Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren,
Mingxing Zhang, Yongwei Wu, Weimin Zheng, Xinran Xu

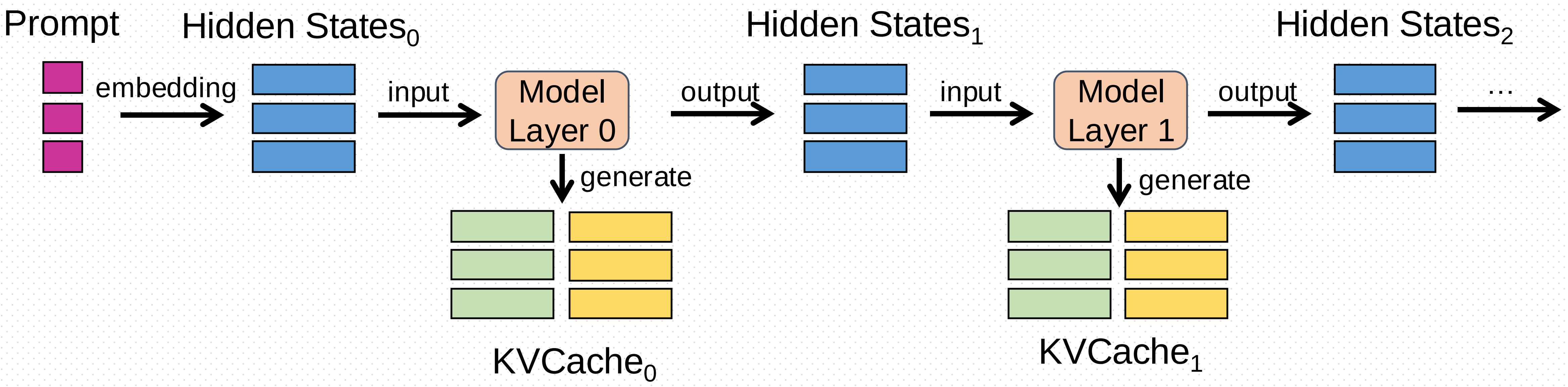
Shared by Juncheng Zhang

Outline

- ❑Prefix Caching
- ❑PD Disaggregation
- ❑Evaluation
- ❑Discussion

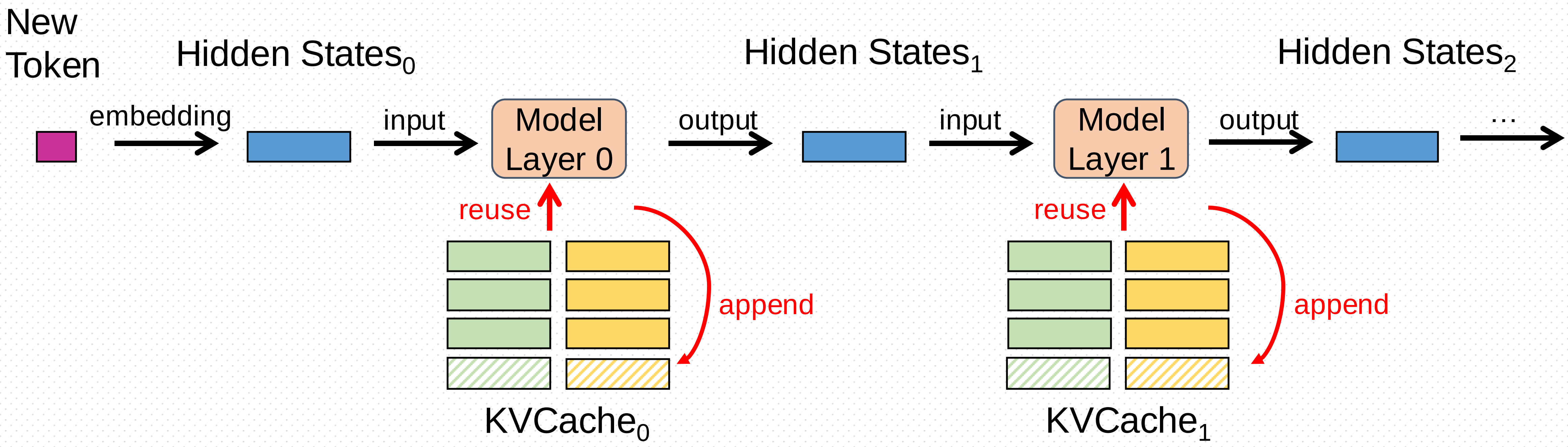
LLM Inference

- ❑LLM inference process can be divided into two phases
  - ❖**Prefill Phase**: generate KVCache and output first token



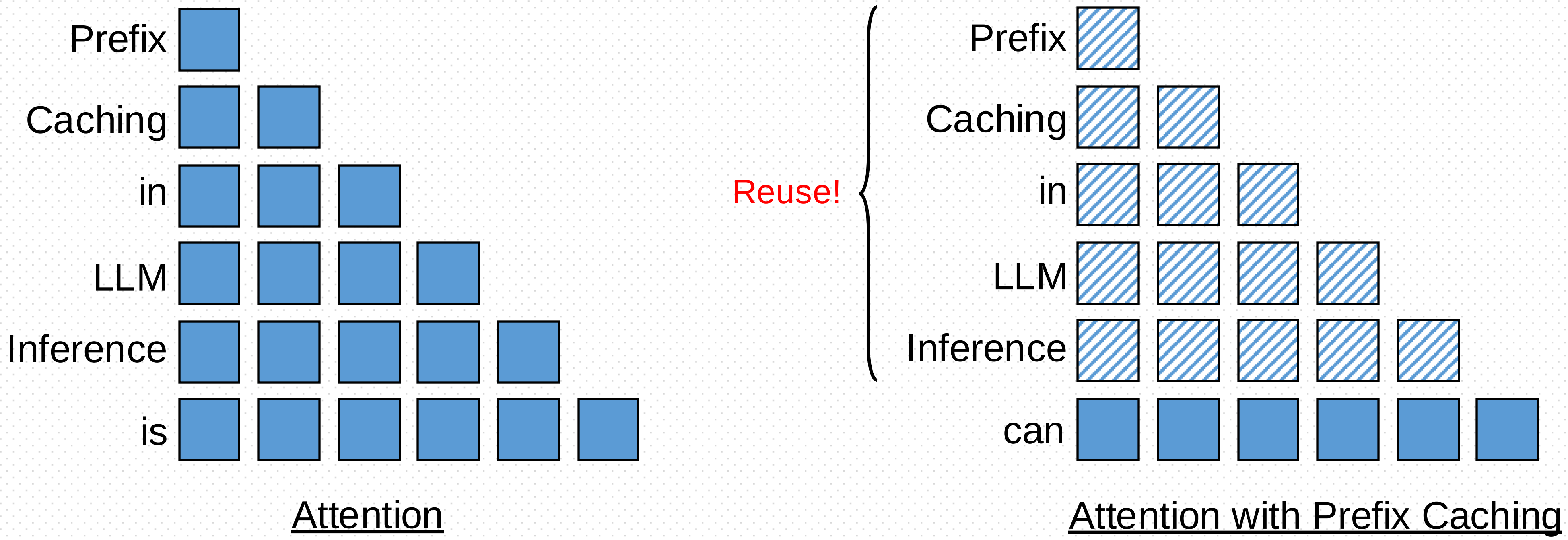
LLM Inference

- ❑LLM inference process can be divided into two phases
  - ❖**Decode Phase**: generate next token



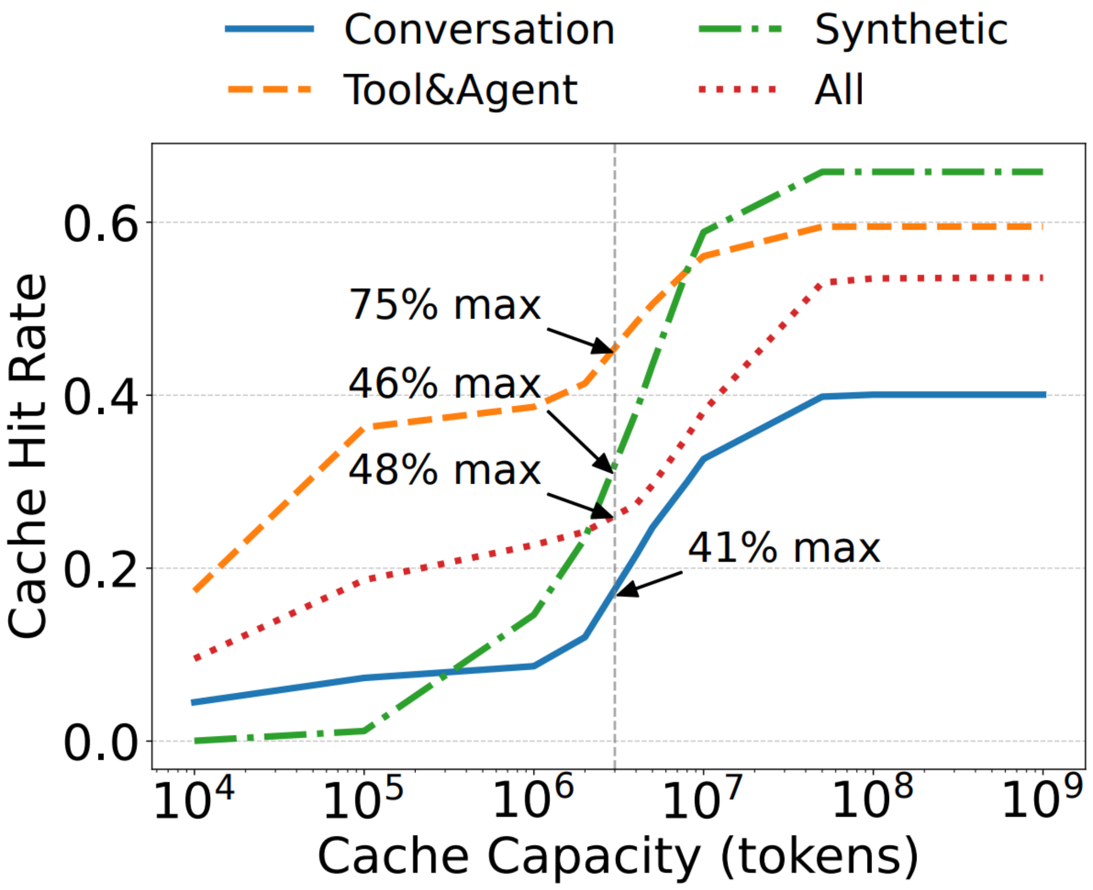
Prefix Caching in LLM Inference

- ❑ Requests with the same prefix can shared the same KVCache
  - ❖ Computation reduction in prefill phase



Prefix Caching in LLM Inference

❑Not easy in Kimi's real system deployment !!!



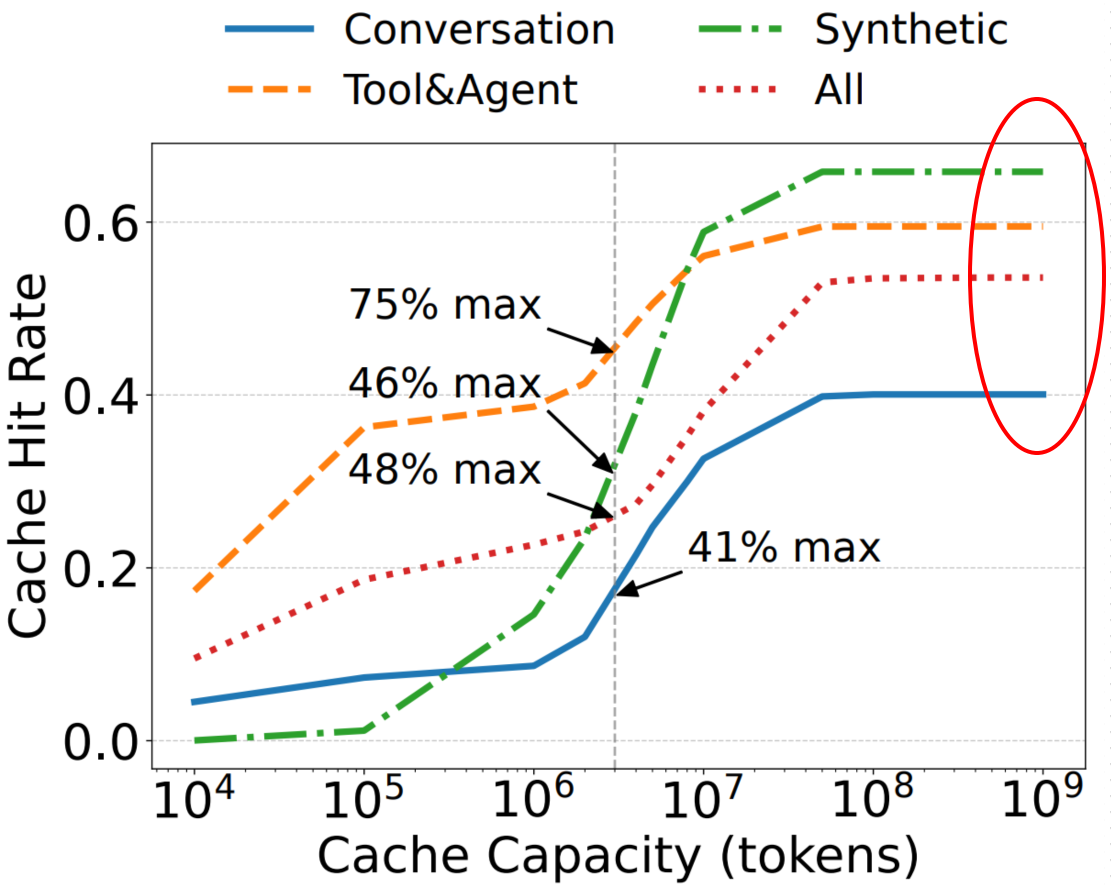
Conversation: collected from real-world online conversation requests

Tool & Agent: collected from real-world online requests that include tool use

Synthetic: synthesized from publicly available long context datasets

Prefix Caching in LLM Inference

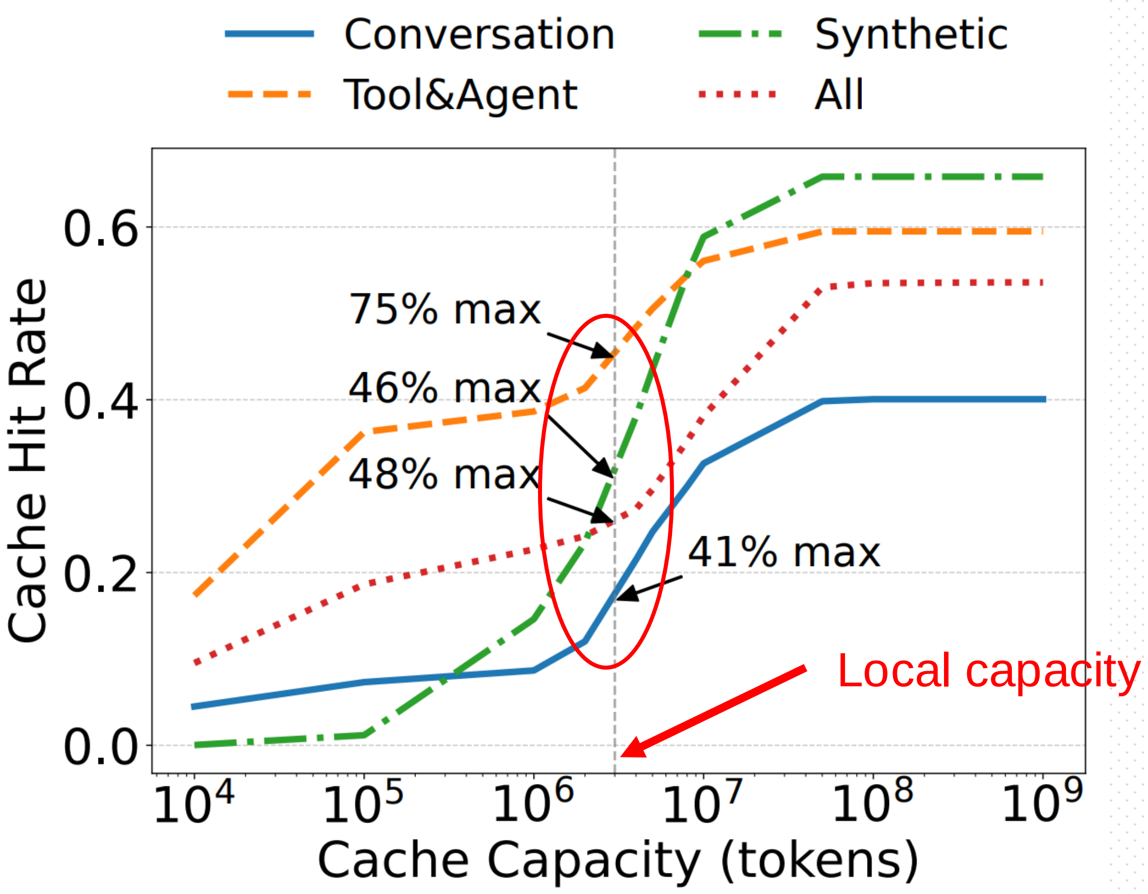
- ❑Not easy in Kimi's real system deployment !!!



