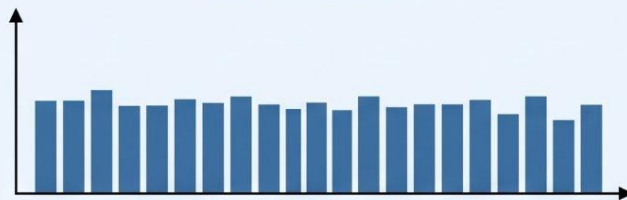
(ϵ, δ) guaranteed

ATTENTION SCORE SCENARIOS (Illustrative Data)

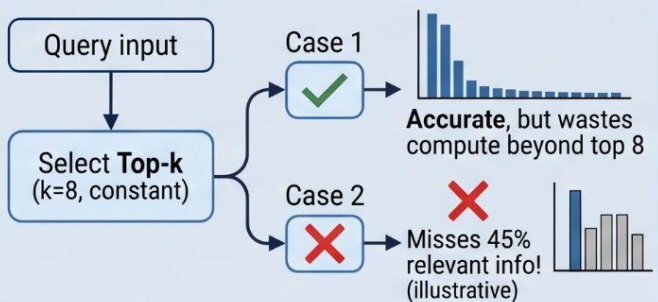
Case 1: Heavy Hitters (Sharp Scores)



Case 2: Flat Scores



FIXED K APPROACH



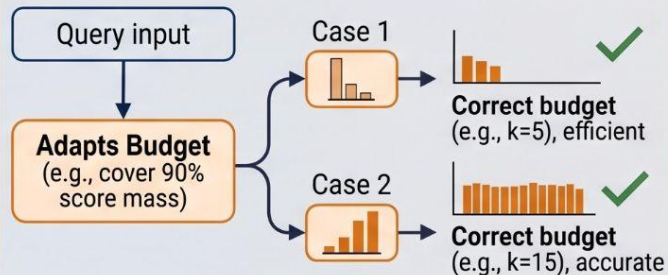
PROS: Simple implementation, predictable latency.

CONS: Low accuracy with flat scores, inefficient with sharp scores.



Latency:
10ms (fixed)

DYNAMIC K APPROACH



PROS: Better accuracy-compute fit, handles diverse distributions.

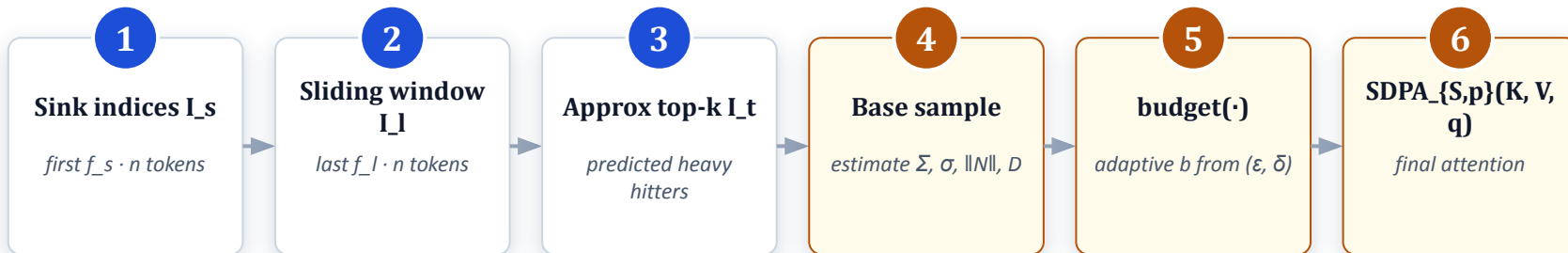
CONS: Harder to predict latency, extra passes needed.



