OSDI 26

ECHO: Efficient KV Cache Offloading with Lossless Prefetching for Serving Native Sparse Attention LLMs

Guangda Liu[1], Wenhao Chen[1], Chengwei Liu[1], Zhenyu Ning[1],
Jing Lin[2], Yiwu Yao[2], Quan Chen[1], Shixuan Sun[1], Jieru Zhao[1*], Minyi Guo[3,1]

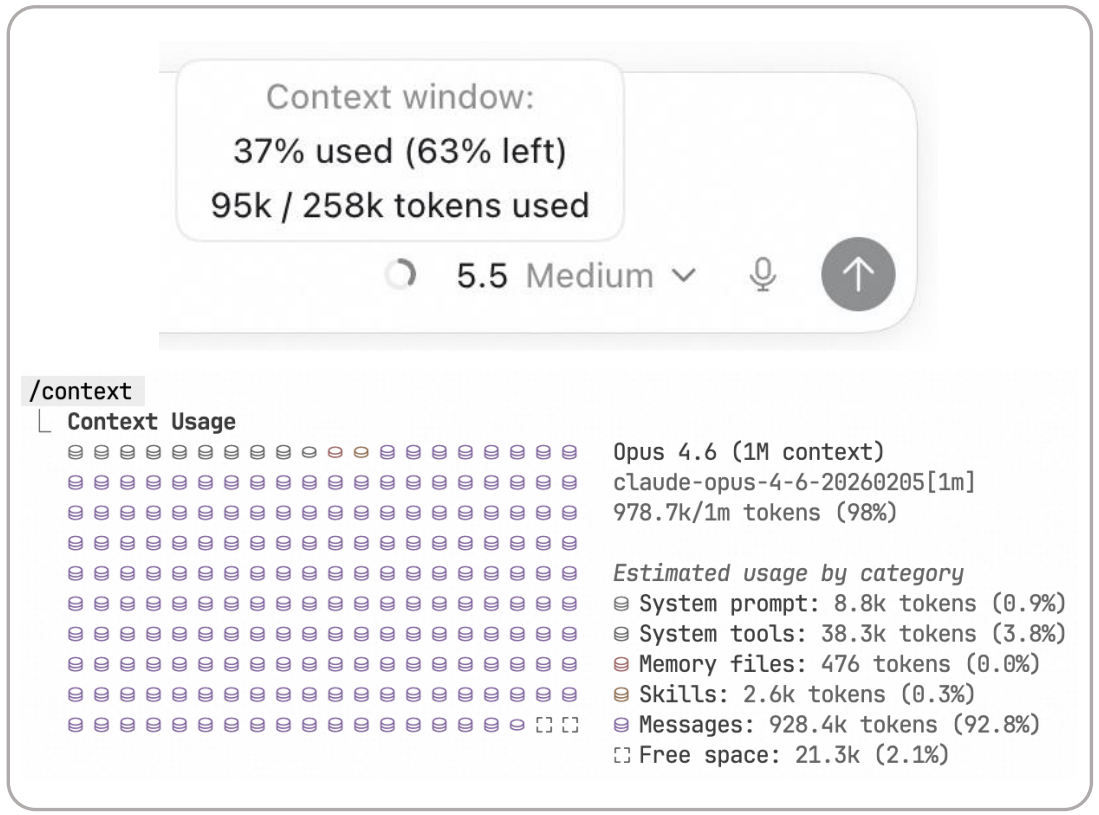
[1]School of Computer Science, Shanghai Jiao Tong University [2]Huawei
[3]Guizhou Provincial Laboratory of Big Data, Guizhou University

*Corresponding author: Jieru Zhao (zhao-jieru@sjtu.edu.cn)

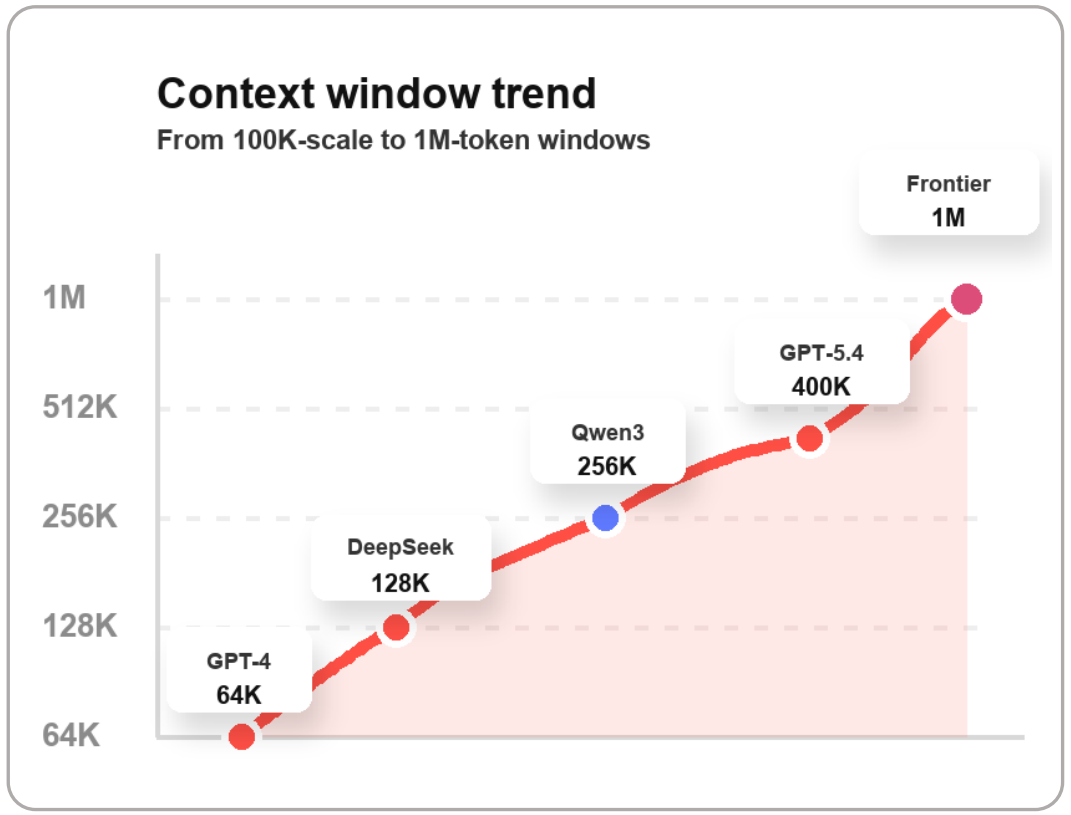


上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Long Context Inference Is Becoming Normal



Application Side



Model Side

1M-token context is becoming a must-have for today's agentic workloads

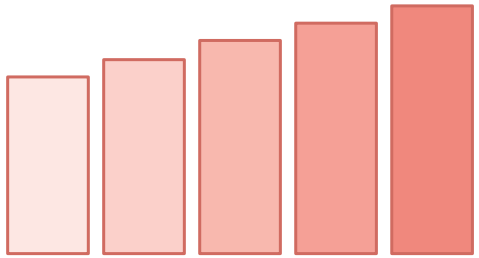
Long-Context Serving Is Expensive

Computation

Self-attention computes scores for each token with every previous token.

$O(L^2)$

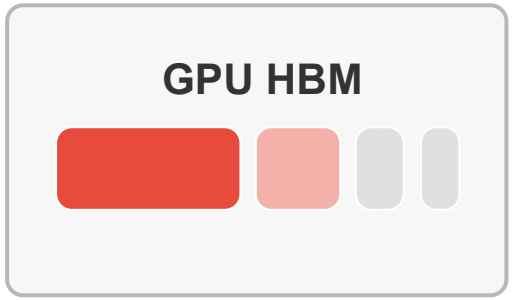
Memory



KV cache for all context tokens must be stored during generation.

$O(L)$

Utilization

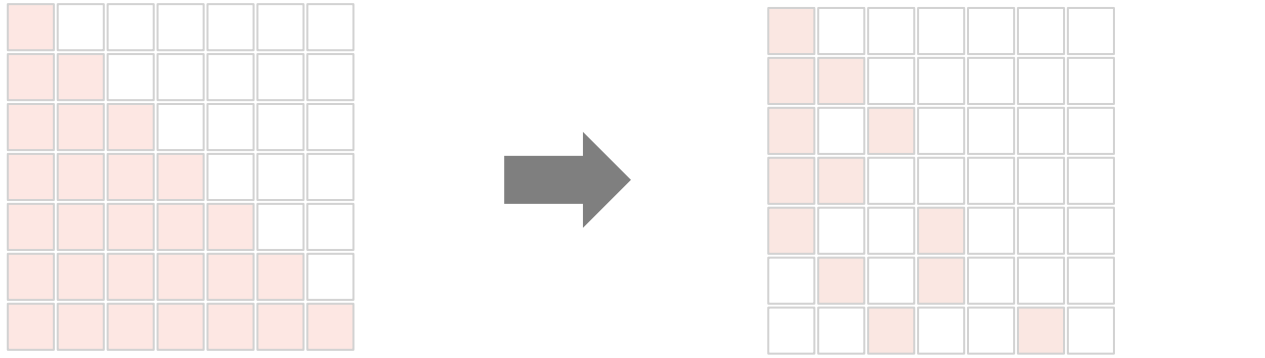


Long context requests occupy more HBM, leaving fewer slots for batching.