In ideal case, around 50% of the token's KVCache can be reused

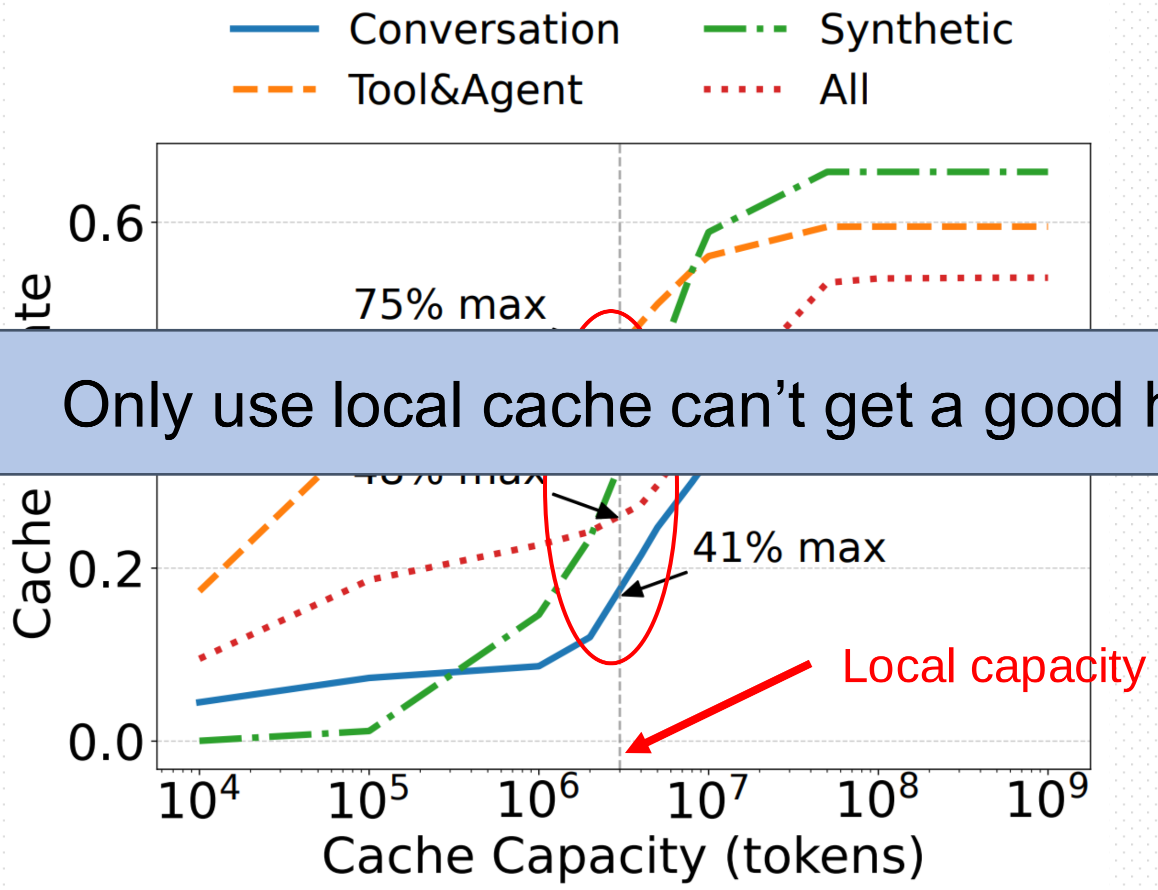
Prefix Caching in LLM Inference

❑Not easy in Kimi's real system deployment !!!



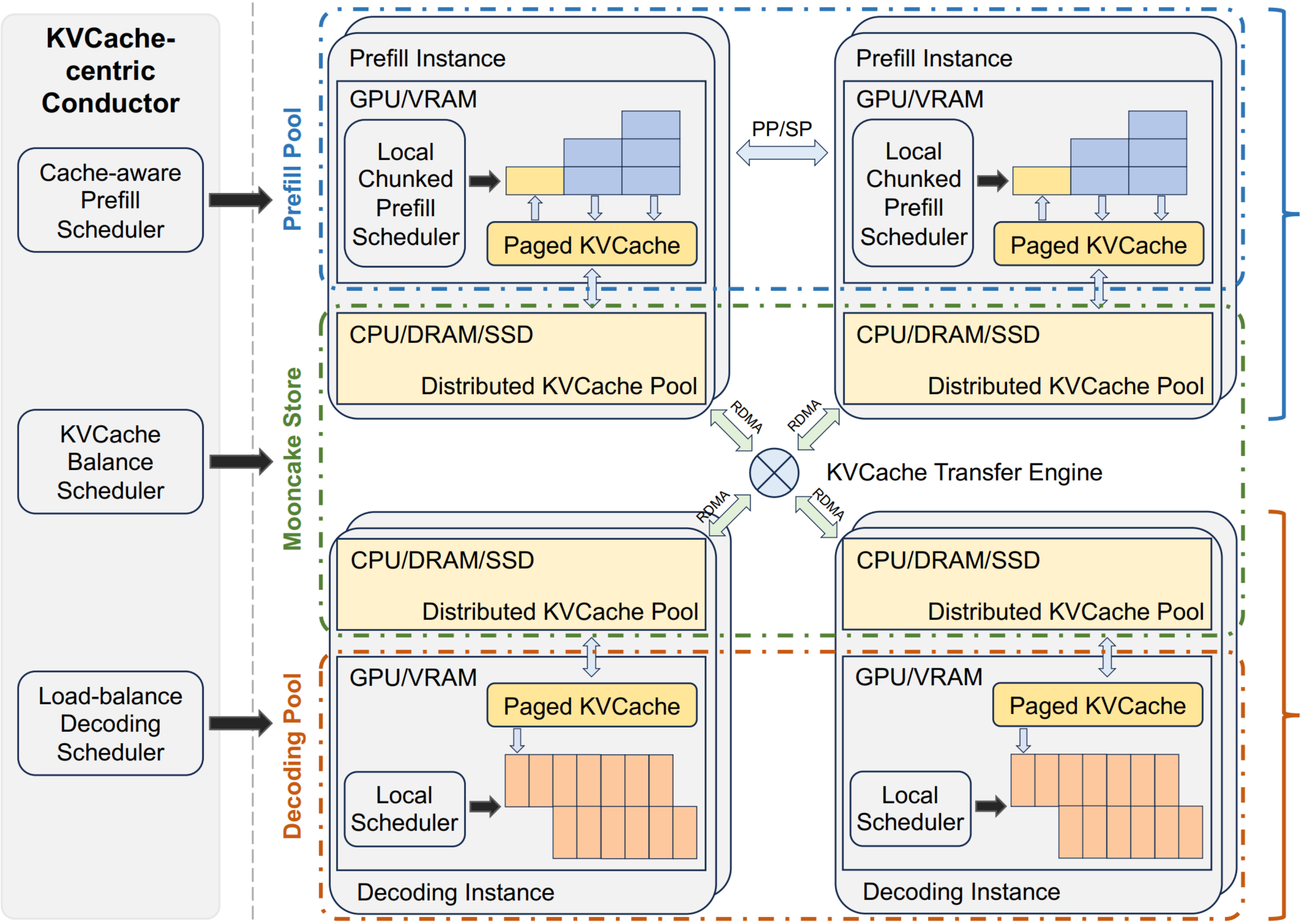
Prefix Caching in LLM Inference

- ❑Not easy in Kimi’s real system deployment !!!



Only use local cache can’t get a good hit ratio, Kimi needs a ***global cache***!