Latency:
5ms - 15ms (variable)

FEATURE	FIXED K	DYNAMIC K
Accuracy (diverse inputs)		
Compute Efficiency		
Latency Predictability		

vAttention Algorithm (Algo. 1)



Deterministic part

sinks + local + top-k cover the heavy hitters

Stochastic part

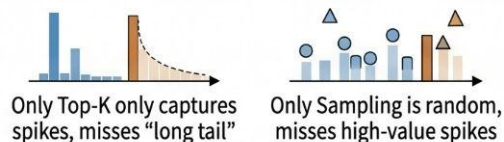
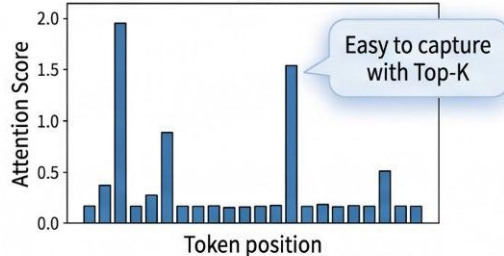
uniform sample from residual n_s tokens

GPU-friendly

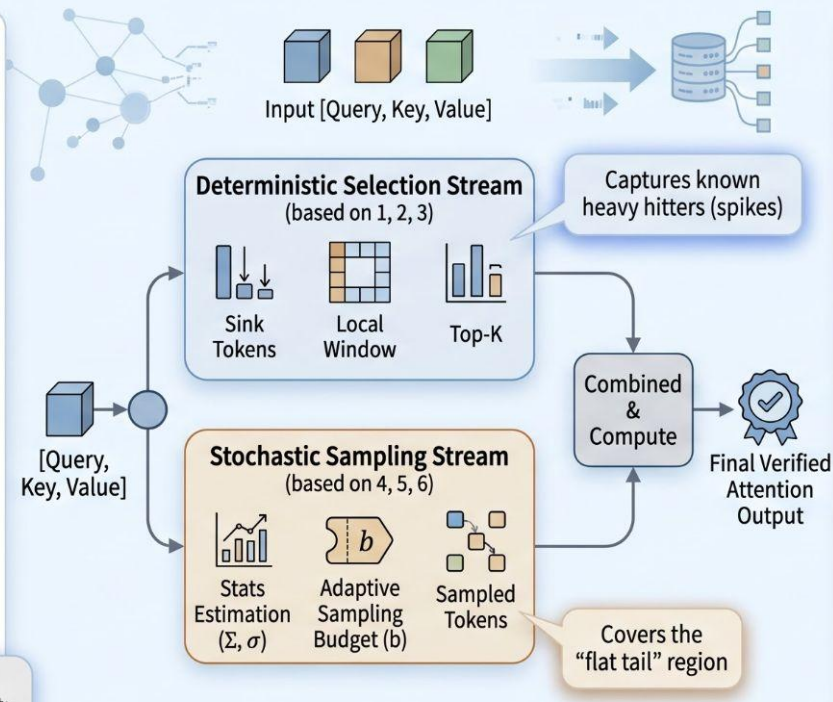
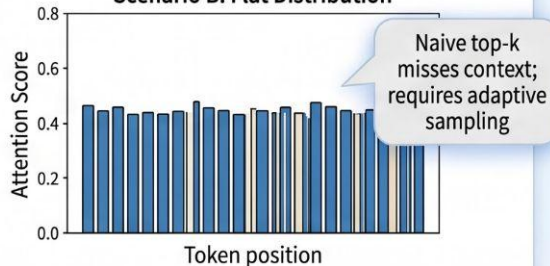
budget b computed entirely on GPU from base-sample stats

