Mooncake: A KVCache-centric Disaggregated 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

Abstract

Mooncake is the serving platform for Kimi, a leading LLM service provided by Moonshot AI. It features a KVCache-centric disaggregated architecture that separates the prefill and decoding clusters. It also leverages the underutilized CPU, DRAM, and SSD resources of the GPU cluster to implement a disaggregated cache of KVCache. The core of Mooncake is its KVCache-centric scheduler, which balances maximizing overall effective throughput while meeting latency-related Service Level Objectives (SLOs). Unlike traditional studies that assume all requests will be processed, Mooncake faces challenges due to highly overloaded scenarios. To mitigate these, we developed a prediction-based early rejection policy. Experiments show that Mooncake excels in long-context scenarios. Compared to the baseline method, Mooncake can achieve up to a 525% increase in throughput in certain simulated scenarios while adhering to SLOs. Under real workloads, Mooncake's innovative architecture enables Kimi to handle 75% more requests.

1 Introduction

1.1 Motivation of Developing Mooncacke

With the rapid adoption of large language models (LLMs) in various scenarios [1, 2, 3, 4], the workloads for LLM serving have become significantly diversified. These workloads differ in input/output length, frequency and distribution of arrival, and, most importantly, demand different kinds of Service Level Objectives (SLOs). As a Model as a Service (MaaS) provider, one of the primary goals of Kimi [5] is to solve an optimization problem with multiple complex constraints. The optimization goal is to maximize overall effective throughput, which directly impacts revenue, while the constraints reflect varying levels of SLOs. These SLOs typically involve meeting latency-related requirements, mainly the time to first token (TTFT) and the time between tokens (TBT).

To achieve this goal, a prerequisite is to make the best use of the various kinds of resources available in the GPU cluster. Specifically, although GPU servers are currently provided as highly integrated nodes (e.g., DGX/HGX supercomputers [6]), it is necessary to decouple and restructure them into several disaggregated resource pools, each optimized for different but collaborative goals. For example, many researchers [7, 8, 9] have suggested separating prefill servers from decoding servers because these two stages of LLM serving have very different computational characteristics, in which the KVCache shifts with requests moving from prefill to decoding servers.

Building on this idea, we found that the scheduling of KVCache is central to LLM serving scheduling. To improve overall throughput, there are typically two general approaches: *1)* reuse KVCache as much as possible to reduce the required computation resources; and *2)* maximize the number of tokens in each batch to improve the Model FLOPs Utilization (MFU). However, reusing KVCache

[1] Ruoyu Qin's part of work done as an intern at Moonshot AI, contributed equally with Zheming Li.
[2] Corresponding to zhang_mingxing@mail.tsinghua.edu.cn, xuxinran@moonshot.ai.

Preprint Tech Report by Moonshot AI and MadSys @ Tsinghua.

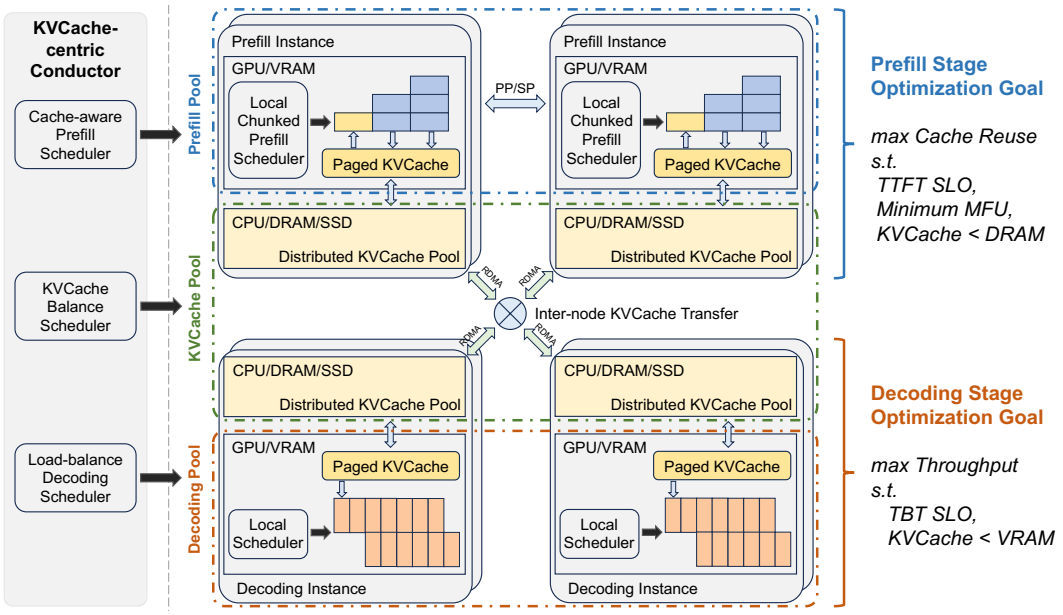


Figure 1: Mooncake Architecture.

from a remote location will prolong the TTFT, and a large batch size will lead to a larger TBT. Thus, the utilization of both these throughput oriented optimizations may lead to violations of latency related SLOs.

According to the above guidelines, we propose a disaggregated design that is centered around KVCache for scheduling and optimization. Figure 1 presents our current **KVCache-centric disaggregated architecture** for LLM serving, named Mooncake. For each request, the global scheduler (Conductor) needs to select a pair of prefill and decoding instances and schedule the request in the following steps: *1)* transfer as much reusable KVCache as possible to the selected prefill instance; *2)* complete the prefill stage in chunks/layers and continuously stream the output KVCache to the corresponding decoding instance; *3)* load the KVCache and add the request to the continuous batching process at the decoding instance for generating request outputs.

Although this process seems straightforward, the selection policy is complex due to many restrictions. In the prefill stage, the main objective is to reuse the KVCache as much as possible to avoid redundant computation. However, waiting for KVCache stored on lower-tier storage may violate the TTFT SLO. Additionally, high demand on the KVCache server can lead to network congestion, prolonging the waiting time. Thus Conductor is also responsible for predicting the future usage of KVCache blocks and executing scheduling operations such as swapping and replication accordingly. The hottest blocks should be replicated to multiple nodes to avoid fetching congestion, while the coldest ones should be swapped out to reduce reserving costs. Prefill scheduling is also constrained by the availability of DRAM space in the prefill node, especially when much of the memory is reserved for the global KVCache pool.

In contrast, the decoding stage has different optimization goals and constraints. The aim is to aggregate as many tokens as possible in a decoding batch to improve MFU. However, this objective is restricted not only by the TBT SLO but also by the total size of aggregated KVCache that can be contained in the VRAM.

More importantly, existing research on LLM serving assumes sufficient resources and focuses on improving resource utilization. In contrast, the current GPU/accelerator supply is limited, and many MaaS providers face severe overload problems, especially during peak times. Scheduling in such scenarios presents unique challenges that existing works have not explored. For example, we need to predict future loads and reject certain requests early if there will be no available decoding slots after the prefill stage, to save wasted computation resources. However, a straightforward implementation of such an early reject policy surprisingly leads to fluctuations in the overloads. This has led us to

aim at predicting the generation length of specific queries and making overall load predictions in the short-term future to implement a better rejection policy. It is also necessary to classify different request priorities to implement priority-based scheduling. In this paper, we summarize these problems as **overload-oriented scheduling** and present our preliminary study results.

1.2 Design and Results of Mooncacke

In the following sections of this paper, we first present an overview of Mooncake's architecture, including its main components and the typical workflow for processing a request (§3). Then, we describe the main design choices made during its implementation, especially those not covered in current research.

First, in §4, we discuss how to implement a separate prefill node pool that seamlessly handles the dynamic distribution of context length. We employ a chunked pipeline parallelism (CPP) mechanism to scale the processing of a single request across multiple nodes, which is necessary for reducing the TTFT of long-context inputs. Compared to traditional sequence parallelism (SP) based solutions, CPP reduces network consumption and simplifies the reliance on frequent elastic scaling. This mechanism is further supplemented with layer-wise prefill that enables stream transferring of KVCache to overlap latency.

Next, in §5, we detail our KVCache-centric request scheduling algorithm, which balances instance loads and user experience as measured by TTFT and TBT SLOs. This includes a heuristic-based automated hot-spot migration scheme that replicates hot KVCache blocks without requiring precise predictions of future KVCache usage. Experimental results show that our cache-aware scheduling can significantly lower TTFT in real-world scenarios. In end-to-end experiments using public datasets, simulated data, and real workloads, Mooncake excels in long-context scenarios. Compared to the baseline method, Mooncake can achieve up to a 525% increase in throughput while meeting SLOs. Under real workloads, Mooncake enables Kimi to handle 75% more requests.