Mooncake Architecture



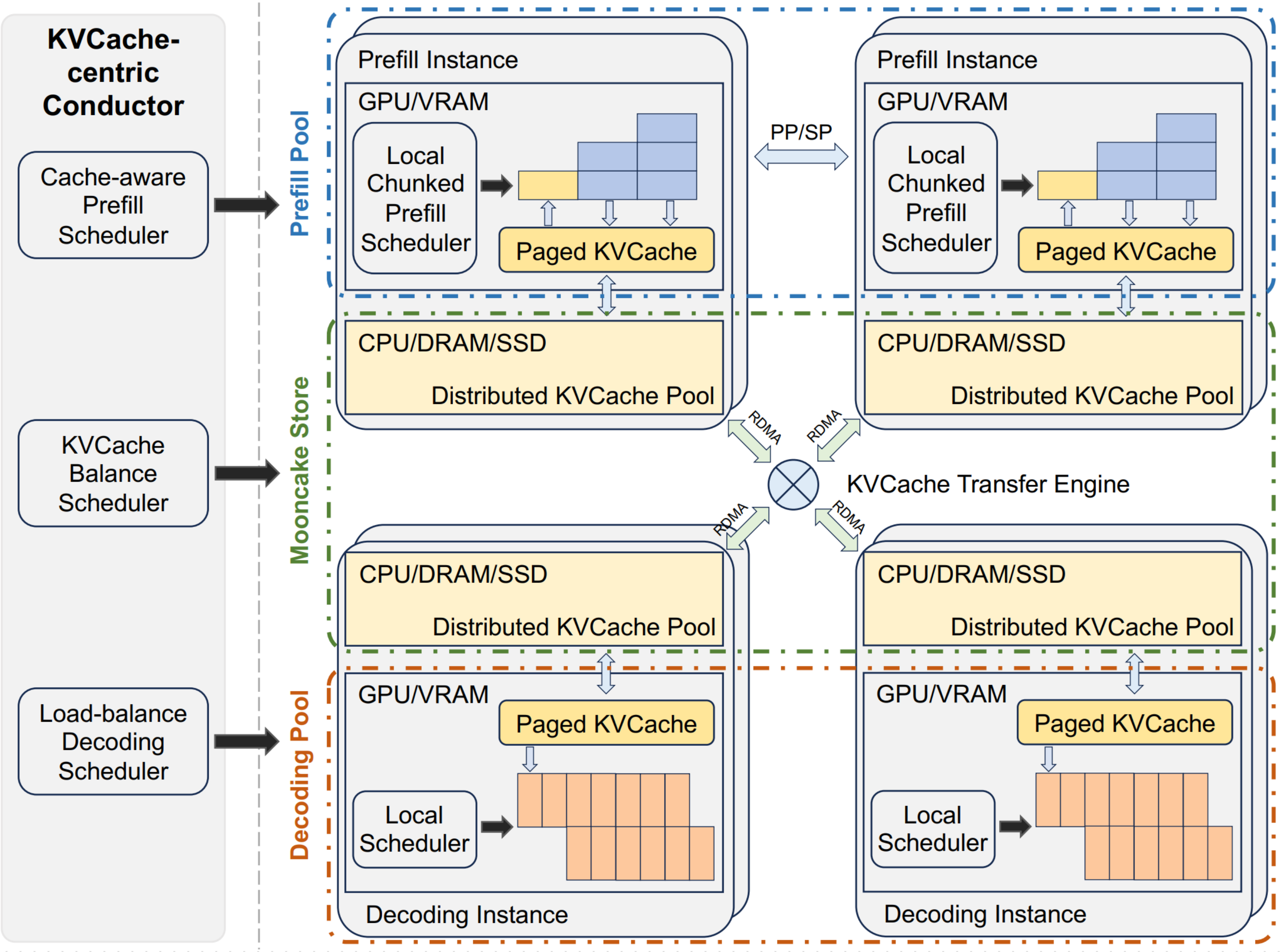
Prefill Stage Optimization Goal

max Cache Reuse
s.t.
TTFT SLO,
MFU Lower Bound,
KVCache < DRAM

Decoding Stage Optimization Goal

max Throughput
s.t.
TBT SLO,
KVCache < VRAM

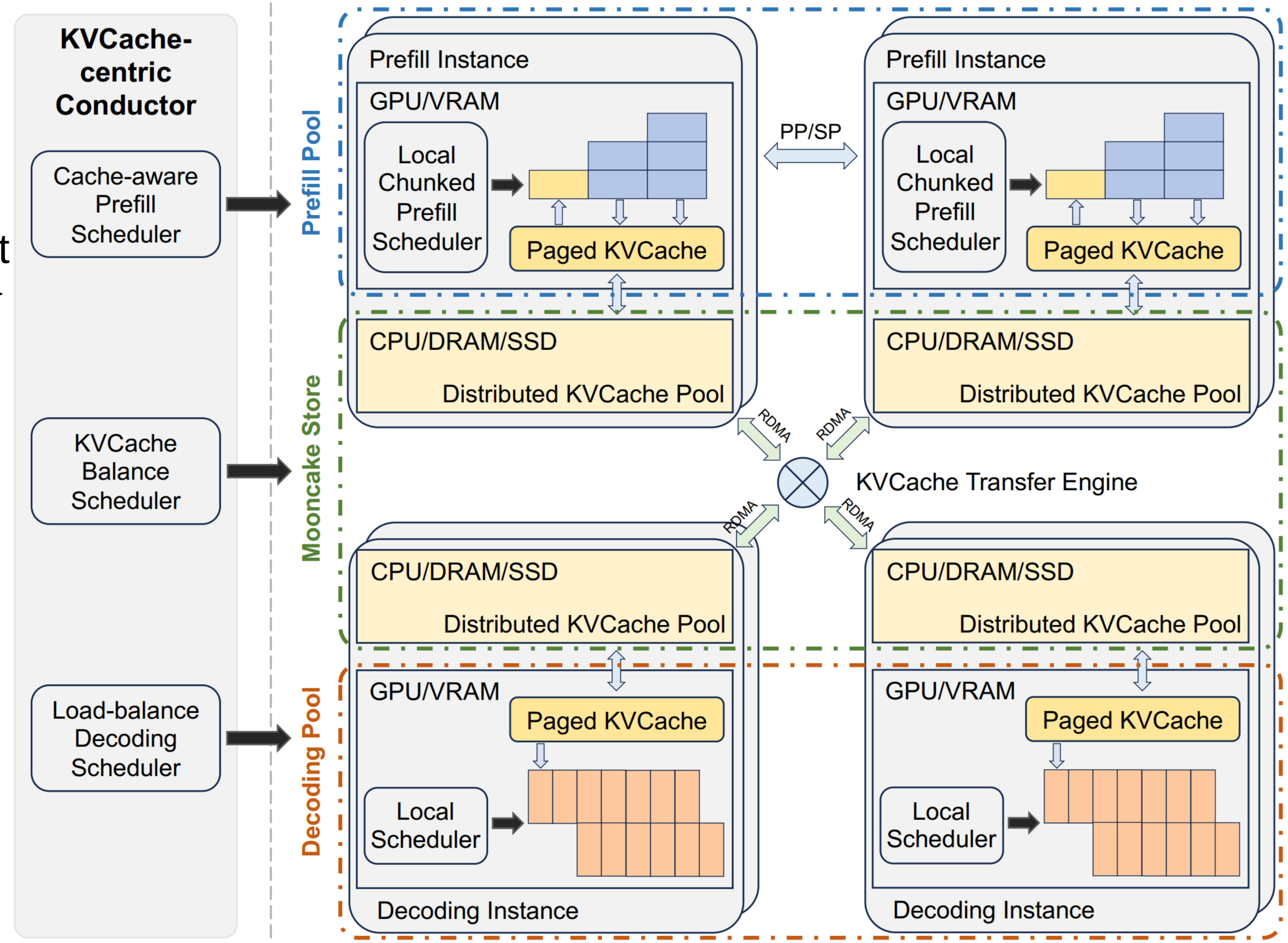
Mooncake Architecture



For simplicity, we leave the PD disaggregation in later discussion

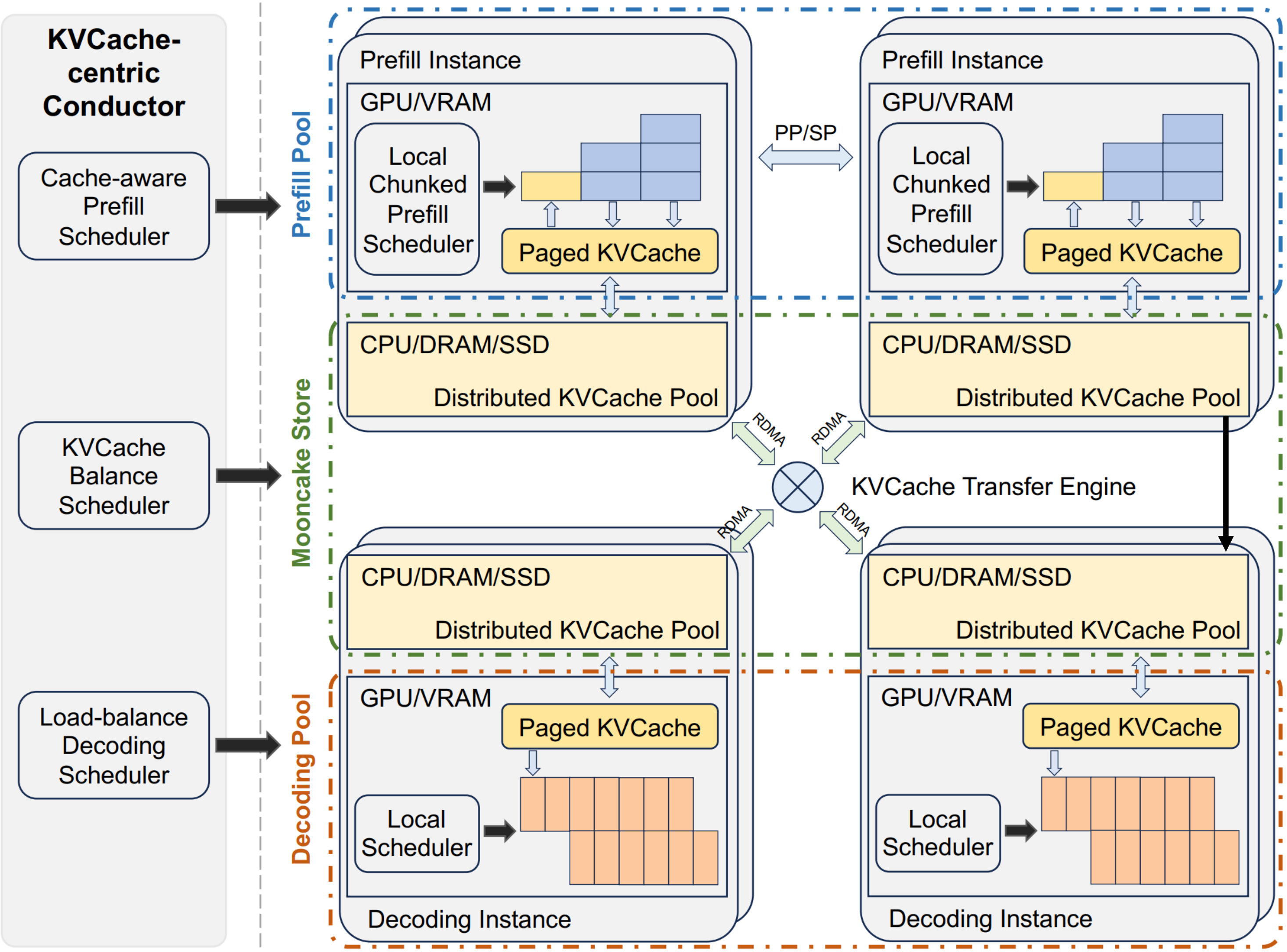
Mooncake Architecture

1. User request



2. Select *Prefill* and *Decode* instances

Mooncake Architecture



3. Prefill phase utilize *Prefix Caching*

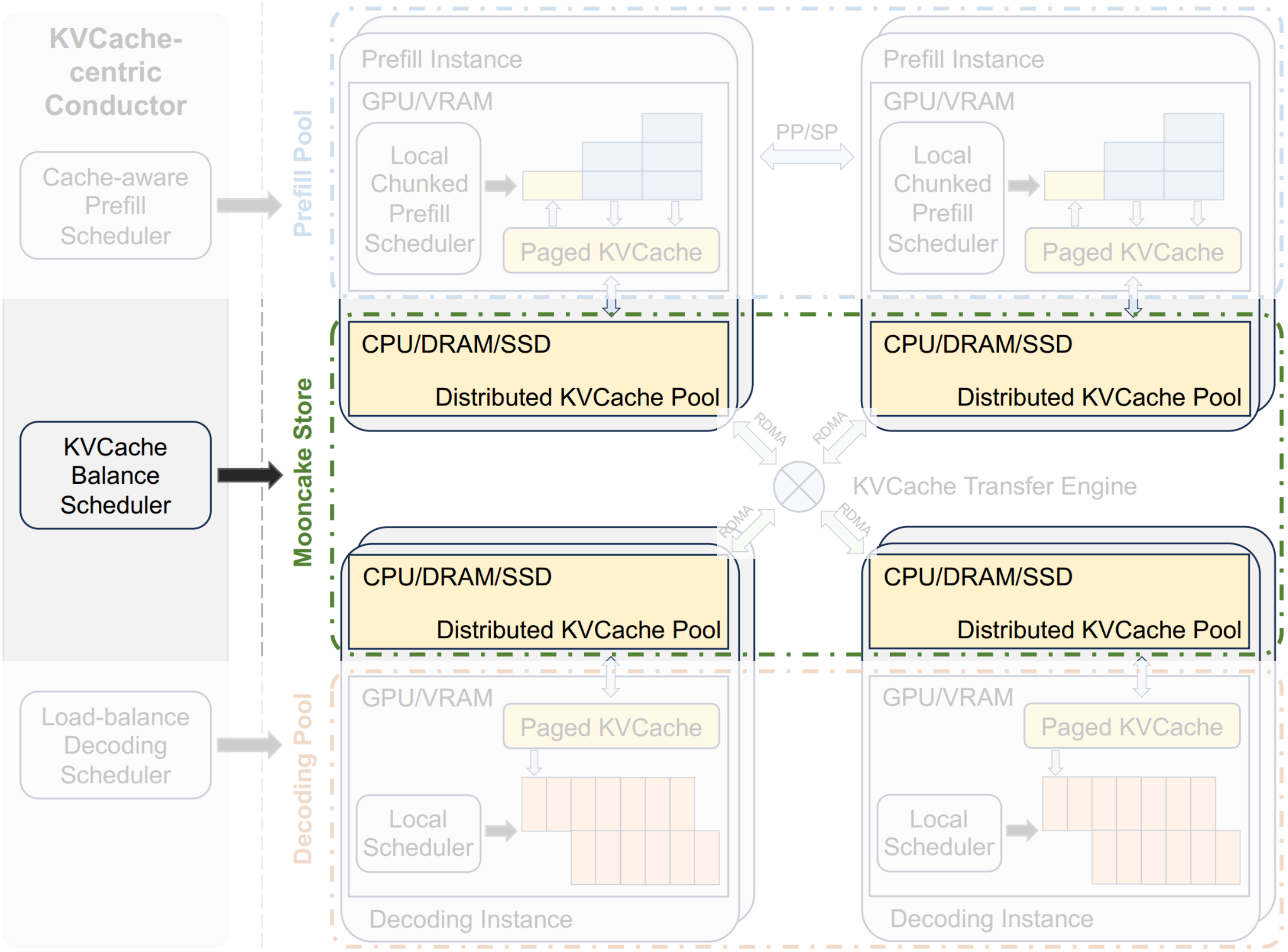
4. Transfer *KVCache* from Prefill instance to Decode instance

5. Finish inference, return to user

Challenge in global cache

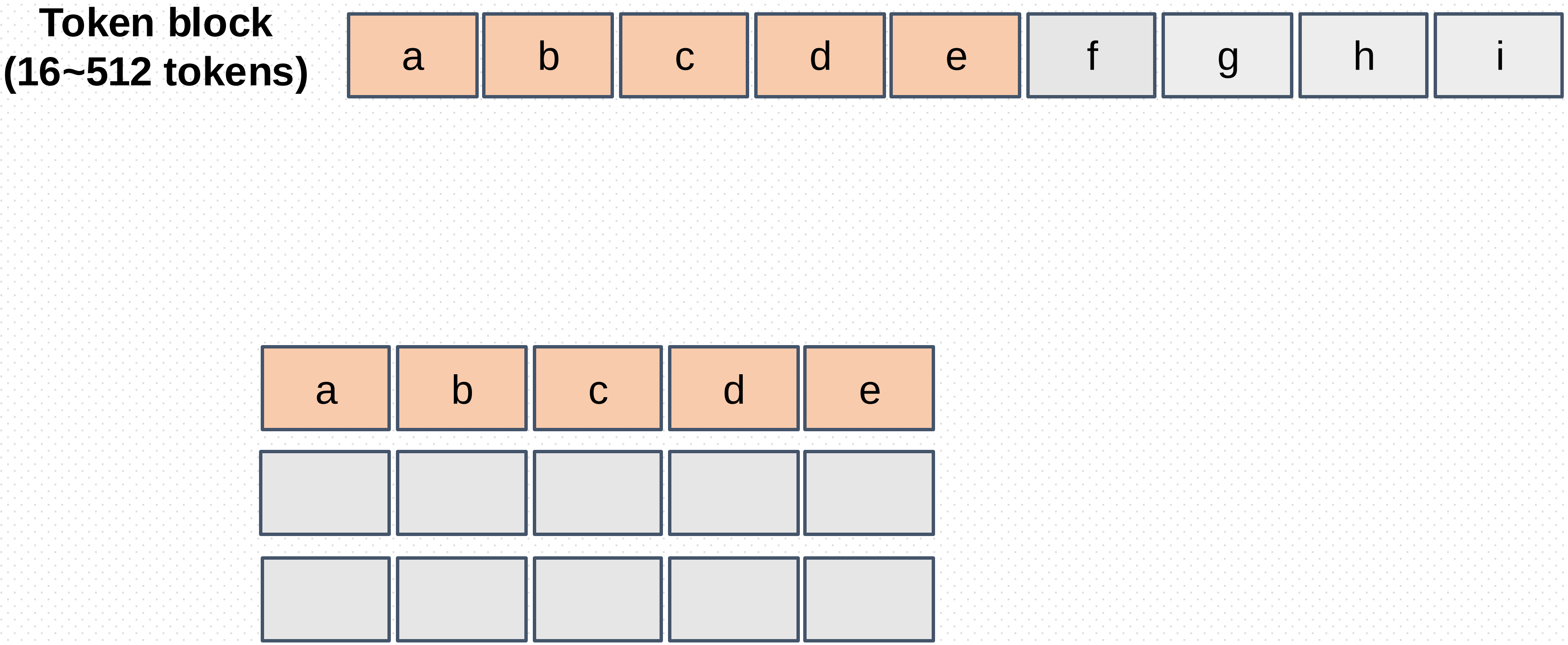
- Aggregate all the usable resource
  - A distributed multi-layer *KVCache pool*
- KVCache needs to be transferred between different machines
  - A low latency, high bandwidth *transfer engine*
- User request dispatch should consider prefix cache
  - *KVCache-aware scheduling*

Mooncake Architecture



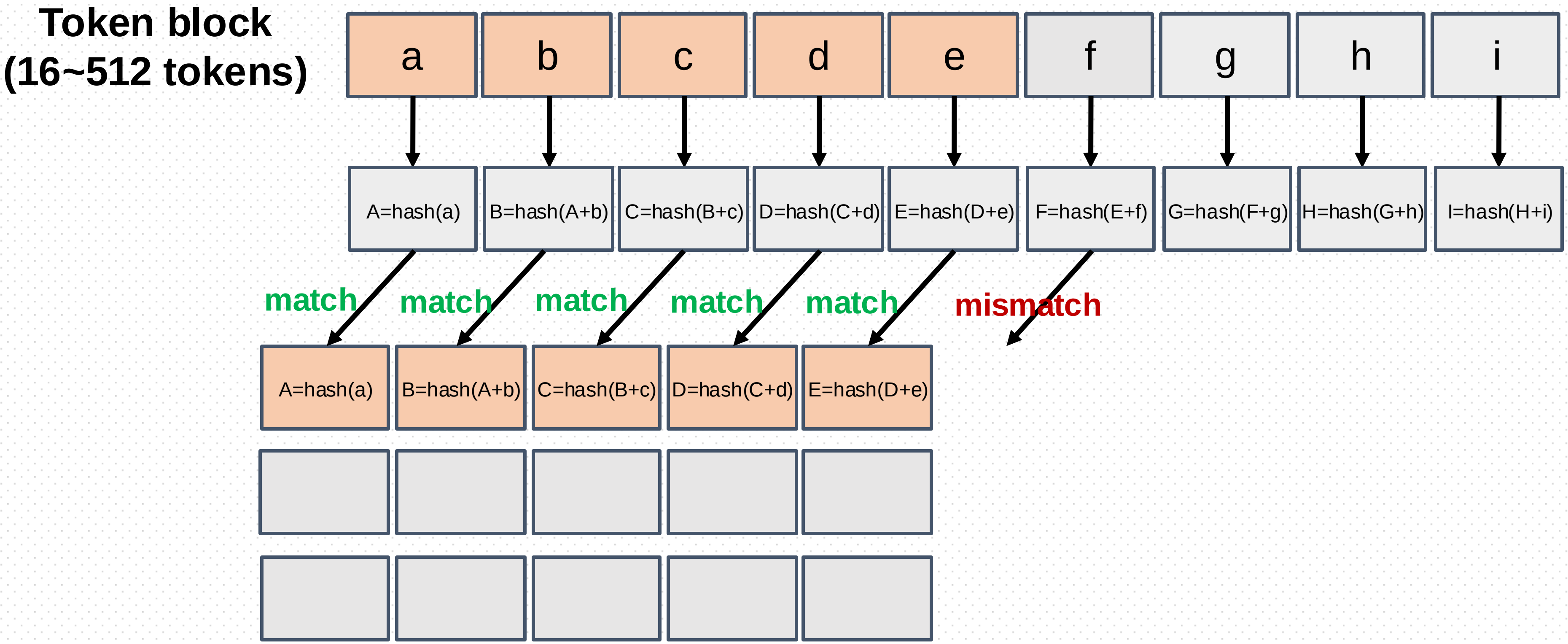
Mooncake Store

- Block-level KVCache management



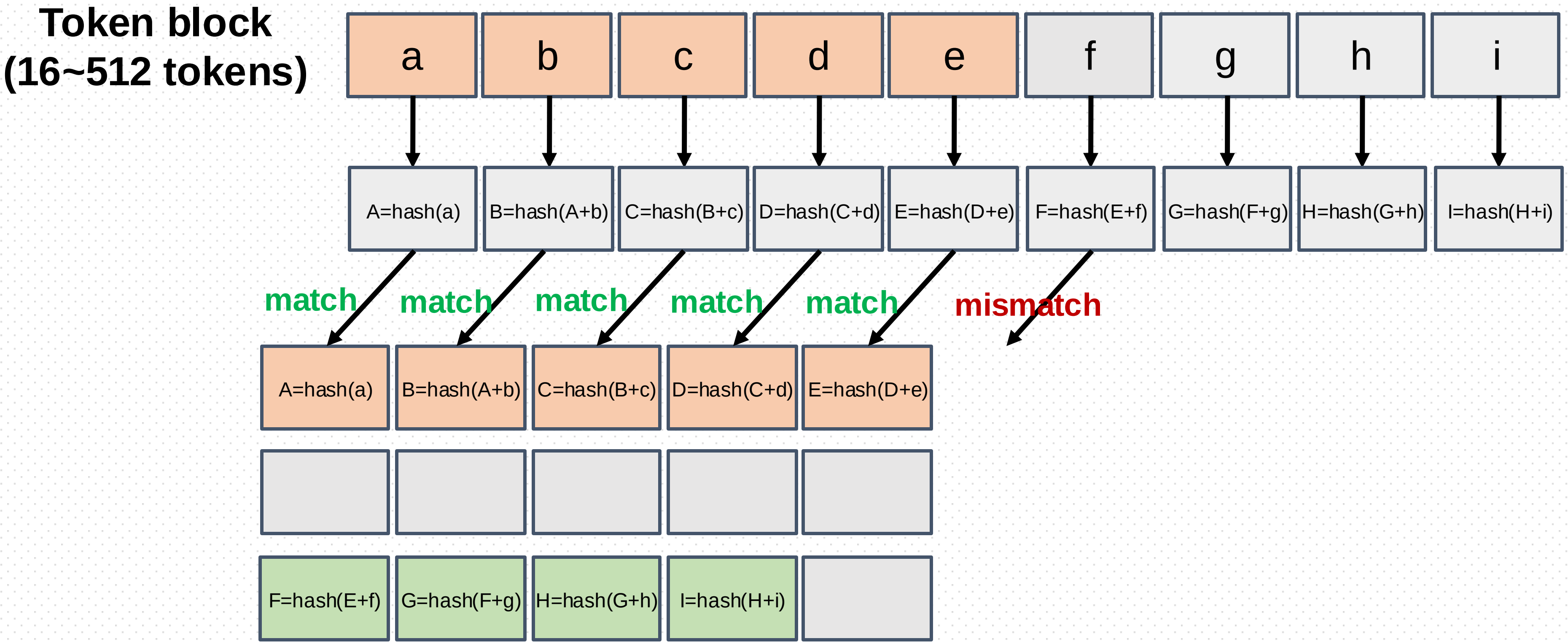
Mooncake Store

- ❑Block-level KVCache management
- ❑Prefix-hashed for fast match and deduplication



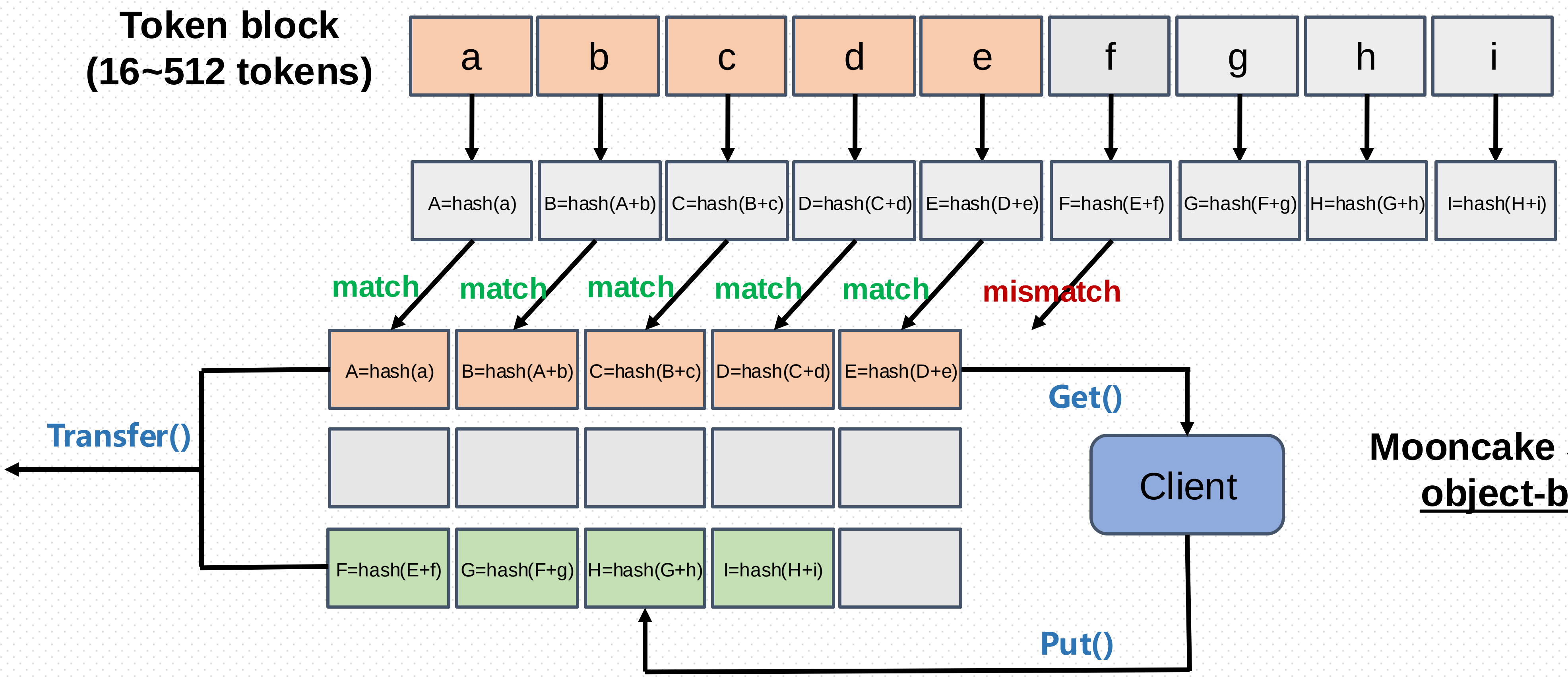
Mooncake Store

- ❑Block-level KVCache management
- ❑Prefix-hashed for fast match and deduplication