Understanding Attention Score Distributions

Scenario A: Heavy Hitter Distribution



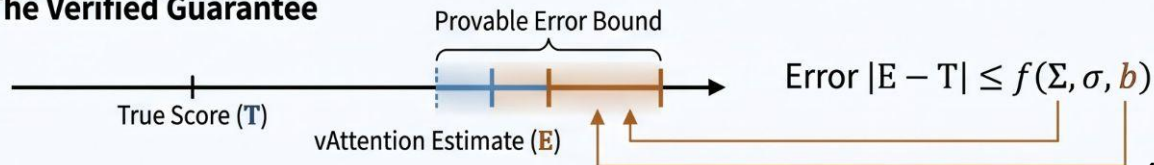
Scenario B: Flat Distribution

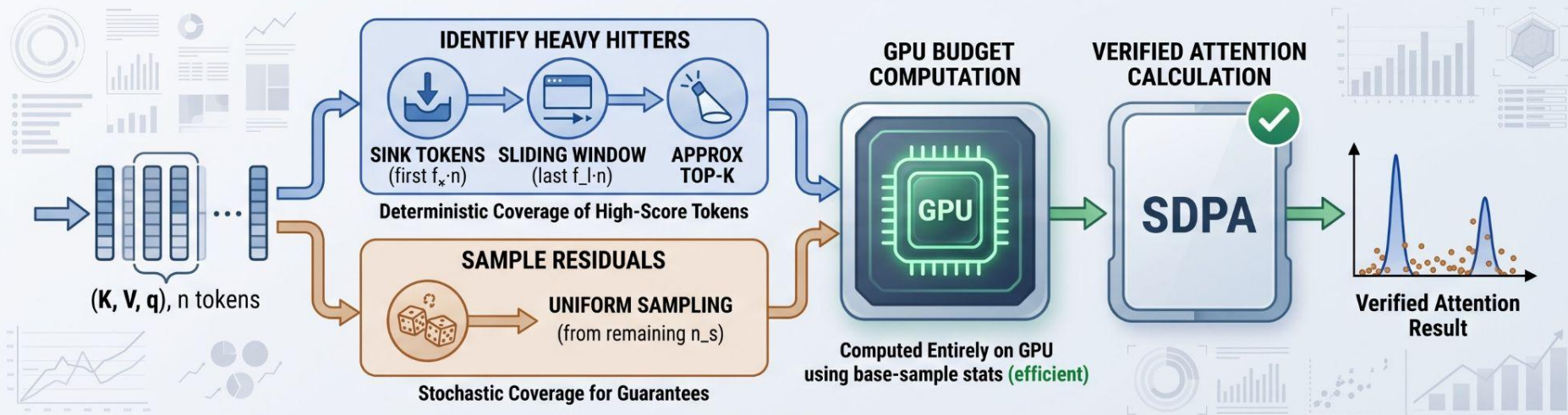


Why not only X? and the Advantages

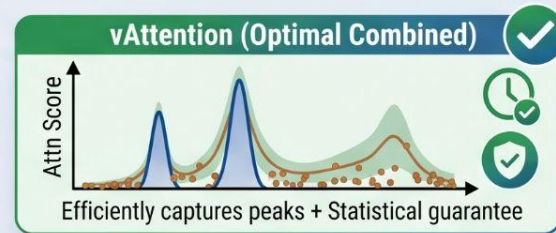
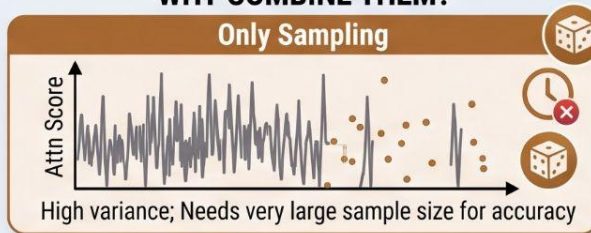
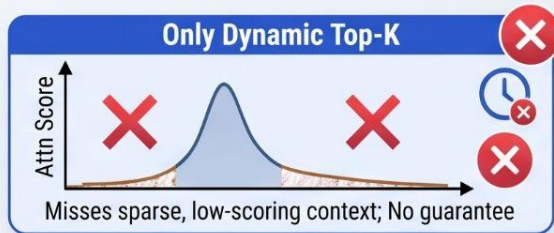
	Dynamic Top-K only	Stochastic Sampling only	vAttention (Verified)
1. Capture Heavy Hitters	✓	✗	✓
2. Cover Flat Tail	✓	✗	✓
3. Speed & GPU-Friendly	✓	✓	✓
4. Provable Error Bound	Yes	No	Guaranteed