Finally, unlike existing work on LLM serving that assumes all requests will be processed, Mooncake consistently faces overload due to Kimi's rapid growth in user requests. Thus, Mooncake's scheduling involves determining whether to accept or reject incoming requests based on the system load. In §6, we discuss our implementation of a unique early rejection policy that reduces wasted computational resources in overloaded scenarios. We further explore the load fluctuation problem caused by straightforward early rejection and how predicting future load can mitigate this issue.

Mooncake is currently the primary platform for serving Kimi and has successfully handled exponential workload growth, proving its effectiveness in scaling out to large and highly overloaded workloads. However, many more problems need to be explored, and these future directions are also included in the paper.

> To protect proprietary information and facilitate reproducibility, all the experimental results reported in this paper are based on replayed traces of real workloads, but using a **dummy model** that follows the same architecture as LLaMA2-70B. The trace includes only the timing of request arrivals, the number of input tokens, and the number of output tokens, without any real user content. The trace will be open-sourced later after following certain internal procedures.

2 Preliminary and Problem Definition

Modern large language models (LLMs) are based on the Transformer architecture, which utilizes attention mechanisms and multilayer perceptrons (MLP) to process input. Popular Transformer-based models, such as GPT [10] and LLaMA [11], employ a decoder-only structure. Each inference request is logically divided into two stages: the prefill stage and the decoding stage.

In the prefill stage, all input tokens are processed in parallel. This stage generates the first output token while storing intermediate results of computed keys and values, referred to as the KVCache. The decoding stage then uses this KVCache to autoregressively generate new tokens, adding new keys and values from the computation to the KVCache. The ability to process input tokens simultaneously in the prefill stage typically makes it computationally intensive, except for short requests. Since the computational complexity of attention networks scales quadratically with input length while

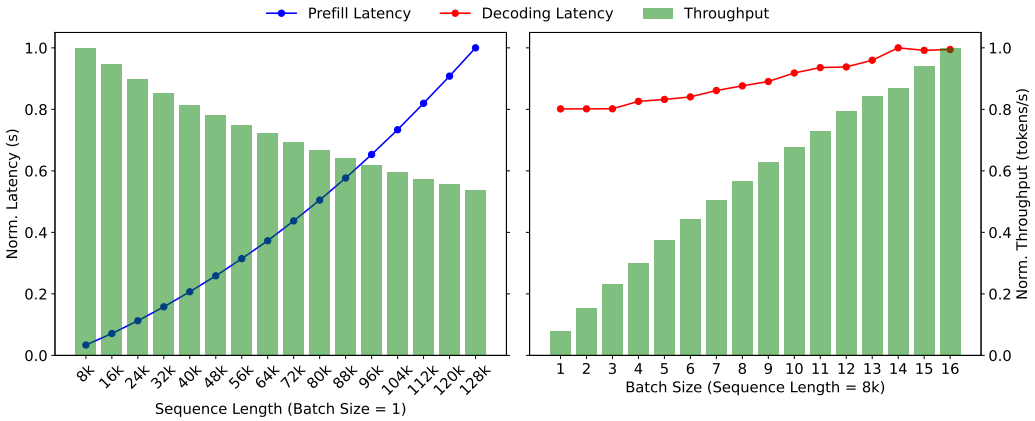


Figure 2: Normalized throughput and latency of prefill and decoding stages with different sequence lengths or batch sizes for the dummy LLaMA2-70B model.

the complexity of MLP scales linearly, computation time in the prefill stage generally increases superlinearly with input length, as shown in the left part of Figure 2.

In contrast, the decoding stage processes only one token at a time per batch due to the limitation of autoregressive generation. This makes it memory-constrained and causes computation time to increase sublinearly with batch size, as shown in the right part of Figure 2. A widely used optimization in the decoding stage is continuous batching [12, 13]. Before each iteration, the scheduler checks the status of all requests, adding newly arrived requests to the batch's prefill stage while removing completed requests.

Due to the distinct characteristics of the prefill and decoding stages, MaaS providers set different metrics to measure their corresponding Service Level Objectives (SLOs). Specifically, the prefill stage is mainly concerned with the latency between the request arrival and the generation of the first token, known as the time to first token (TTFT). On the other hand, the decoding stage focuses on the latency between successive token generations for the same request, referred to as the time between tokens (TBT).

As a MaaS provider, it is crucial to ensure quality assurance by meeting SLO metrics defined by service agreements. For example, a metric such as $\text{TTFT}_{P90} = 4\times$ indicates that 90% of inference requests will have a TTFT no greater than four times that of a single request running under the same conditions without interference. Specifically, in the end-to-end experiment of this paper (§7.1), we set $\text{TTFT}_{P90} = 10\times$ and $\text{TBT}_{P90} = 5\times$. In real deployments, we set fixed SLOs of TTFT and TBT. If monitoring detects unmet SLOs, we either add inference resources or reject some incoming requests.

However, due to the current contingent supply of GPUs, elastically scaling out the inference cluster is typically unfeasible. Therefore, deciding which requests to reject becomes a core issue in overload-oriented scheduling. Our main objective is to maximize overall throughput while adhering to SLOs, a concept referred to as goodput in other research [8, 14]. Our approach differs in that only requests that fully complete their execution are counted in the measure of goodput. Otherwise, all previously consumed/generated tokens are not counted, and the corresponding resources are wasted. In other words, a request should be rejected as early as possible if it cannot finish its full execution under the SLO. Achieving this goal involves not only optimizing the architecture of both the prefill and decoding stages but also developing a capability to predict short-term future loads.

3 Overview of Mooncake's Disaggregated Archtecture

As depicted in Figure 1, Mooncake employs a disaggregated architecture that not only separates prefill from decoding nodes, but also groups the CPU, DRAM, SSD, and RDMA resources of the GPU cluster to implement a disaggregated KVCache. This disaggregated cache harnesses underutilized

resources to provide ample cache capacity and transfer bandwidth, enabling efficient near-GPU prefix caching without additional costs.

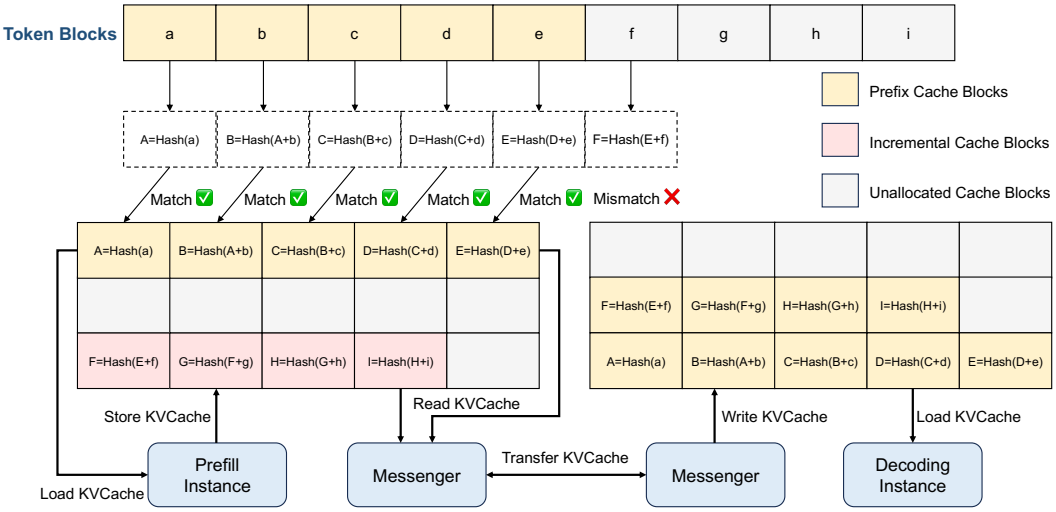


Figure 3: The KVCache pool in CPU memory. Each block is attached with a hash value determined by both its own hash and its prefix for deduplication.

Figure 3 illustrates the storage and transfer logic of the KVCache blocks. In CPU memory, KVCache is stored as paged blocks. Depending on the request patterns, it can use cache eviction algorithms such as LRU (Least Recently Used), LFU (Least Frequently Used), or algorithms based on request characteristics. The transfer of these KVCache blocks across CPUs and GPUs is handled by a separate (GPUDirect) RDMA-based component called Messenger. This archtecture also enables us to provide the context caching API to outside users for a higher reusement of KVCache.

