ShadowServe: Interference-Free KV Cache Fetching for Distributed Prefix Caching

Xingyu Xiang, Raj Joshi, Yuhan Liu, Jiayi Yao, Chenxingyu Zhao, Junchen Jiang, Yang Zhou, Eddie Kohler, Minlan Yu

Harvard University, University of Chicago, University of Washington, UC Davis

Jingbin Lin | June 1, 2026

Authors

Xingyu Xiang: Second-year Ph.D. student in Computer Science, Harvard University

Raj joshi: Postdoctoral Fellow at Harvard SEAS

Yuhan Liu: Ph.D. Candidate, Computer Science, The University of Chicago

Minlan Yu: Gordon McKay Professor of Computer Science at Harvard SEAS

Contents

1 Long-Context LLM Serving

2 KV Cache & Prefix Caching

3 The Bottleneck: GPU Interference

4 ShadowServe Design

5 Evaluation, Critique & Discussion

Long-Context LLM Serving

Why long prompts make LLM inference expensive?

What makes a request “long-context”?

What makes a request “long-context”?

Input is not just a short user question

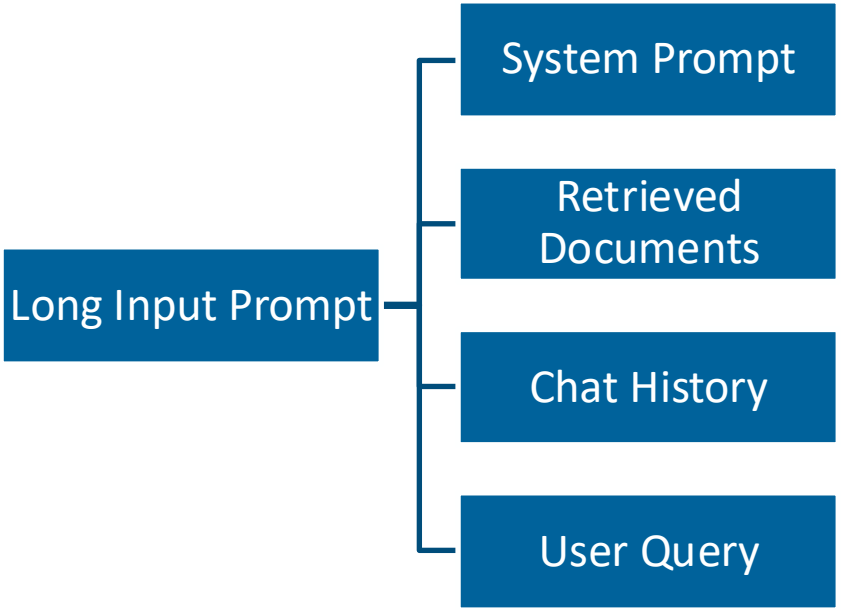
Examples:

