Mooncake: Kimi's KVCache-centric Architecture for LLM Serving

(W6) Paper Reading

Nan Lin

Shanghai Jiao Tong University

2024-08-10

Mooncake: Kimi's KVCache-centric Architecture for LLM Serving

Ruoyu Qin[♠][♡][1] Zheming Li[♠][1] Weiran He[♠]
Mingxing Zhang[♡][2] Yongwei Wu[♡] Weimin Zheng[♡] Xinran Xu[♠][2]
[♠]Moonshot AI [♡]Tsinghua University

- Click the link to open: https://arxiv.org/html/2407.00079v1

Outline

Outline

Review 1: Transformer-based Inference Workflow

Generating from an input text sequences (**prompts**) to text (**completion**) consists of:

1. Loading weights to GPU
2. *Tokenizing* **prompts** on CPU
3. Transfering the tokenized prompts to GPU

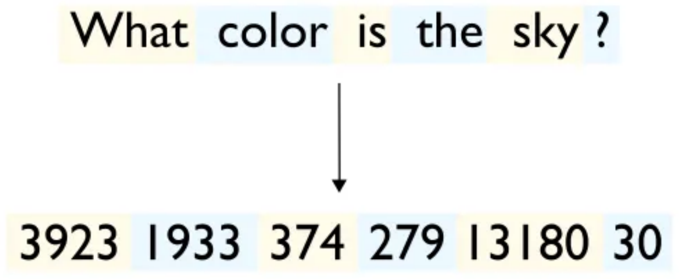


Figure 2: Tokenization

Review 1: Transformer-based Inference Workflow

(cont.)

4. **Prefill stage** (or **Initialization stage**): Generate the first token
5. **Decoding stage** (or **Generation stage**, **Auto-regression stage**, **Incremental stage**): Repeat …
   - Append the generated token to the sequence of input tokens
   - Using it as a new input to generate the second token

   End repeat if EOS token generated or maximum token sequence length reached.

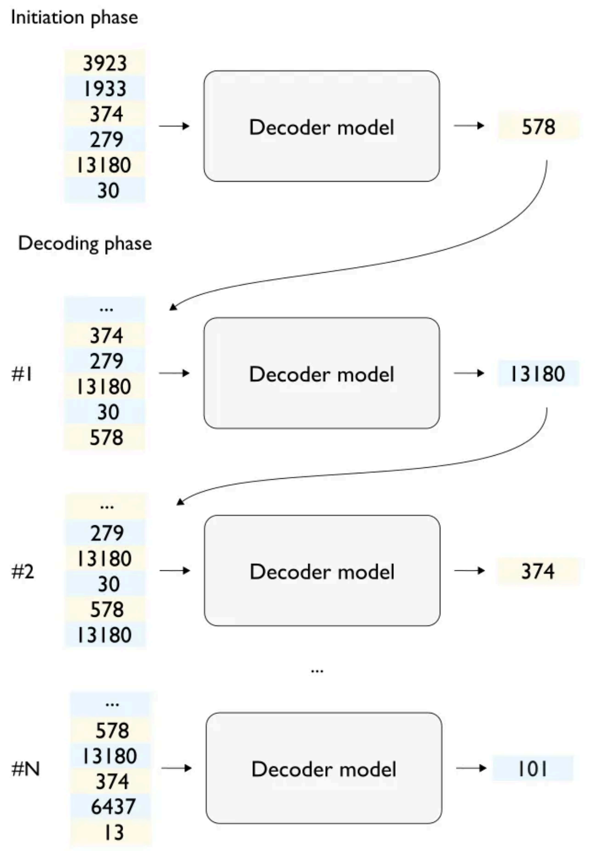


Figure 3: Prefill Stage and Decoding Stage

Review 1: Transformer-based Inference Workflow

(cont.)

6. *Detokenizing* the tokenized incremented token to get your generated text.

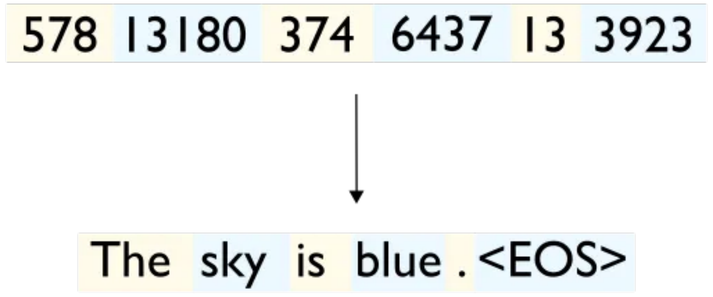


Figure 4: Detokenization

Review 2: KVCache

Motivation:

- **Masking** technique: *Causality*, we cannot use token generated of later words to influence the former ones.