To schedule all these disaggregated components, at its center, Mooncake implements a global scheduler named Conductor. Conductor is responsible for dispatching requests based on the current distribution of the KVCache and workloads. It also replicates or swaps certain blocks of the KVCache if it is beneficial for future inference. Specifically, Figure 4 demonstrates the typical workflow of a request. Once tokenizing is finished, the conductor selects a pair of prefill nodes and a decoding node, and starts a workflow comprising four steps:

1) KVCache Reuse: The selected prefill node (group) receives a request that includes the raw input, the block IDs of the prefix cache that can be reused, and the block IDs of the full cache allocated to the request. It loads the prefix cache from remote CPU memory into GPU memory based on the prefix cache block IDs to bootstrap the request. This step is skipped if no prefix cache exists. This selection balances three objectives: reusing as much KVCache as possible, balancing the workloads of different prefill nodes, and guaranteeing the TTFT SLO. It leads to a KVCache-centric scheduling that will be further discussed in §5.

2) Incremental Prefill: The prefill node (group) completes the prefill stage using the prefix cache and stores the newly generated incremental KVCache back into CPU memory. If the number of uncached input tokens exceeds a certain threshold ($prefill\_chunk$), the prefill stage is split into multiple chunks and executed in a pipeline manner. This threshold is selected to fully utilize the corresponding GPU's computational power and is typically larger than 1000 tokens. The reason for using chunked but still disaggregated prefill nodes is explained in §4.1.

3) KVCache Transfer: The aforementioned Messenger service is deployed in each node to manage and transfer these caches. Each Messenger operates as an independent process within its respective inference instance, receiving signals to facilitate high-speed, cross-machine KVCache transfer. This step is asynchronously executed and overlapped with the above incremental prefill step, streaming the KVCache generated by each model layer to the destination decoding node's CPU memory to reduce waiting time.

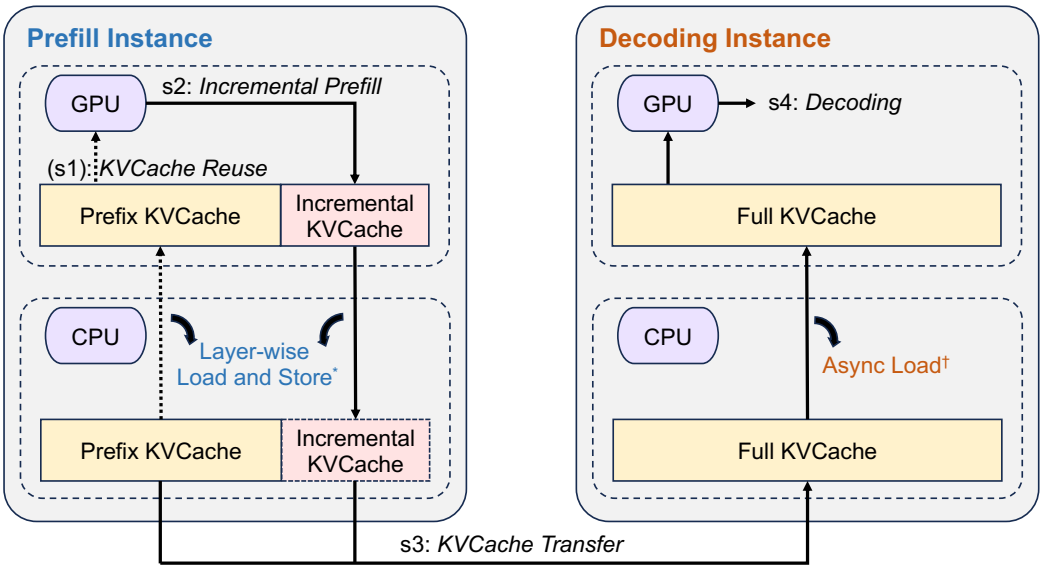


Figure 4: Workflow of inference instances. (∗) For prefill instances, the load and store operations of the KVCache layer are performed layer-by-layer and in parallel with the prefill computation to mitigate transmission overhead (see §4.2). (†) For decoding instances, asynchronous loading is performed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

4) Decoding: After all the KVCache is received in the CPU DRAM of the decoding node, the request joins the next batch in a continuous batching manner. Conductor pre-selects the decoding node based on its current load to ensure it does not violate the TBT SLO. However, this SLO is double-checked by the local scheduler because the anticipated load may have changed after the prefill stage. This double-checking may lead to the rejection of the request, in which case the corresponding prefill costs are wasted.

4 Implementation of the Prefill Pool

Unlike the inviolable decoding nodes, the necessity and best practices for designing a separate and elastic prefill pool remain under debate. For example, although many researchers [7, 8, 9] share our intuition to use a disaggregated architecture, it is worth discussing whether this separation is still necessary with the introduction of chunked prefill [15]. Chunked prefill divides the input tokens into multiple small chunks that join the continuous batch process. This approach has two clear benefits: *1)* Without separation, all nodes are treated equally, making scheduling easier; *2)* Inlining chunked prefill into the decoding batch can improve the computational intensity of the decoding batch, leading to better MFU.

However, after careful consideration, we decided to maintain Mooncake's disaggregated architecture. A request's prefill is inlined into the decoding batch only when it can be forwarded without chunking and without compromising the TBT SLO. There are two main reasons for this decision: *1)* Prefill nodes require different cross-node parallelism settings to handle long contexts (§4.1). *2)* It presents a unique opportunity to save VRAM (§4.2).

4.1 Multi-node Prefill

The available context length of recent LLMs is increasing rapidly, from 8k to 128K and even 1M [16]. Typically, for such long context requests, the input tokens can be 10 to 100 times larger than the output tokens, making optimizing the TTFT crucial. Due to the abundant parallelism in long context prefill, using more than a single 8x GPU node to process them in parallel is desirable. However, extending tensor parallelism (TP) across more than one node requires two expensive RDMA-based all-reduce operations per layer, significantly reducing the MFU of prefill nodes.

Recently, many works have proposed sequence parallelism (SP) [17, 18, 19, 20, 21, 22, 23]. SP partitions the input sequences of requests across different nodes to achieve acceleration. These SP methods take advantage of the associative property of the attention operator and require cross-node communication at least once per layer during the implementation of Ring Attention [18] or Striped Attention [19]. This greatly reduces network consumption and improves MFU.

However, adopting SP still results in a worse MFU compared to using single-node TP only. A desired deployment organizes prefill nodes into two groups: one with TP only and the other with SP. Requests are dispatched to the SP group only when necessary to meet the TTFT SLO. This further disaggregation leads to problems in dynamically adjusting the number of nodes in each group, as a static parallelism setting can result in low utilization across the cluster. Recent research [14] proposes elastic sequence parallelism to dynamically scale up or down the SP group. Although possible, this adds complexity to our architecture. For example, it requires establishing a global communication group in advance and complicates Conductor's design when considering metrics like cache reuse utilization and SLO requirement violations during adjustments. This makes it challenging for our situations that require frequent on-the-fly scalability during deployment. Additionally, SP still requires frequent cross-node communication, which lowers the MFU and competes with network resources for transferring KVCache across nodes.

To address this, Mooncake leverages the autoregressive property of decoder-only transformers and implements chunked pipeline parallelism (CPP) for long context prefill. We group every X nodes in the prefill cluster into a pipelined prefill node group. For each request, its input tokens are partitioned into chunks, each no longer than the $prefill\_chunk$. Different chunks of the same request can be processed simultaneously by different nodes, thus parallelizing the processing and reducing TTFT.

CPP offers two main benefits: *1)* Similar to pipeline parallelism in training, it requires cross-node communication only at the boundaries of each pipeline stage, which can be easily overlapped with computation. This leads to better MFU and less network resource contention with KVCache transfer. *2)* It naturally fits both short and long contexts, bringing no significant overhead for short context prefill and avoiding frequent dynamic adjustment of node partitioning. This pipeline-based acceleration method has been explored in training systems [24], but to our knowledge, this is the first application in the inference stage, as long context inference has only recently emerged.