RAG

Chatbots

Document QA

Legal/medical document analysis



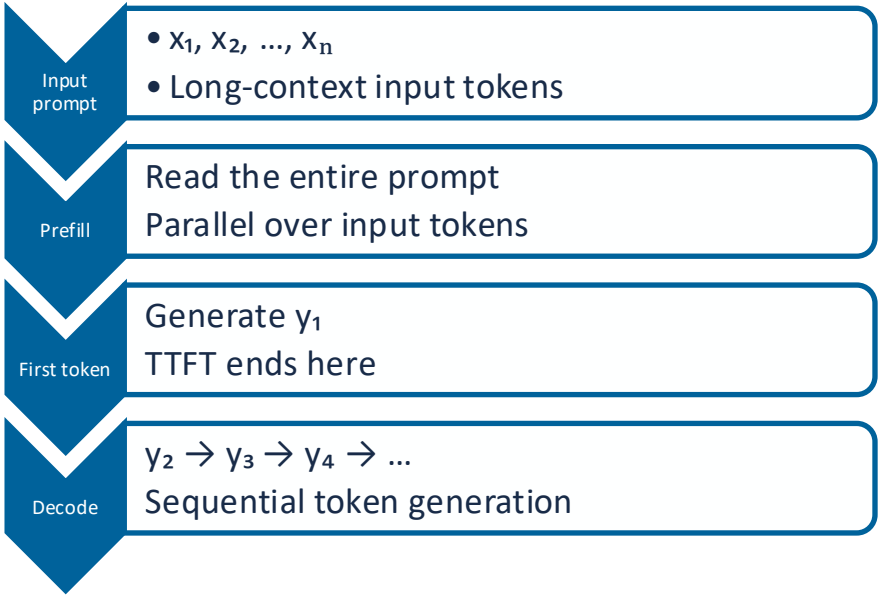
LLM Serving - Two Phases

All prompt tokens are known, so prefill can run as a batch

The model computes an attention matrix for all prompt tokens at once

Each token can only attend to previous tokens and itself

	x_1	x_2	x_3	x_4
x_1	✓	X	X	X
x_2	✓	✓	X	X
x_3	✓	✓	✓	X
x_4	✓	✓	✓	✓



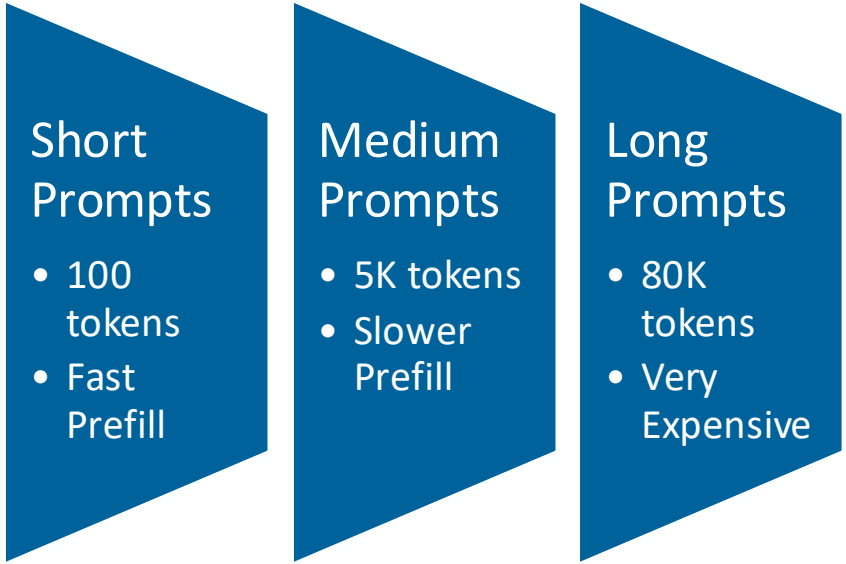
Long context makes prefill expensive

Long context makes prefill expensive:

More input tokens:
1. More attention computation
2. Larger intermediate state
3. Slower first token (TTFT)

What does “faster serving” mean?

TTFT: wait until first token
TPOT: time per generated token
Throughput: requests per second



KV Cache & Prefix Caching

If long-context prefill is expensive, what computation can we reuse?

Attention creates Q, K, and V

Prefill computes K/V for every prompt token

Decode:

New token brings Q

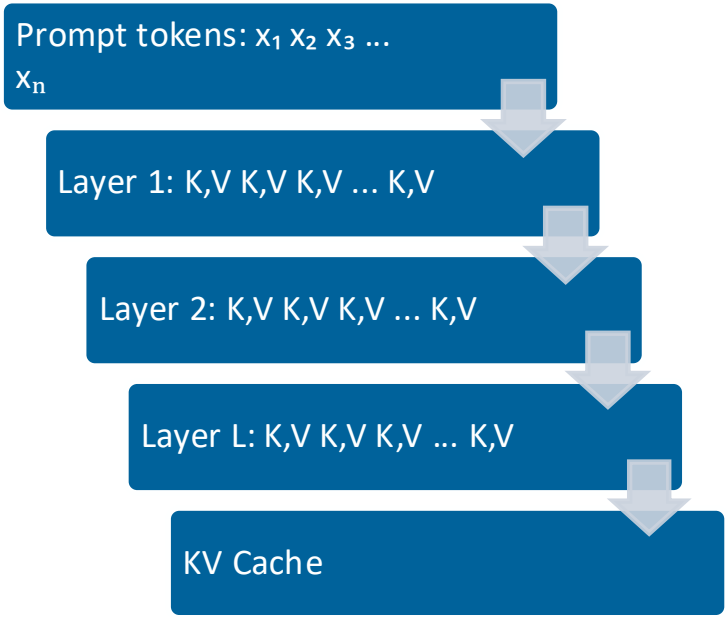
Past tokens provide cached K/V

Query: what am I looking for? (Question)

Key: what do I match against? (Labels)

Value: what information do I retrieve? (Content)

New token y_1 with Q ---- Attends to cached K/V



Prefix Caching: Reuse Shared Prompt Prefixes

Prefix Caching Reuses Shared Beginnings:

Same prefix, different requests → Reuse prefix KV cache

Shared prefix → cached KV → reused

Unique question → compute normally

Request A	Request B	Request C
• System Prompt • Long Document • Question A	• System Prompt • Long Document • Question B	• System Prompt • Long Document • Question C

Why not keep all cache locally?