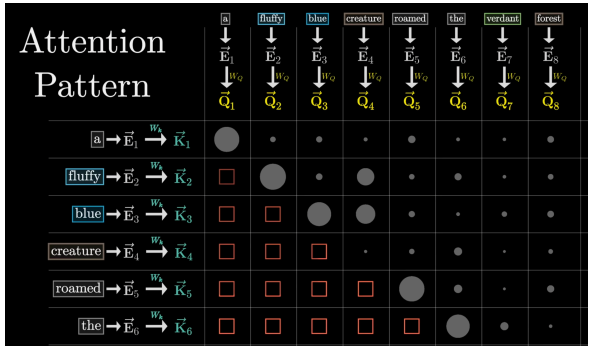


Figure 5: Masking

- Previous tokens are identical across all the stages.

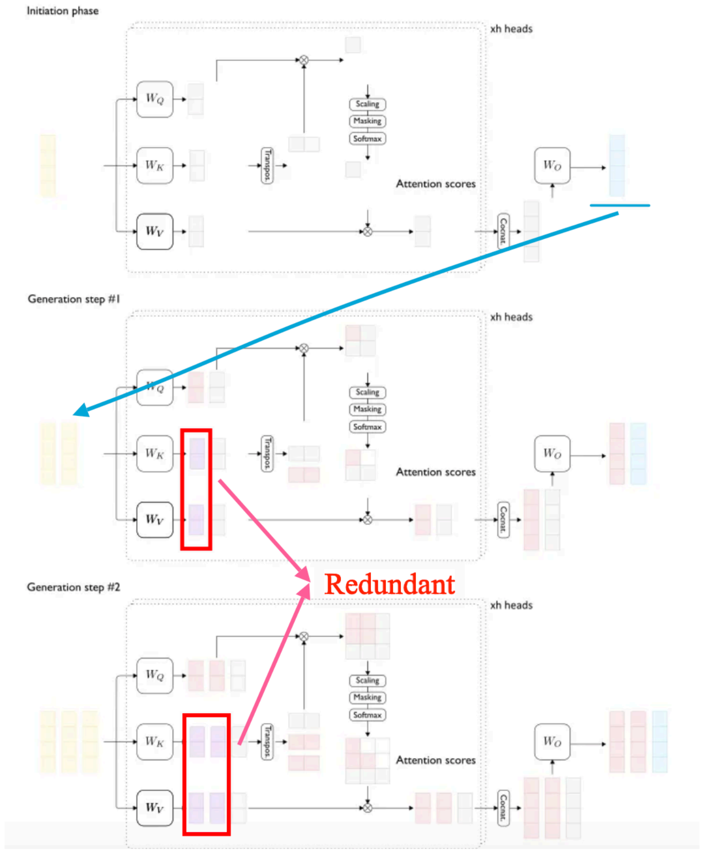


Figure 6: Excessive Datas in Prefill Stage and Decoding Stage

Review 2: KVCache

Changes to the original workflow:

- No longer need tokens but their key and value vectors.
- Take the KVCache and the last generated token as input.
- Initilization phase could be renamed as *pre-fill* stage since it prepares all the key and value vectors of input text sequence.

Issues of KVCache:

- *Large memory consumption in GPU*
- *Low GPU Utilization problem*: data transmission time > calculation time

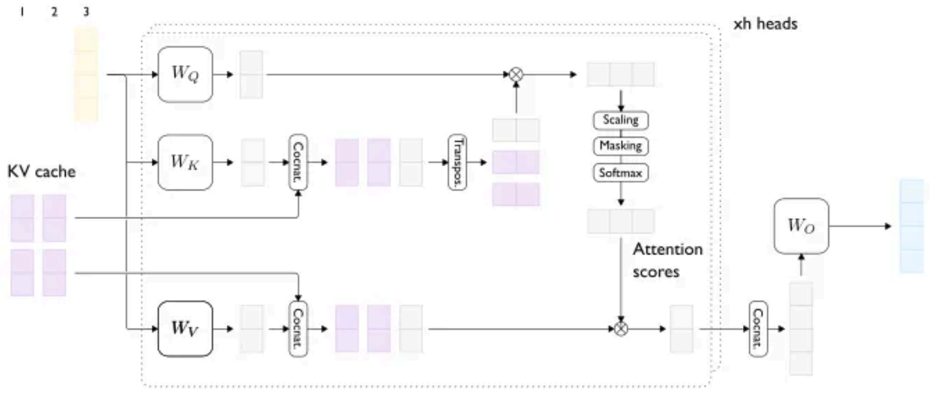


Figure 7: Generating Steps with KVCache Enabled

Outline

Modelization

Primary Goal

Diversified workload of LLM models $\rightarrow$ *Optimization problem* with multiple constraints:

- Maximize overall throughput
- Constraints of latency-related SLOs:

Workload difference in:

- I/O length
- Frequency
- Distribution of arrival
- SLOs

Two Types of SLOs

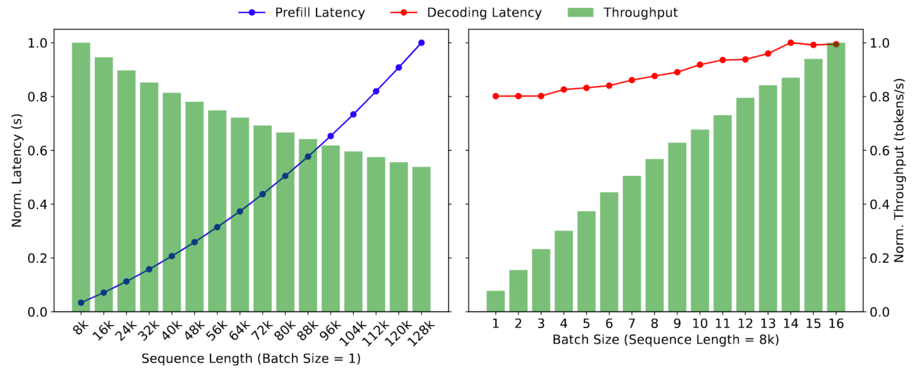


Figure 2: Normalized throughput and latency of prefill and decoding stages with different sequence lengths or batch sizes for the dummy LLaMA2-70B model.

Computation time increases:

- At *prefill stage*, superlinearly with input length due to parallel processing of tokens → **Time to first token (TTFT)**
- At *decoding stage*, sublinearly with batch size due to auto-regressive processing on one token at a time per batch → **Time between tokens (TBT)**

Going More Concrete

Disaggregation Idea

Make best use of resources →

1. Scheduling of KVCache is central to LLM serving scheduling;
2. Decouple and reconstruct nodes for different but collabrative goals

Current approaches:

- Reuse KVCache as much as possible
- Maximize the number of tokens in each batch (to improve **Model FLOPs Utilization (MFU)**)

However, corresponding issues:

- If tokens stored in remote places → TTFT problem
- Large batch size → TBT problem

Another Objective

Limited GPU/accelerator supply while excessive demand.

- Early reject policy if no available slots could be use
- However, fluctuation in the workloads

Their work:

Overload-Oriented Scheduling

- Prediction of generation length
- Load prediction in the short-tern future
- Classify request priorities.