The Verified Guarantee

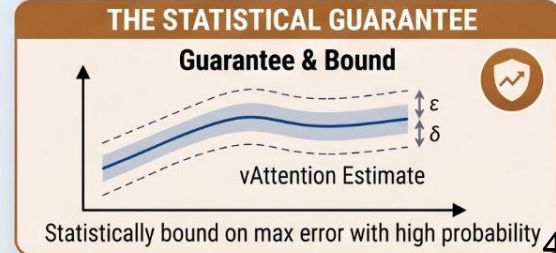
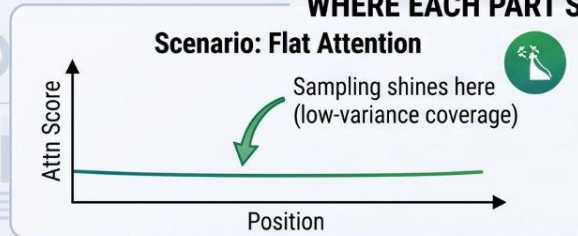




WHY COMBINE THEM?



WHERE EACH PART SHINES & THE GUARANTEE



Foundations: Estimating a Sum via Sampling(modify equations for visibility)

LEMMA 4.1 — VECTOR-SUM ESTIMATOR

Let $\mathbf{s} = \sum_{i=1}^{n_s} \mathbf{r}_i \in \mathbb{R}^d$ be a sum of n_s vectors to be estimated from a sample I_b of size b . Let Σ be the covariance of $\{\mathbf{r}_i\}$. The unbiased estimator is:

$$\hat{\mathbf{s}}_b = (n_s / b) \cdot \sum_{i \in I_b} \mathbf{r}_i \quad \text{CLT} \Rightarrow b \geq \lceil \Phi^{-1}(1 - \delta/2) \cdot \sqrt{(n_s \cdot \text{Tr}(\Sigma)) / \tau^2} \rceil \Rightarrow \Pr(\|\hat{\mathbf{s}} - \mathbf{s}\|_2 > \tau) \leq \delta$$

1 Two-knob control

τ sets the error tolerance; δ sets the failure probability.

2 Why $\text{Tr}(\Sigma)$?

Spread of the residuals — wider spread needs more samples.

3 Scaling

$b \propto \text{Tr}(\Sigma) / \tau^2$: halving the error quadruples the budget.

4 Next

We apply this lemma to the numerator N to derive b_N .

The Estimator: The scaled estimate is given by $\hat{s}_b = \frac{n_s}{b} \sum_{i \in I_b} r_i$.

The Guarantee: By leveraging the **Central Limit Theorem (CLT)**, if your sample size b satisfies:

$$b \geq \left(\Phi^{-1} \left(1 - \frac{\delta}{2} \right) \frac{n_s \sqrt{\text{Tr}(\Sigma)}}{\tau} \right)^2$$

then the probability that the error vector exceeds a tolerance threshold τ is bounded by δ :

$$\Pr(\|\hat{s} - s\|_2 > \tau) \leq \delta$$

ϵ - δ Proof Sketch

Lemma 4.2: Combining Numerator and Denominator Approximations

What it says: Scaled Dot Product Attention consists of both a **Numerator** (N) (a sum of vectors weighted by attention scores) and a **Denominator** (D) (a sum of scalar attention scores). Lemma 4.2 explains what happens when you approximate *both* simultaneously.

The Guarantee: Suppose your numerator budget ensures an (ϵ_1, δ_1) approximation and your denominator budget ensures an (ϵ_2, δ_2) approximation (where $\epsilon_2 < 0.5$). If you pick a joint sample budget $b = \max(b_D, b_N)$, the relative error between the true attention vector $\frac{N}{D}$ and your approximated attention vector $\frac{\hat{N}}{\hat{D}}$ is bounded by:

$$\Pr \left(\left\| \frac{N}{D} - \frac{\hat{N}}{\hat{D}} \right\|_2 > 2(\epsilon_1 + \epsilon_2) \left\| \frac{N}{D} \right\|_2 \right) \leq (\delta_1 + \delta_2)$$

Verified Attention — Theorem 4.3 (Add: why top k is imp?)