Distributed Prefix Caching:

KV cache can be huge

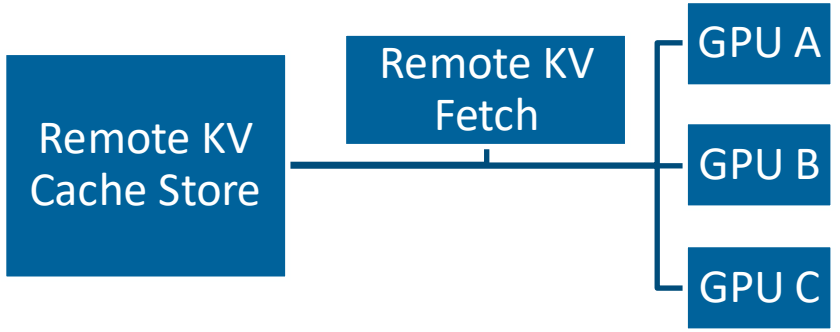
Many GPUs want to share it

Remote cache improves reuse

Fetch KV over the network

Key: Fetch must be faster than recomputing prefill.

Compression – Network – Decompression (Bottleneck)



(a) Prior work: KV cache fetching with GPU decompression.

(b) ShadowServe: KV cache fetching with SmartNIC decompression.

Figure 1: Solutions for distributed prefix caching.

The Bottleneck: GPU Interference

Distributed caching saves prefill compute.

But cache fetching now competes with model serving.

Remote KV cache fetch vs. recompute

Remote caching is only useful if fetching is faster.

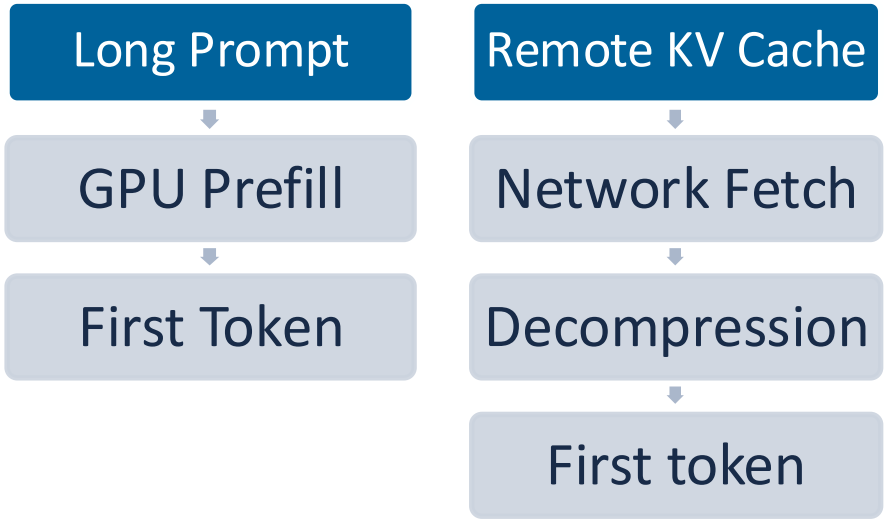
Why Compression:

KV cache is large

Less network traffic

But extra decompression stage

Where should decompression happen?



But decompression competes with LLM decode

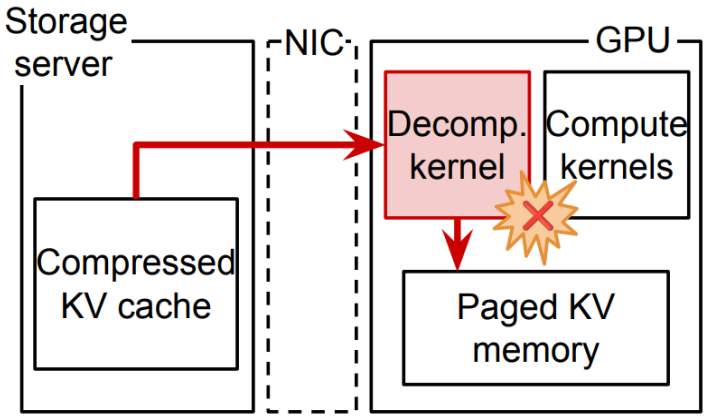
LLM decode and decompression share GPU resources

Result:

Decode slows down

Decompression slows down

Serving latency increases



(a) Prior work: KV cache fetching with GPU decompression.

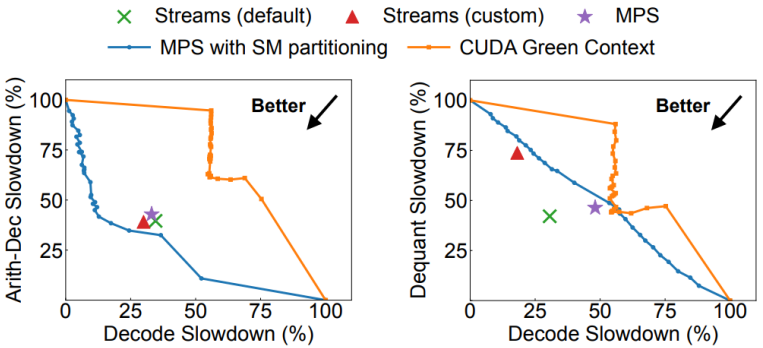
GPU decompression is not “free”

Same GPU:

LLM decode + KV decompression

Result:

Bidirectional slowdown



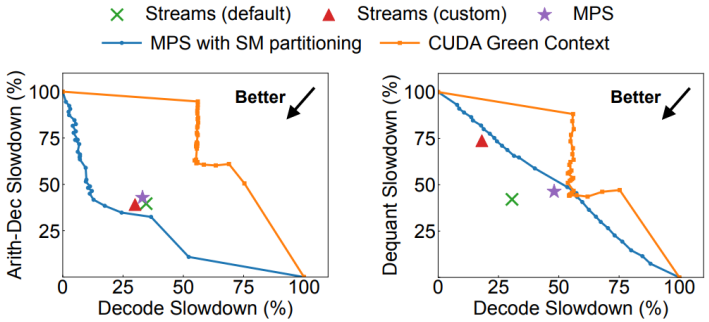
(a) Interference between arithmetic decoding and LLM decode.

(b) Interference between dequantization and LLM decode.

Figure 3: Decompression and LLM decode interfere on GPU. No interference would correspond to 0% slowdown for either task; we observe significant slowdowns across all GPU multitasking mechanisms. For CUDA streams, we launch LLM decode either in the default stream, or in a custom stream like decompression. MPS [80] and Green Context [76] enable SM partitioning, and the two curves are plotted by assigning different numbers of SMs to the two tasks. Green Context performs poorly as it is an experimental feature and fails to partition memory bandwidth effectively [109].

Discussion Questions

If even GPU multitasking tools still cause large slowdown, what does that tell us about the limit of same-device optimization?



(a) Interference between arithmetic decoding and LLM decode.

(b) Interference between dequantization and LLM decode.

Figure 3: Decompression and LLM decode interfere on GPU. No interference would correspond to 0% slowdown for either task; we observe significant slowdowns across all GPU multitasking mechanisms. For CUDA streams, we launch LLM decode either in the default stream, or in a custom stream like decompression. MPS [80] and Green Context [76] enable SM partitioning, and the two curves are plotted by assigning different numbers of SMs to the two tasks. Green Context performs poorly as it is an experimental feature and fails to partition memory bandwidth effectively [109].

Why not decompress on the CPU?

CPU avoids GPU interference,

but it is not a free resource.

Problems:

Limited decompression throughput

Busy with serving-side tasks

Extra data movement

GPU Decompression	CPU Decompression
• Close to Model • Interference with Decode	• Avoid GPU Kernel • Busy CPU • Extra Copies • Low Throughput

ShadowServe: Decompress on NIC

Move KV-cache preparation off the GPU

ShadowServe separates control and data

CPU control plane:

Schedule and coordinate fetches

SmartNIC data plane:

Fetch, Decompress, Dequantize, Transfer

GPU:

Model compute, Paged KV memory

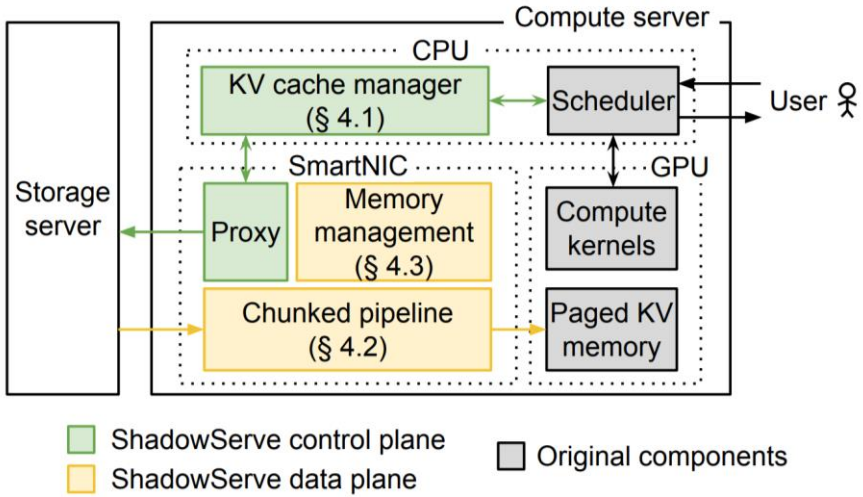


Figure 5: ShadowServe overview.