Outline

Overview: KVCache-Centric Disaggregated Architecture

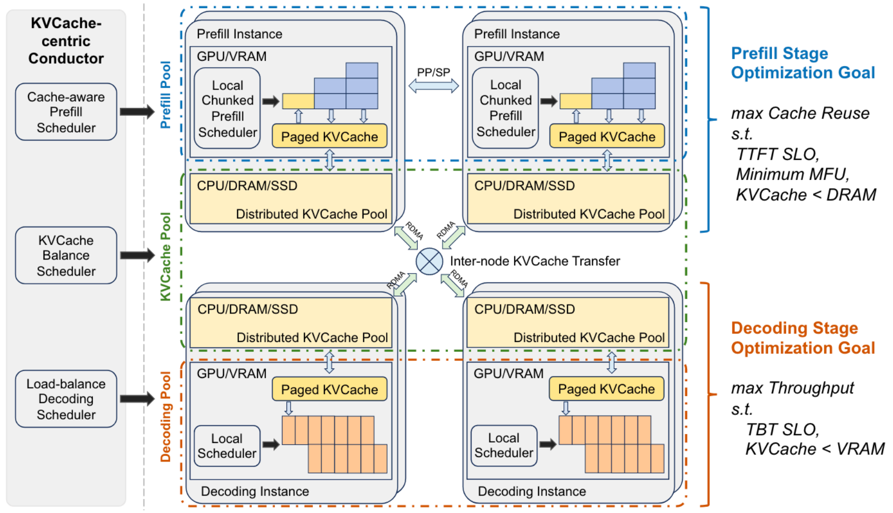

KVCache-centric Conductor
Cache-aware Prefill Scheduler
KVCache Balance Scheduler
Load-balance Decoding Scheduler
Prefill Pool
KVCache Pool
Decoding Pool
Prefill Instance
GPU/VRAM
Local Chunked Prefill Scheduler
Paged KVCache
CPU/DRAM/SSD
Distributed KVCache Pool
PP/SP
RDMA
Inter-node KVCache Transfer
Local Scheduler
Decoding Instance
Prefill Stage Optimization Goal
max Cache Reuse s.t. TTFT SLO, Minimum MFU, KVCache < DRAM
Decoding Stage Optimization Goal
max Throughput s.t. TBT SLO, KVCache < VRAM

Figure 1: Mooncake Architecture.

Quick-reminder: reuse KVCache / max num of tokens in each batch.

Conductor: The Global Scheduler

Selects a pair of prefill and decoding instances and schedule the request to:

1. Transfer reusable KVCache
2. (Prefill Stage) Complete in chunks/layers, stream the output to the corresponding decoding instance
3. (Decoding Stage) Load KVCache and add it to the batching process, generate output

Storage and Transfer Logic of the KVCache Blocks

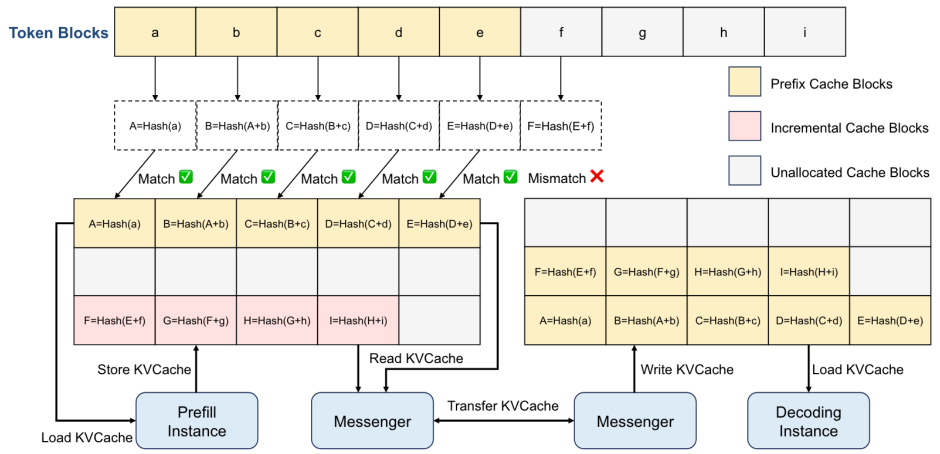


Figure 3: The KVCache pool in CPU memory. Each block is attached with a hash value determined by both its own hash and its prefix for deduplication.

- In CPU memory, KVCache stored as page blocks.
- Cache eviction algorithms used
- Transfer across CPUs and GPUs handled by a RDMA-based component called *Messenger*
- Global scheduler: *Conductor*

Workflow of a Request (C3)

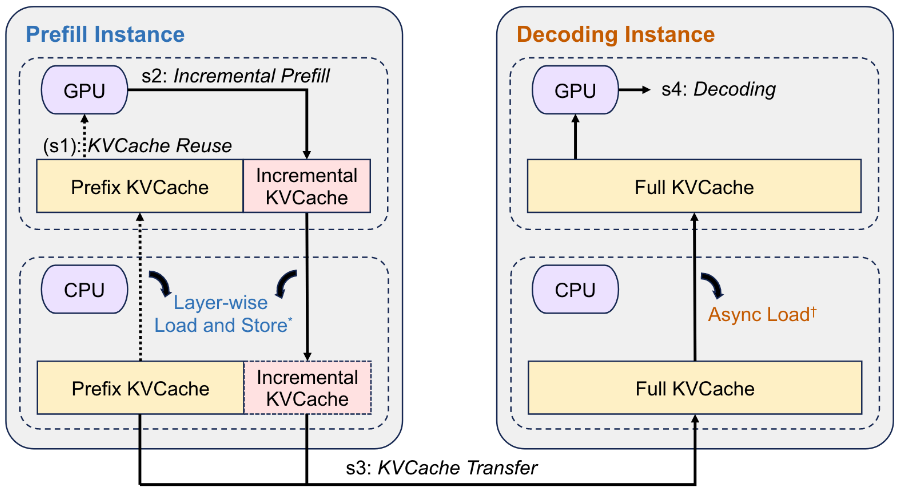


Figure 4: Workflow of inference instances. (∗) For prefill instances, the load and store operations of the KVCache layer are performed layer-by-layer and in parallel with the prefill computation to mitigate transmission overhead (see §4.2). (†) For decoding instances, asynchronous loading is performed concurrently with GPU decoding to prevent GPU idle time.