LEMMA 4.2 · COMBINE N + D

The same Lemma 4.1 template applied to the scalar denominator $D = \sum \exp\langle K[i], q \rangle$ yields $b_D(\epsilon, \delta, \sigma, D)$. If b_N gives (ϵ_1, δ_1) on the numerator and b_D gives (ϵ_2, δ_2) on the denominator ($\epsilon_2 < 0.5$), then $b = \max(b_N, b_D)$ ensures:

$$\Pr(\|N/D - \tilde{N}/D\|_2 > 2(\epsilon_1 + \epsilon_2) \cdot \|N/D\|_2) < \delta_1 + \delta_2$$

THEOREM 4.3 · (ϵ, δ) -VERIFIED SDPA

Choosing $b \geq \min\{\epsilon', \delta'\} \max(b_D(\epsilon'/2, \delta'), b_N((\epsilon-\epsilon')/2, \delta-\delta'))$ guarantees:

$$\Pr(\|v\text{Attention}(K, V, q) - \text{SDPA}(K, V, q)\|_2 > \epsilon \cdot \|\text{SDPA}(K, V, q)\|_2) \leq \delta$$

PRACTICAL RELAXATION

In deployment, vAttention uses a **denominator-only relaxation**: the denominator bound alone assigns per-head budgets and shows **>0.99 correlation** with observed errors. Cheaper because the denominator is a scalar (small base sample suffices) and accurate denominator estimates reduce bias, which otherwise compounds across sequence length.

Amit's part begins from next slide

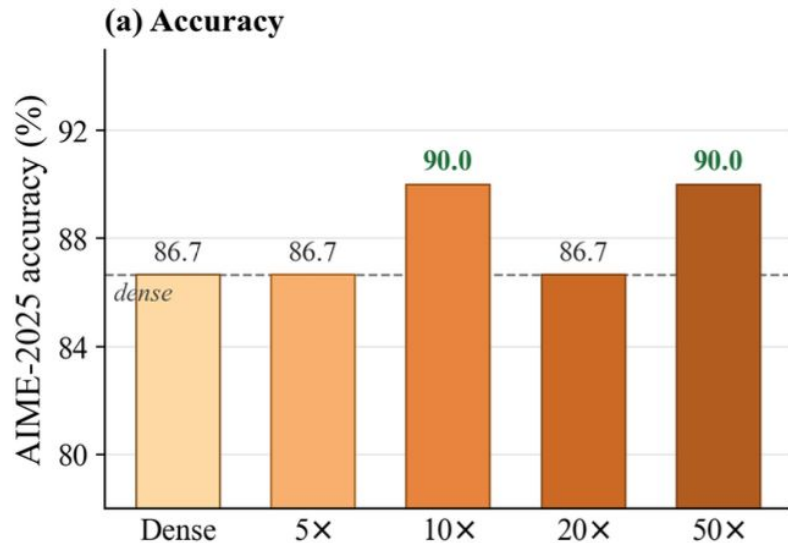
AIME results with sparsity (Qwen-3.5-27B)

Find n if $133^5 + 110^5 + 84^5 + 27^5 = n^5$.

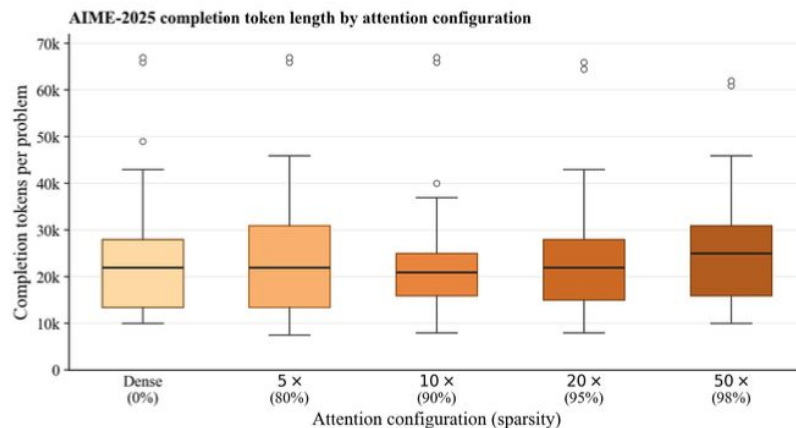
Google Images for query "AIME exemplar problems"