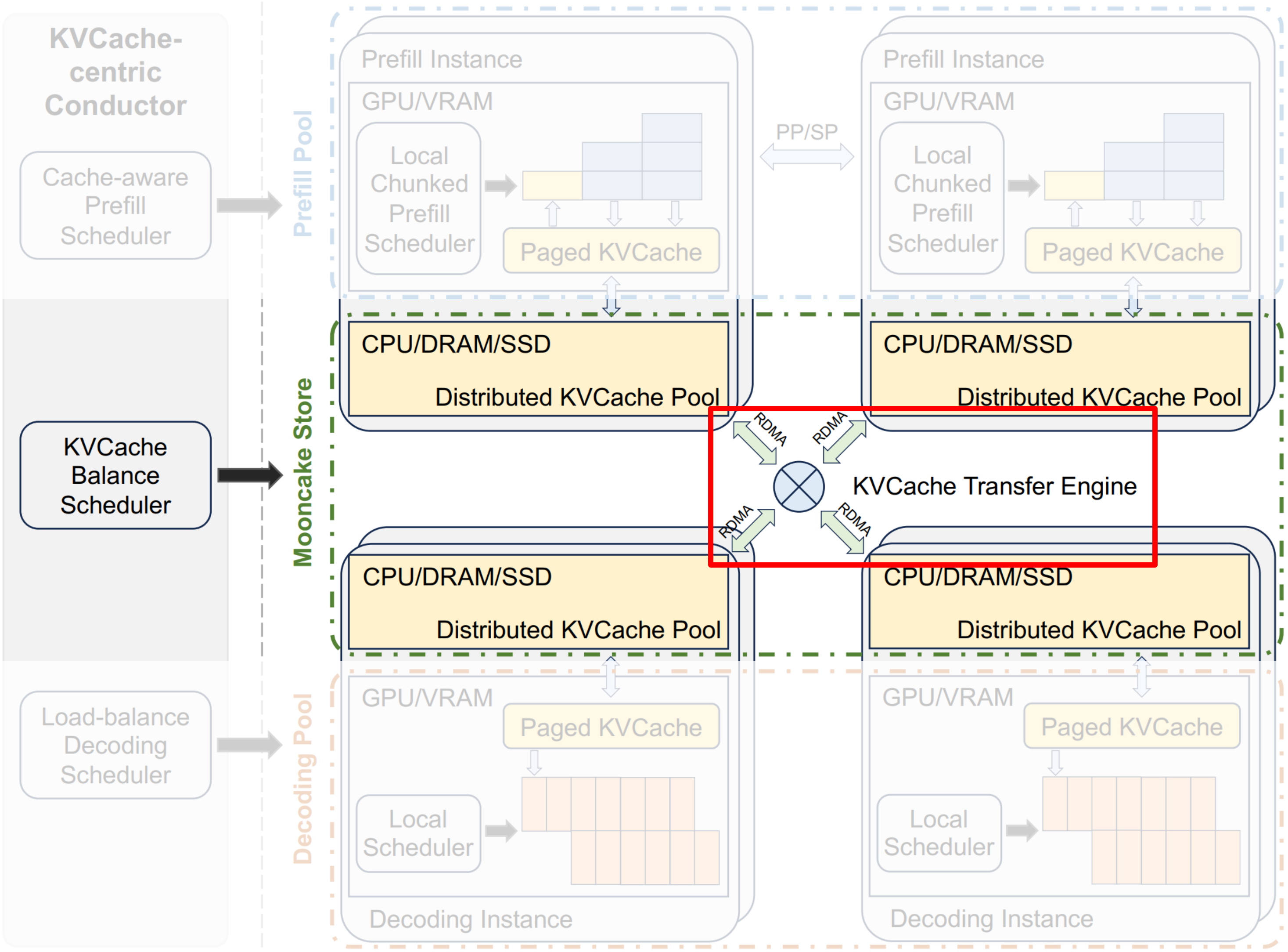
Mooncake Store

- ❑Block-level KVCache management
- ❑Prefix-hashed for fast match and deduplication



Mooncake Store offers <u>object-based</u> API

MoonCake Architecture



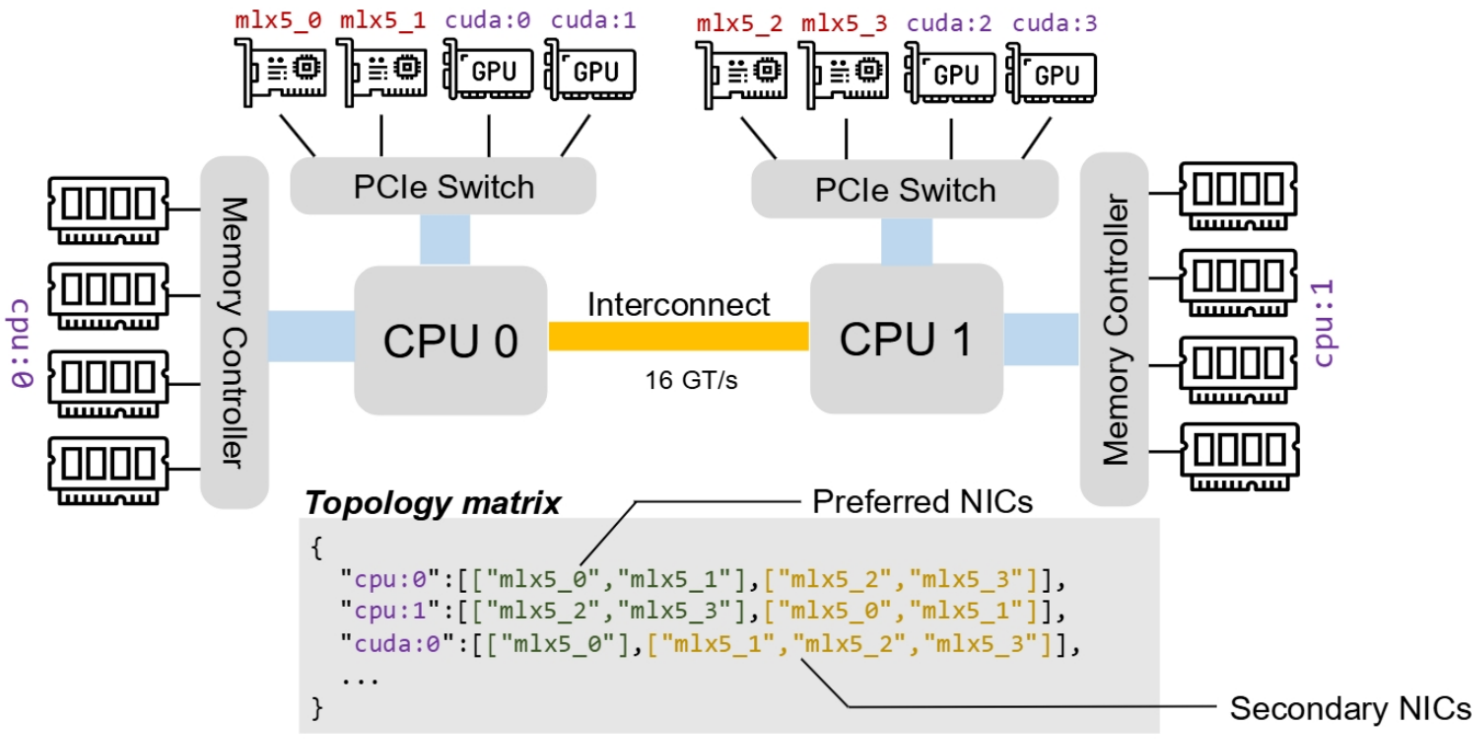
Transfer Engine

❑Topology Aware Path Selection

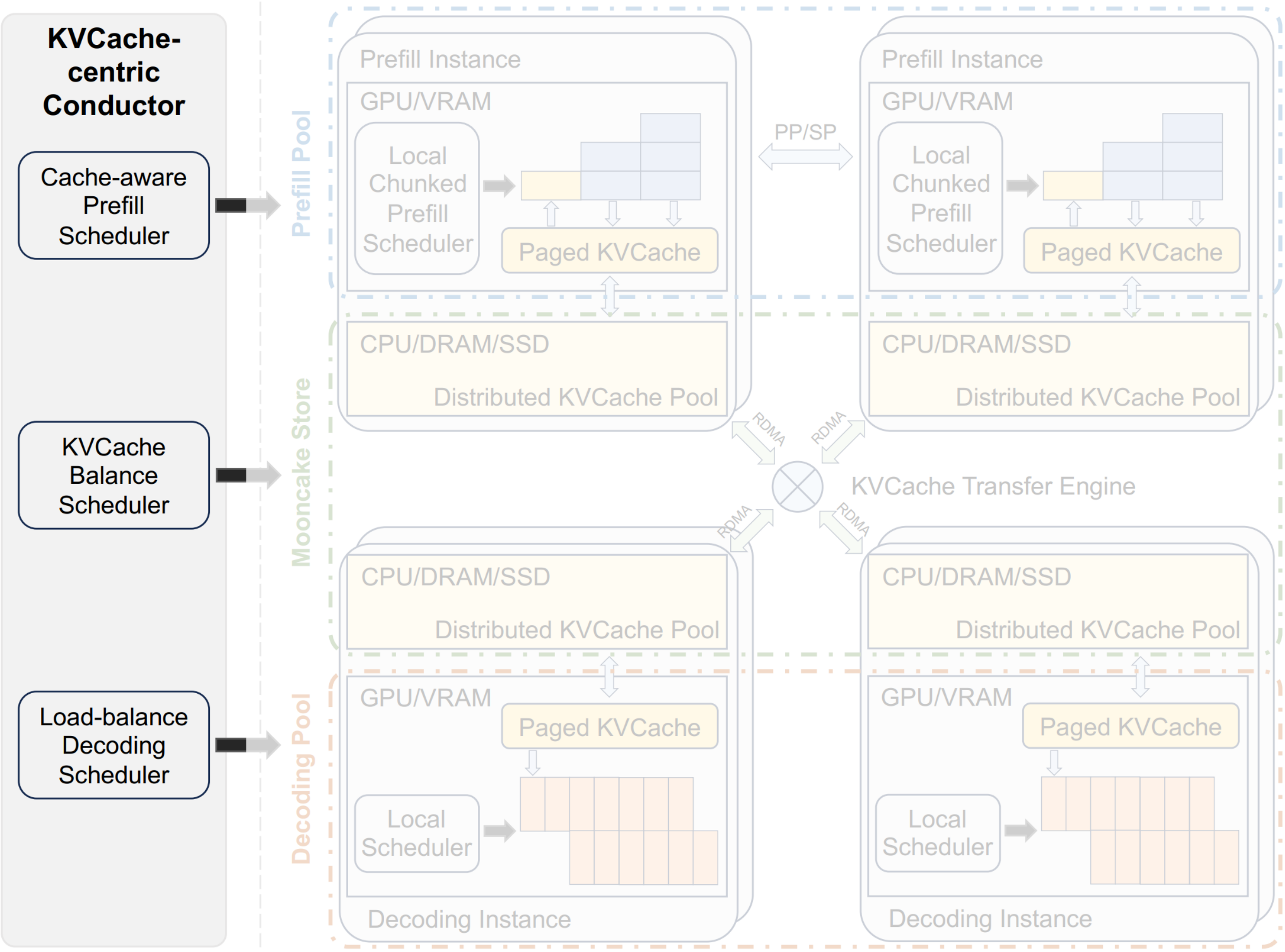
- ❖ Broadcast the topology matrix across the cluster

❑Endpoint pooling

- ❖ Use SIEVE for eviction



Mooncake Architecture



KVCache-centric Scheduling

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.

Output: the prefill and decoding instances (p, d) to process R.

$block\_keys \leftarrow \text{PrefixHash}(R.prompt\_tokens, B)$
$TTFT, p \leftarrow \text{inf}, \emptyset$
$best\_len, best\_instance \leftarrow \text{FindBestPrefixMatch}(P, block\_keys)$
for $instance \in P$ **do**
  if $\frac{best\_len}{instance.prefix\_len} >$ **kvcache_balancing_threshold then**
    $prefix\_len \leftarrow best\_len$
    $transfer\_len \leftarrow best\_len - instance.prefix\_len$
    $T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(transfer\_len)$
  else
    $prefix\_len \leftarrow instance.prefix\_len$
    $T_{transfer} \leftarrow 0$
  $T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$
  $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}(\text{len}(R.prompt\_tokens), prefix\_len)$
  if $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
    $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
    $p \leftarrow instance$
$d, TBT \leftarrow \text{SelectDecodingInstance}(D)$
if $TTFT > TTFT\_SLO$ **or** $TBT > TBT\_SLO$ **then**
  reject R; **return**
if $\frac{best\_len}{p.prefix\_len} >$ **kvcache_balancing_threshold then**
  $\text{TransferKVCache}(best\_instance, p)$
return (p, d)

KVCache-centric Scheduling

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.

Output: the prefill and decoding instances (p, d) to process R.

Estimate Transfer Time

```
block_keys ← PrefixHash(R.prompt_tokens, B)
TTFT, p ← inf, ∅
best_len, best_instance ← FindBestPrefixMatch(P, block_keys)
for instance ∈ P do
    if best_len / instance.prefix_len > kvcache_balancing_threshold then
        pre...
        transfer_len ← best_len − instance.prefix_len
        T_transfer ← EstimateKVCacheTransferTime(transfer_len)
    else
        prefix_len ← instance.prefix_len
        T_transfer ← 0
```

Estimate Queue and Prefill Time

```
    T_queue ← EstimatePrefillQueueTime(instance)
    ...
        len(R.prompt_tokens), prefix_len)
```

Choose Prefill and Decode Instance

```
    if TTFT > T_transfer + T_queue + T_prefill then
        TTFT ← T_transfer + T_queue + T_prefill
        ...
d, TBT ← SelectDecodingInstance(D)
```

Other

```
if TTFT > TTFT_SLO or TBT > TBT_SLO then
    reject R; return
if best_len / p.prefix_len > kvcache_balancing_threshold then
    TransferKVCache(best_instance, p)
return (p, d)
```

KVCache-centric Scheduling

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.

Output: the prefill and decoding instances (p, d) to process R.

$block\_keys \leftarrow \text{PrefixHash}(R.prompt\_tokens, B)$
$TTFT, p \leftarrow \text{inf}, \emptyset$
$best\_len, best\_instance \leftarrow \text{FindBestPrefixMatch}(P, block\_keys)$
for $instance \in P$ **do**
    if $\frac{best\_len}{instance.prefix\_len} >$ **kvcache_balancing_threshold then**
        $prefix\_len \leftarrow best\_len$
        $transfer\_len \leftarrow best\_len - instance.prefix\_len$
        $T_{transfer} \leftarrow \text{EstimateKVCacheTransferTime}(transfer\_len)$
    else
        $prefix\_len \leftarrow instance.prefix\_len$
        $T_{transfer} \leftarrow 0$
    $T_{queue} \leftarrow \text{EstimatePrefillQueueTime}(instance)$
    $T_{prefill} \leftarrow \text{EstimatePrefillExecutionTime}($
            $\text{len}(R.prompt\_tokens), prefix\_len)$
    if $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ **then**
        $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
        $p \leftarrow instance$
$d, TBT \leftarrow \text{SelectDecodingInstance}(D)$
if $TTFT > TTFT\_SLO$ **or** $TBT > TBT\_SLO$ **then**
    reject R; **return**
if $\frac{best\_len}{p.prefix\_len} >$ **kvcache_balancing_threshold then**
    $\text{TransferKVCache}(best\_instance, p)$
return (p, d)

Design1: TTFT prioritized scheduling

KVCache-centric Scheduling

Algorithm 1 KVCache-centric Scheduling Algorithm

Input: prefill instance pool P, decoding instance pool D, request R, cache block size B.