After tokenizing is finished, conductor selects a pair of prefill nodes and a decoding node:

1. **KVCache reuse**: loads the prefix cache from remote CPU memory into GPU memory
2. **Incremental Prefill**: Prefill node complete prefill stage using the prefix cache, afterwards it stores the *newly generated and incremental* KVCache back into CPU memory

Workflow of a Request (C3)

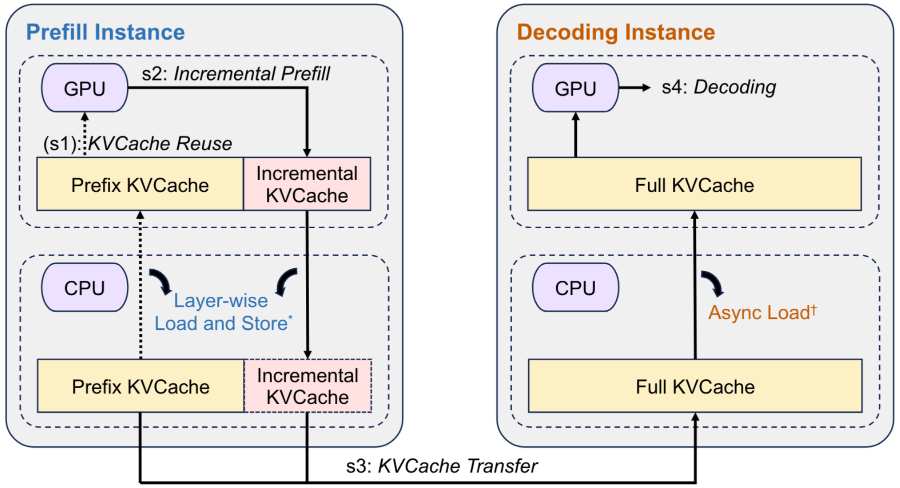


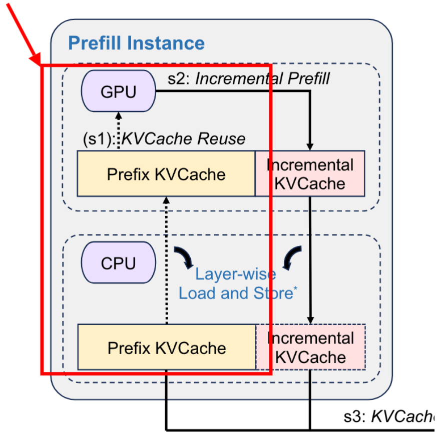
Figure 4: Workflow of inference instances. (∗) For prefill instances, the load and store operations of the KVCache layer are performed layer-by-layer and in parallel with the prefill computation to mitigate transmission overhead (see §4.2). (†) For decoding instances, asynchronous loading is performed concurrently with GPU decoding to prevent GPU idle time.

3. **KVCache Transfer**: Operated by *Messenger*, cross-machine KVCache transformer, asynchronously exectued and streaming the KVCache
4. **Decoding**: KVCache is received in the CPU DRAM of the decoding node, request joins the next batch in a continuous batching manner

KVCache-centric Scheduling

Prefill Global Scheduling

Selection of prefill instances depend not only on load, but also *prefix cache hit length* and *distribution of reusable KVCache blocks.*