SmartNIC offload is not automatically fast

SmartNIC constraints:

Limited compute

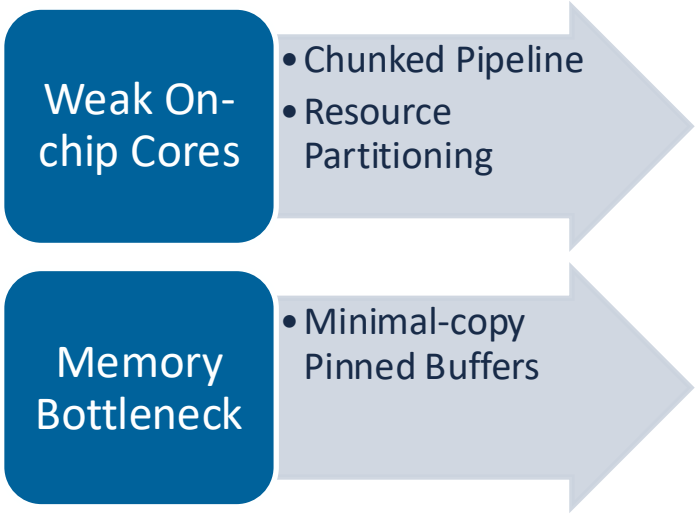
Limited memory subsystem

Need:

Pipeline + careful memory management

Offloads decompression and DMA to SmartNIC hardware accelerators

Partitions the Arm cores between network and dequantization work



Control Plane: fetch in the background

1. Scheduler forms a prefill batch
2. KV cache manager intercepts requests with cached KV
Requests waiting for KV fetch
3. Data plane fetches in background, then
Completion queue:
manager restores requests
Requests ready to resume

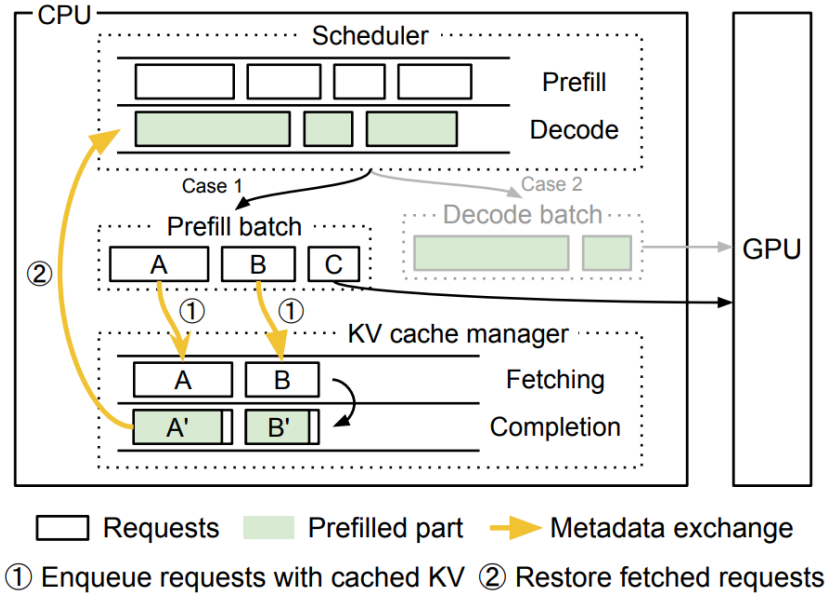


Figure 6: ShadowServe's asynchronous fetching.

Discussion Question

What could go wrong if remote fetching was synchronous?

GPU stalls

Lower throughput

Higher TPOT

Scheduler blocked by I/O

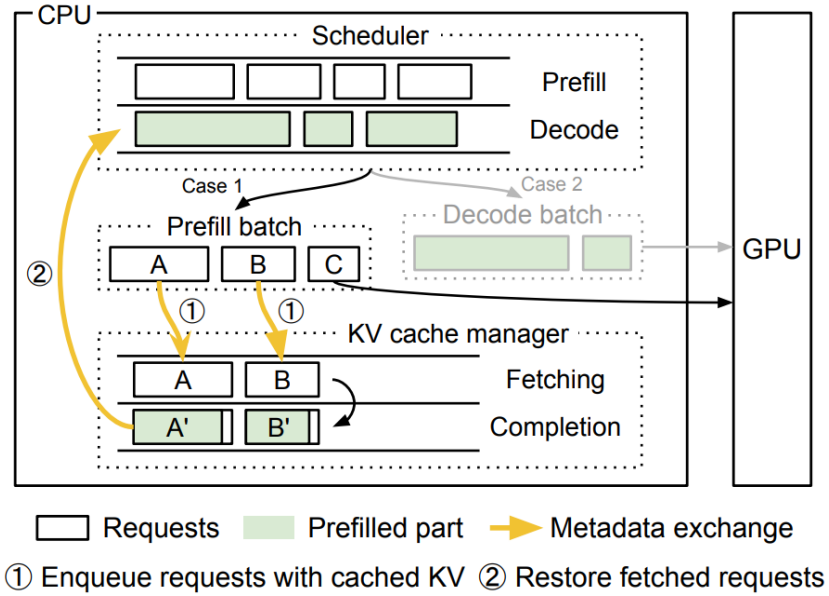


Figure 6: ShadowServe's asynchronous fetching.