Lower Concurrency & Utilization

Sparse Attention: From Training-free to Native

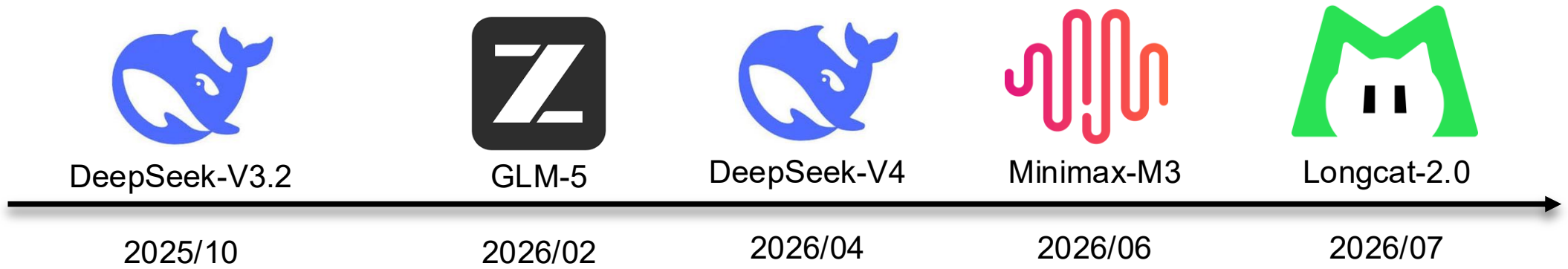
- **Attention is naturally sparse:** a small set of tokens dominate each query
- Sparse attention computes only selected tokens
  - Select top-k most important tokens
  - Reduce attention computation from **$O(L^2)$** to **$O(Lk)$**



Full attention: $O(L^2)$ **Sparse Attention: $O(Lk)$**

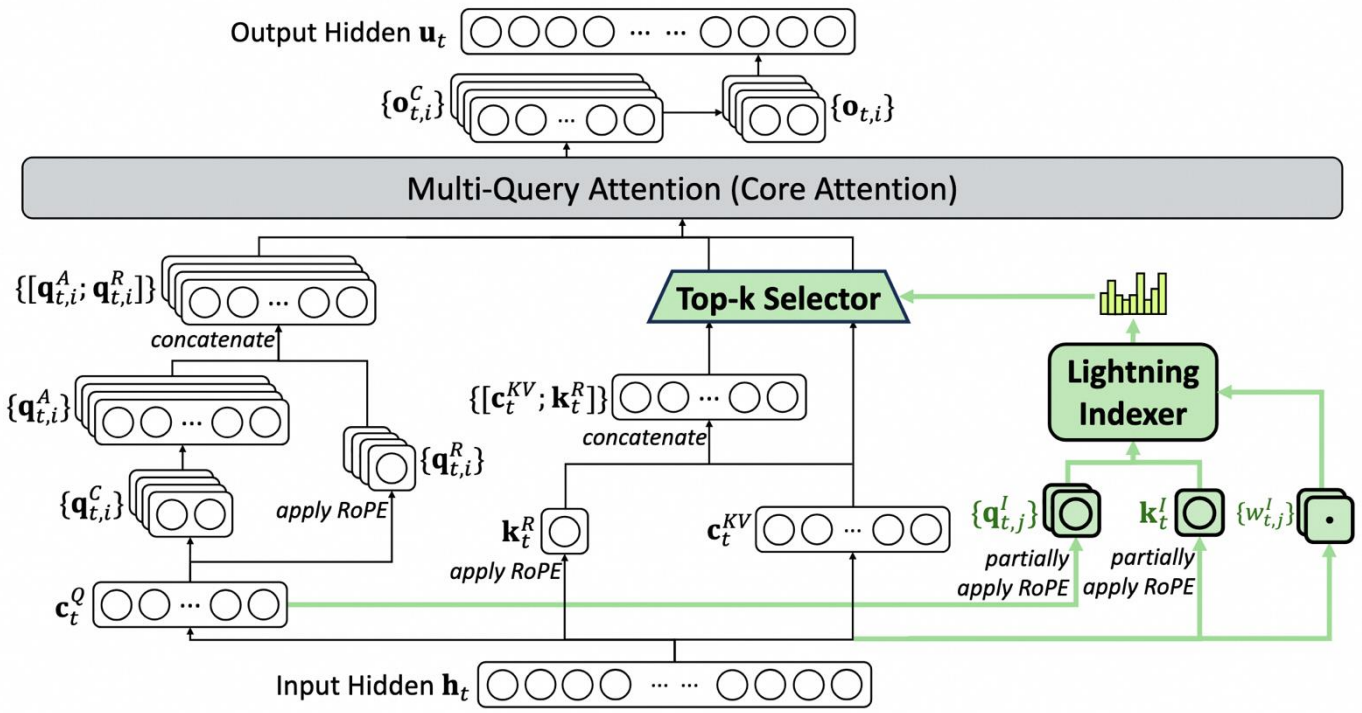
Sparse Attention: From Training-free to Native

- **Attention is naturally sparse:** a small set of tokens dominate each query
- Sparse attention computes only selected tokens
- **Training-free methods apply sparsity only at inference time**
  - May change model behavior and incur accuracy losses
- **Native sparse attention trains the model with sparsity**
  - Avoids inference-time accuracy loss
- **Frontier models are adopting native sparse attention**

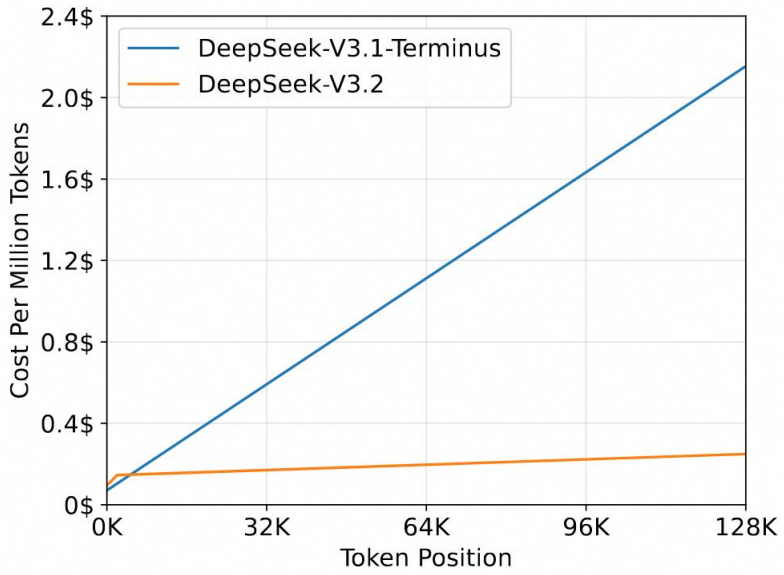


DSA Makes Native Sparsity Practical

- **DeepSeek Sparse Attention (DSA)** is validated on large-scale models
- DSA introduces a Lightning Indexer to select top-k tokens
  - Reduces attention computation by around **10x** at a 128K context length



Computation

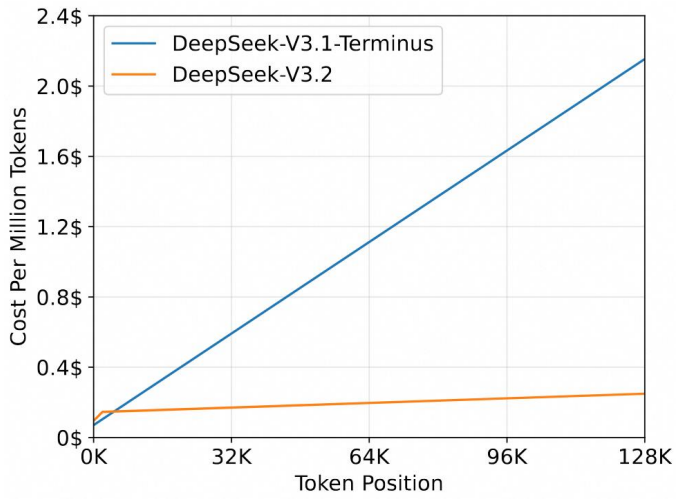


Source: DeepSeek V3.2 Technical Report

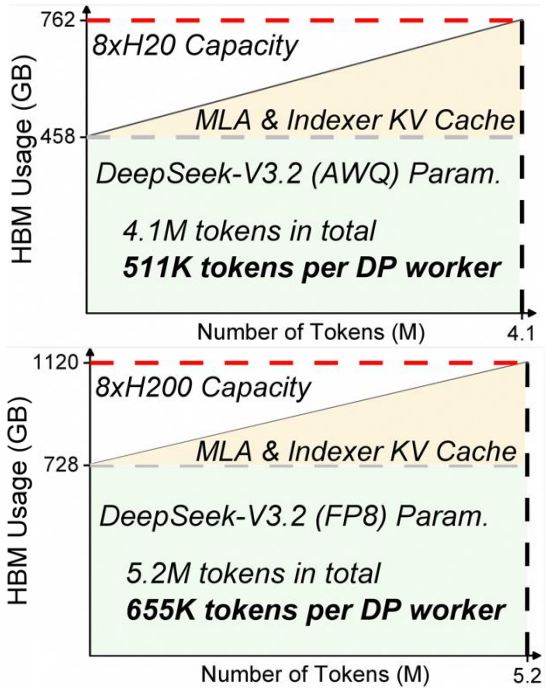
Sparse Attention Exposes Memory Bottlenecks

- **Sparse attention reduces compute, not KV cache memory**
- In DSA, the indexer adds about 20% more KV cache memory
  - Only 500-600K tokens per GPU, i.e., up to 5 requests with 128K context

Computation



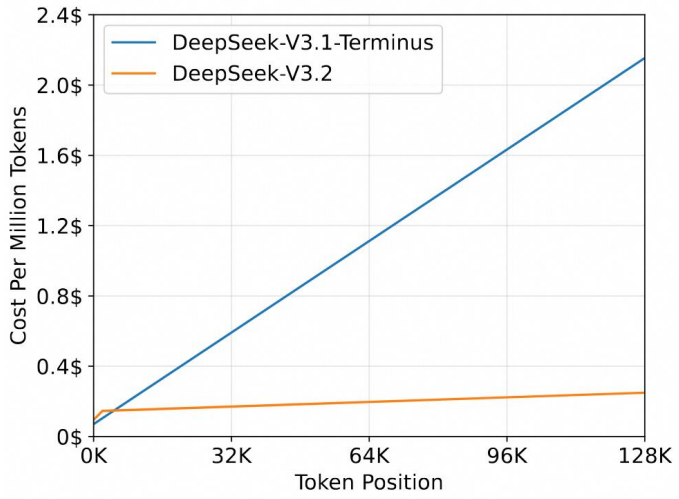
Memory



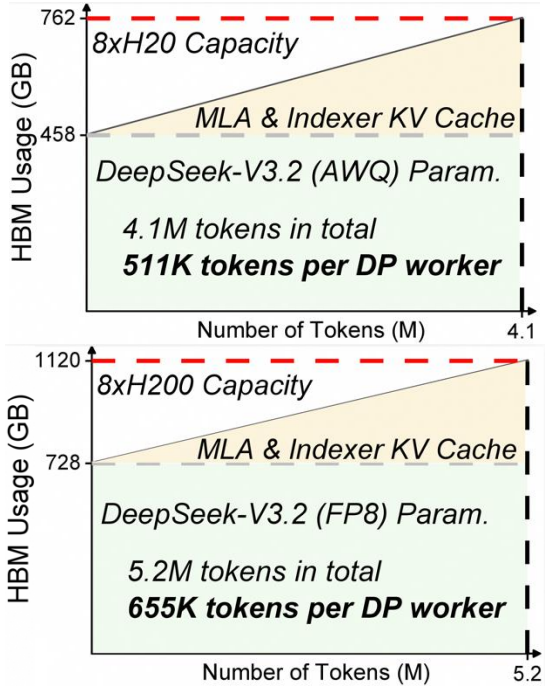
Sparse Attention Exposes Memory Bottlenecks

- **Sparse attention reduces compute, not KV cache memory**
- Sparse attention kernels have even lower hardware utilization
  - Due to indexing logic and non-contiguous memory access