Output: the prefill and decoding instances (p,d) to process R.

```
block_keys ← PrefixHash(R.prompt_tokens, B)
TTFT, p ← inf, ∅
best_len, best_instance ← FindBestPrefixMatch(P, block_keys)
for instance ∈ P do
    if best_len / instance.prefix_len > kvcache_balancing_threshold then
        prefix_len ← best_len
        transfer_len ← best_len − instance.prefix_len
        T_transfer ← EstimateKVCacheTransferTime(transfer_len)
    else
        prefix_len ← instance.prefix_len
        T_transfer ← 0
    T_queue ← EstimatePrefillQueueTime(instance)
    T_prefill ← EstimatePrefillExecutionTime(
                    len(R.prompt_tokens), prefix_len)
    if TTFT > T_transfer + T_queue + T_prefill then
        TTFT ← T_transfer + T_queue + T_prefill
        p ← instance
d, TBT ← SelectDecodingInstance(D)
if TTFT > TTFT_SLO or TBT > TBT_SLO then
    reject R; return
if best_len / p.prefix_len > kvcache_balancing_threshold then
    TransferKVCache(best_instance, p)
return (p, d)
```

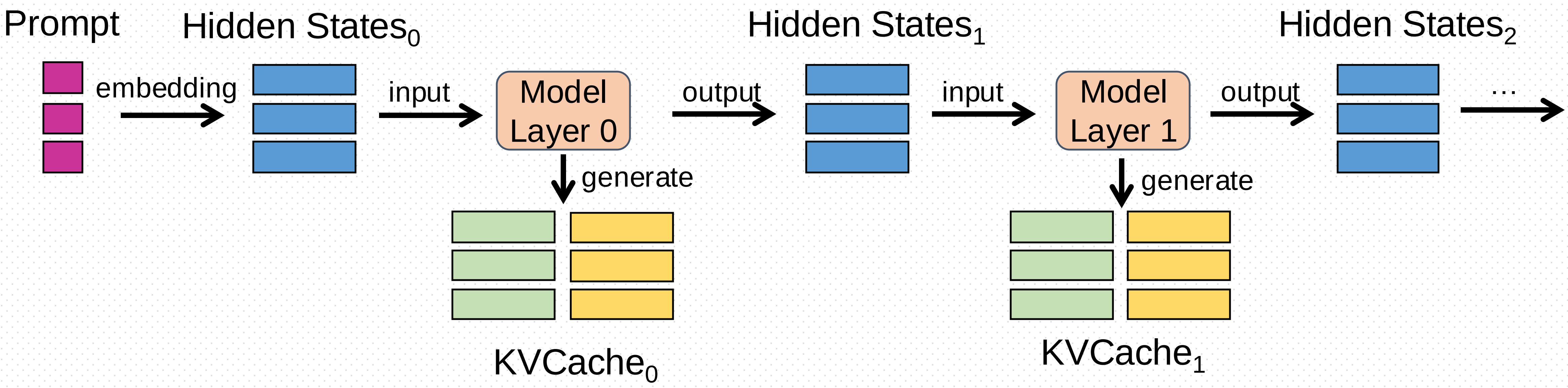
Design2: KVCache load balancing

Summary of Prefix Caching

- Mooncake Store
  - Prefix-hashed, Object-based API
- Transfer Engine
  - Topology-aware path selection, endpoint pooling
- KVCache-centric Scheduling
  - TTFT prioritized scheduling, KVCache load balance

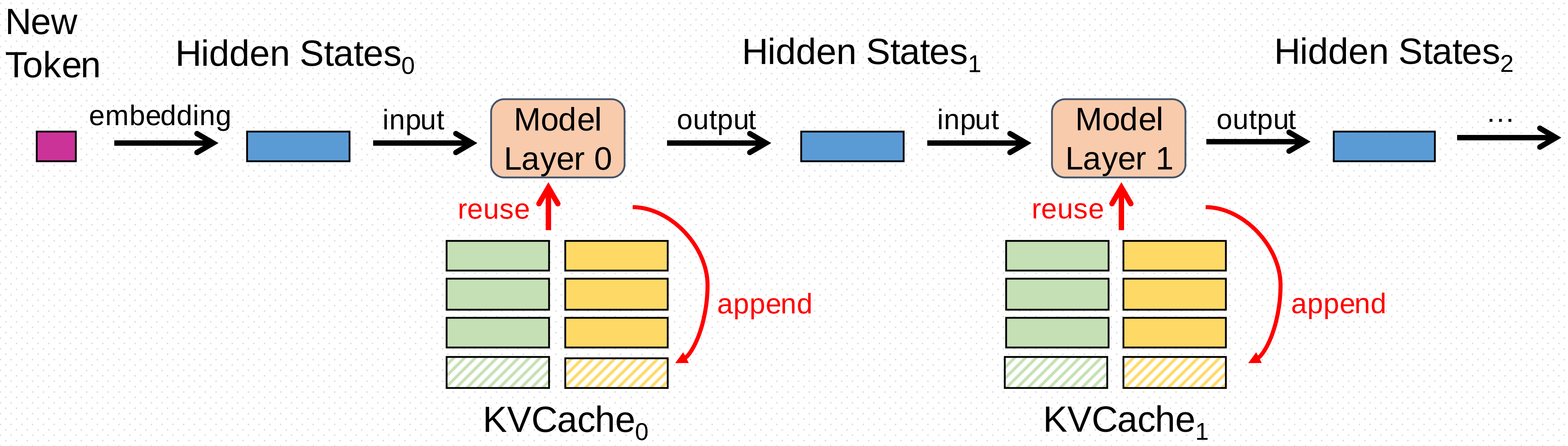
LLM Inference

- ❑LLM inference process can be divided into two phases
  - ❖**Prefill Phase**: generate KVCache and output first token



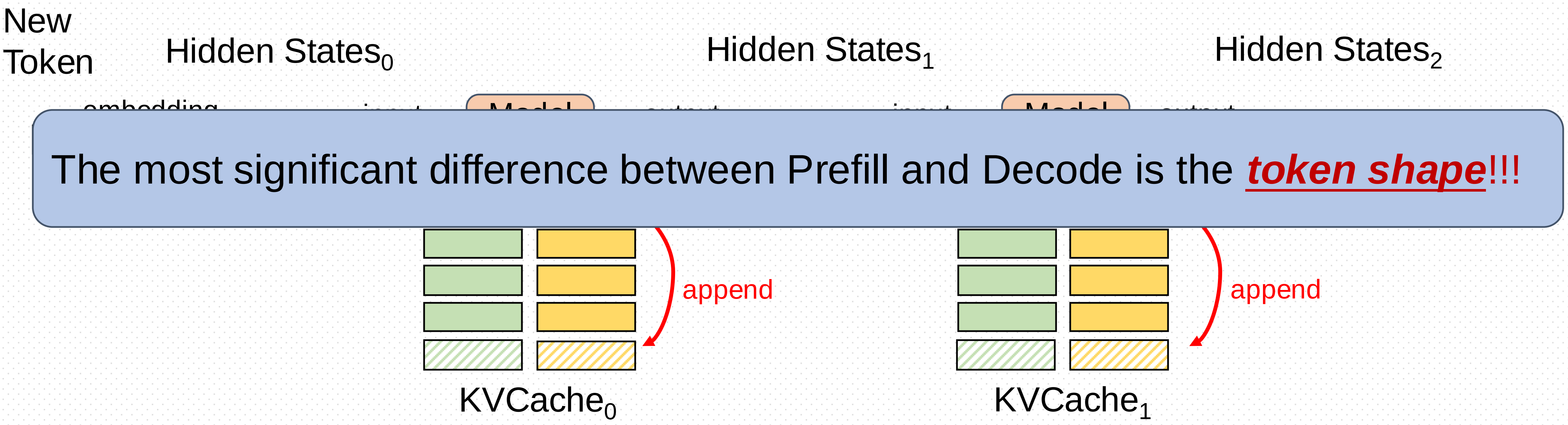
LLM Inference

- LLM inference process can be divided into two phases
  - **Decode Phase**: generate next token



LLM Inference

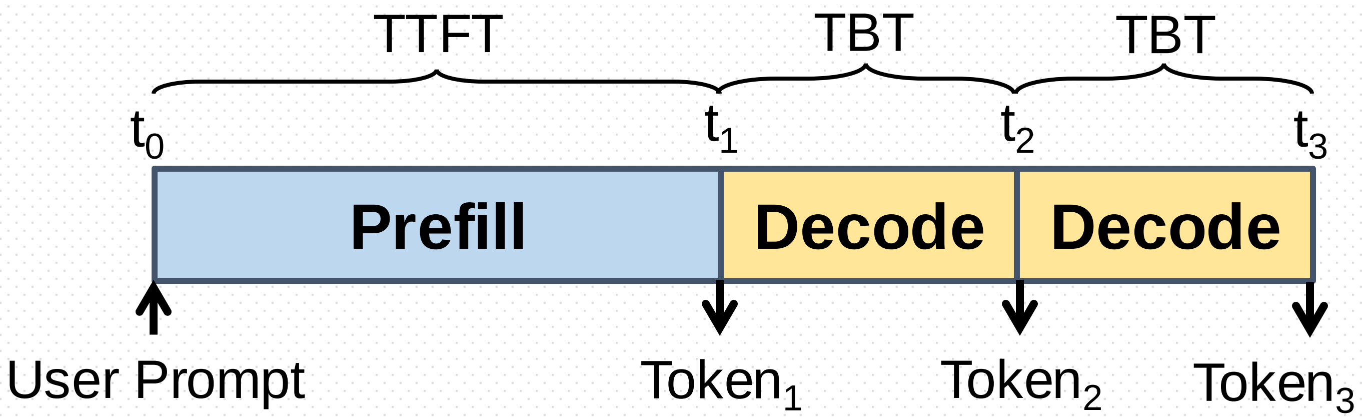
- LLM inference process can be divided into two phases
  - **Decode Phase**: generate next token



The most significant difference between Prefill and Decode is the ***token shape***!!!

Performance Metrics

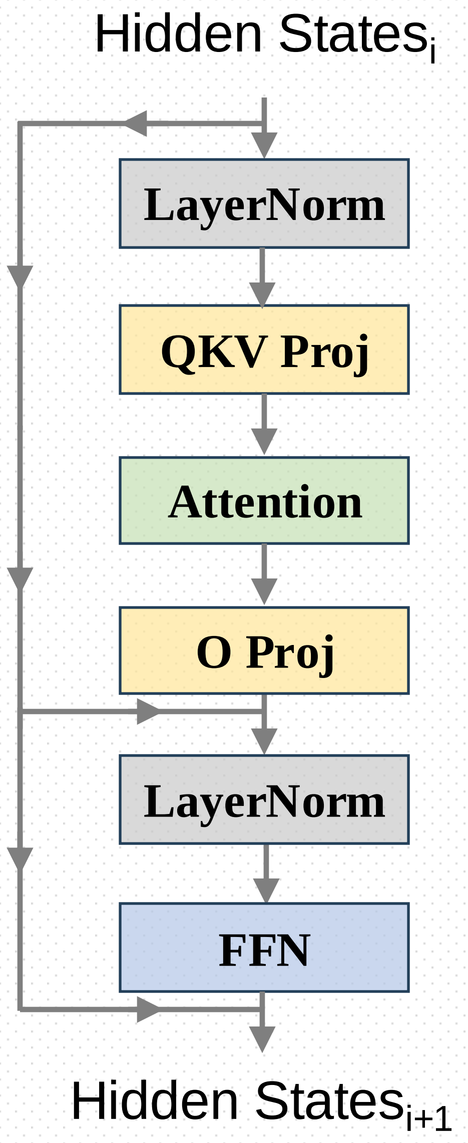
- Latency
  - Time-to-first-token (*TTFT*)
    - Latency metric for *Prefill*
  - Time-between-token (*TBT*)
    - Latency metric for *Decode*



- Throughput
  - Model-Flops-Utilization (*MFU*)
    - Measured Flops / Theoretical Upper Bound

Layer Computation Details

- ❑There are four operations in single layer
  - ❖Proj
  - ❖Attention
  - ❖FFN
  - ❖Layer Norm



Layer Computation Details

❑Some notations on model parameters

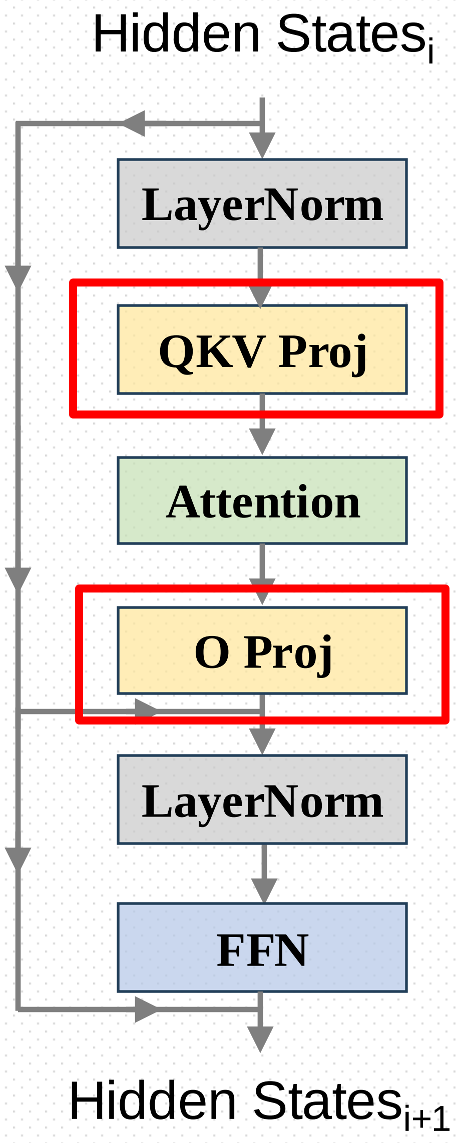
Parameter	Notation
Batch size	$\boldsymbol{b}$
Num head	$\boldsymbol{n}$
Sequence length	$\boldsymbol{s}$
Head dimension	$\boldsymbol{h}$
Hidden dimension	$\boldsymbol{d}$

Layer Computation Details

❑Proj

- ❖Shape: [d, d]
- ❖Input
  - ➢Prefill: [b, s, d]
  - ➢Decode: [b, 1, d]
- ❖HBM load num:
  - ➢Prefill: d^2+bsd
  - ➢Decode: d^2+bd
- ❖Computation Ops:
  - ➢Prefill: bsd^2
  - ➢Decode: bd^2

Parameter	Notation
Batch size	b
Num head	n
Sequence length	s
Head dimension	h
Hidden dimension	d



Layer Computation Details

❑Proj

❖Shape: [***d***, ***d***]

❖Input

➢Prefill: [***b, s, d***]

➢Decode: [***b, 1, d***]

❖HBM load num:

➢Prefill: ***d^2+bsd***

➢Decode: ***d^2+bd***

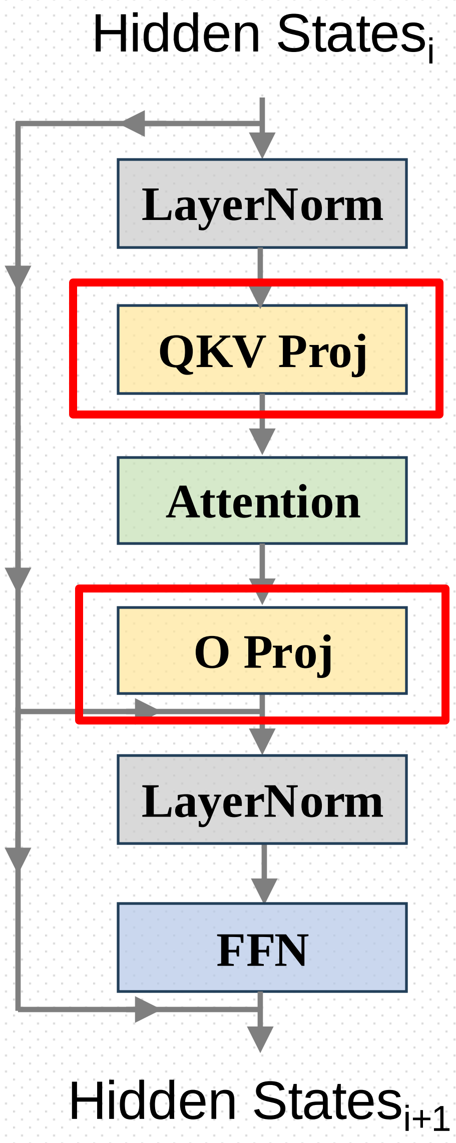
❖Computation Ops:

➢Prefill: ***bsd^2***

➢Decode: ***bd^2***

Parameter	Notation
Batch size	***b***
Num head	***n***
Sequence length	***s***
Head dimension	***h***
Hidden dimension	***d***

As ***b*** and ***s*** increases, ***proj*** quickly becomes ***comp-bound*** task



Layer Computation Details

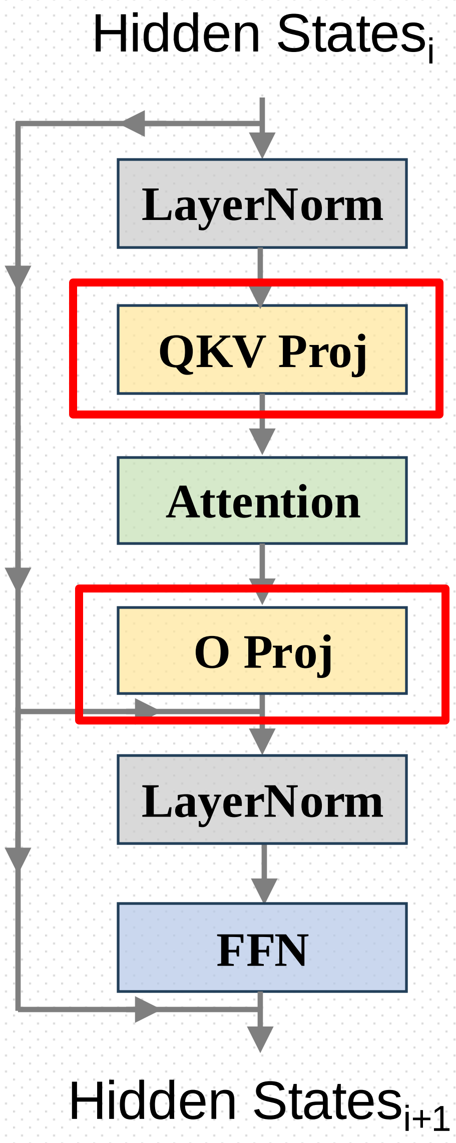
❑Proj

- ❖Shape: [***d***, ***d***]
- ❖Input
  - ➢Prefill: [***b***, ***s***, ***d***]
  - ➢Decode: [***b***, ***1***, ***d***]
- ❖HBM load num:
  - ➢Prefill: d^2+bsd
  - ➢Decode: d^2+bd
- ❖Computation Ops:
  - ➢Prefill: bsd^2
  - ➢Decode: bd^2

Parameter	Notation
Batch size	b
Num head	n
Sequence length	s
Head dimension	h
Hidden dimension	d

As ***b*** and ***s*** increases, ***proj*** quickly becomes ***comp-bound*** task

In other words, even with small ***b***, ***proj*** can use up comp resource



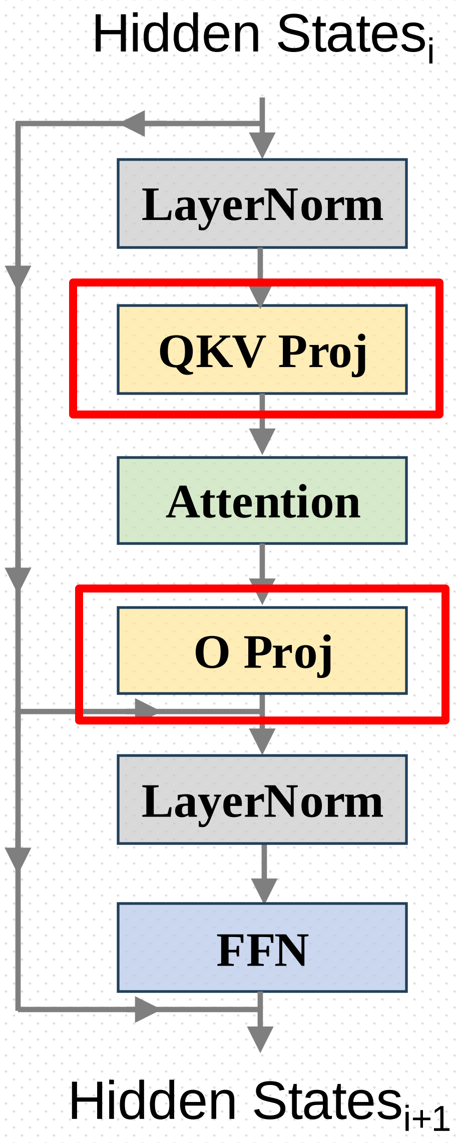
Layer Computation Details

❑Proj

- ❖Shape: [*d*, *d*]
- ❖Input
  - ➢Prefill: [***b*, *s*, *d***]
  - ➢Decode: [***b*, 1, *d***]
- ❖HBM load num:
  - ➢Prefill: d^2+bsd
  - ➢Decode: d^2+bd
- ❖Computation Ops:
  - ➢Prefill: bsd^2
  - ➢Decode: bd^2