Fetched KV cache is not a complete prefill

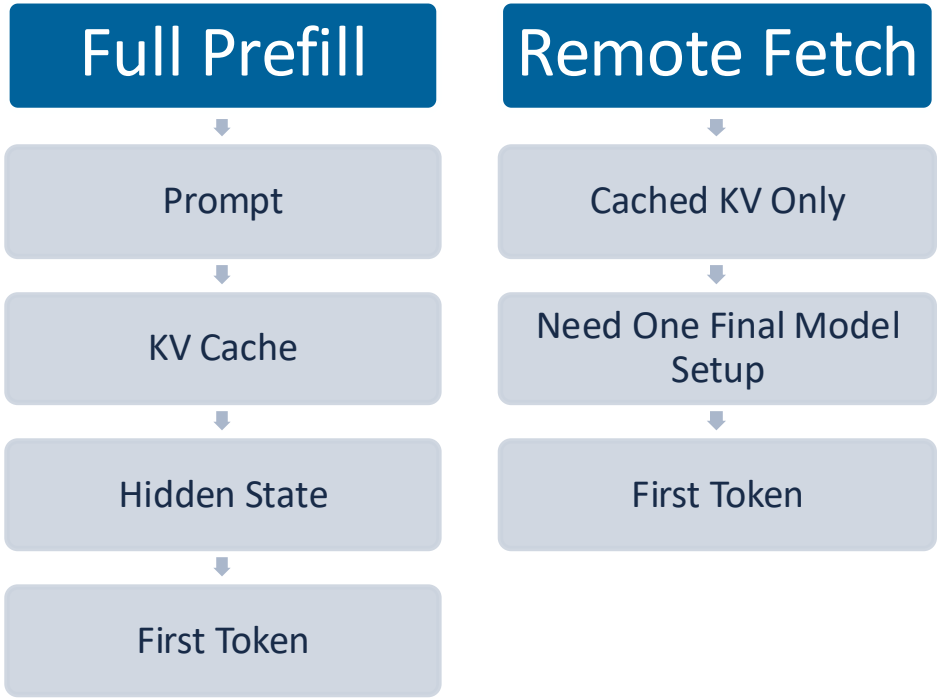
KV cache = intermediate attention state

Full prefill also produces:

The first output token

ShadowServe workaround:

Mark last token as not yet prefilled



Data Plane: four-stage chunked pipeline

Network → Decompress → Dequantize → DMA

Split KV cache into chunks

Pipeline chunks across SmartNIC resources

Smart NIC (Network Interface Card)

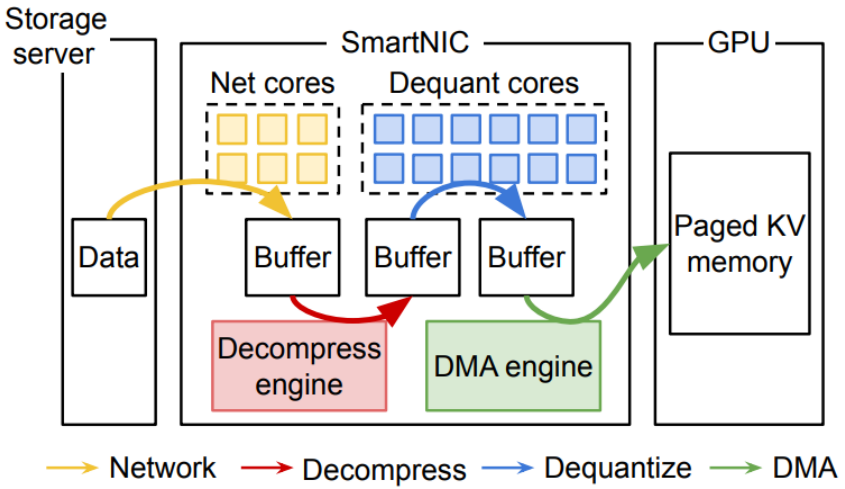


Figure 7: ShadowServe's chunked pipeline.

Memory Management: keep the data path tight

1. Compressed chunks arrive from storage

2. Decompression writes into decompression/dequantization buffers

3. Dequantization writes into DMA source buffer

4. DMA copies contiguous data into GPU destination buffer

5. Lightweight scatter places data into paged KV memory

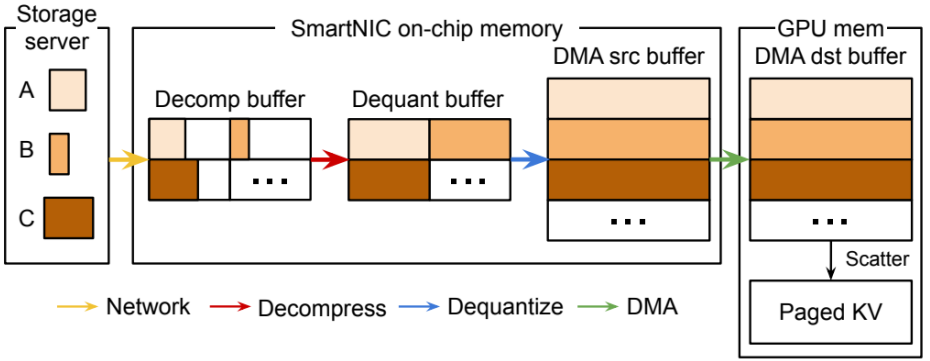


Figure 8: ShadowServe's memory management. It is also possible to perform DMA directly into paged KV memory, but this results in scattered data copies which significantly degrade DMA throughput.