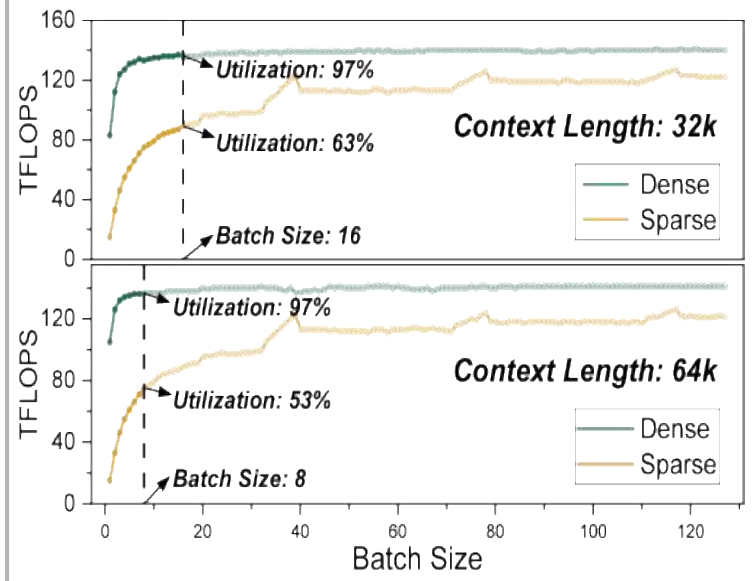
Computation



Memory



Utilization



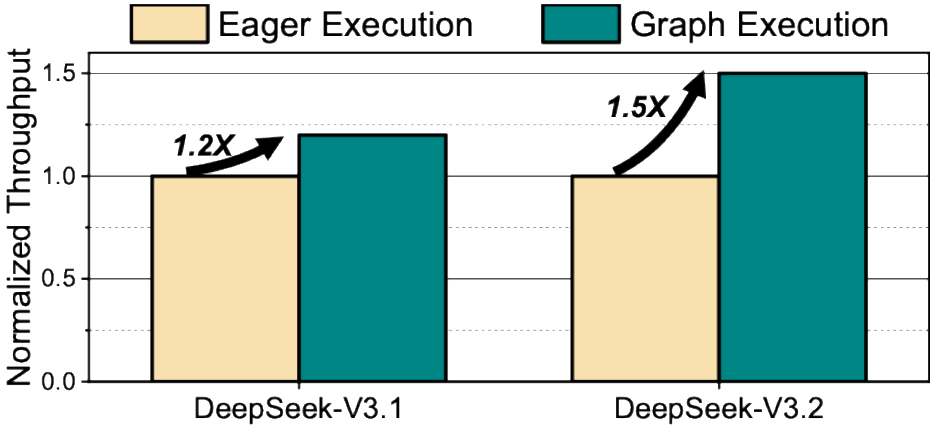
Mitigate Memory Bottleneck with KV Offloading

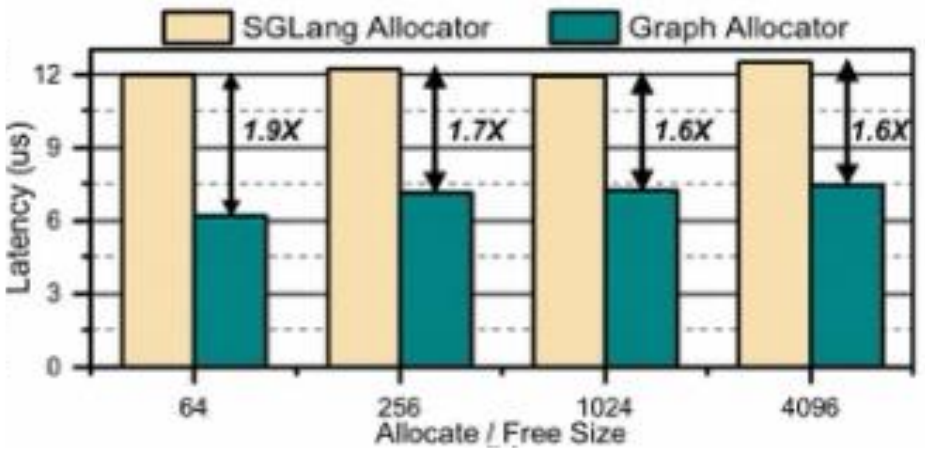
- **Keep only hot KV entries on GPU**
  - The full KV cache is offloaded to large DRAM
  - **Dynamically *recall*** selected tokens from host to GPU during inference
- Dynamic offloading differs from static offloading (e.g., Strata)
  - They are complementary and can be enabled together

	Static	**Dynamic**
Goal	More prefix KV cache reuse	Larger KV cache pool for higher concurrency
Effect	Reduce TTFT	Improve generation throughput
Granularity	Each request	Each model forward
Models	General	Sparse attention LLMs

Challenges for Dynamic KV Offloading

- **Challenge 1: cache management must be graph-friendly**
- **Graph execution (e.g., CUDA graph) is critical for practical serving**
  - Eliminate kernel launch overhead
  - Particularly important for sparse-attention models with more kernels
- **Existing cache managers break graph execution**
  - Dynamic tensor slicing and concatenation require CPU-side control



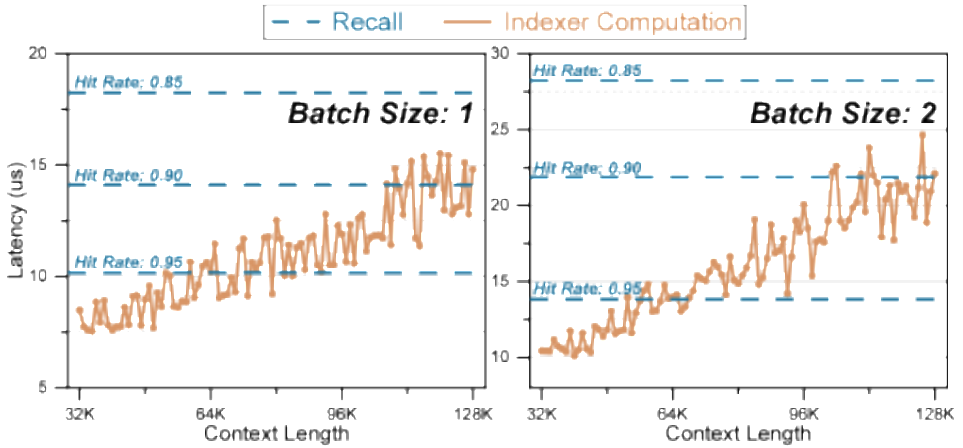


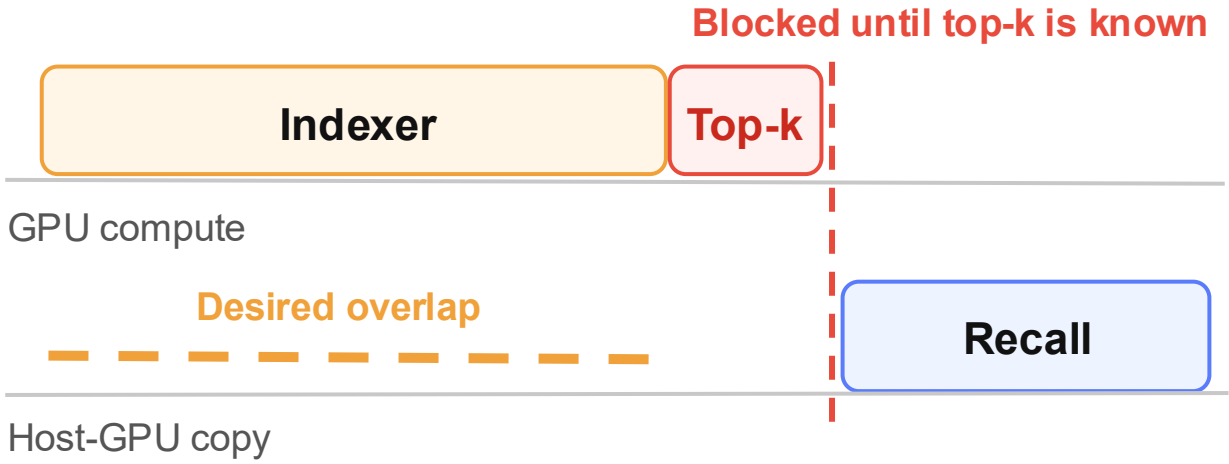
Challenges for Dynamic KV Offloading

- **Challenge 2: KV recall can be expensive**
  - Low interconnect bandwidth between host and GPU (e.g. PCIe)
- **Prior systems use pre-selection and prefetching to mitigate recall overhead**
  - InfiniGen (OSDI'24) leverages inter-layer similarity
  - FreeKV (ICLR'25) leverages inter-query similarity
- **Approximate prefetching can change selected tokens and incur accuracy losses**

Challenges for Dynamic KV Offloading

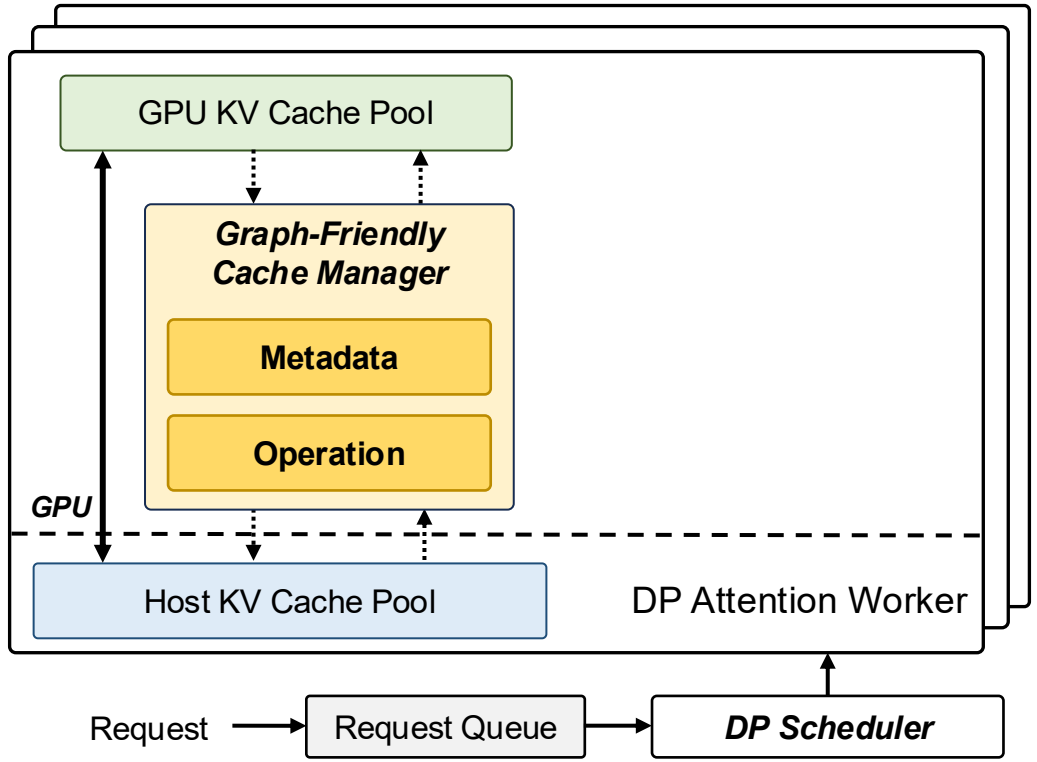
- **Challenge 2: KV recall can be expensive**
- **Can we prefetch in a lossless manner?**
  - Indexer: **O(L)**, grows linearly with context length
  - Recall: **O(k)**, constant overhead
  - **At long context lengths, recall can be hidden behind indexer computation**
- **However, top-k is a blocking operator**
  - The selected KV entries are unknown until indexer computation finishes