Parameter	Notation
Batch size	b
Num head	n
Sequence length	s
Head dimension	h
Hidden dimension	d

But if we give a small ***b*** in decode phase, ***proj*** wound become an ***IO-bound*** task



Layer Computation Details

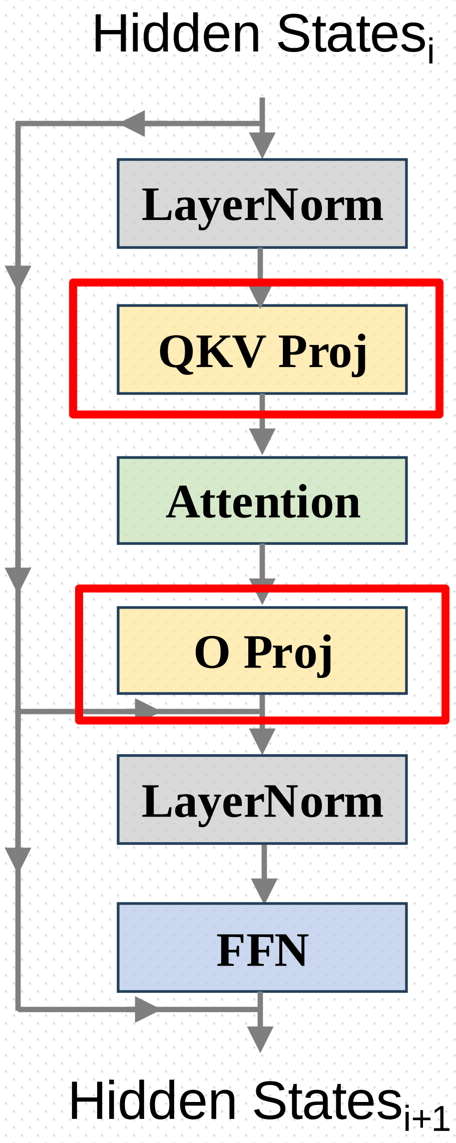
❑Proj

- ❖Shape: [***d***, ***d***]
- ❖Input
  - ➢Prefill: [***b, s, d***]
  - ➢Decode: [***b, 1, d***]
- ❖HBM load num:
  - ➢Prefill: d^2+bsd
  - ➢Decode: d^2+bd
- ❖Computation Ops:
  - ➢Prefill: bsd^2
  - ➢Decode: bd^2

Parameter	Notation
Batch size	***b***
Num head	***n***
Sequence length	***s***
Head dimension	***h***
Hidden dimension	***d***

But if we give a small ***b*** in decode phase, ***proj*** wound become an ***IO-bound*** task

On the other hand, ***proj*** wound become more ***comp-intensive*** if we give a bigger ***b***

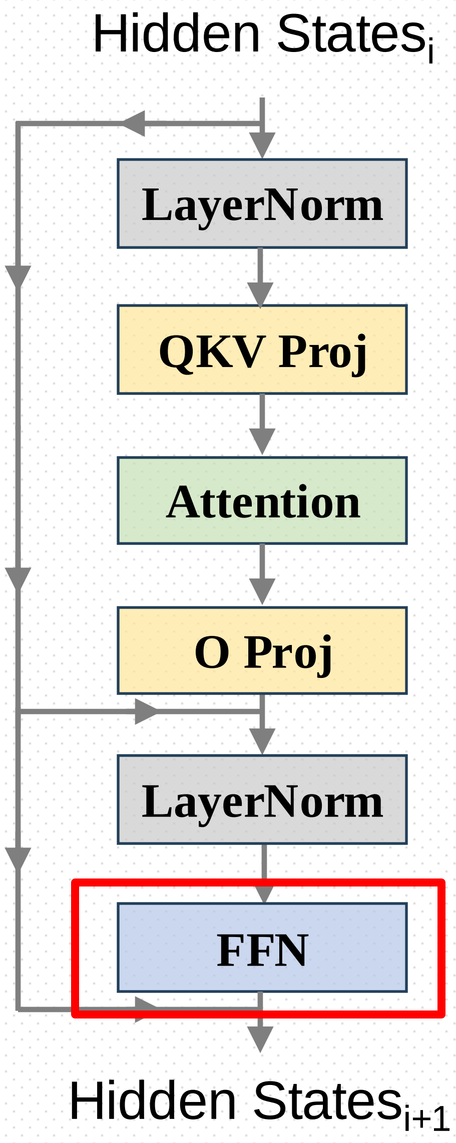


Layer Computation Details

❑FFN

- ❖Shape: [d, $4d$] and [$4d$,d]
- ❖Input
  - ➢Prefill: [b, s, d]
  - ➢Decode: [b, 1, d]
- ❖HBM load num:
  - ➢Prefill: $8d^2+bsd$
  - ➢Decode: $8d^2+bd$
- ❖Computation Ops:
  - ➢Prefill: $8bsd^2$
  - ➢Decode: $8bd^2$

Parameter	Notation
Batch size	b
Num head	n
Sequence length	s
Head dimension	h
Hidden dimension	d



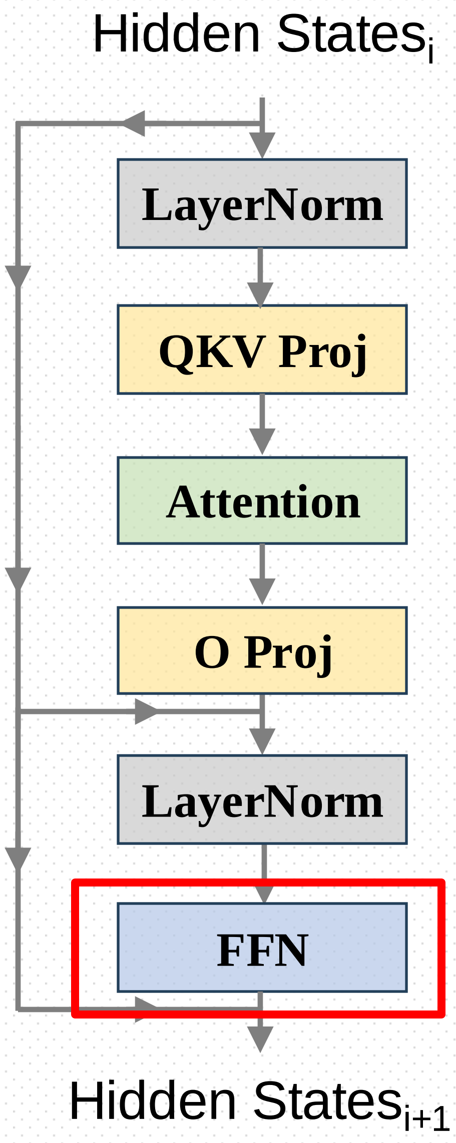
Layer Computation Details

❑FFN

- ❖Shape: [*d*, *4d*] and [*4d*,*d*]
- ❖Input
  - ➢Prefill: [*b, s, d*]
  - ➢Decode: [*b, 1, d*]
- ❖HBM load num:
  - ➢Prefill: $8d^2+bsd$
  - ➢Decode: $8d^2+bd$
- ❖Computation Ops:
  - ➢Prefill: $8bsd^2$
  - ➢Decode: $8bd^2$

Similar to ***proj***, ***FFN*** is ***comp-bound*** in prefill, ***IO-bound*** in decode

Parameter	Notation
Batch size	b
Num head	n
Sequence length	s
Head dimension	h
Hidden dimension	d



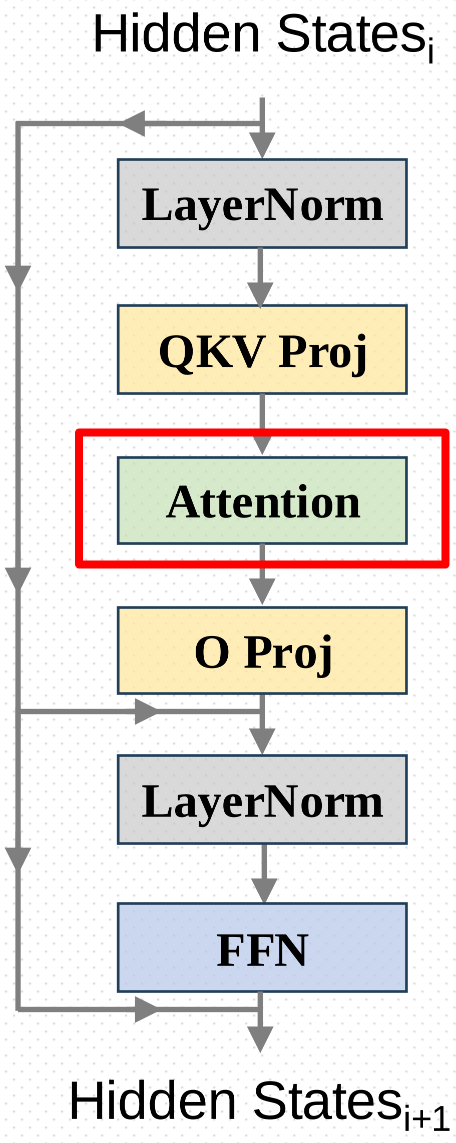
Layer Computation Details

❑Attention

- ❖Input
  - ➢Prefill: [***b, n, s, h***] for QKV
  - ➢Decode: [***b, n, 1, h***] for Q, [***b, n, s, h***] for KV
- ❖HBM load num:
  - ➢Prefill: ***3bnsh***
  - ➢Decode: ***2bnsh+bnh***
- ❖Computation Ops:
  - ➢Prefill: $2bns^2h$
  - ➢Decode: ***2bnsh***

Parameter	Notation
Batch size	***b***
Num head	***n***
Sequence length	***s***
Head dimension	***h***
Hidden dimension	***d***

$$Attn = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



Layer Computation Details

❑Attention

❖Input

➢Prefill: [***b, n, s, h***] for QKV

➢Decode: [***b, n, 1, h***] for Q, [***b, n, s, h***] for KV

Parameter	Notation
Batch size	***b***
Num head	***n***
Sequence length	***s***
Head dimension	***h***
Hidden dimension	***d***

❖HBM load num:

$$Attn = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

comp-bound

➢Prefill: ***3bnsh***

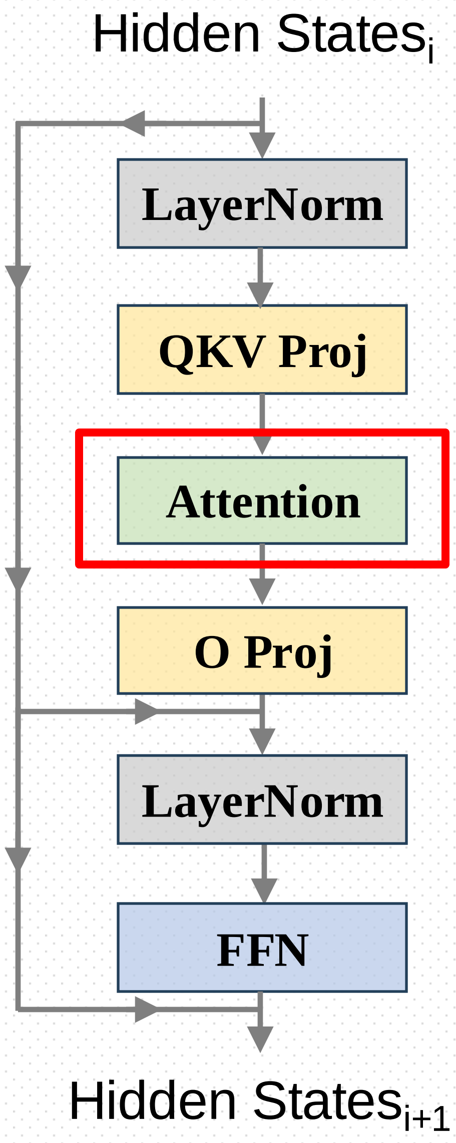
➢Decode: ***2bnsh+bnh***

❖Computation Ops:

IO-bound

➢Prefill: ***2bns²h***

➢Decode: ***2bnsh***



Decode Need Bigger Batch Size

- A40, KV tokens=1000

Batch Size	QKV Proj	Attn	O Proj	FFN
1	0.35%	0.29%	0.35%	0.37%
2	0.68%	0.35%	0.67%	0.72%
4	1.34%	0.37%	1.34%	1.43%
8	2.69%	0.36%	2.67%	2.85%
16	5.35%	0.36%	5.33%	5.62%
32	10.72%	0.39%	10.72%	11.34%
64	21.13%	0.40%	21.00%	21.10%
128	36.39%	0.40%	35.63%	37.50%

Bigger batch size can increate decode MFU

Decode Need Bigger Batch Size

- A40, KV tokens=1000

Batch Size	QKV Proj	Attn	O Proj	FFN
1	20.82%	3.97%	6.97%	68.24%
2	20.18%	6.29%	6.80%	66.73%
4	19.19%	11.23%	6.43%	63.15%
8	16.89%	20.78%	5.66%	56.67%
16	14.15%	33.87%	4.72%	47.25%
32	10.94%	49.27%	3.65%	36.15%
64	7.43%	64.07%	2.48%	26.02%
128	5.17%	75.57%	1.77%	17.50%

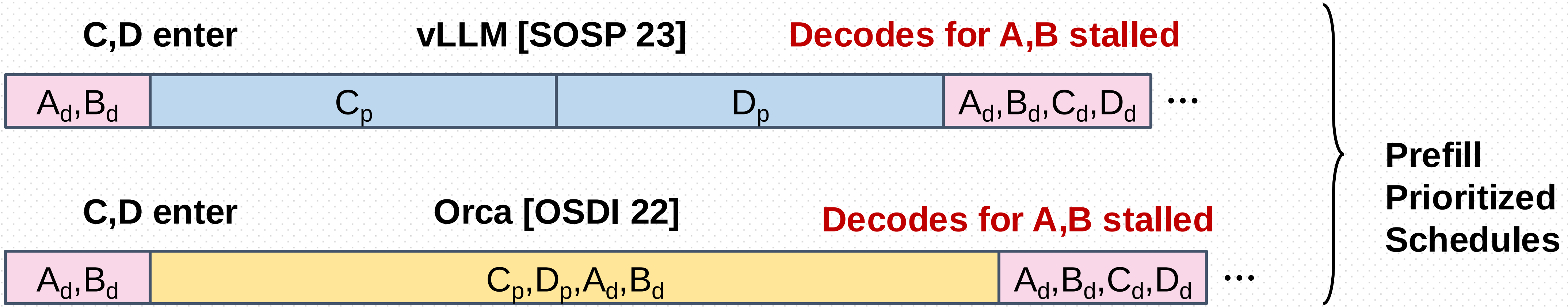
MFU increase has an upper bound

Workload Feature

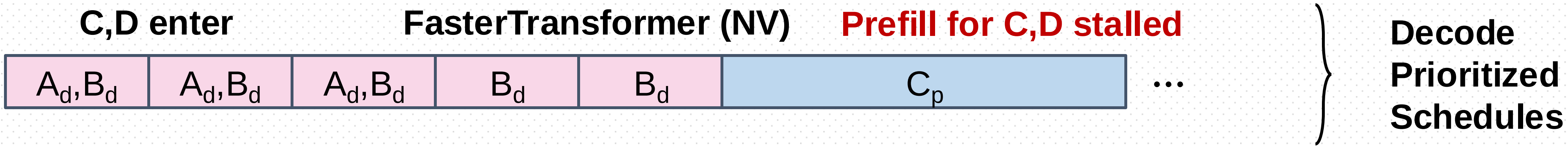
- ***Prefill***
  - *Comp-bound*
  - Small batch size can utilize most computation resource
- **Decode**
  - *IO-bound*
  - Most time in model weight loading

Serving Scheduling

❑Tradeoff between Latency and Throughput



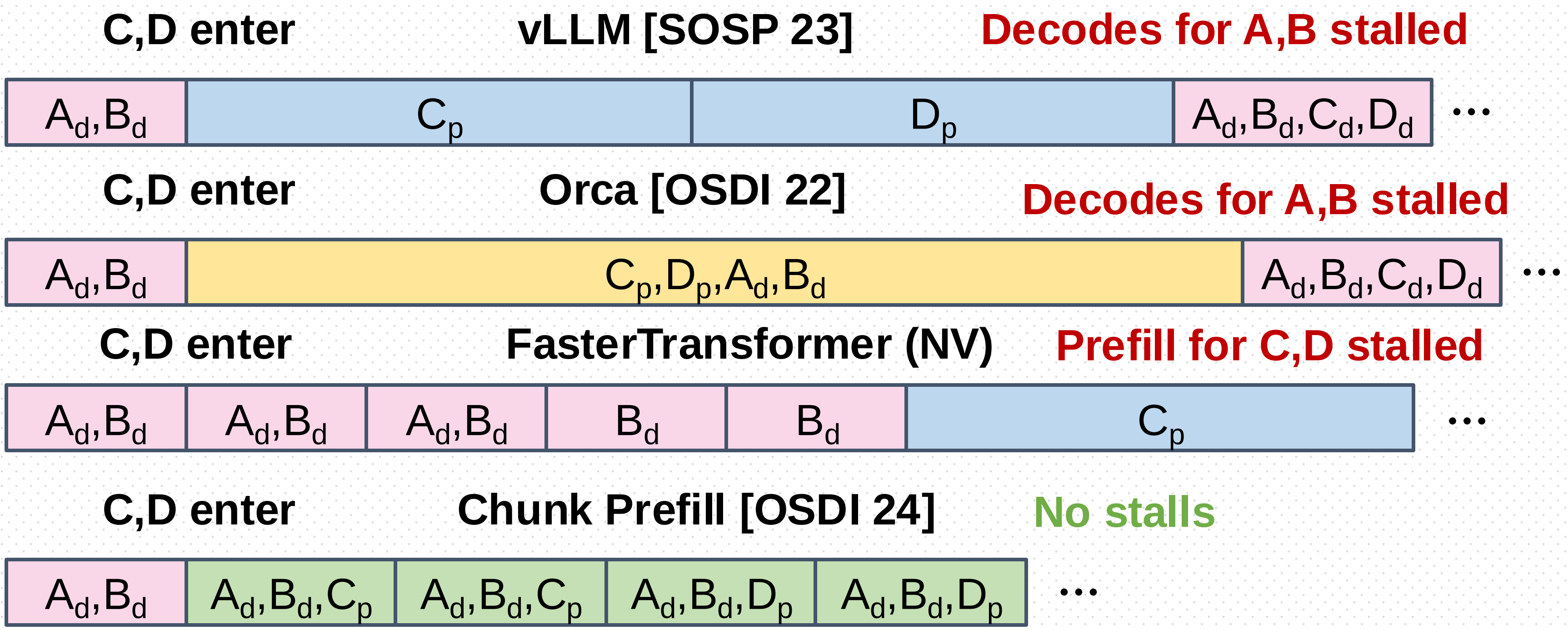
Prefill finish early -> Large decode batch size -> High Throughput
Decode stall -> High Latency