4.2 Layer-wise Prefill

Beyond computational power, the limited size of VRAM is also a precious resource, and we aim to minimize the VRAM occupation by states, primarily the KVCache. Theoretically, if the KVCache size of a request is S and the processing time is T, its occupation cost is $S * T$. If a request is chunked and the processing of each chunk is inlined with other decoding requests in chunked prefill, T will increase, leading to a larger occupation cost.

Moreover, since prefill is processed layer-by-layer and is computation-bound, it is possible to overlap the transferring and dumping of KVCache with computation, further reducing its occupation cost. In Mooncake, KVCache loading and storing are executed asynchronously via launch and wait operations. Before each layer's attention computation begins, the model waits for the asynchronous loading of that layer's KVCache to complete and triggers the next layer's asynchronous KVCache loading. After the attention calculation is complete, asynchronous storage of that layer's KVCache is launched. Once all layers' computations are finished, the process waits for the completion of all asynchronous storage operations. Transfer overlapping allows the prefill instance's execution time to be roughly equivalent to either the KVCache loading time or the standard prefilling time, depending on the prefix cache proportion relative

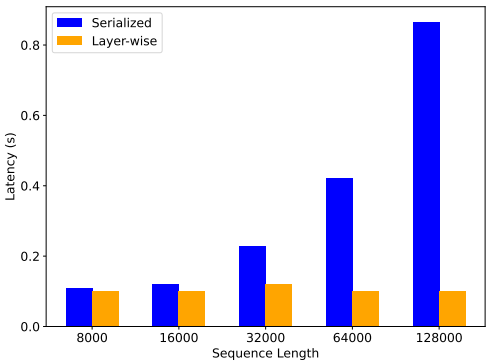


Figure 5: Latency of storing KVCache of different request lengths (Layer-wise latency refers to the difference in latency between Layer-wise Prefill and Prefill without storing KVCache).

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)

to the input length. The experimental result of KVCache storing latency, as shown in Figure 5, demonstrates that the layer-wise prefill can effectively reduce the latency for long-context requests.

The main advantage of this overlap effectiveness is that it enables us to disregard the available VRAM size in prefill scheduling, as long as it can contain a single request. As shown in Figure 1, the scheduling of prefill nodes only considers the KVCache distribution and the available DRAM size.

In the future, we intend to explore more uses for this free VRAM. For example, OpenAI recently proposed the use of batch APIs [25], which enable users to send asynchronous groups of requests at 50% lower costs, but with only a clear 24-hour turnaround time. This service is ideal for processing jobs that do not require immediate responses. Since there is no stringent TBT for these batch requests, we can inline even the decoding stage of these requests into prefill processing for better MFU, if there is enough VRAM space to hold the corresponding KVCache.

5 KVCache-centric Scheduling

In this section, we mainly discuss how Conductor schedules the requests and KVCache blocks under normal conditions, leaving the discussion on overload scenarios for the next section.

5.1 Prefill Global Scheduling

Previous research on LLM serving typically uses a load-balancing strategy that evaluates the load on each instance based on the number of assigned requests. In Mooncake, however, the selection of prefill instances considers additional factors—not just load but also the prefix cache hit length and the distribution of reusable KVCache blocks. While there is a preference to route requests to

prefill instances with longer prefix cache lengths to reduce computation costs, it may be beneficial to schedule them to other nodes to ensure overall system balance and meet TTFT SLOs. To address these complexities, we propose a cache-aware global scheduling algorithm that accounts for both the prefill time due to the prefix cache and the queuing time associated with the load on the instance.

Algorithm 1 details the mechanism for our cache-aware prefill scheduling. For every new request, its input tokens are divided into several blocks, and a hash key is computed for each block. This involves generating a hash key of tokens in a block concatenated with the hash key of the previous block (if available). The request's block keys are then compared one by one against each prefill instance's cache keys to identify the prefix match length ($prefix\_len$). Similar reuse logic is already implemented in vLLM, but the open-source version of vLLM only supports local KVCache caching.

With this matching information, Conductor estimates the corresponding execution time based on the request length and $prefix\_len$ (which varies by instance). It then adds the estimated waiting time for that request to get the TTFT on that instance. Finally, Conductor assigns the request to the instance with the shortest TTFT and updates the cache and queue times for that instance accordingly. If the SLO is not achievable, Conductor directly returns the HTTP 429 Too Many Requests response status code to the upper layers.

The backbone of this scheduling framework is straightforward, but complexities are hidden in the engineering implementation of various components. For example, to predict the computation time of the prefill stage for a request, we employ a predictive model derived from offline test data. This model estimates the prefill duration based on the request's length and prefix cache hit length. Thanks to the regular computation pattern of Transformers, the error bound of this prediction is small as long as enough offline data is available. The queuing time for a request is calculated by aggregating the prefill times of all queued requests. In practical implementations, TTFTs are computed in parallel, rendering the processing time negligible compared to the inference time.

More difficulty lies in predicting the transfer time because it is determined not only by the size of the transferred data but also by the current network status, especially whether the sending node is under congestion. This also necessitates the replication of hot KVCache blocks, which will be discussed in the next section.

5.2 Cache Load Balancing

In our Mooncake cluster, each prefill machine manages its own set of local prefix caches. The usage frequency of these caches varies significantly. For example, system prompts are accessed by almost every request, whereas caches storing content from a local long document may be used by only one user. As discussed in §5.1, Conductor's role is crucial in achieving an optimal balance between cache matching and instance load. Thus, from the perspective of the distributed cache system, load balancing also plays an important role. Specifically, it involves strategizing on how to back up caches to ensure that global prefill scheduling can achieve both high cache hits and low load.

A straw-man solution to this KVCache scheduling problem could be collecting the global usages of each block, using a prediction model to forecast their future usages, and making scheduling decisions accordingly. However, unlike the estimation of prefill time, workloads are highly dynamic and change significantly over time. Especially for a MaaS provider experiencing rapid growth in its user base, it is impossible to accurately predict future usages. Thus, we propose a heuristic-based automated hot-spot migration scheme to enhance cache load balancing.

As previously noted, requests may not always be directed to the prefill instance with the longest prefix cache length due to high instance load. In such cases, the conductor forwards the cache's location and the request to an alternative instance if the estimated additional prefill time is shorter than the transfer time. This instance proactively retrieves the KVCache from the holder and stores it locally. More importantly, we prefer to compute the input tokens if the best remote prefix match length is no larger than the current local reusable prefix multiplied by a threshold[1] Both strategies not only reduce the prefill time for requests but also facilitate the automatic replication of hot-spot caches, allowing for their broader distribution across multiple machines.

[1]This threshold is currently adjusted manually but can be adaptively adjusted by an algorithm in the future.

To validate the effectiveness of our strategy, we conducted a scheduling experiment that compares random scheduling and load-balancing scheduling with our strategy. We further compare the cache-aware scheduling described in §5.1 and the KVCache-centric scheduling described in this section that considers cache load balancing. In random scheduling, a prefill instance is selected arbitrarily for each request. In load-balancing scheduling, the instance with the lightest load is chosen. To evaluate, we built a Mooncake cluster consisting of 8 prefill instances and 8 decoding instances, using idle machines overnight, and replayed 23,000 real-world requests for the experiment. We assessed the performance of each scheduling algorithm using the average TTFT and the TTFT SLO attainment rate. The experimental results, depicted in Figure 6, demonstrate that both the cache-aware strategy and the cache load balancing strategy significantly reduce the TTFT of requests. Our KVCache-centric scheduling algorithm outperforms both random and load-balancing scheduling across both metrics. More experiment results can be found in §7.

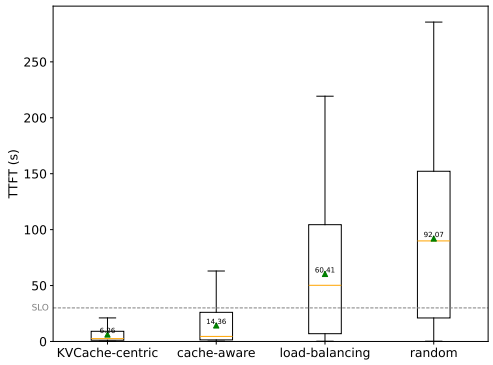


Figure 6: The prefill scheduling experiment in the Mooncake cluster.

6 Overload-oriented Scheduling

Most existing work on LLM serving assumes that all requests will be processed, optimizing the throughput or the TTFT and TBT of requests accordingly. However, in real scenarios, processing every incoming request is neither economical nor realistic. For commercial inference services facing rapidly increasing volumes of user requests, the growth rate of the cluster's inference resources is far slower than the increase in incoming requests. As a result, overload is a common issue in current LLM serving, especially during peak times.