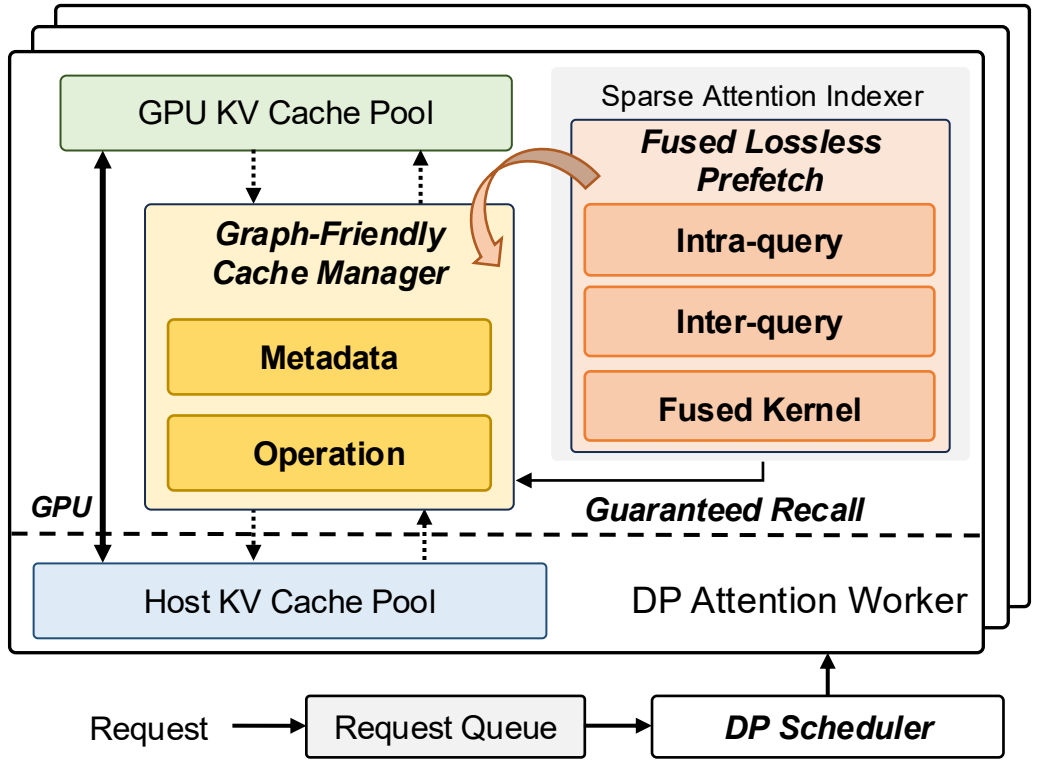
Our Solution: ECHO

- Efficient KV cache offloading
- **Graph-friendly Cache Manager**
  - Coordinate GPU and host KV cache pools



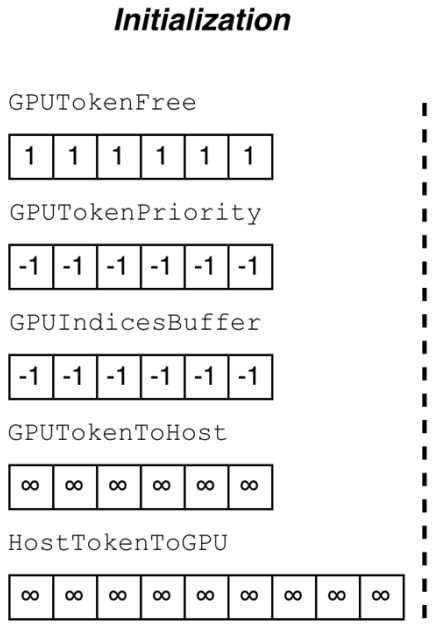
Our Solution: ECHO

- **Fused lossless prefetch**
  - Intra-query for decoding, inter-query for prefill
  - Guaranteed recall for missed tokens to ensure accuracy



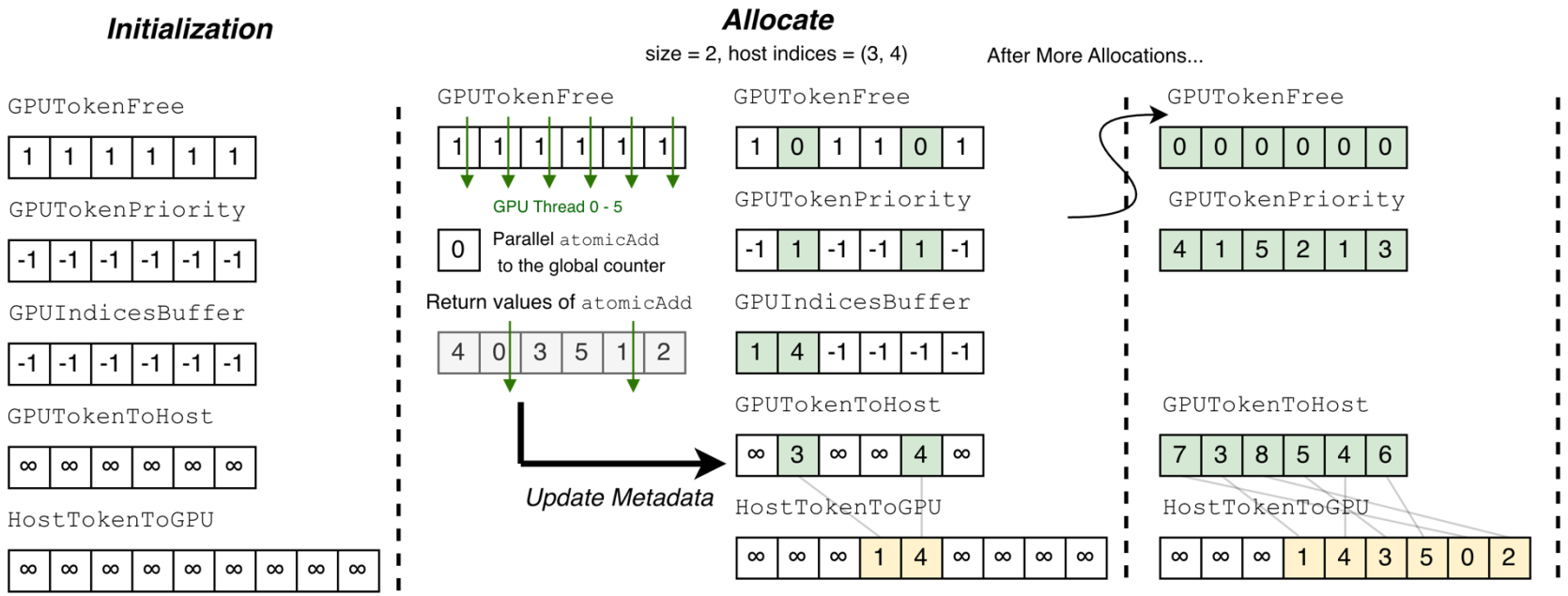
Design 1: Graph-Friendly Cache Manager

- **All metadata is stored as fixed-length tensors on GPU**
  - **GPUTokenFree:** Marks whether each GPU KV slot is free or occupied
  - **GPUTokenPriority:** Tracks eviction priority for GPU-resident KV entries (LRU-like policy)
  - **GPUTokenToHost/HostTokenToGPU:** Maps each GPU/host KV slot to its corresponding host/GPU KV slot



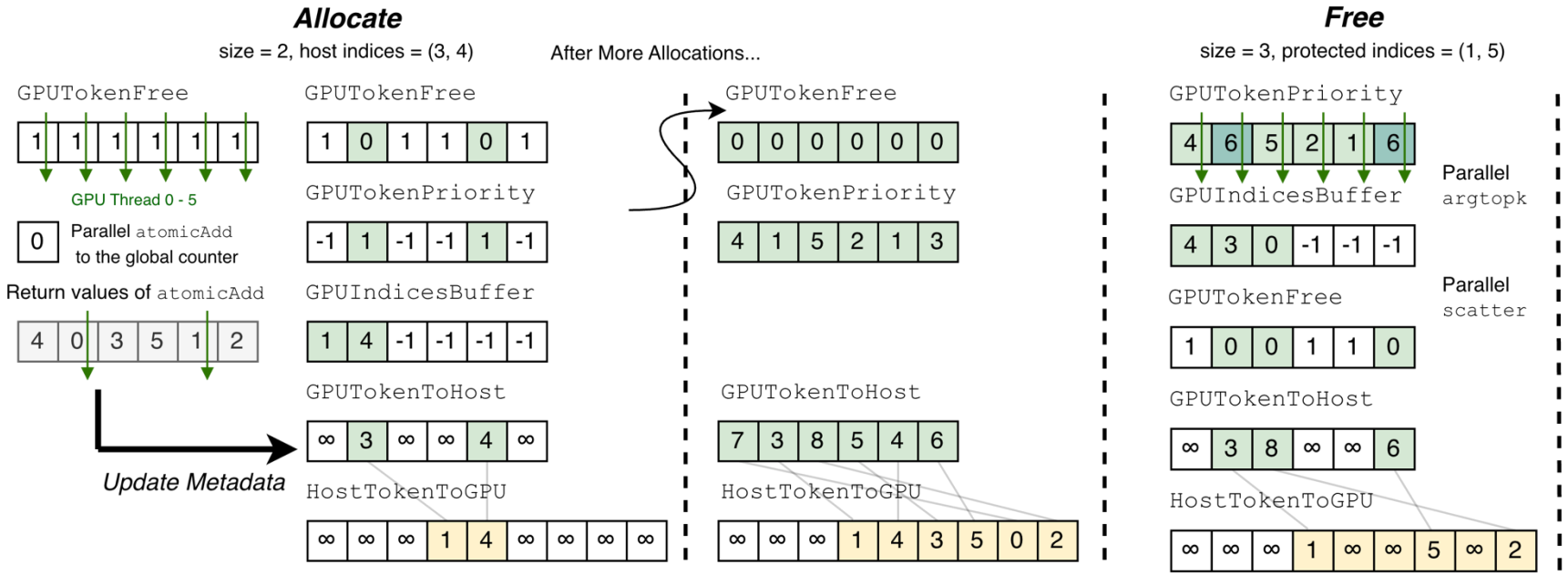
Design 1: Graph-Friendly Cache Manager

- **Allocate:** parallel scan for free slots
  - Allocated indices are output to fixed-size buffer