Decode without interference -> Low Latency
Prefill Stall -> Small batch in decode -> Poor throughput

Serving Scheduling

- ❑Amortize weight loading overhead in decode phase

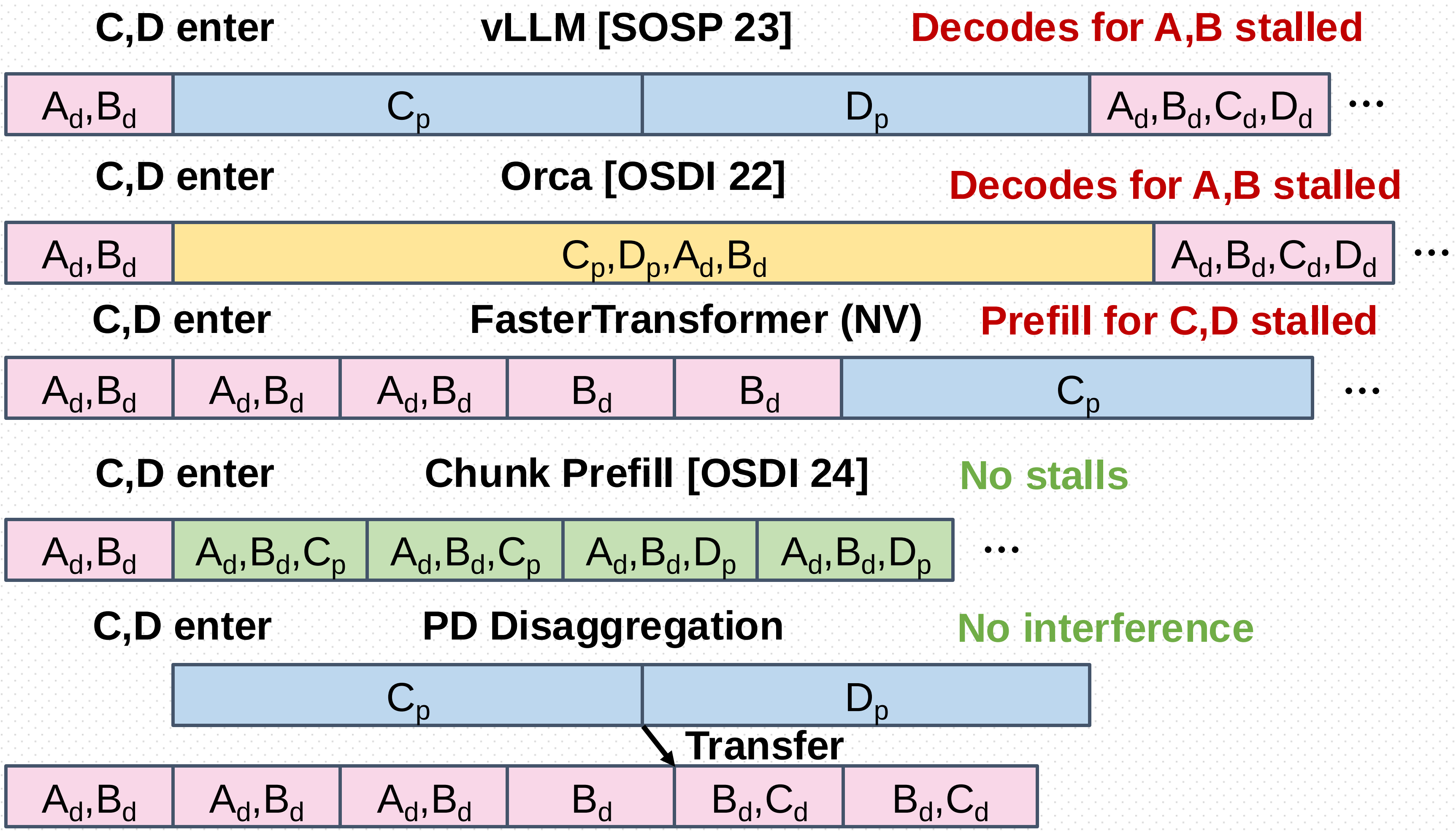


Serving Scheduling

- Prefill and Decode share the same model weight
  - QKV Proj, O Proj, FFN
  - Amortized model weight loading overhead
- Attention is processed separately
- Increased latency to both Prefill and Decode
  - Prefill need to load more KV from HBM
  - More computation in Decode phase

Serving Scheduling

- Amortize weight loading overhead in decode phase



Serving Scheduling

- Minimize Prefill and Decode interference
    - Additional KVCache transfer overhead
    - Still low MFU in Decode phase

Serving Scheduling —— Summary

	Prefill prioritized	Decode prioritized	Chunk-Prefill	PD Disaggregation
TTFT	+++	---	++	+++
TBT	---	+++	++	+++
Prefill-MFU	+++	+++	++	+++
Decode-MFU	+	-	++	+

Serving Scheduling —— Summary

Suitable for *relax SLO*
throughput-oriented scenario

	Prefill prioritized	Decode prioritized	Chunk-Prefill	PD Disaggregation
TTFT	+++	---	++	+++
TBT	---	+++	++	+++
Prefill-MFU	+++	+++	++	+++
Decode-MFU	++	-	++	+

Suitable for *stringent SLO*
latency-oriented scenario

Some other discussion

❑Long context support

- ❖Emphasize the importance of *TTFT* optimization

❑Parallelism Choice

- ❖***TP*** -> require two RDMA-based all-reduce in cross-node setting
- ❖***SP*** -> not suitable for short input
- ❖***ESP*** -> add complexity to architecture design
- ❖***CPP*** -> ***C***hunk ***P***ipeline ***P***arallelism
  - ➢Less communication overhead
  - ➢Fit both short and long contexts

Evaluation

❑Testbed

- ❖16 nodes each with 8 * A800-80GB, 4 * 200 Gbps RDMA NICs

❑Metric

- ❖TTFT (SLO: 30s)
- ❖TBT (SLO: 100ms, 200ms, 300ms)

❑Baseline

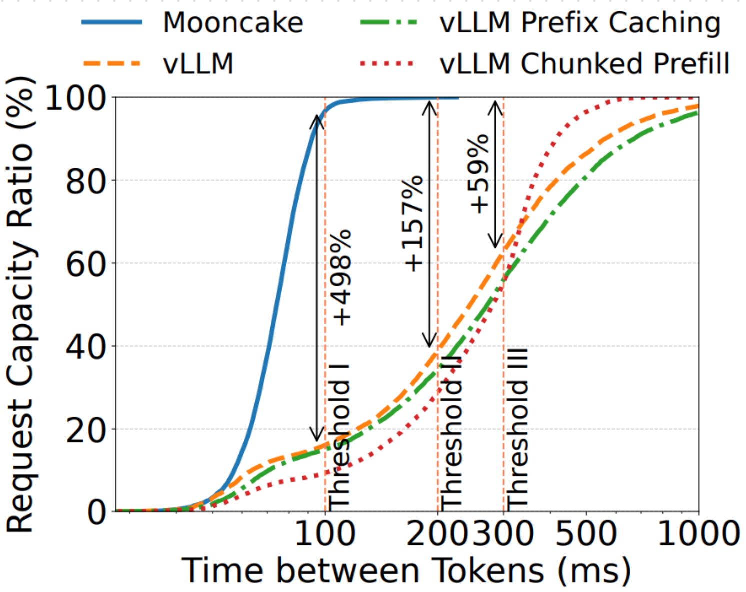
- ❖vLLM
- ❖vLLM with prefix caching
- ❖vLLM with chunk prefill

Evaluation

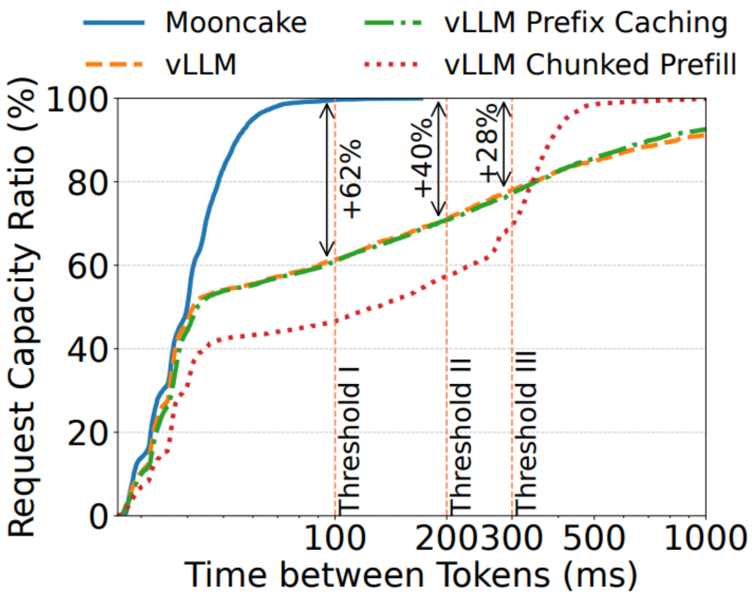
❑Workload

	Conversation	Tool&Agent	Synthetic
Avg Input Len	12035	8596	15325
Avg Output Len	343	182	149
Cache Ratio	40%	59%	66%
Arrival Pattern	Timestamp	Timestamp	Poisson
Num Requests	12031	23608	3993

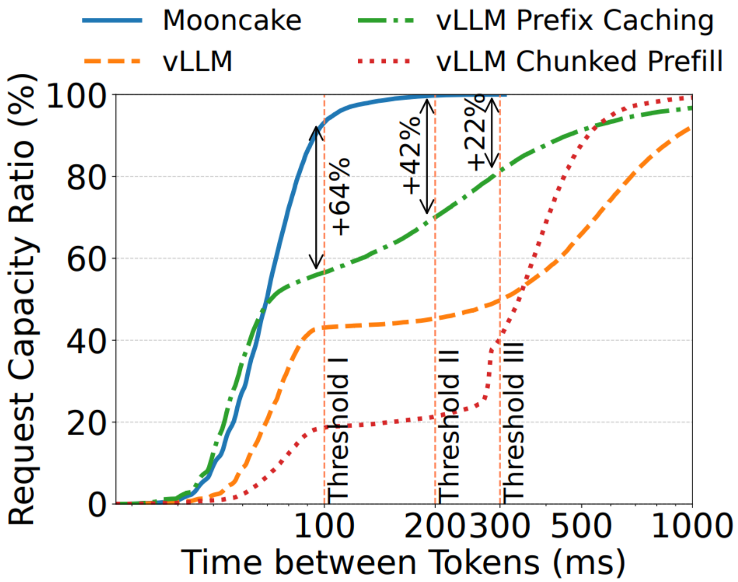
Evaluation —— TBT



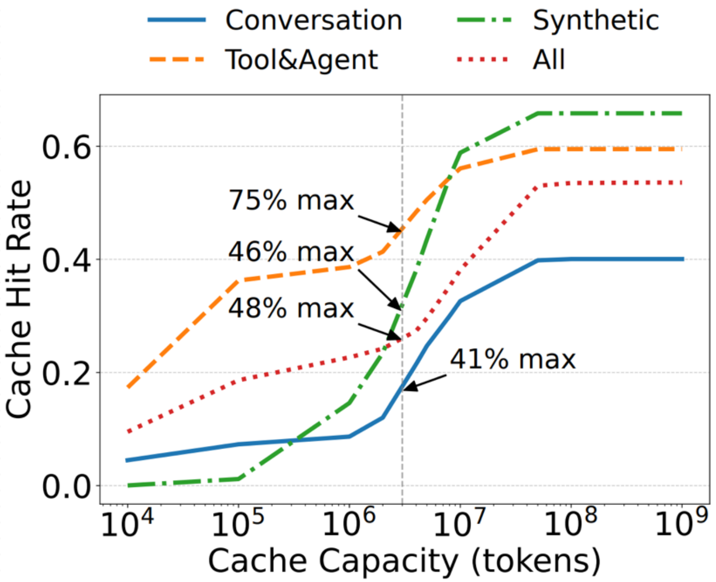
Conversation
Input:12035
Output:343
Cache:40%



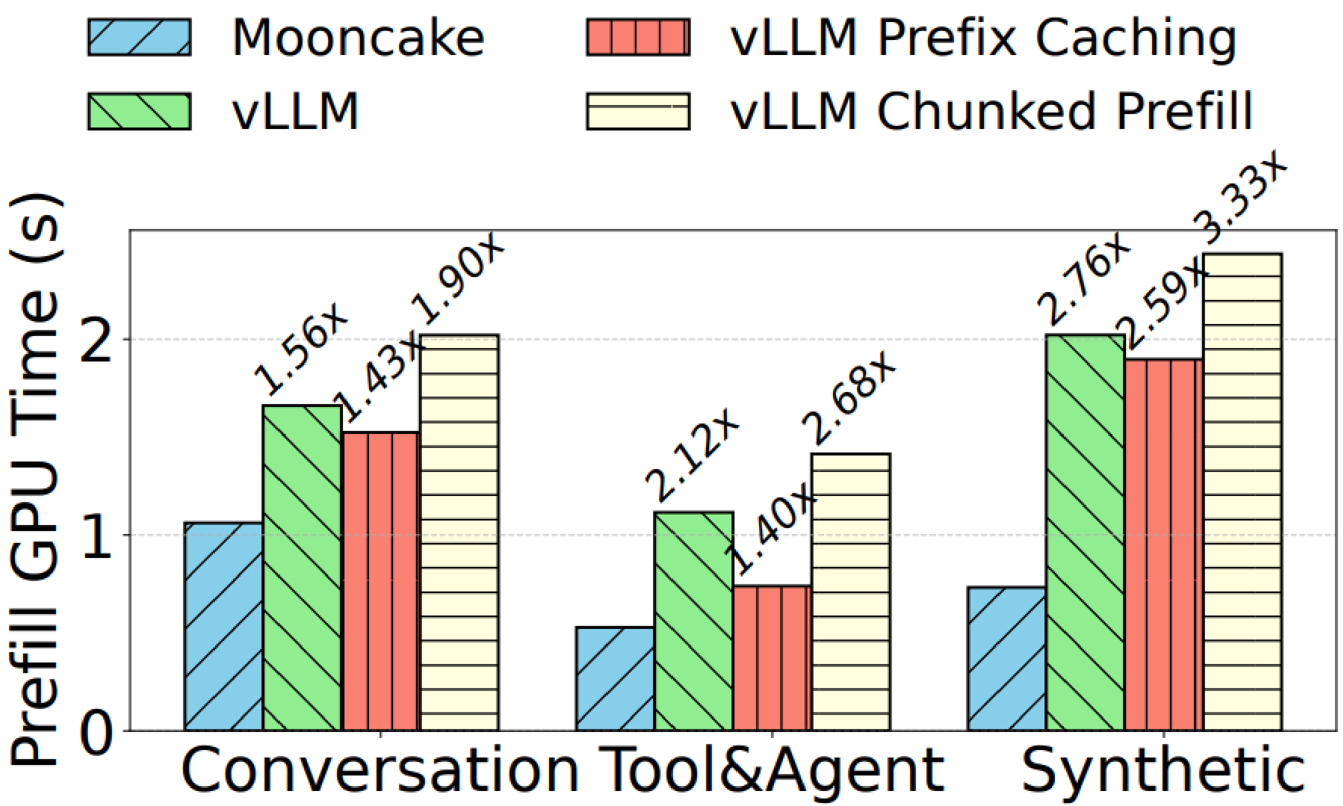
Synthetic
Input:15325
Output:149
Cache:66%