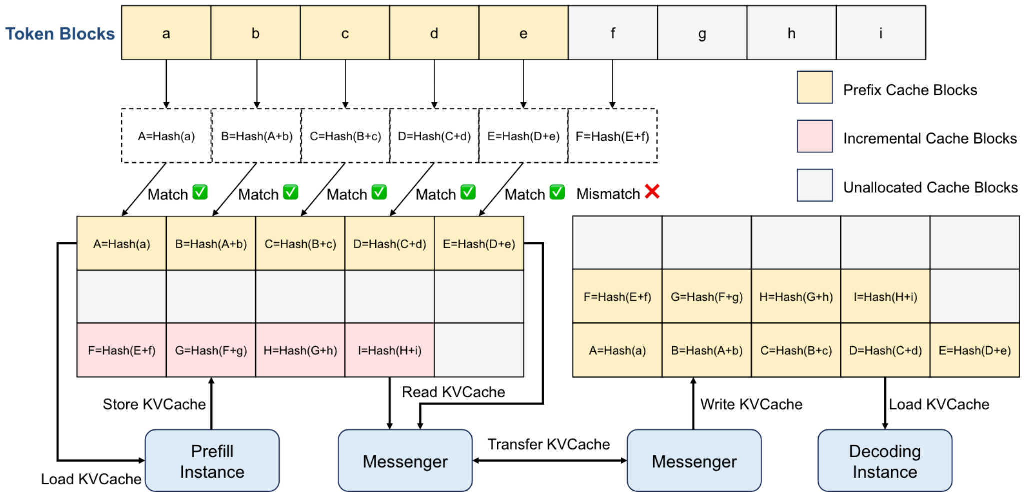
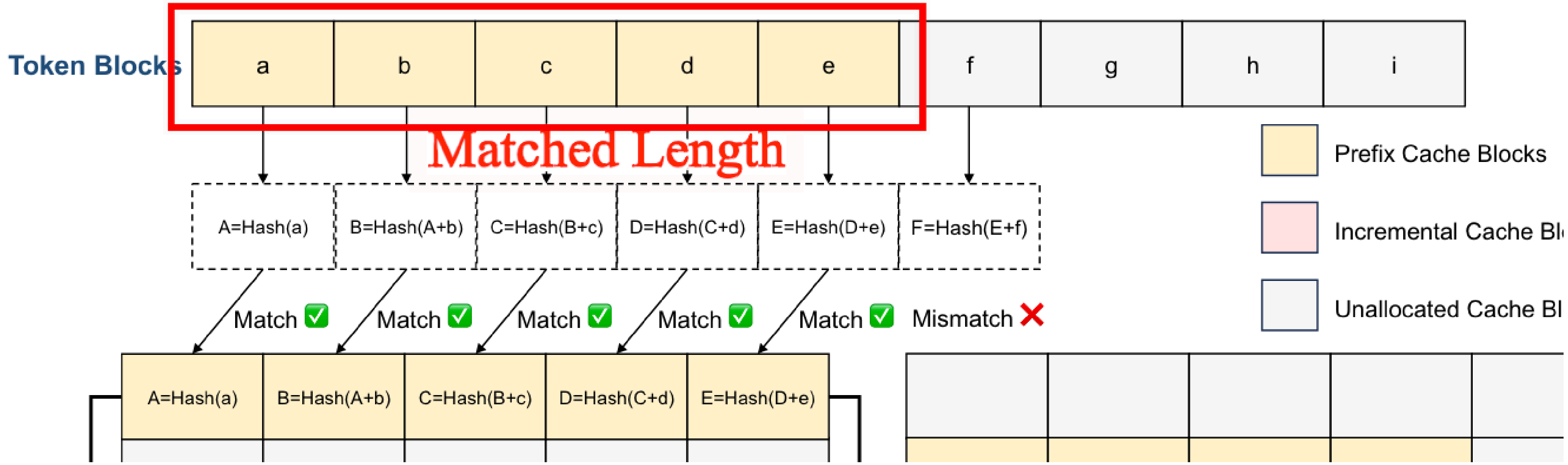