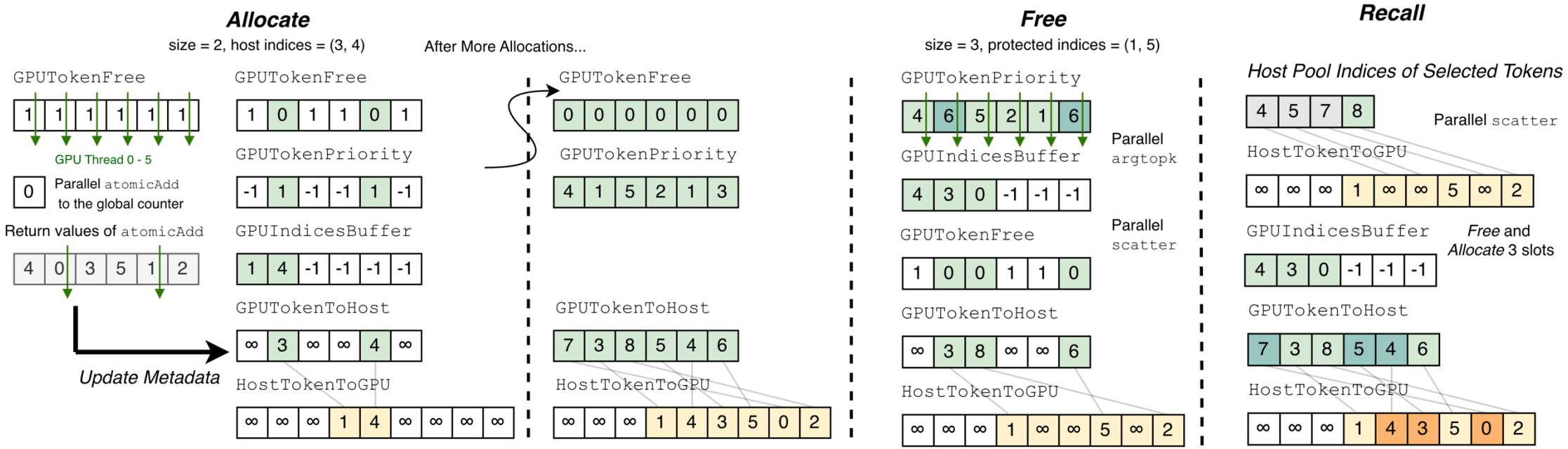
Design 1: Graph-Friendly Cache Manager

- **Allocate:** parallel scan for free slots
- **Free:** parallel ArgTopK and scatter for eviction
  - Free indices are output to fixed-size buffer



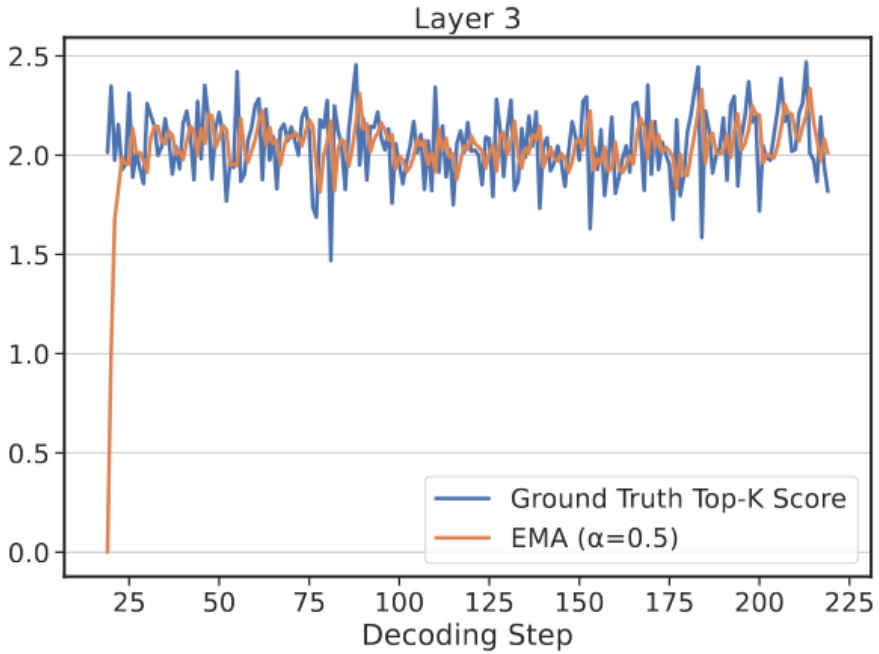
Design 1: Graph-Friendly Cache Manager

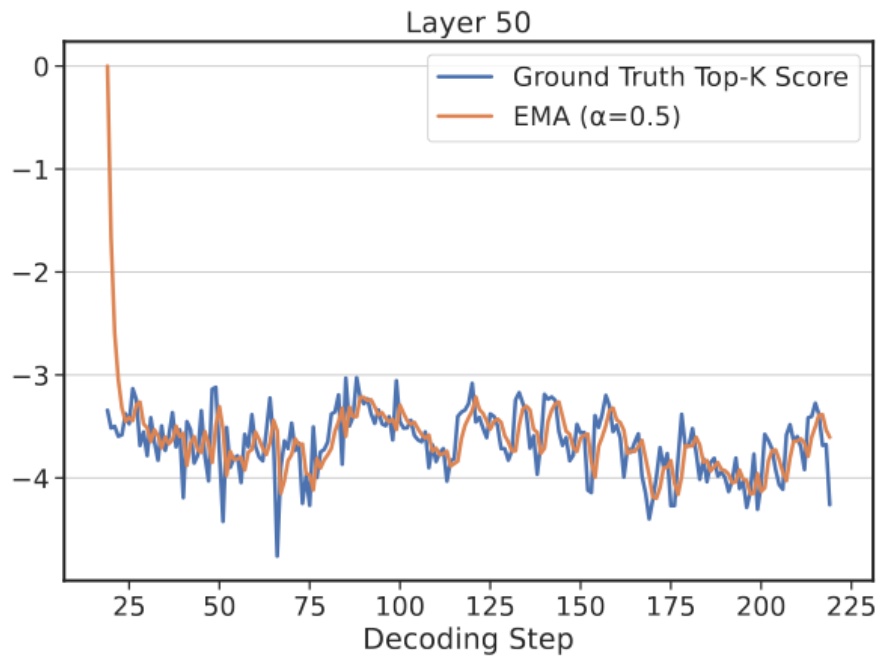
- **Allocate:** parallel scan for free slots
- **Free:** parallel argtopk and scatter for eviction
- **Recall:** parallel scatter + free + allocate
- **All operations run on GPU with fixed-size tensors and do not break graph**



Design 2.1: Intra-Query Prefetch for Decode

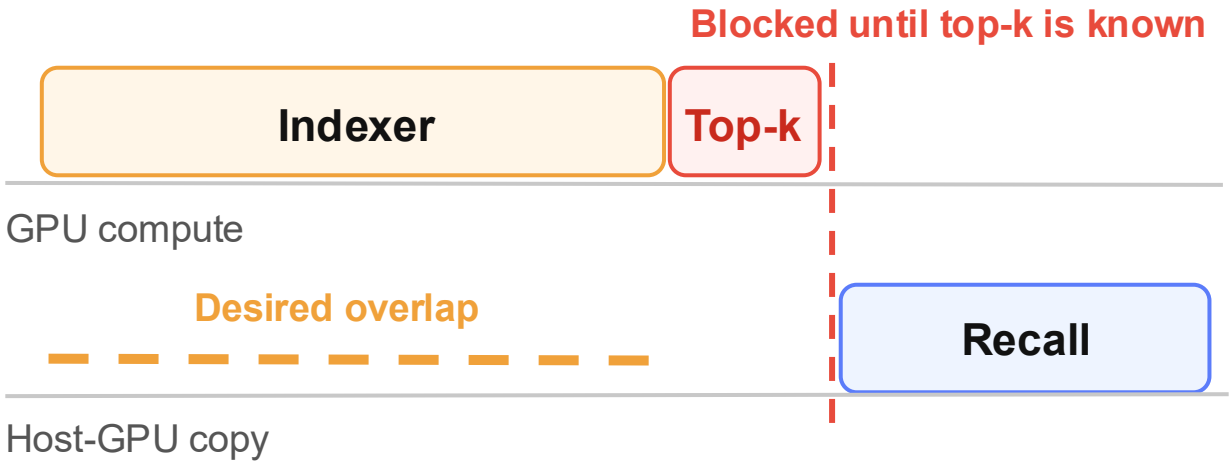
- **How to break the blocking of the top-k operator?**
- **The k-th highest score at the current decoding step is highly predictable from historical indexer scores**
  - With a simple exponential moving average (EMA)