To balance costs and user experience, the system should process as many requests as possible until the system load reaches a predefined threshold. After this point, remaining requests will be either directly rejected or deferred for later retry. Mooncake, implemented as a disaggregated inference system, allows for more flexible scheduling strategies but also confronts unique scheduling challenges not present in non-disaggregated systems and not mentioned in previous works[7, 8, 9].

In this section, we describe an early rejection policy designed specifically for a disaggregated architecture and address the load fluctuation caused by this approach. We then explore how predicting the generation length is necessary to mitigate these problems.

6.1 Scheduling in Overload Scenarios

In scenarios where system overload occurs, scheduling involves determining whether to accept or reject incoming requests based on the system load. A critical aspect of this process is defining what constitutes the "system load", as this definition influences the threshold at which requests are rejected. In conventional coupled systems, the prediction of TTFT and TBT can be complicated by interference between the prefill and decoding stages. Therefore, the load is often measured simply by the ratio of the number of requests being processed to the system's maximum capacity.

In contrast, Mooncake, with its disaggregated architecture, processes the prefill and decoding stages independently. Thus we use SLO satisfaction as a direct load measurement. Specifically, we define l_{ttft} and l_{tbt} as the TTFT and TBT SLO constraints for requests, respectively. The load for prefill and decoding instances is then determined by comparing the predicted maximum TTFT and TBT on an instance against l_{ttft} and l_{tbt}. With these two criteria, Mooncake's scheduling requires two key decisions: first, whether to accept the prefill stage based on the prefill instance's load, and second, whether to proceed with the decoding stage depending on the decoding instance's load.

6.2 Early Rejection

In practice, the individual load on prefill or decoding instances does not accurately reflect the actual number of requests processed by the system. This discrepancy arises due to a time lag between scheduling prefill and decoding 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, the computational resources expended during the prefill stage are wasted. Consequently, the actual number of successfully processed requests during prefill is less than that indicated by the load metric.

To address this issue, it is natural to advance the load assessment of the decoding instance to precede the beginning of the prefill stage. We refer to this strategy as **Early Rejection**. Upon the arrival of a request, Conductor evaluates whether to accept the request based on the greater load between the prefill and decoding pools. Early Rejection significantly reduces ineffective computations from rejected requests and enhances load balancing.

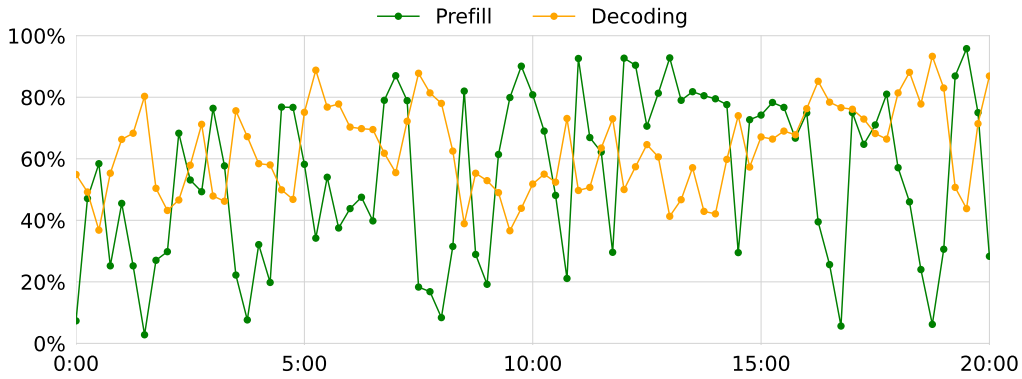


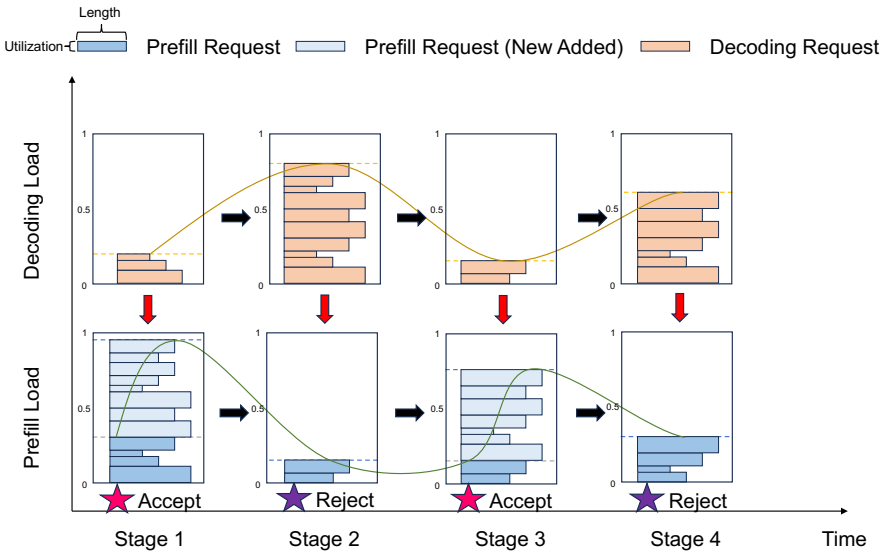
Figure 7: The load of prefill and decoding instances over 20 minutes, before using the prediction-based early rejection.

6.3 Load Fluctuation Caused by Early Rejection

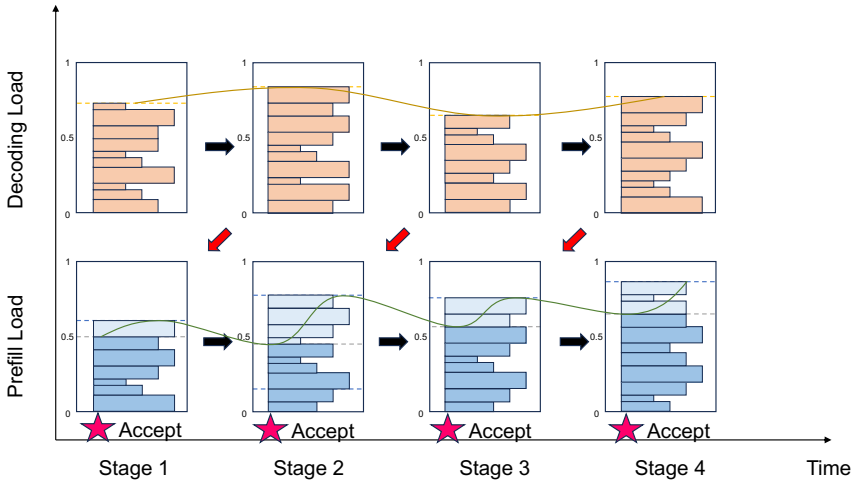
However, Early Rejection introduces new challenges. Figure 7 shows the observed real-world instance load over a 20-minute period in a cluster of 20 machines after using the Early Rejection strategy. It highlights significant anti-phase fluctuations between prefill and decoding machines. This phenomenon becomes more pronounced in clusters with fewer prefill machines and in scenarios where the prefill stage takes longer.

Upon further exploration, we found that this load fluctuation problem is rooted in the time lag between predicting the decoding load and its actual execution. Scheduling based on the current decoding load is inherently delayed. This delay causes fluctuations and phase staggering between the loads on prefill and decoding instances, as illustrated in the theoretical example described in Figure 8a. The green curve represents the load of prefill instances (scaled from 0 to 1), and the yellow curve represents the load of decoding instances.

In Stage 1, the load on both prefill and decoding instances is low, so Conductor accepts a large number of requests until the load on prefill instances reaches its limit. In Stage 2, requests processed by prefill instances are scheduled to decoding instances, causing the load on decoding instances to be high. Consequently, Conductor rejects incoming requests, leading to a lower load on prefill instances. In Stage 3, no new requests enter the decoding stage, resulting in a decreased load. At this point, Conductor again accepts a large number of requests until the prefill instances are fully loaded. In Stage 4, as the load on decoding instances increases, Conductor rejects requests, causing a low load on prefill instances. This severe fluctuation in load between prefill and decoding instances results in poor resource utilization of the inference cluster.



(a) Early Rejection.



(b) Early Rejection Based on Prediction.

Figure 8: Instance load when applying Early Rejection and Early Rejection Based on Prediction.