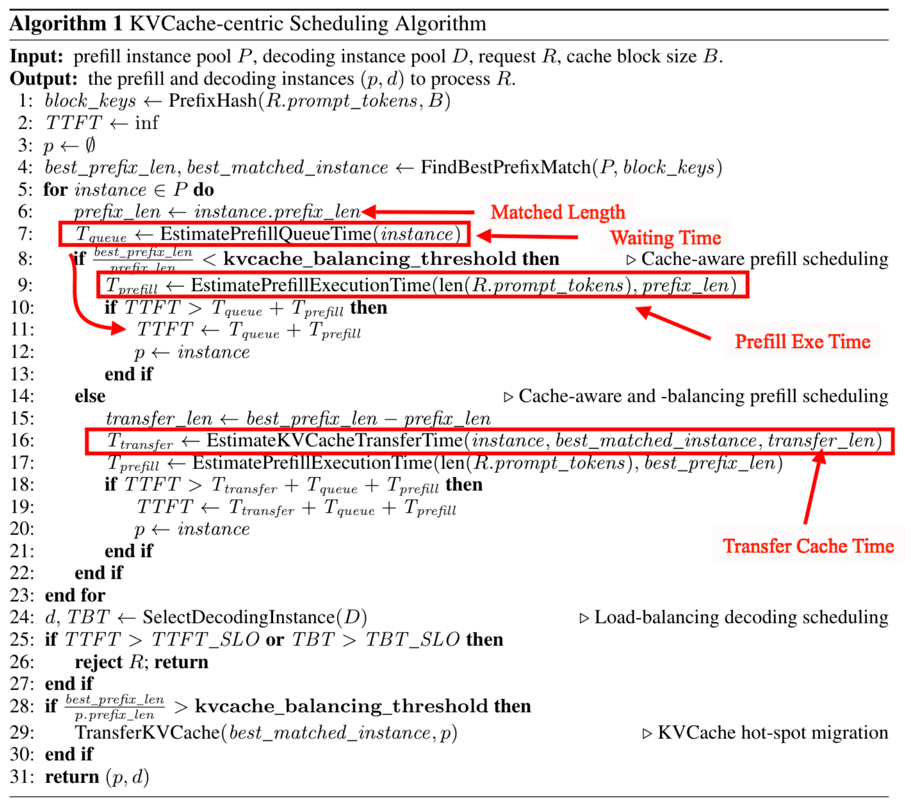
Figure 3: The KVCache pool in CPU memory. Each block is attached with a hash value determined by both its own hash and its prefix for deduplication.

KVCache-centric Scheduling

- Input token for each request divided into blocks, assign hash keys one by one.
- Compare one by one to identify the prefix match length, named `prefix_len`



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.

```
$block\_keys \leftarrow$ PrefixHash($R.prompt\_tokens, B$)
$TTFT \leftarrow$ inf
$p \leftarrow \emptyset$
$best\_prefix\_len, best\_matched\_instance \leftarrow$ FindBestPrefixMatch($P, block\_keys$)
for $instance \in P$ do
    $prefix\_len \leftarrow instance.prefix\_len$
    $T_{queue} \leftarrow$ EstimatePrefillQueueTime($instance$)
    if $\frac{best\_prefix\_len}{prefix\_len} <$ kvcache_balancing_threshold then      ▷ Cache-aware prefill scheduling
        $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len($R.prompt\_tokens$), $prefix\_len$)
        if $TTFT > T_{queue} + T_{prefill}$ then
            $TTFT \leftarrow T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    else      ▷ Cache-aware and -balancing prefill scheduling
        $transfer\_len \leftarrow best\_prefix\_len - prefix\_len$
        $T_{transfer} \leftarrow$ EstimateKVCacheTransferTime($instance, best\_matched\_instance, transfer\_len$)
        $T_{prefill} \leftarrow$ EstimatePrefillExecutionTime(len($R.prompt\_tokens$), $best\_prefix\_len$)
        if $TTFT > T_{transfer} + T_{queue} + T_{prefill}$ then
            $TTFT \leftarrow T_{transfer} + T_{queue} + T_{prefill}$
            $p \leftarrow instance$
        end if
    end if
end for
$d, TBT \leftarrow$ SelectDecodingInstance($D$)      ▷ Load-balancing decoding scheduling
if $TTFT > TTFT\_SLO$ or $TBT > TBT\_SLO$ then
    reject $R$; return
end if
if $\frac{best\_prefix\_len}{p.prefix\_len} >$ kvcache_balancing_threshold then
    TransferKVCache($best\_matched\_instance, p$)      ▷ KVCache hot-spot migration
end if
return $(p, d)$
```

- Find shortest TTFT
- Update T_{cache} and T_{queue}
- If SLO not achievable, return `HTTP 429 TOO MANY REQUESTS`.

KVCache-centric Scheduling

- Each prefill machine mangages its own set of prefix caches.
- Goal: achieve balance between cache matching and instance load (e.g. system prompts vs long document)
- Collect global usages of each block, use for forecasting ? (No way, requests fluctuation)

Cache Load Balancing

Requests may not always be directed to the prefill instance with the longest prefix cache length due to high instance load.

Conductor forwards the cache's location and the request to an alternative instance.

- Improve replication of hot-spot caches.