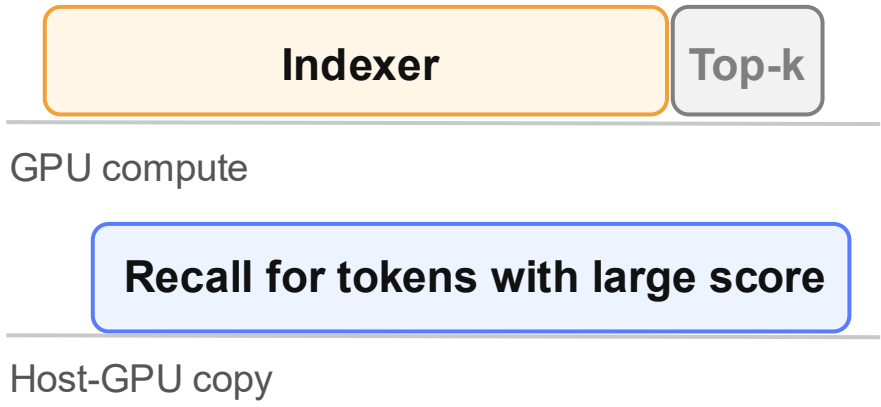




Design 2.1: Intra-Query Prefetch for Decode

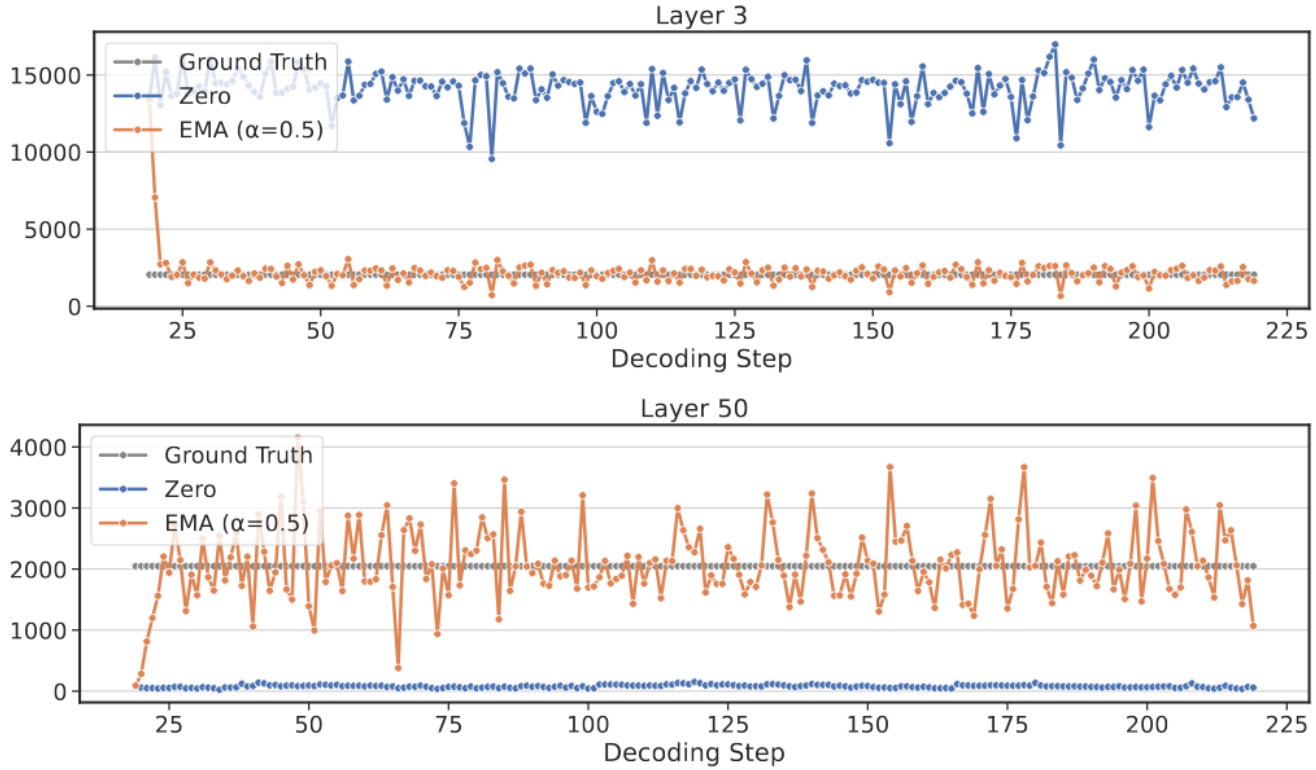
- **How to break the blocking of the top-k operator?**
- **The top-k selection can be reformulated to threshold-based selection**
  - Select and prefetch tokens whose indexer scores exceed the predicted k-th highest score





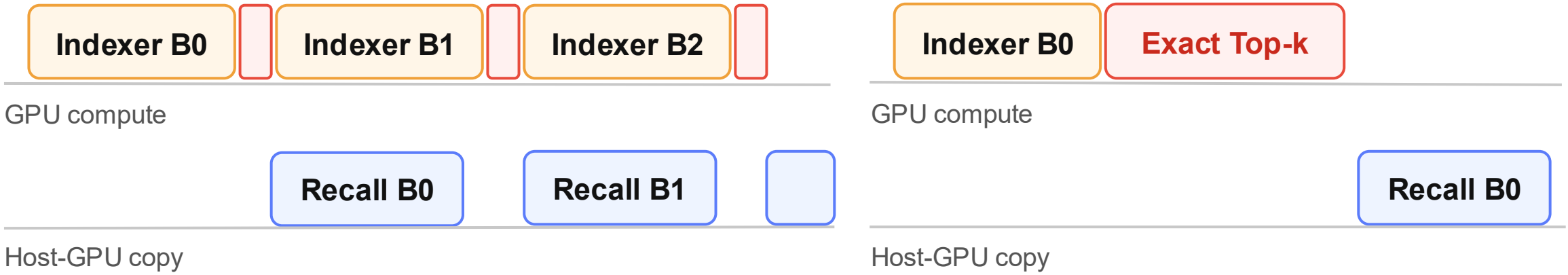
Design 2.1: Intra-Query Prefetch for Decode

- **How to break the blocking of the top-k operator?**
- **The top-k selection can be reformulated to threshold-based selection**
  - Select and prefetch tokens whose indexer scores exceed the predicted k-th highest score
- The number of prefetch candidates close to k = 2048



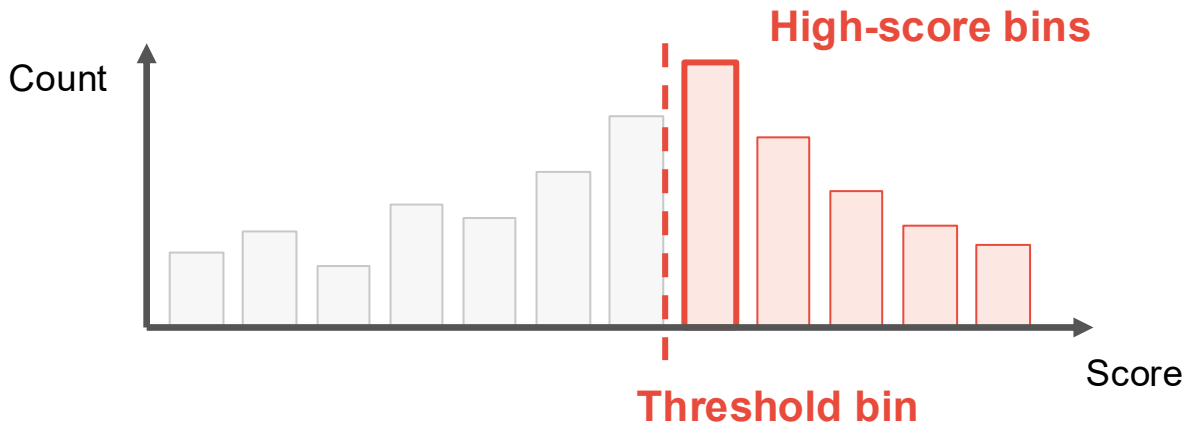
Design 2.2: Inter-Query Prefetch for Prefill

- Queries are processed sequentially block by block
- **Prefetch block i while computing scores for block i+1**
- **However, using exact top-k can significantly degrade performance**
  - Shared memory conflicts



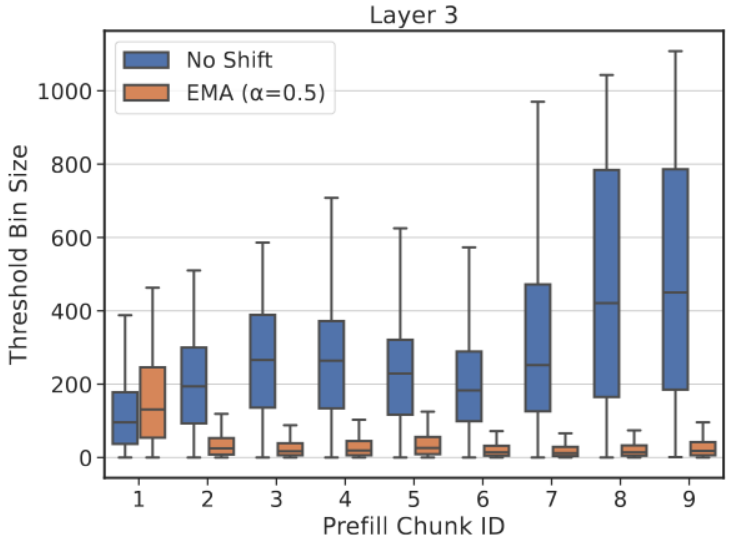
Design 2.2: Inter-Query Prefetch for Prefill

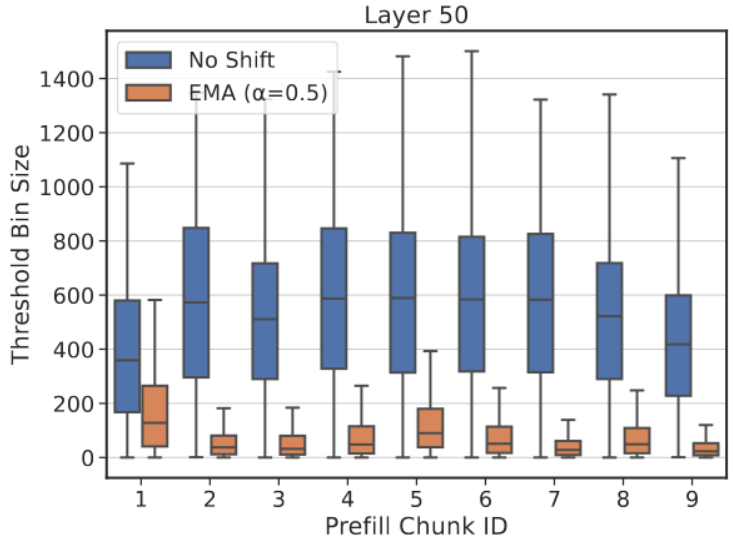
- **Prefetch for block i when computing score for block i+1**
- **However, using exact top-k can significantly degrade performance**
- **Use approximate top-k to obtain a subset of top-k tokens for prefetching**
  - Radix top-k: build histogram and filter in multiple rounds (coarse to fine)
  - We build and filter for **only one round**
- **(k – b) tokens can be selected and prefetched**
  - b is the size of the threshold bin