6.4 Early Rejection Based on Prediction

To solve the load fluctuation problem, we propose a framework of Early Rejection Based on Prediction to address scheduling challenges in overload scenarios for disaggregated LLM serving systems like Mooncake. As illustrated in Figure 8b, this framework predicts the decoding load after the prefill stage of incoming requests and uses this prediction to decide whether to accept the requests, which helps mitigate the fluctuation problem. The core component of this strategy is the accurate prediction of the decoding load for the subsequent period. We introduce two approaches for this:

Request level: Previous work highlights a significant challenge in predicting loads for LLM serving: the unknown output length of each request. If we could determine the output length in advance, it would be possible to estimate the TTFT and TBT much more accurately. This, in turn, would help predict the number of requests a decoding instance can complete and the number of new requests that will be added after a specified time, thereby obtaining the load at that time. However, predicting each request's output length is challenging due to high costs [9] or low accuracy, especially under overload conditions where resources are scarce and accurate predictions are necessary, making request-level predictions particularly difficult.

System level: In contrast to request-level predictions, system-level predictions do not attempt to predict the completion time for individual requests. Instead, they estimate the overall batch count or the TBT status for instances after a specified time. This type of prediction is ongoing and requires less precision, making it more appropriate for overload scenarios.

In Mooncake, we currently utilize a system-level prediction strategy: we assume that each request's decoding stage takes a uniform time t_d. First, for a given moment t, requests that can be completed by the prefill instances at t are added to the uniform decoding instances. Next, requests that will be completed (i.e., their execution time exceeds t_d) before t are removed from the decoding instances. Finally, the average TBT ratio of all decoding instances to l_{tbt} is calculated to predict the load. The exploration of request-level prediction is left for future work.

7 Evaluation

7.1 End-to-end Performance

Table 1: Datasets used in the end-to-end experiment.

Dataset	Avg Input Length	Avg Output Length	Cache Ratio	Arrival Pattern
ArXiv Summarization [26]	8088	229	~0%	Poisson Process
L-Eval [27]	19019	72	>80%	Poisson Process
Simulated Data	16k, 32k, 64k, 128k	512	50%	Poisson Process
Real Data	7955	194	~50%	Timestamp-based

This section evaluates the end-to-end performance of Mooncake under different datasets and various workloads. As stated before, to protect proprietary information and facilitate reproducibility, all the experimental results reported in this paper are based on a dummy model that follows the same architecture as LLaMA2-70B.

Testbed During the experiments, the system was deployed on a high-performance computing node cluster to test performance. Each node in the cluster is configured as follows: 8 NVIDIA-A800-SXM4-80GB GPUs, each with 80GB HBM, connected by NVLINK; equipped with RDMA network cards that supporting up to 800 Gbps of interconnect bandwidth between nodes. Each node deploys either a prefill instance or a decoding instance according to the startup parameter.

Dataset and Workload Building upon previous research [15, 8, 14], we selected or designed the datasets as outlined in Table 1. In addition to utilizing public datasets, we generated a batch of simulated data featuring predefined lengths and prefix cache ratios for our experiments. To examine performance in real-world scenarios, we constructed a dataset consisting of 23,000 real request traces, each annotated with an arrival timestamp. Experiments involving real request traces were conducted by replaying these requests according to their actual arrival times. For other scenarios, we simulated requests using a Poisson arrival process and controlled the request rate through RPS (Requests per Second).

Metric In the experiments, we focus on the throughput performance of various systems under defined SLOs. We measure the TTFT and TBT across different RPS rates, where a higher RPS signifies improved throughput. To assess whether the majority of requests satisfy the SLOs, we use the 90th percentile (P90) values of TTFT and TBT as the ultimate metrics. As mentioned in §2, the thresholds for TTFT and TBT are set by multiplying the lowest observed RPS values by factors of 10 and 5, respectively. Exceeding these thresholds indicates a failure to meet the SLOs and the corresponding consumed resources are considered as wasted. For ease of comparison, we normalize all TTFT and TBT values against these upper limits, establishing a baseline of 1.0.

Baseline We employ vLLM, one of the state-of-the-art open-source LLM serving systems, as our experimental baseline. vLLM incorporates continuous batching and PagedAttention technologies, significantly boosting inference throughput. Despite its strengths, vLLM's design, which couples the prefill and decoding stages of inference requests, can cause disruptions during decoding in scenarios involving long contexts.

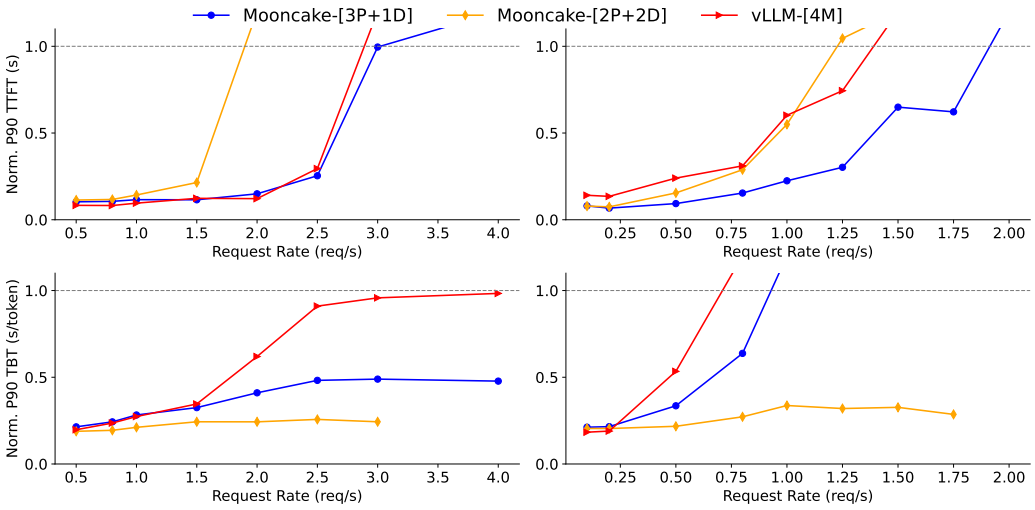


Figure 9: End-to-end experiments of Mooncake and vLLM on the ArXiv Summarization and L-Eval datasets

7.1.1 Public Datasets

This section evaluates the performance of Mooncake and vLLM in end-to-end tests on public datasets using ArXiv Summarization and L-Eval. We establish a baseline using a cluster of four vLLM instances, denoted as vLLM-[4M]. In contrast, Mooncake is configured in two distinct setups: one cluster consists of three prefill instances and one decoding instance, labeled Mooncake-[3P+1D], and the other has two prefill and two decoding instances, labeled Mooncake-[2P+2D]. The results, depicted in Figure 9, demonstrate that on the ArXiv Summarization and L-Eval datasets, Mooncake-[3P+1D] achieves throughput improvements of 20% and 40%, respectively, over vLLM-[4M] while satisfying SLOs. Moreover, Mooncake's throughput on the L-Eval dataset is further enhanced by prefix caching, which significantly reduces prefill time. However, despite having lower TBT latency, Mooncake-[2P+2D] does not perform as well on the TTFT metric compared to Mooncake-[3P+1D] and vLLM-[4M]. This discrepancy arises from an imbalance in the load between prefill and decoding instances. In real-world clusters, the demand for prefill and decoding instances generally remains stable over certain periods, with only minor temporary imbalances. Thus, the proportion of prefill and decoding instances can be preset. Future research will explore more flexible deployment and conversion methods.

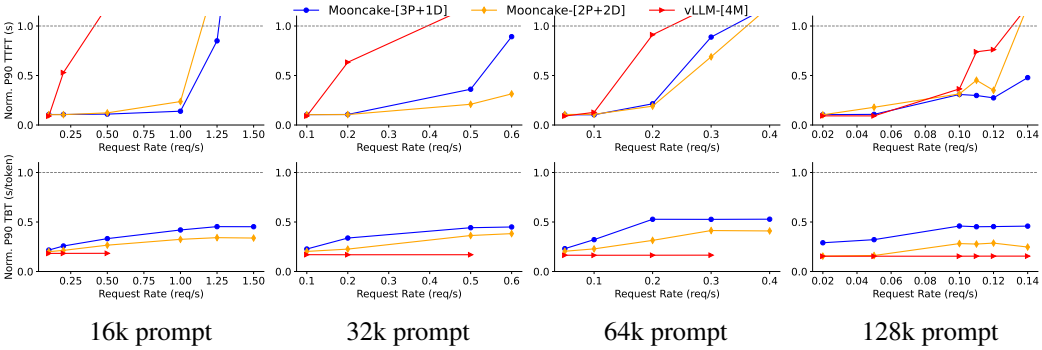


Figure 10: End-to-end experiments of Mooncake and vLLM on simulated data.