What ShadowServe changes

Prior systems:

GPU decompresses compressed KV cache

ShadowServe:

SmartNIC fetches + decompresses + transfers KV cache

Result:

GPU focuses on model computation

Evaluation, Critique & Discussion

Did ShadowServe solve the bottleneck, where does it win, and where does it still fail?

Evaluation setup

Baselines:

vLLM = recompute KV cache

CacheGen-Async = GPU decompression

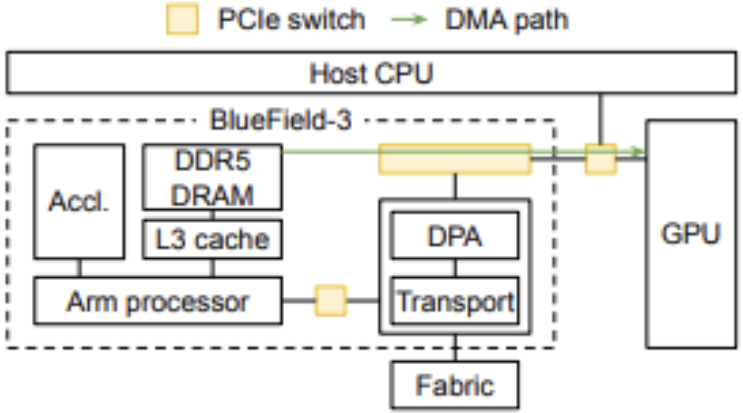
ShadowServe = SmartNIC offload

Metrics:

TTFT, TPOT, throughput

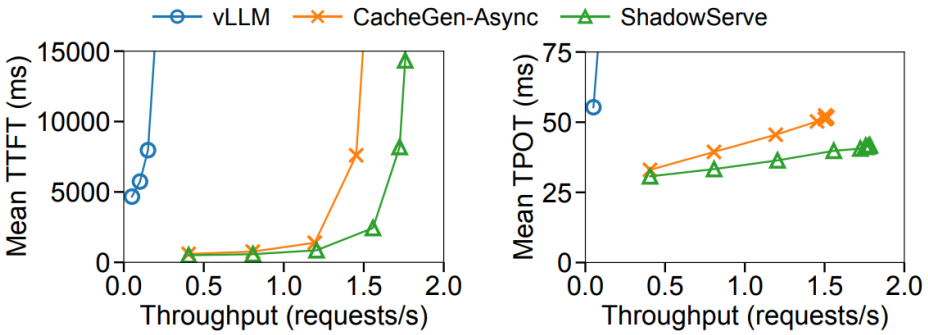
NIC: NVIDIA BlueField-3 DPU

Decode: NVIDIA L40S GPU



End-to-end: ShadowServe reduces latency under load

- 20 Gbps, output length = 32
- vs. CacheGen-Async:
- 16% lower unloaded TTFT
- 20% lower loaded TPOT
- 18% higher max throughput

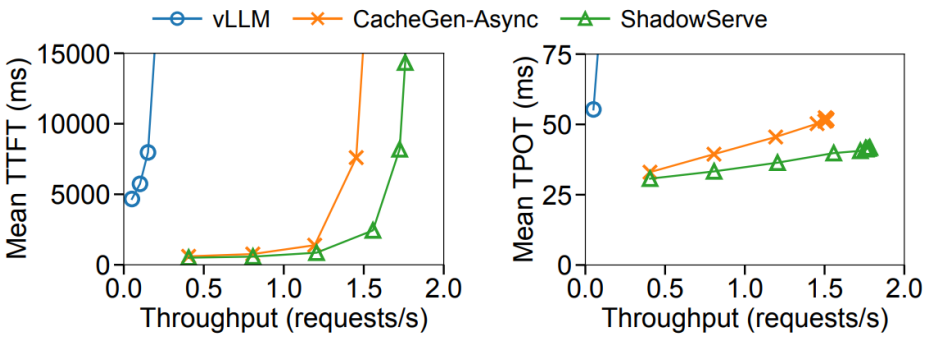


(a) Mean TTFT vs. throughput. (b) Mean TPOT vs. throughput.

Figure 9: Load-latency curves for output length = 32 and network bandwidth = 20 Gbps. The TPOT curves do not show turning points, because TPOT does not include the queueing delay, and therefore does not increase further after the systems become fully loaded.

Discussion Question

In a real product, would you care more about TTFT or TPOT, and would that change your decision to deploy ShadowServe?