Let x, y, z be positive numbers such that $xyz = 1$.

$$x + \frac{1}{z} = 5, y + \frac{1}{x} = 29, z + \frac{1}{y} = ?$$

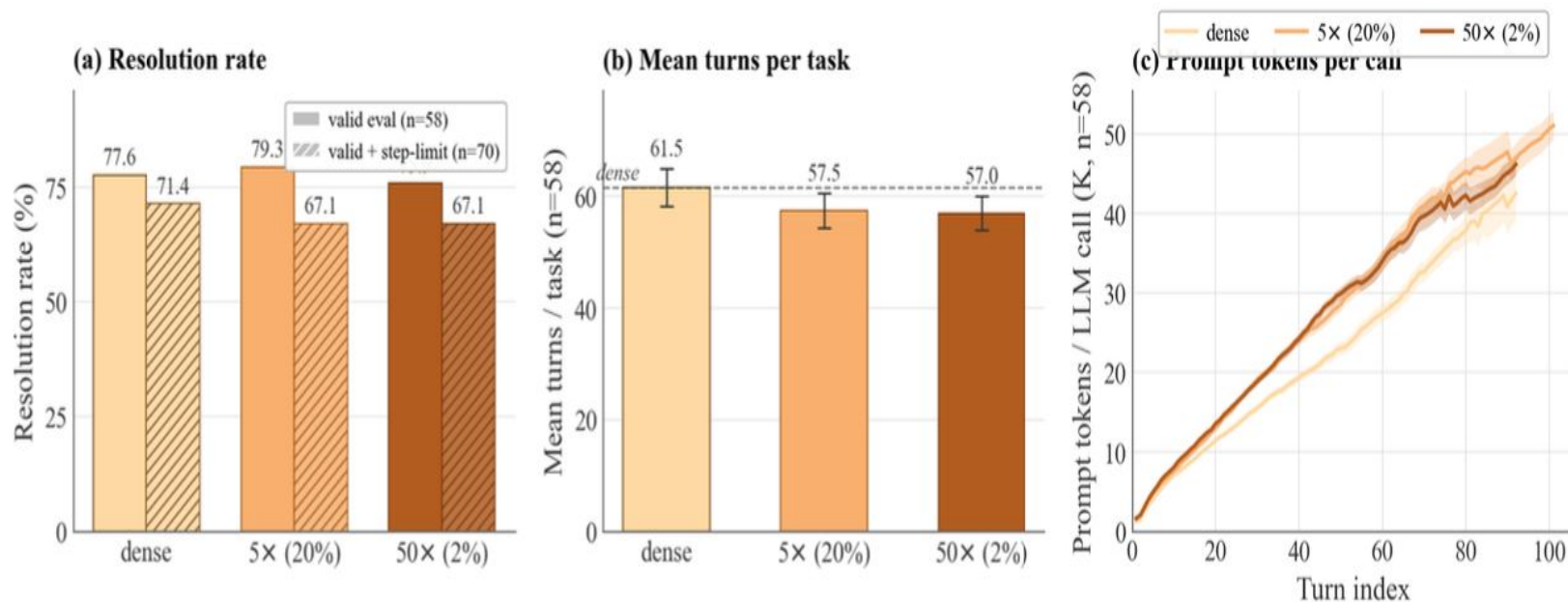


65K token length hard cap



SWE Results with Sparsity

SWE-Bench Django subset(114), hard limit 250 turns per task



Sparsity and Hardware

- Traditional View: To achieve speeds ups at hardware level block sparsity should be present
- Contradicting evidence: Even with unstructured sparsity (per token, per head, per layer) custom kernels can reap the benefits of sparse attention.