Design 2.2: Inter-Query Prefetch for Prefill

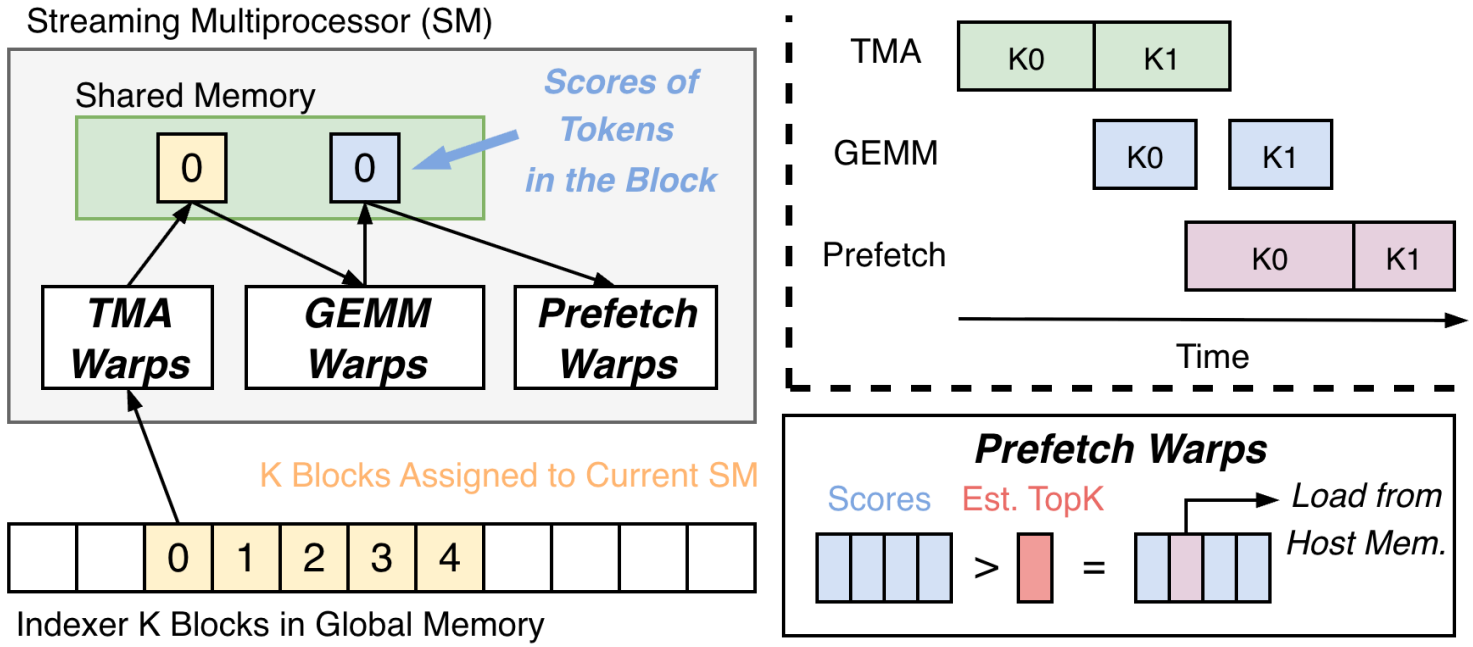
- **(k – b) tokens can be selected and prefetched**
- The size of the threshold bin can be large
  - Only a small number of tokens can be prefetched
- **EMA also helps reduce the size of the threshold bin**
  - Shift the scores by subtracting the k-th highest score predicted from the previous prefill chunk





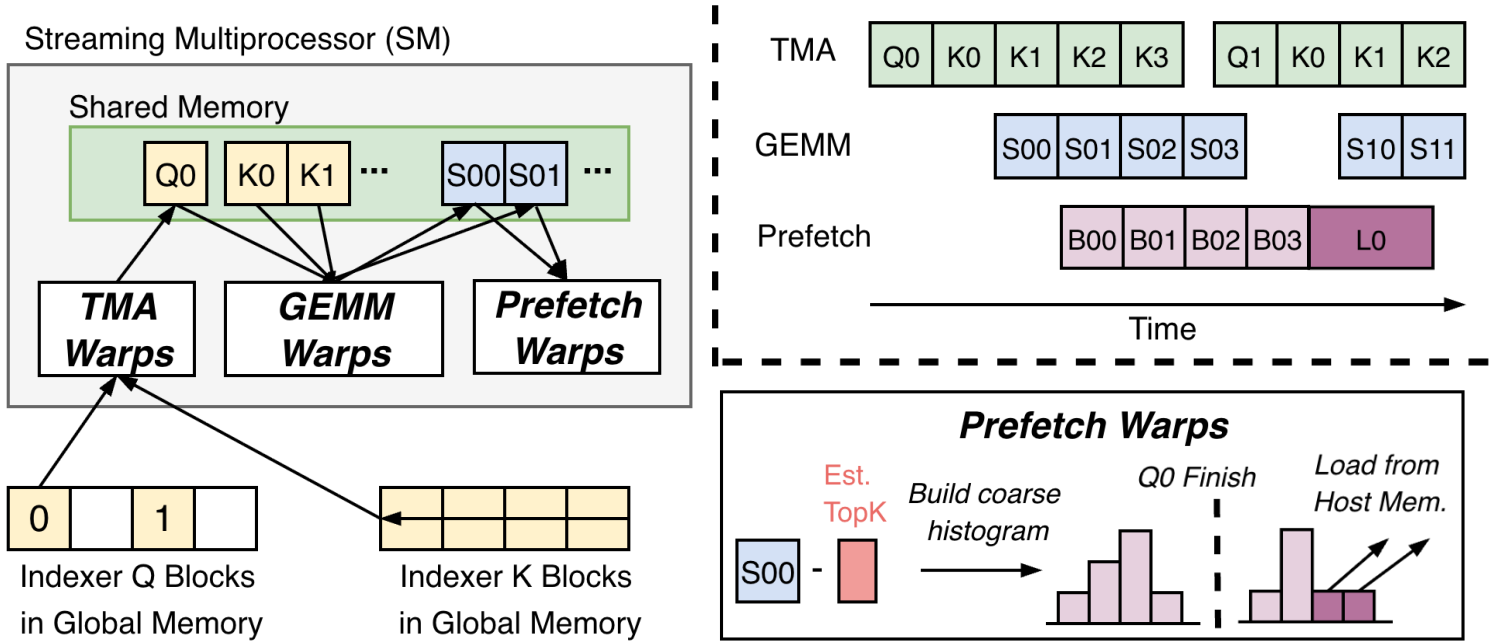
Design 2.3: Fused Kernels for Prefetching

- Build based on DeepGEMM
  - With warp specialization and software pipeline (TMA, GEMM, and Prefetch warps)
- **Intra-query (Split-k for decoding):** Each SM processes a partial K block
  - Scan score blocks and compare them with the EMA-predicted k-th highest score
  - Issue UVA loads inside the kernel



Design 2.3: Fused Kernels for Prefetching

- Build based on DeepGEMM
  - With warp specialization and software pipeline (TMA, GEMM, and Prefetch warps)
- **Inter-query (Split-Q for prefill):** Each SM processes QK pairs sequentially
  - Build coarse histogram for scores for each QK pair with score shifting
  - Filter and prefetch when a Q block iterates over all K blocks

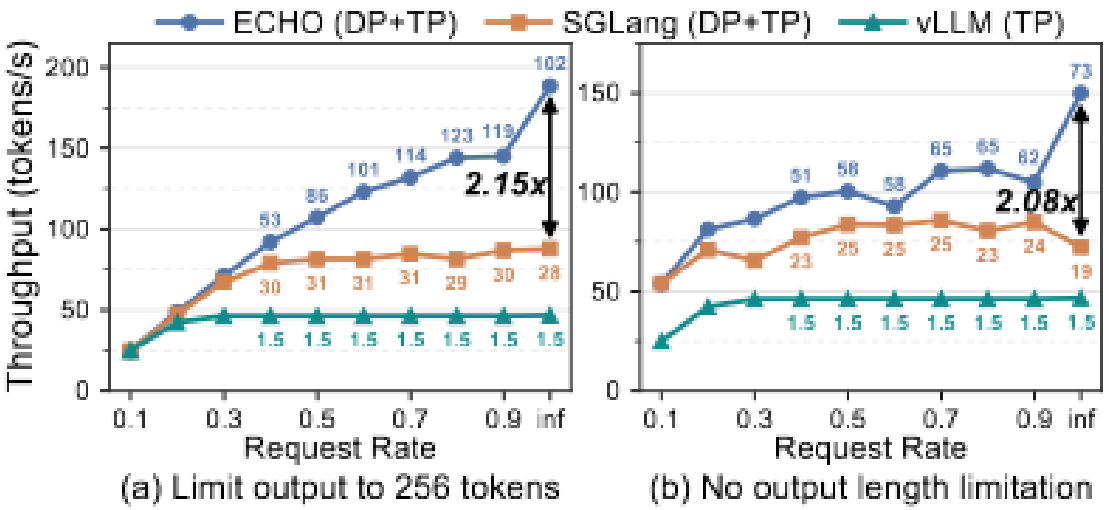


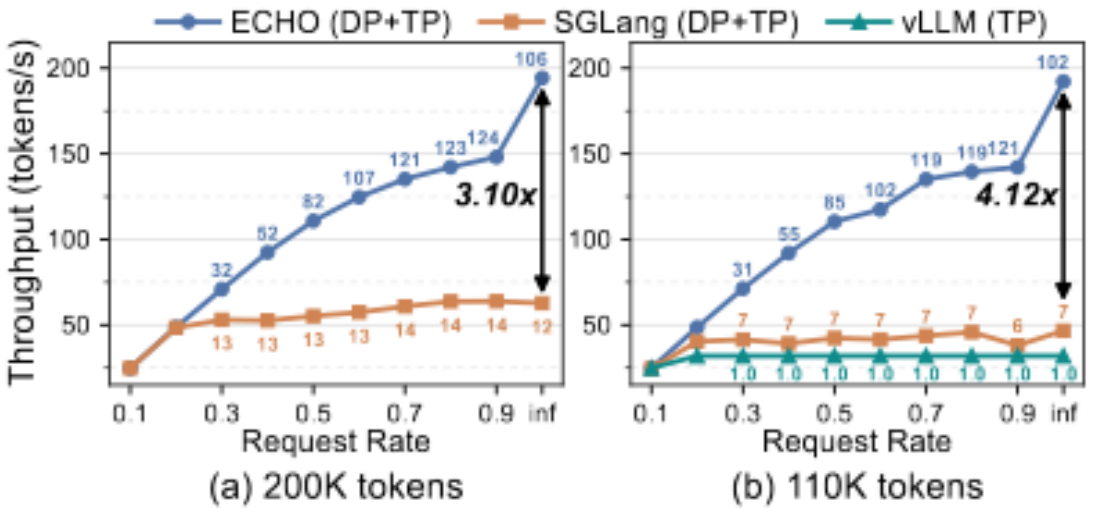
Evaluation: Setup

- Hardware
  - 8x NVIDIA H20 (96GB)
  - Connected via PCIe Gen5, 1.5TB host DRAM
- Workload
  - Model: DeepSeek-V3.2-Exp AWQ (INT4)
  - Benchmarks: ShareGPT and InfiniteBench (80-100K tokens)
- Baselines
  - SGLang v0.5.4: DP8 attention and TP8 MoE
  - vLLM v0.11.1: TP8 (DP attention was not supported at submission time)