Overload-Oriented Scheduling (C6)

Overload: Unrealistic to process every incoming request.

System should process as many requests as possibles until the system load reaches a predefined threshold.

Load = Ratio of num of requests being processed to the system's max capacity

Decision making:

- Whether to accept the prefill stage based on the *prefill instance's load*
- Whether to proceed with the decoding stage based on the *decoding instance's load*

Overload-Oriented Scheduling (C6)

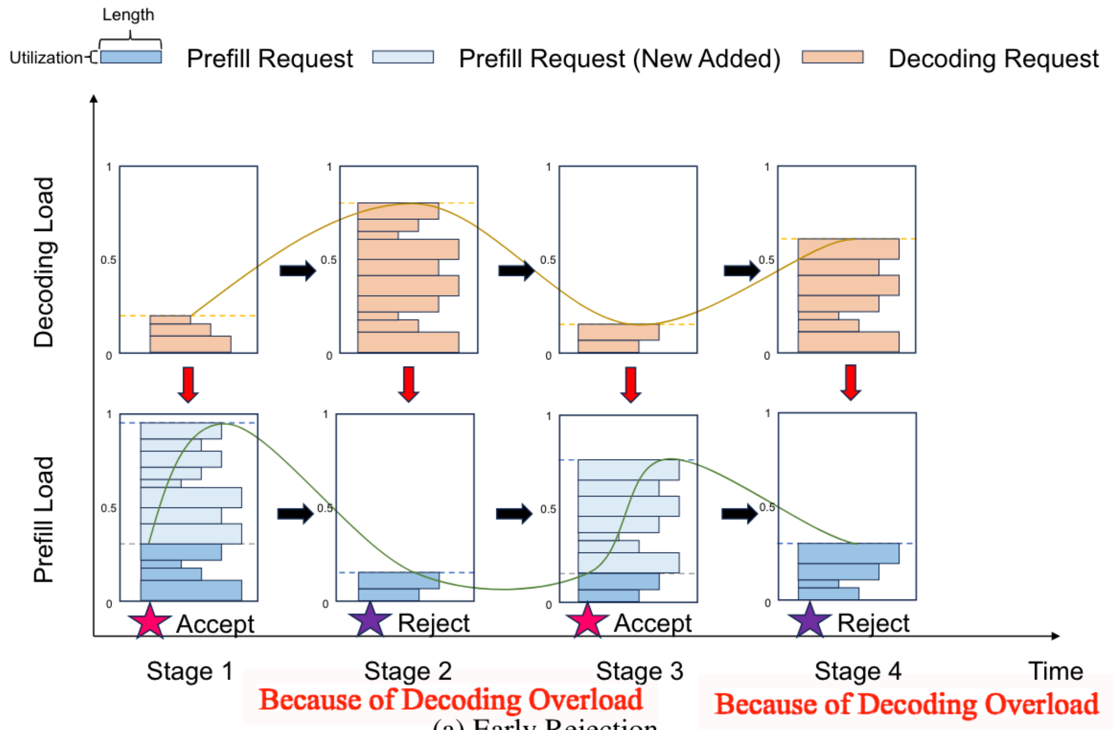
Time-lag between two instances for a single request: If a request is rejected by the decoding instance due to high load after the prefill stage has been completed → Waste of resources

Early Rejection

Advance the load assessement of the decoding instance to precede the beginning of the prefill stage.

Overload-Oriented Scheduling (C6)

Problems of Early Rejection: Fluctuation



(a) Early Rejection.

Overload-Oriented Scheduling (C6)

Early Rejection based on Prediction

Request level

Predict Output Length, thus estimate accurately TTFT and TBT.

System level

Estimate the overall batch count or the TBT status for instances after a specified time.

Their work: System level, request level left for future work.

Outline

Evaluations

Mooncake demonstrates significantly higher throughput, with enhancements ranging from 50% to 525%, while adhering to the same TTFT and TBT SLO constraints compared to vLLM.

Mooncake can mitigate load fluctuations, increasing the request handling capacity.

Outline

References

- Review: LLM Inference
  - https://medium.com/@plienhar/llm-inference-series-2-the-two-phase-process-behind-llms-responses-1ff1ff021cd5
  - https://medium.com/@plienhar/llm-inference-series-3-kv-caching-unveiled-048152e461c8
  - https://www.bilibili.com/video/BV1TZ421j7Ke/?spm_id_from=333.337.search-card.all.click&vd_source=5ade9da381cec8d2c191f450ccd0cf57