Disaggregating Prefill and Decoding: A System Optimization Approach for Large Language Model Serving

224040362, *Wang Ye*

***Abstract*—Large language models (LLMs) have achieved remarkable success across a wide range of natural language processing tasks, driving significant advancements in artificial intelligence. However, the growing scale and complexity of these models introduce significant challenges in serving them efficiently, particularly during the inference phase. Traditional methods for LLM serving often rely on colocation of the prefill and decoding stages, which can lead to inefficiencies due to their differing computational patterns and performance requirements. This survey addresses these challenges by systematically reviewing state-of-the-art optimization techniques that disaggregate the prefill and decoding stages of LLM inference. We categorize existing approaches into three key areas: disaggregated architecture design, key-value (KV) cache pool management for distributed memory optimization, and scheduling strategies. Additionally, we provide a comprehensive analysis of experimental results, evaluating the effectiveness of these techniques in improving system efficiency and resource utilization. The paper aims to offer valuable insights into the current landscape of LLM serving optimization and identifies promising directions for future research in this field.**

***Index Terms*—Large language models (LLMs), model serving optimization, disaggregated architecture, distributed memory management, scheduling strategies.**

I. Introduction

GENERATIVE large language models (LLMs) have become a transformative force in the field of artificial intelligence (AI), significantly advancing the capabilities of machine learning systems across a variety of language-intensive tasks. These models have shown exceptional performance in areas such as natural language processing (NLP), image captioning, visual recognition, and even multimodal tasks. As such, they have emerged as a critical component in the evolution of artificial general intelligence (AGI), shaping both theoretical research and practical applications in AI.

Among the many contributions of LLMs to AI, the advent of transformer-based architectures has marked a paradigm shift in the way machines understand and generate human language. Notable examples include the GPT family (Generative Pre-trained Transformer) [1], the LLaMA family [2], and other cutting-edge models such as OPT [3], BLOOM [4], Mistral [5], DeciLM [6], Baichuan [7], and GLM [8]. These transformer-based LLMs distinguish themselves from traditional neural networks and earlier architectures in several key ways. One of the most striking differences is their sheer scale. The size of the representative model grows astonishingly over time. For instance, as shown in Fig. 1, while earlier models such as T5 consisted of 11B parameters, state-of-the-art LLMs like PanGu can contains over 1G parameters, resulting in nearly 100 times of magnitude increase in model complexity. This massive scale enables LLMs to perform at unprecedented levels but also introduces substantial challenges in terms of resource utilization, including memory requirements, computational costs, and communication overheads.

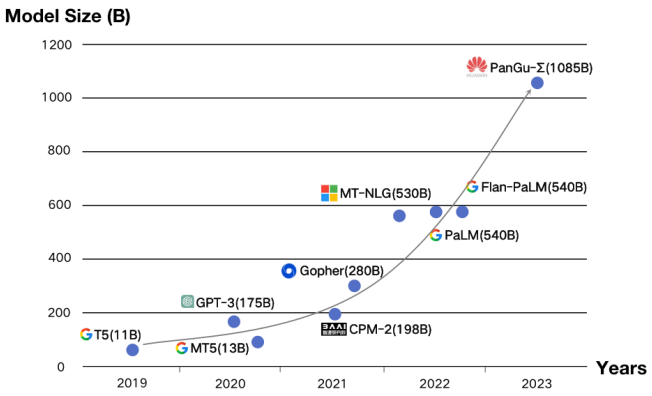


Fig. 1: The scaling trend of models over time.

However, this exponential growth in model size does not imply that every part of the transformer architecture contributes equally to the overall scaling trend. Transformer models are generally composed of two primary stages: the attention mechanism and the feed-forward network (FFN). Each of these stages plays a distinct role in the model's ability to process and generate language. Moreover, the inference process of LLMs can be divided into two additional stages: the prefill stage, during which initial context is loaded, and the decoding stage, where the model generates predictions based on the context. These two stages, though part of a continuous process, differ significantly in their computational characteristics, which introduces unique challenges for efficient model deployment.

Given the complexity and diversity of operations within transformer-based models, optimizing the inference process for LLMs has become a critical area of research. Traditional optimization approaches such as parallelization [9], quantization [10], and kernel optimization [11] have been explored to mitigate the computational burden of LLMs. However, these methods often overlook the heterogeneous nature of

This paper was produced by the IEEE Publication Technology Group. They are in Piscataway, NJ.

Manuscript received April 19, 2021; revised August 16, 2021.