Main Result: Higher Long-Context Throughput

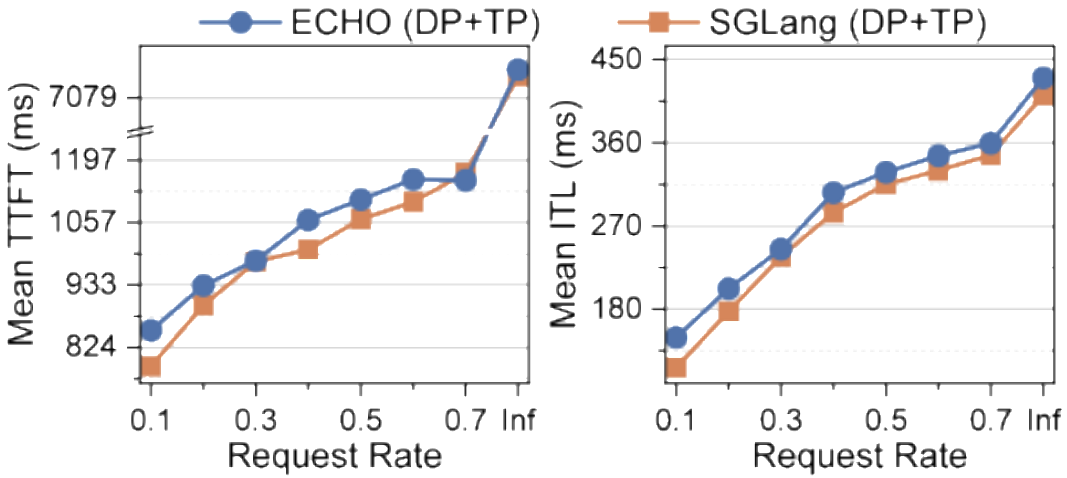
- **Full HBM: ECHO up to 2.15x vs SGLang, 4.1x vs vLLM**
  - **3-4x larger effective batch size** by extending KV pool from **330K tokens to 1.8M tokens**
  - With output length limited to 256 tokens
  - The improvement decreases without the output-length limit due to DP worker idleness
- **Constrained GPU pool: up to 3.10x (200K) and 4.12x (110K) vs SGLang**

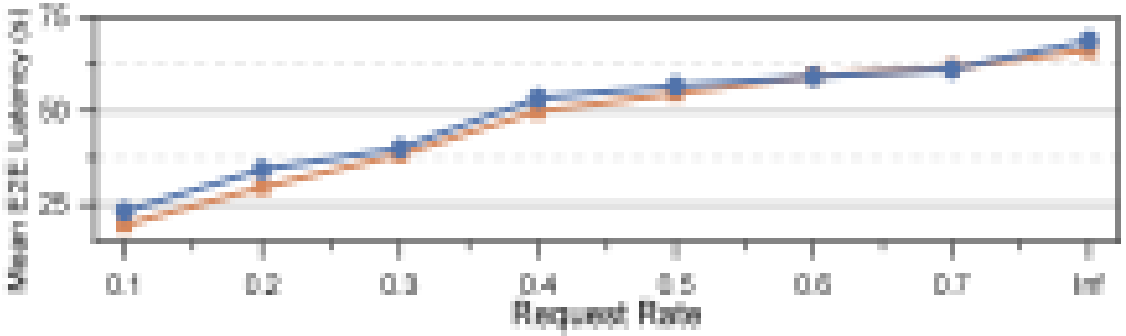




Inference Latency for Light Workload

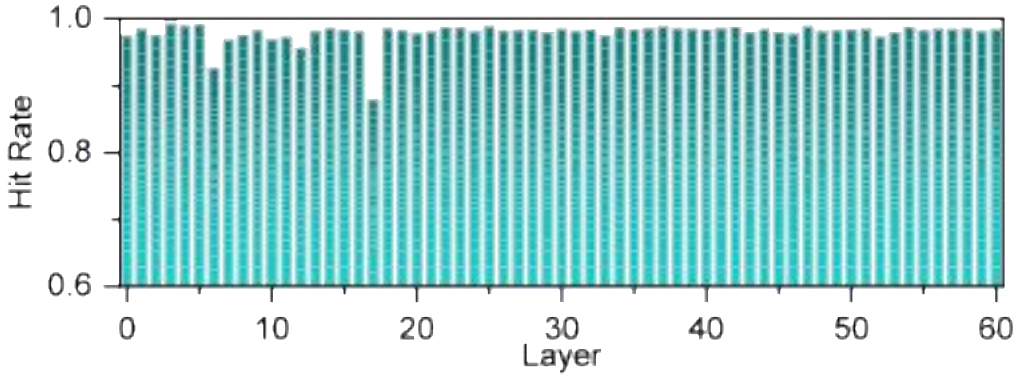
- TTFT increases by up to 7.9%
  - Due to offloading KV to host DRAM
- ITL increases by 2.7% - 27.8%
  - More visible at light load due to less computation to amortize management overhead
- E2E latency increases by up to 7.2% at request rates of 0.3 and above

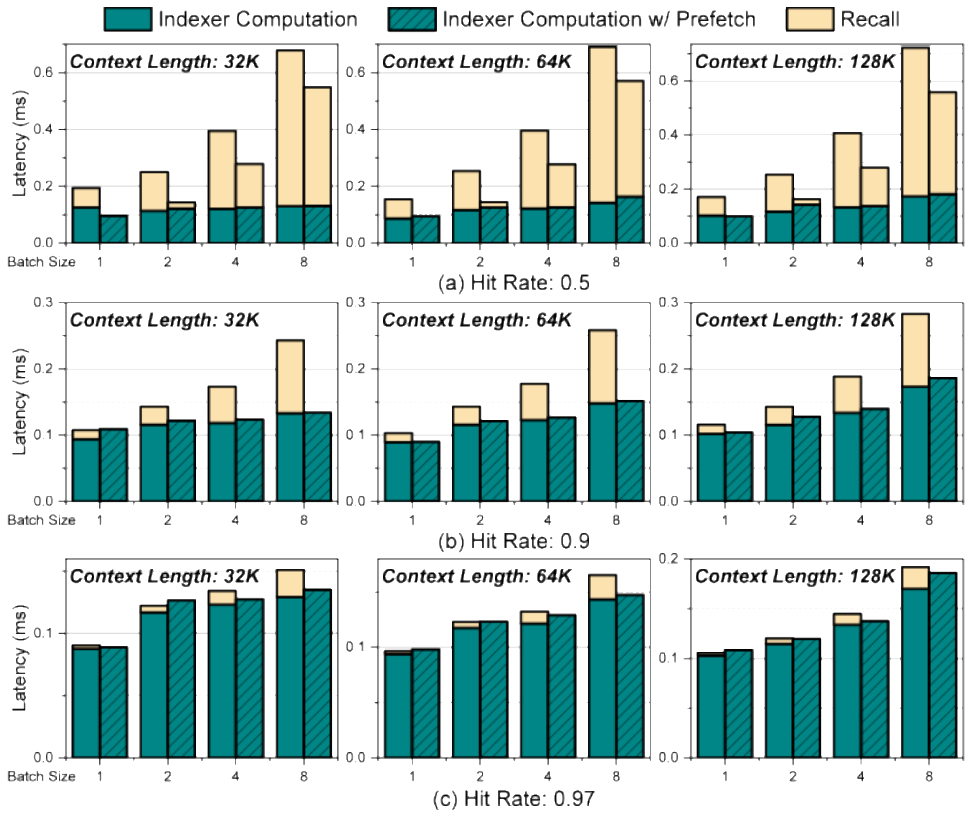




In-depth Analysis

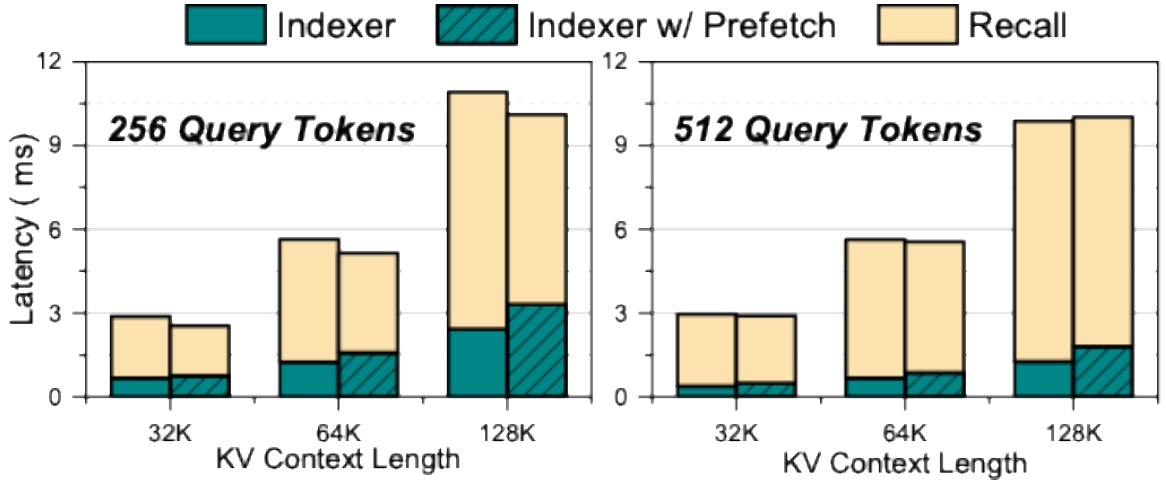
- GPU pool hit rates: typically **0.97-0.99** for 100K context
  - Benefits from the global GPU pool and LRU eviction
- Intra-query prefetch under different hit rates
  - Up to **1.29x** and **1.51x** speedups





In-depth Analysis

- GPU pool hit rates: typically **0.97-0.99** for 100K context
  - Benefits from the global GPU pool and LRU eviction
- Intra-query prefetch under different hit rates
  - Up to **1.29x** and **1.51x** speedups
- Inter-query prefetch: up to 1.1x speedup
  - Limited by too many recalled tokens (up to *the number of query tokens* × k)



Discussion

- **Generalizability to other sparse-attention models**
  - The offloading & cache manager is a general substrate for dynamic sparse-attention serving
  - Prefetching relies on properties of the DSA indexer
    - We are exploring the effectiveness on other attention variants and models
- **SGLang now officially supports dynamic offloading (HiSparse)**

	HiSparse	ECHO
GPU Cache Pool	Per-request: each request maintains the buffer individually	Global: all requests share the GPU cache pool
Recall Overhead	Reversed indexing: smaller memory overhead but longer latency	Direct indexing: larger memory overhead but shorter latency
Prefetch	N/A	✅

Summary

- Sparse attention shifts the bottleneck from compute to HBM and concurrency
- ECHO makes dynamic KV offloading practical for native sparse attention serving
- Key ideas
  - Graph-friendly GPU-side cache manager
  - Lossless prefetching by exploiting indexer predictability
  - Fused pipelined kernels with prefetching
- **Final takeaway：ECHO breaks the HBM capacity bottleneck and improves long-context generation throughput up to 2.1x over SGLang**
- Code is open-sourced at

https://github.com/sjtu-zhao-lab/ECHO