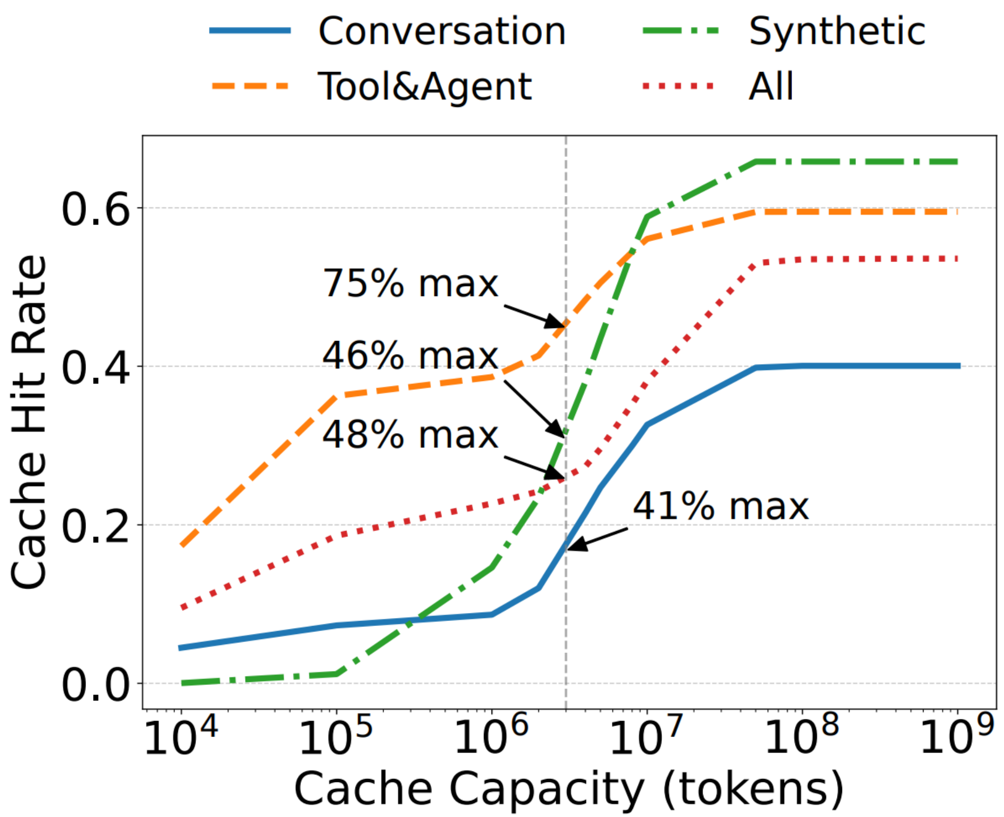


Tool & Agent
Input:8596
Output:182
Cache:59%

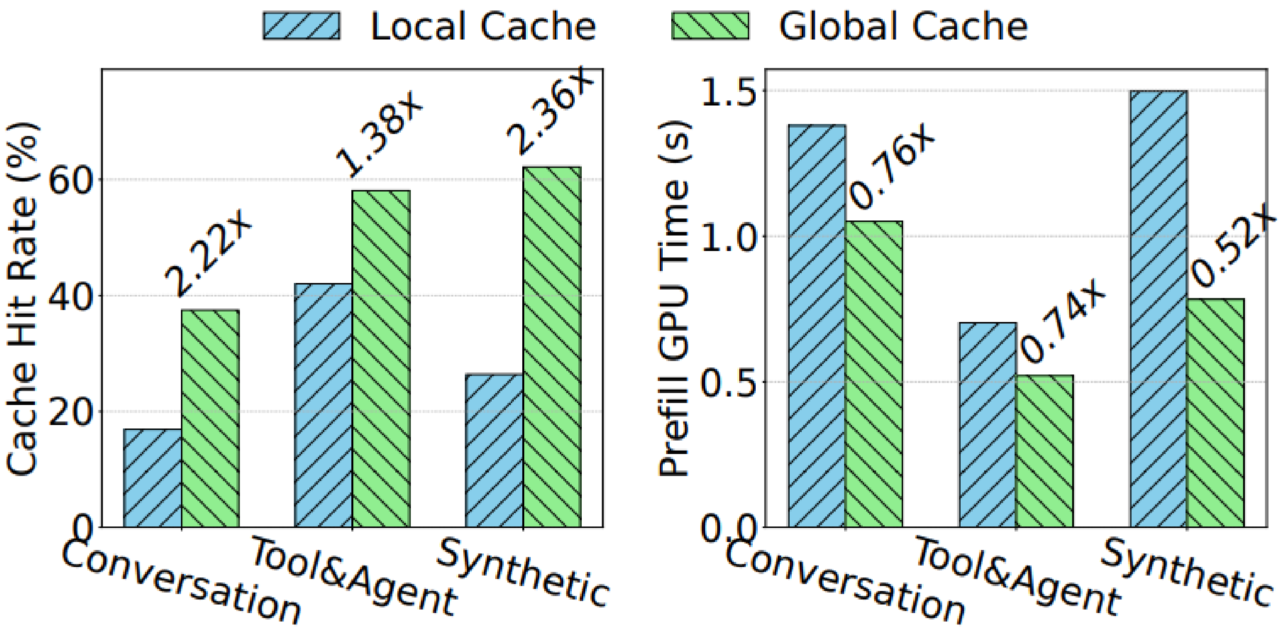


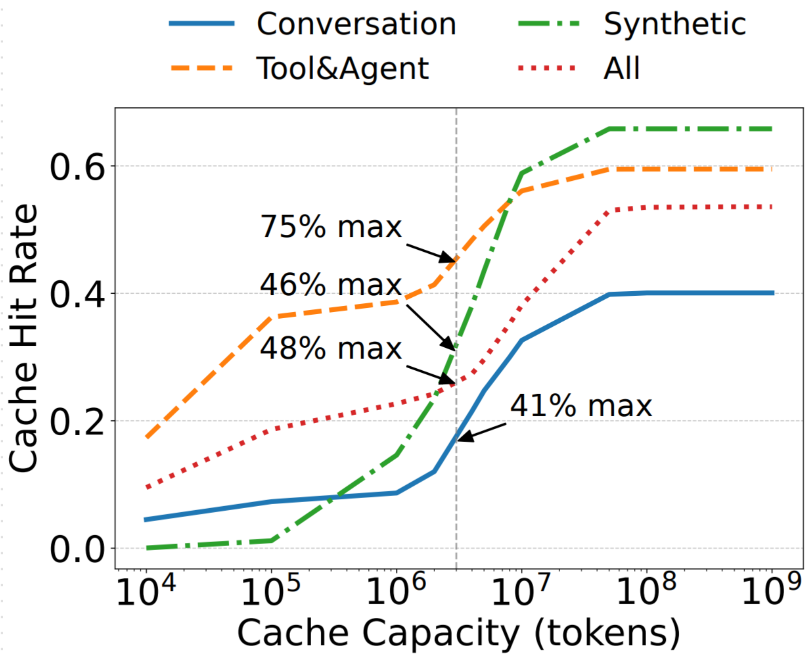
Evaluation —— TTFT





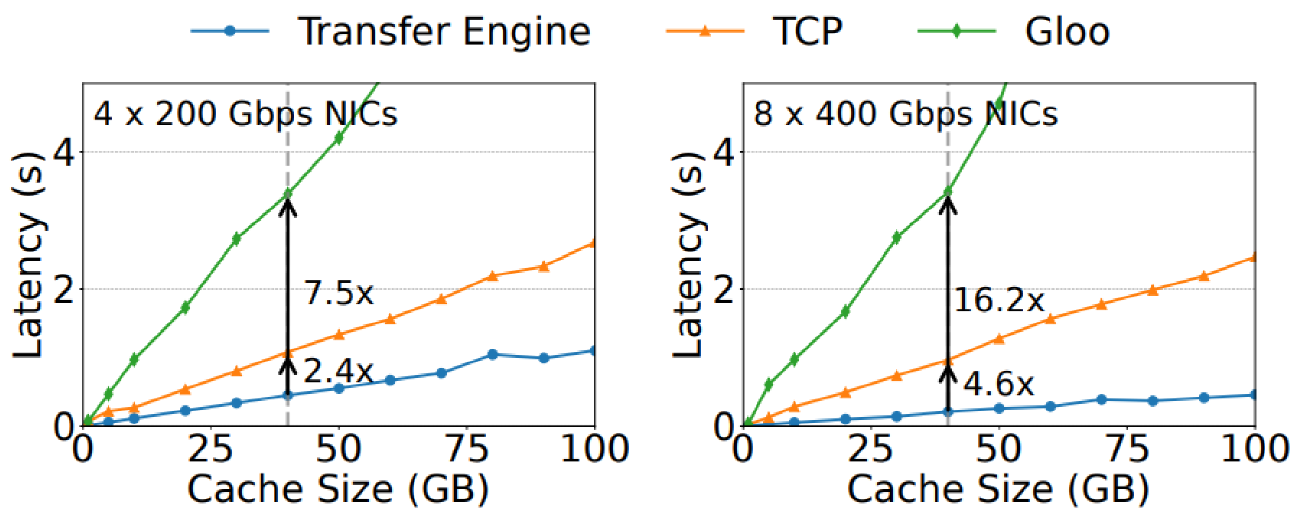
Evaluation —— Global Cache Efficiency



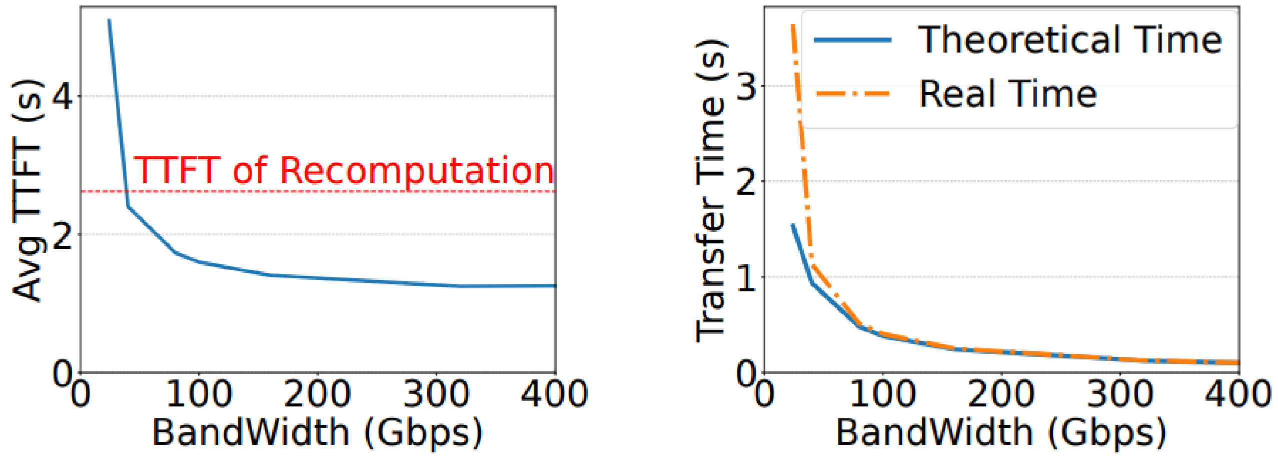


Evaluation —— Transfer Engine

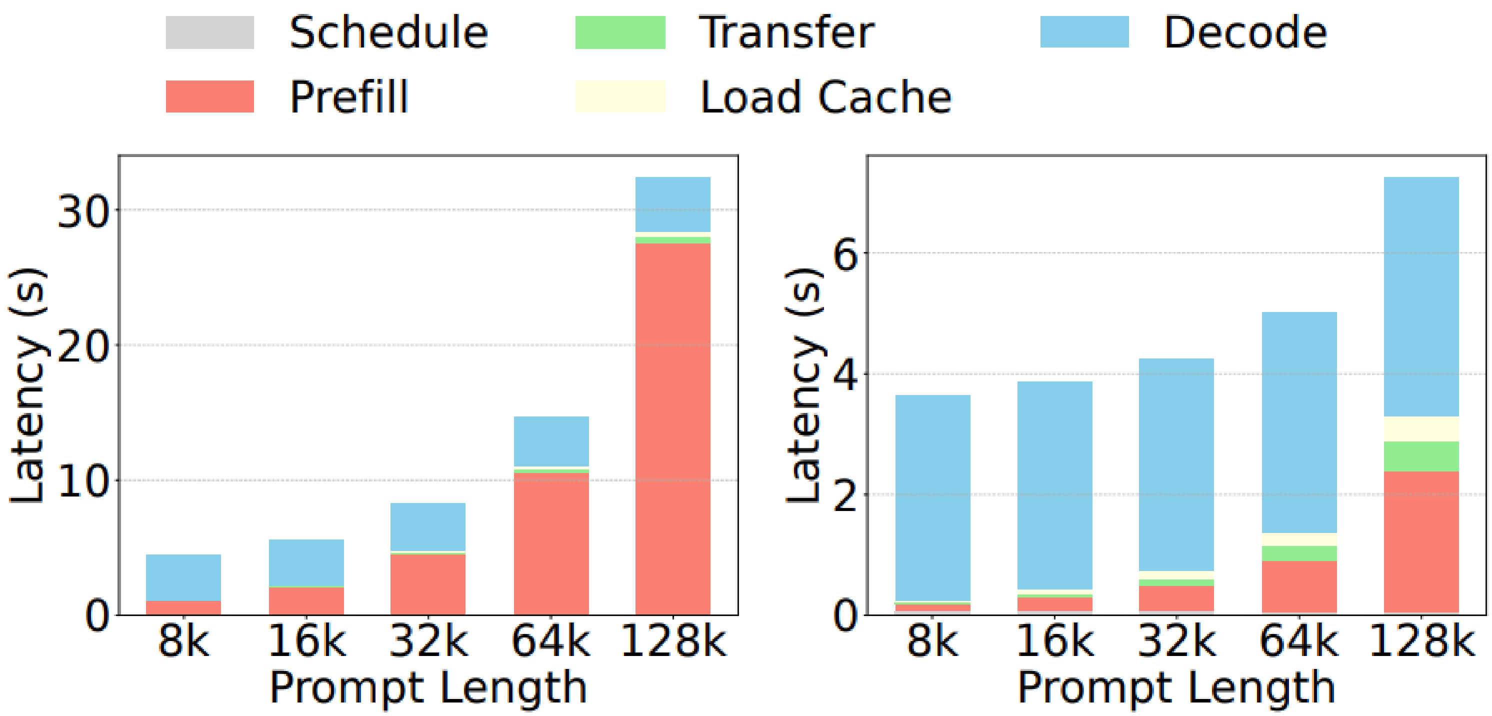
Transfer Engine Performance



Bandwidth Analyze

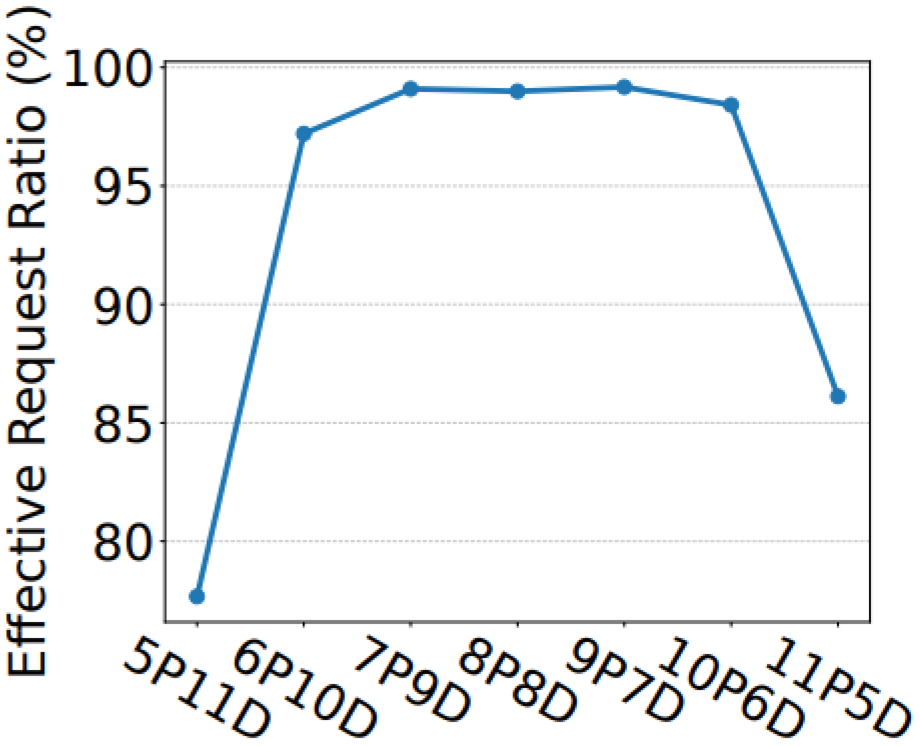


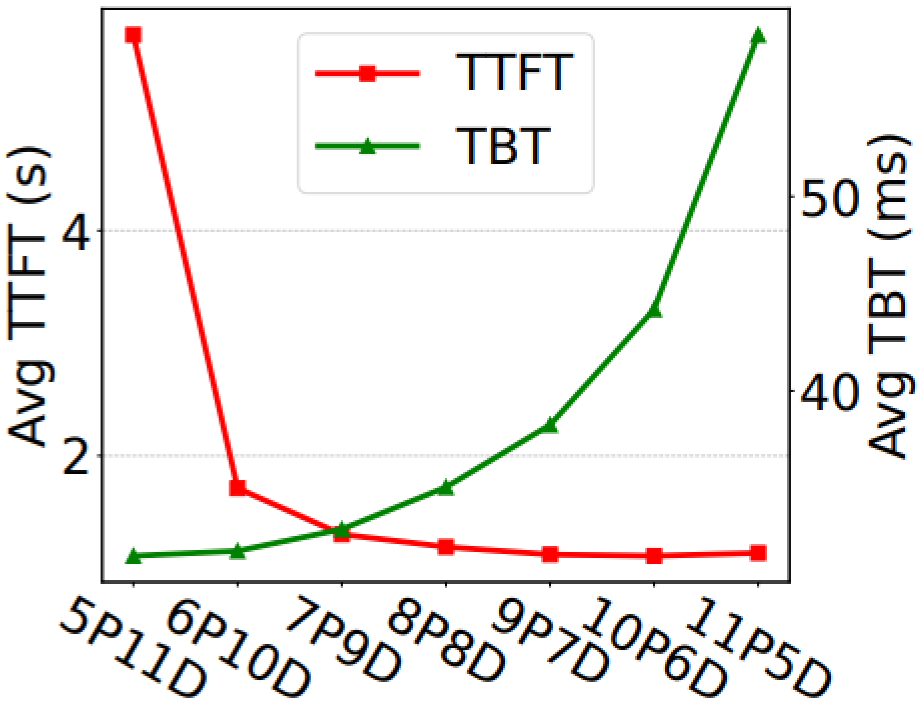
Evaluation —— Breakdown



Prefix Cache Ratio 0% **Prefix Cache Ratio 95%**

Evaluation —— Breakdown





Summary & Discussion

- ❑Prefix-cache based PD disaggregation LLM serving system
  - ❖Mooncake Store
  - ❖Transfer Engine
  - ❖Scheduling
- ❑Provide an industrial view on comparison between ***chunk prefill*** and ***PD disaggregation***
- ❑Open sourced LLM serving trace and codebase