Table 1: Per-query, per-head irregular sparse decode is up to 76× faster than FlashInfer at extreme sparsity Speedup over FlashInfer [51] on H100 80GB HBM3 (FP16, GQA $H_q=32$, $H_{kv}=8$, $D=128$, page size 16, NHD, 128K context). $S\times$ sparsity implies each query-head attends to $1/S$ fraction of total tokens. The Speed up of $< 1\times$ denotes overhead. We generally break-even at $10\times$ sparsity for every batch with huge speedups of $10\text{--}76\times$ in the $50\text{--}500\times$ regime. Note that no block structure imposed on the sparsity.

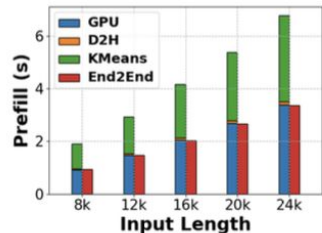
B	FlashInfer (ms)	2x	4x	10x	20x	50x	100x	200x	500x
1	0.19	0.32×	0.63×	1.45×	2.58×	5.57×	10.25×	11.05×	11.14×
4	0.72	0.33×	0.66×	1.64×	3.18×	7.45×	13.36×	24.25×	42.04×
8	1.50	0.38×	0.77×	1.90×	3.75×	8.88×	16.82×	29.64×	76.14×
16	3.08	0.45×	0.89×	2.21×	4.35×	10.54×	20.09×	37.32×	76.77×

Sparsity and Hardware

Table 2: Sparse decode is net-positive with indexer cost included. We use DoubleSparsity as an example here. Speedup over FlashInfer on H100 (FP16, 128K, page size 16) using Double Sparsity [48] (8×16 -bit channels, untuned) for index selection, under MHA ($H_q=H_{kv}=32$) and GQA ($H_q=32, H_{kv}=8$). MHA: break-even at $2\times$ sparsity with $4.17\times$ speedup at $100\times$ sparsity. GQA: break-even at $10\times$, with $2.81\times$ speed up at $100\times$. A lighter indexer (HashAttention, PQCache, low-precision Double Sparsity) is expected to widen the margin. The upper limit can be seen from Table 1

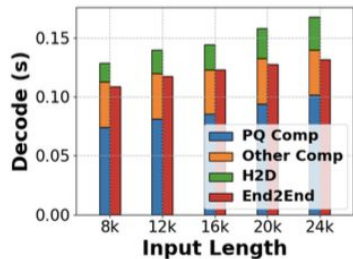
<i>B</i>	FlashInfer (ms)	2x	5x	10x	20x	50x	100x
<i>GQA</i> ($H_q=32, H_k=8$)							
1	0.19	0.28×	0.56×	0.83×	1.12×	1.46×	1.65×
4	0.72	0.32×	0.67×	1.06×	1.52×	2.11×	2.45×
8	1.50	0.36×	0.75×	1.18×	1.68×	2.30×	2.66×
16	3.08	0.41×	0.85×	1.31×	1.82×	2.46×	2.81×
<i>MHA</i> ($H_q=H_k=32$)							
1	0.70	0.91×	1.62×	2.18×	2.68×	3.14×	3.37×
4	2.79	1.02×	1.86×	2.56×	3.17×	3.74×	4.00×
8	5.61	1.12×	2.00×	2.71×	3.32×	3.86×	4.11×
16	11.18	1.22×	2.13×	2.84×	3.43×	3.94×	4.17×

PQ Cache Advantages

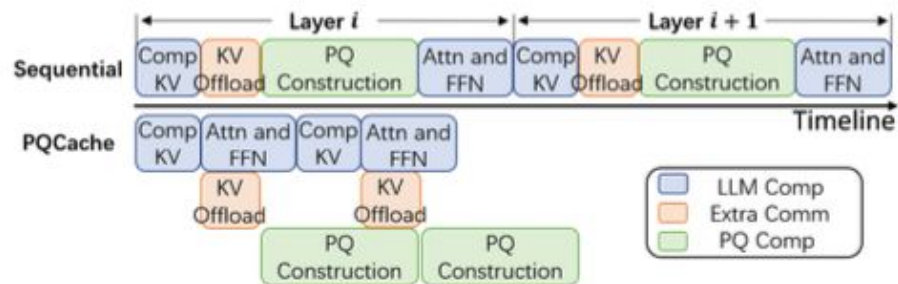


(a) Prefilling time.