7.1.2 Simulated Data

In this section, we employ simulated data for an end-to-end experiment. The cluster configuration is the same as in §7.1.1, utilizing Mooncake configurations of [3P+1D], [2P+2D], and vLLM-[4M]. Notably, the long-context requests in simulated data significantly disrupt the decoding stage of vLLM. To counteract this, vLLM processes requests individually, rather than in batches. The results of the experiment are presented in Figure 10. Although Mooncake employs batch processing, its two-stage disaggregation design effectively minimizes the impact of the prefill stage on the decoding stage, ensuring it never breaks the TBT SLO. 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.

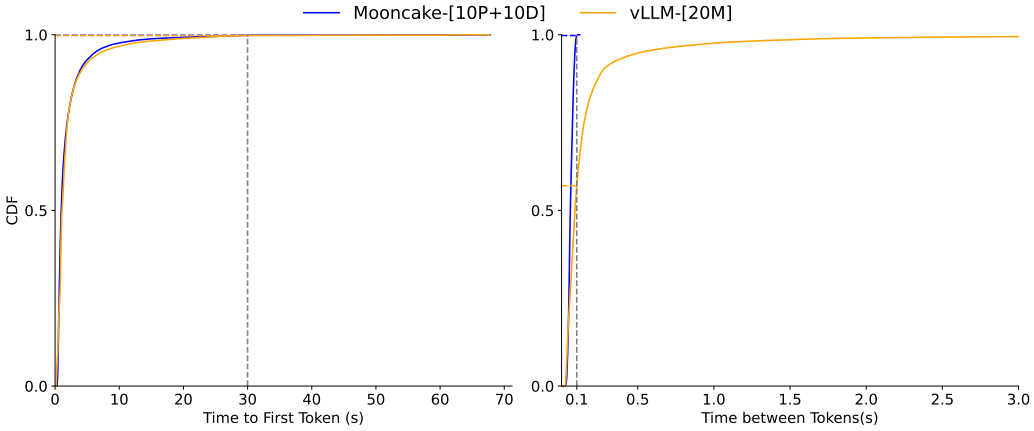


Figure 11: Request TTFT and TBT distributions of Mooncake and vLLM under real workloads

7.1.3 Real Workload

We further utilize 10 prefill instances and 10 decoding instances, labeled Mooncake-[10P+10D], along with 20 instances of vLLM, referred to as vLLM-[20M], to replay **real request traces** and conduct load tests on both Mooncake and vLLM. In this experimental setup, the upper limit for the TTFT is set at 30 seconds, while the TBT threshold is capped at 0.1 seconds per token. Figure 11 presents the CDF (Cumulative Distribution Function) plots for the TTFT and TBT for the two systems. The TTFT distributions for both Mooncake-[10P+10D] and vLLM-[20M] are nearly identical, with almost 100% of requests meeting the TTFT SLO. However, while approximately 100% of the requests for Mooncake-[10P+10D] satisfy the TBT SLO, only 57% of the requests for vLLM-[20M] meet this criterion, with some requests exhibiting extremely high TBTs. In this experiment, Mooncake can process approximately 75% more requests while adhering to the SLOs.

7.2 Performance in Overload Scenarios

In this section, we evaluate performance under overload scenarios, focusing on the maximum number of requests the system can handle, as discussed in §6. The baseline strategy, which rejects requests based on load before both stages start, leads to resource wastage by rejecting requests already processed in the prefill stage. In contrast, we propose the Early Rejection and Early Rejection based on Prediction strategies, detailed in §6.2 and §6.4, respectivly. These strategies take the system's load into comprehensive consideration, and hence reducing unnecessary request rejections.

Specifically, We built a Mooncake cluster with 8 prefill instances and 8 decoding instances and tested it using real traces from 23,000 requests. To simulate overload scenarios, we increased the replay speed to 2x.

Table 2 shows Mooncake's performance under different strategies. With the baseline strategy, the system rejects 4,183 requests. In contrast, under the Early Rejection and Early Rejection based on Prediction strategies, Mooncake rejects 3,771 and 3,589 requests, respectively. This demonstrates that by rejecting requests early, Mooncake can avoid unnecessary prefill computations, thereby improving

Table 2: Number of requests rejected by the system under the overloaded-scenario experiment.

	Baseline	Early Rejection	Early Rejection based on Prediction
Number of rejected requests	4183	3771	3589

the effective utilization of system resources. Furthermore, by predicting the load of decoding instances, Mooncake can mitigate load fluctuations, increasing the request handling capacity.

8 Related Work

Significant efforts have been dedicated to enhancing the efficiency of LLM serving systems through scheduling, memory management, and resource optimization. Production-grade systems like FasterTransformer [28], TensorRT-LLM [29], and DeepSpeed Inference [30] are designed to significantly boost throughput. Orca [12] employs iteration-level scheduling to facilitate concurrent processing at various stages, while vLLM [13] leverages dynamic KV cache management to optimize memory. FlexGen [31], SARATHI [15], and FastServe [32] incorporate innovative scheduling and swapping strategies to distribute workloads effectively across limited hardware, often complementing each other's optimizations.

Our design of Mooncake builds on these developments, particularly drawing from the open-source community of vLLM, for which we are deeply appreciative.

Moreover, recent research shares our insight into separating the prefill and decoding stages, leading to a disaggregated architecture that enhances system throughput. The arXiv publication of Splitwise [7] is at the early stage of the development of Mooncake, which further motivated our progress. Many concurrent works corroborate our findings, including DistServe [8], which optimizes resource allocation and parallel strategies for each stage to maximize GPU goodput, and TetriInfer [9], which incorporates both chunked prefill and two-stage disaggregation along with a predictive two-stage scheduling algorithm to optimize resource utilization.

Prefix caching is also widely adopted to enable the reuse of KV caches across multiple requests, reducing computational overhead in LLM inference systems [29, 13]. Prompt Cache [33] precomputes and stores frequently used text KV caches on inference servers, facilitating their reuse and significantly reducing inference latency. SGLang [34] leverages RadixAttention, which uses a least recently used (LRU) cache within a radix tree structure to efficiently enable automatic sharing across various reuse patterns.

Among these approaches, AttentionStore [35], a concurrent work with us, proposes a hierarchical KV caching system that utilizes cost-effective memory and storage media to accommodate KV caches for all requests. The architecture of Mooncake shares many design choices with AttentionStore. However, in long-context inference, the KV cache becomes extremely large, requiring high capacity and efficient data transfer along with KVCache-centric global scheduling. Additionally, Mooncake is not a standalone cache service, it incorporates both a memory-efficient cache storage mechanism and a cache-aware scheduling strategy, further improving prefix caching efficiency.

Furthermore, recent research [36] has started exploring the scheduling of prompts, which is essentially KVCache-centric scheduling. We corroborate many results in this area, although the real reusability in our online traces is much smaller than results reproduced by open-source benchmarks. Theoretically, up to only 50% of the KVCache can be reused in our current workloads, even if we assume both the capacity of storage and the TTFT SLO are infinite. However, this reusability highly depends on the application scenario and can be as large as 90% for certain scenarios, such as our chat-to-paper service `https://papers.cool/`. We also emphasize the need for overload-oriented scheduling subject to SLOs, rather than merely throughput-oriented scheduling.

9 Future Work

Disaggregating different parts of LLM serving into dedicated resource pools is key to Mooncake's high resource utilization. In the future, we plan to explore more opportunities along this path, particularly the potential use of heterogeneous accelerators. Current flagship accelerators balance

multiple metrics such as computational power, memory bandwidth, and capacity, making them versatile but not optimal in every single metric. For instance, considering only bandwidth per dollar or bandwidth per watt, current GDDR and even LPDDR solutions can be an order of magnitude better than flagship accelerators. We are also particularly interested in new technologies that use process-in-memory [37, 38, 39, 40] or hybrid bonding [41, 42, 43, 44, 45] techniques to implement memory-oriented devices that could offer both high bandwidth and high capacity in the near future. These technologies would be ideal for reducing the cost of executing memory-bound operations in the decoding phase.