(a) Mean TTFT vs. throughput.

(b) Mean TPOT vs. throughput.

Figure 9: Load-latency curves for output length = 32 and network bandwidth = 20 Gbps. The TPOT curves do not show turning points, because TPOT does not include the queueing delay, and therefore does not increase further after the systems become fully loaded.

Where does ShadowServe win?

- TPOT: better across all settings
- TTFT: better at ≤ 20 Gbps,
- Worse above 20 Gbps
- Throughput:
- Best for low bandwidth or longer outputs

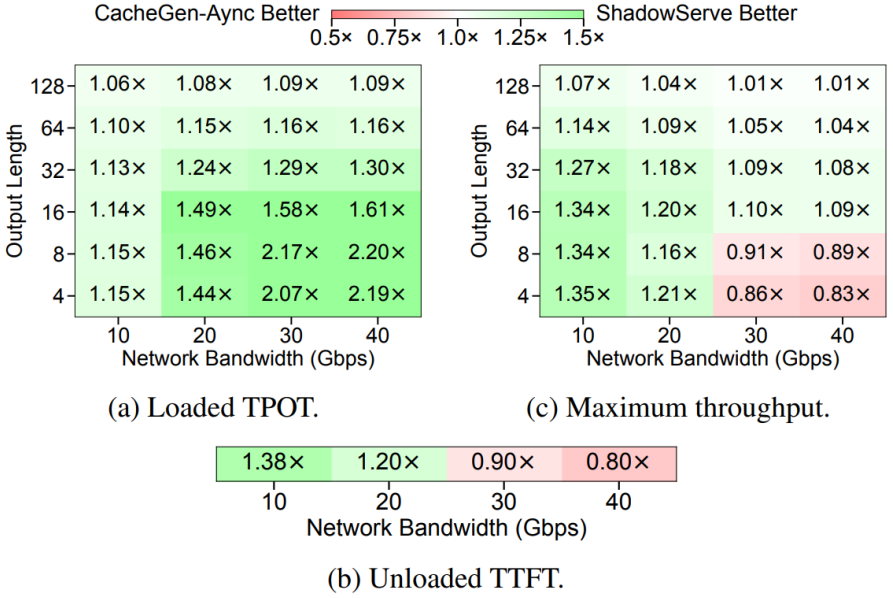


Figure 10: Performance comparison between ShadowServe and CacheGen-Async across different output lengths and network bandwidths. Unloaded TTFT is not affected by output length as it measures the latency before the first output token.

Why the results change by workload

Short output:

TTFT dominates

Longer output:

TPOT dominates

Low bandwidth:

Compression/offload helps

High bandwidth:

SmartNIC fetching becomes bottleneck

	Short output	Longer output
Low bandwidth	ShadowServe often wins	ShadowServe wins
High bandwidth	CacheGen-Async can win	Mixed / ShadowServe often wins

Offloading removes one bottleneck, but exposes another

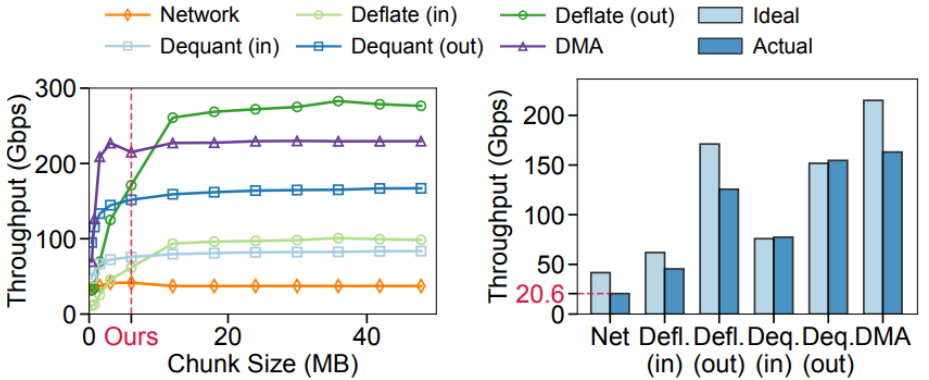
Network stage becomes the bottleneck

Actual network throughput:

37.3 Gbps standalone → 20.6 Gbps in pipeline

Likely cause:

SmartNIC memory subsystem contention



(a) Standalone (microbenchmark) performance for each pipeline stage under different chunk sizes.

(b) Standalone vs. actual performance for each pipeline stage using our chunk size (256 tokens).

Figure 13: SmartNIC microbenchmark and actual performance. There are two throughput numbers for Deflate and dequantization because they have different input and output data sizes. The results show that the pipeline is currently bottlenecked by the network stage.

Final takeaway

ShadowServe is strongest when:

network bandwidth is limited

outputs are long enough for TPOT to matter

GPU decode interference is expensive

Main limitation:

SmartNIC memory/network bottleneck

UC San Diego

Thank you!