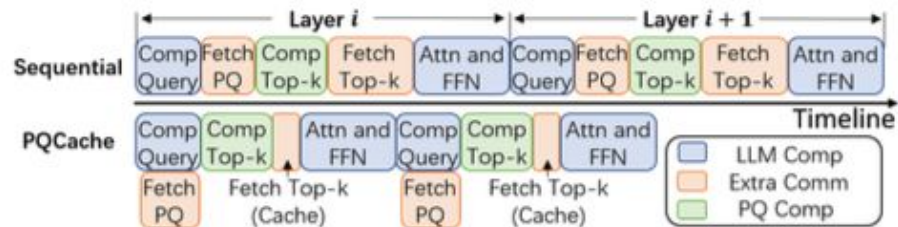
⇒ Task: HotPotQA,
 ⇒ SAR: 0.2,
 ⇒ Model: Mistral-7B-IT-v0.2
 ⇒ Conf: GQA
 ⇒ Native CW = 32K tokens



(b) Decoding time.



(a) Prefilling.



(b) Decoding.

Complexity Analysis (Prefill and decode)

- Prefill

- Memory footprint (MFPT) reduction = $mb / (16 * dh)$
- Normal attention compute complexity (CC) (seq to seq + projection) = $O(dh * s^2 + s * d^2)$
- K means clustering (Additional compute): $O(s * T * hkv * dh * 2^b)$, Linear in s, can be offloaded to CPU $\Rightarrow$ no additional computational burden on GPU

- Decode

- Original attention computational complexity: $a = \text{softmax}(qK^T)$; CC = $O(s * dh)$; MFPT = $O(s * dh)$
- For PQ cache, construct $LUT[j, c] = q(j^{\text{th}} \text{ partition}) * u(j^{\text{th}} \text{ partition}, c^{\text{th}} \text{ centroid}) \Rightarrow CC = MFPT = O(m * dm * 2^b)$
- Then, $a_{\text{apprx}} = \text{softmax}(\sum_j \{LUT[j, u(j, c)]\})$; for this and prev step $\Rightarrow CC = O(dh * 2^b + s * m)$; MFPT = $O(m * 2^{b/8} * dm + m * s * b / 8)$
- Based on a_{apprx} retrieve top-k KV pairs and get final attention $\Rightarrow MFPT = O(k * dh)$, CC = $O(s * \log k)$
- Reduction factor in MFPT = dh / m (only consider coefficient of s because context length is dominant, ignore $k * dh$ as top k is very small compared to full KV)

PQ Cache (Limitations)

- Fixed number of centroids.
- Initial K means clustering overhead may require some adaptive iterative scaling.
- With longer context length, the PQ needs to be reconstructed.
- Appropriate CPU-GPU ratio; no bottlenecks from either side
- Other IR techniques like HNSW may provide more efficient retrieval.

End of Amit's part

Add part of what can be offloaded to CPU and what benefits it leads. Focus on systems part. In general: Reduce text, add figures,

References (not exhaustive)

- HashAttention: Semantic Sparsity for Faster Inference, Desai et al., 2025
- MagicPIG: LSH Sampling for Efficient LLM Generation, Chen et al., 2024
- PQCache: Product Quantization-based KVCache for Long Context LLM Inference, Zhang et al., 2025
- vAttention: Verified Sparse Attention, Desai et al., 2026
- Squeezed Attention: Accelerating Long Context Length LLM Inference, Hooper et al., 2025
- Inference Time Context Sparsity: Illusion or Opportunity?, Joshi et al., 2026
- <https://www.pinecone.io/learn/series/faiss/product-quantization>

Thank You, Questions?

When someone asks: Why are you being
so dense?

Tell 'em: Because I cannot (efficiently)
manage sparsity induced overheads!