Furthermore, in a heterogeneous accelerator environment that includes both computation-oriented and bandwidth-oriented accelerators, we can explore more advanced disaggregation architectures. For example, unlike other linear transformation operators, the arithmetic intensity of the attention operator in the decoding phase is only proportional to the number of attention heads divided by the number of key/value heads. This intensity cannot be increased by increasing the batch size and is typically more memory-bound than other operators. Therefore, it is possible to separate the attention operator from other linear operators to improve resource utilization further. According to our preliminary simulated results [46], this architecture has great potential to increase overall throughput. Additionally, the recently proposed MLA operator by DeepSeek-v2 [47] directly increases arithmetic intensity, solving this problem from another angle and showing great promise.

As an orthogonal direction, many algorithms aim to reduce the size of KVCache, benefiting Mooncake in two important ways: 1) increasing the batch size for better utilization and 2) improving the KVCache cache hit ratio to reduce prefill costs. This is currently a very active area, including different methods for compressing KVCache [48, 49, 50, 51, 52, 53], selecting important tokens by various metrics [54, 55, 56, 57, 58, 59, 60], sharing KVCache across different layers [61, 62, 63], or using hybrid architectures with operators that do not use KVCache [64, 65, 66, 67, 68, 69].

In terms of scheduling, we are developing an advanced policy that accounts for varying request priorities and scenarios with different TTFT/TBT SLOs. This policy is designed to enhance the responsiveness and efficiency of our system under diverse operational conditions. Effective management of KVCache, including replication, migration, and specialized eviction policies for partial hits and expiration scenarios, is also crucial for optimizing cache reuse. Additionally, we plan to dynamically balance prefill and decoding instances and investigate strategies for utilizing idle resources through batch-oriented offloading tasks. This approach will allow us to maximize resource utilization during fluctuating workloads.

10 Conclusion

This paper presents Mooncake, a KVCache-centric disaggregated architecture designed for efficiently serving LLMs, particularly in handling long contexts and overloaded scenarios. We discuss the necessity, challenges, and design choices involved in balancing the goal of maximizing overall effective throughput while meeting latency-related SLO requirements.

References

[1] OpenAI. Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022.

[2] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023. *URL https://arxiv. org/abs/2307.09288*, 2023.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[5] Moonshot AI. Kimi. `https://kimi.moonshot.cn`, 2023.

[6] NVIDIA. Nvidia h100 tensor core gpu architecture. `https://resources.nvidia.com/en-us-tensor-core`, 2022.

[7] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[8] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

[9] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[10] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[11] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[12] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[14] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism, 2024.

[15] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

[16] Google. Our next-generation model: Gemini 1.5. `https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024`, 2024.

[17] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[18] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[19] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers, 2023.

[20] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Joseph E Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. Lightseq:: Sequence level parallelism for distributed training of long context transformers. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.

[21] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[22] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, 2023.

[23] Jiarui Fang and Shangchun Zhao. Usp: A unified sequence parallelism approach for long context generative ai, 2024.

[24] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[25] OpenAI. Batch api. https://platform.openai.com/docs/guides/batch, 2024.

[26] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, 2018.

[27] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. L-eval: Instituting standardized evaluation for long context language models, 2023.

[28] NVIDIA Corporation. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2019.

[29] NVIDIA Corporation. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[30] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[31] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[32] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[33] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *arXiv preprint arXiv:2311.04934*, 2023.

[34] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

[35] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024.

[36] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving. https://escholarship.org/uc/item/1bm0k1w0, May 2024.

[37] Xiaoxuan Yang, Bonan Yan, Hai Li, and Yiran Chen. Retransformer: Reram-based processing-in-memory architecture for transformer acceleration. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Ann Franchesca Laguna, Arman Kazemi, Michael Niemier, and X. Sharon Hu. In-memory computing based accelerator for transformer networks for long sequences. In *2021 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 1839–1844, 2021.

[39] Wantong Li, Madison Manley, James Read, Ankit Kaul, Muhannad S. Bakir, and Shimeng Yu. H3datten: Heterogeneous 3-d integrated hybrid analog and digital compute-in-memory accelerator for vision transformer self-attention. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(10):1592–1602, 2023.

[40] Shrihari Sridharan, Jacob R. Stevens, Kaushik Roy, and Anand Raghunathan. X-former: In-memory acceleration of transformers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(8):1223–1233, 2023.

[41] Negar Akbarzadeh, Sina Darabi, Atiyeh Gheibi-Fetrat, Amir Mirzaei, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. H3dm: A high-bandwidth high-capacity hybrid 3d memory design for gpus. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(1), feb 2024.

[42] The Korea Economic Daily. Samsung to unveil 3D DRAM in 2025 to lead AI chip market. https://www.kedglobal.com/korean-chipmakers/newsView/ked202404020016, 2023.

[43] Jishen Zhao, Guangyu Sun, Gabriel H. Loh, and Yuan Xie. Optimizing gpu energy efficiency with 3d die-stacking graphics memory and reconfigurable memory interface. *ACM Trans. Archit. Code Optim.*, 10(4), dec 2013.

[44] Jishen Zhao, Qiaosha Zou, and Yuan Xie. Overview of 3-d architecture design opportunities and techniques. *IEEE Design & Test*, 34(4):60–68, 2017.

[45] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[46] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and economic large language model inference with attention offloading, 2024.

[47] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[48] Hao Yu, Zelan Yang, Shen Li, Yong Li, and Jianxin Wu. Effectively compress kv heads for llm, 2024.

[49] Zefan Cai., Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling, 2024.

[50] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization. 2023.

[51] Yefei He, Luoming Zhang, Weijia Wu, Jing Liu, Hong Zhou, and Bohan Zhuang. Zipcache: Accurate and efficient kv cache quantization with salient token identification, 2024.

[52] Ruikang Liu, Haoli Bai, Haokun Lin, Yuening Li, Han Gao, Zhengzhuo Xu, Lu Hou, Jun Yao, and Chun Yuan. Intactkv: Improving large language model quantization by keeping pivot tokens intact, 2024.

[53] Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models, 2024.

[54] Zhiyu Guo, Hidetaka Kamigaito, and Taro Watanabe. Attention score is not all you need for token importance indicator in kv cache reduction: Value also matters, 2024.

[55] Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. A simple and effective l_2 norm-based strategy for kv cache compression, 2024.

[56] Yao Yao, Zuchao Li, and Hai Zhao. Sirllm: Streaming infinite retentive llm, 2024.

[57] Dongjie Yang, XiaoDong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. Pyramidinfer: Pyramid kv cache compression for high-throughput llm inference, 2024.

[58] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference, 2024.

[59] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024.

[60] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

[61] Zayd Muhammad Kawakibi Zuhri, Muhammad Farid Adilazuarda, Ayu Purwarianti, and Alham Fikri Aji. Mlkv: Multi-layer key-value heads for memory efficient transformer decoding, 2024.

[62] Haoyi Wu and Kewei Tu. Layer-condensed kv cache for efficient inference of large language models, 2024.

[63] Character.AI. Optimizing AI Inference at Character.AI. `https://research.character.ai/optimizing-inference/`, 2023.

[64] Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models, 2024.

[65] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024.

[66] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwkv: Reinventing rnns for the transformer era, 2023.

[67] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024.

[68] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024.

[69] Aleksandar Botev, Soham De, Samuel L Smith, Anushan Fernando, George-Cristian Muraru, Ruba Haroun, Leonard Berrada, Razvan Pascanu, Pier Giuseppe Sessa, Robert Dadashi, Léonard Hussenot, Johan Ferret, Sertan Girgin, Olivier Bachem, Alek Andreev, Kathleen Kenealy, Thomas Mesnard, Cassidy Hardin, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Armand Joulin, Noah Fiedel, Evan Senter, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, David Budden, Arnaud Doucet, Sharad Vikram, Adam Paszke, Trevor Gale, Sebastian Borgeaud, Charlie Chen, Andy Brock, Antonia Paterson, Jenny Brennan, Meg Risdal, Raj Gundluru, Nesh Devanathan, Paul Mooney, Nilay Chauhan, Phil Culliton, Luiz GUStavo Martins, Elisa Bandy, David Huntsperger, Glenn Cameron, Arthur Zucker, Tris Warkentin, Ludovic Peran, Minh Giang, Zoubin Ghahramani, Clément Farabet, Koray Kavukcuoglu, Demis Hassabis, Raia Hadsell, Yee Whye Teh, and Nando de Frietas. Recurrentgemma: Moving past transformers for efficient open language models, 2024.