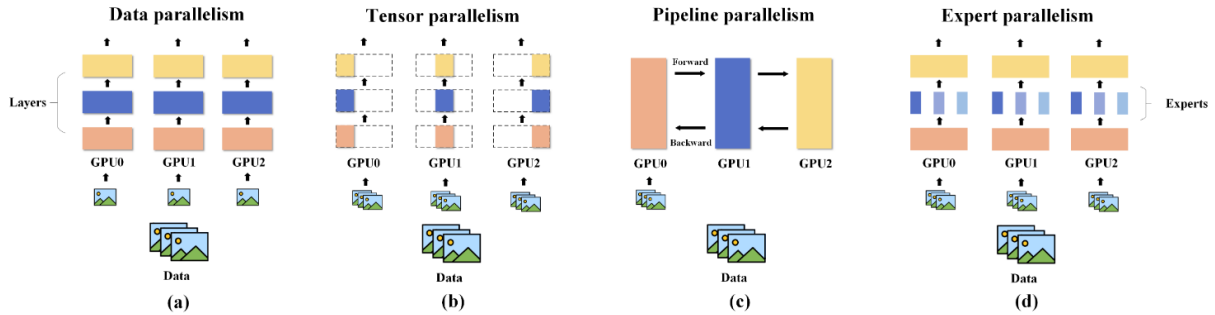


Fig. 2: Schematic diagram of parallelization strategies in machine learning systems. Different color blocks indicate different layers in the network.

the inference process, particularly the distinct demands of the prefill and decoding stages. Addressing this heterogeneity is crucial for improving both the efficiency and scalability of LLM serving.

This paper presents a comprehensive survey of state-of-the-art methods for optimizing LLM serving by disaggregating the prefill and decoding stages. We provide a systematic categorization of existing approaches, focusing on three key aspects: disaggregated architecture realization, key-value (KV) cache pool management (distributed memory optimization), and scheduling strategies. Additionally, we include a detailed analysis of the experimental results of various methods, offering insights into their relative performance and applicability. Our goal is to provide both researchers and practitioners with a deeper understanding of the current landscape of LLM serving optimization and highlight promising directions for future work.

The remainder of this paper is organized as follows: Section 2 reviews related works and provides the necessary background for understanding the challenges in LLM serving. Section 3 outlines the motivations and theoretical foundation for disaggregating the prefill and decoding stages. Section 4 categorizes the various disaggregation techniques and discusses the associated works within each category. Section 5 examines the evaluation methodologies used to assess the performance of these optimization methods. Finally, Section 6 concludes the survey and proposes future directions for research in this area.

II. Related Work

A. Conventional Machine Learning System (MLSys) Optimizations

Parallelization strategies have long been a cornerstone in optimizing machine learning (ML) systems, particularly for large-scale model serving. These techniques leverage the parallel processing capabilities of modern hardware architectures, distributing computations across multiple cores or devices to achieve significant speedups during inference. Prior to recognizing the distinct computational demands of prefill and decoding stages, parallelization was applied across the entire inference pipeline in a uniform manner.

Most model parallelism techniques were initially developed for distributed training of large deep neural networks (DNNs), particularly transformer-based models. Figure 2 illustrates several parallelism strategies commonly employed in distributed training. For example, tensor model parallelism (TP) [12] partitions model layers—such as attention and feed-forward networks (FFN)—into smaller chunks, with each chunk deployed across separate devices (e.g., GPUs). This approach can greatly reduce inference latency, particularly in environments with high-speed interconnects like NVLink, allowing for efficient use of multiple GPUs within the same machine. Moreover, when applied to multi-query attention with a single head for key-value pairs, TP can be combined with data parallelism in a hybrid tensor partitioning strategy.

In contrast, pipeline model parallelism (PP) [13] organizes model layers into a sequence, with each device responsible for processing one or more consecutive layers. While PP can increase the throughput (i.e., the number of inputs processed per unit of time), it does not inherently reduce the time required to process a single input from start to finish, as TP does. Sequence parallelism (SP) represents another variation, with its key innovation being the distribution of long sequences across multiple GPUs along the sequence length dimension [14]. Each GPU processes a portion of the sequence, helping to balance computational and memory loads. Different parallelism approaches introduce varying levels of communication overhead and computational latency [15], and achieving optimal performance often requires careful consideration of these trade-offs.

To automate the selection of the most efficient parallelization strategy, several frameworks for distributed training have been proposed, such as Alpa [16], FlexFlow [17], and Galvatron [18]. These systems aim to optimize resource utilization and improve overall system efficiency by dynamically adjusting parallelism strategies based on the characteristics of the model and the underlying hardware.

B. Resources in Cluster

The successful deployment of LLMs also relies on the effective utilization of resources within a computational cluster. These resources include computational, communication, and memory resources, each of which plays a critical role in optimizing the performance of distributed systems.

- **Node-to-Node Communication Resources:** Remote Direct Memory Access (RDMA) [19] is a key technology

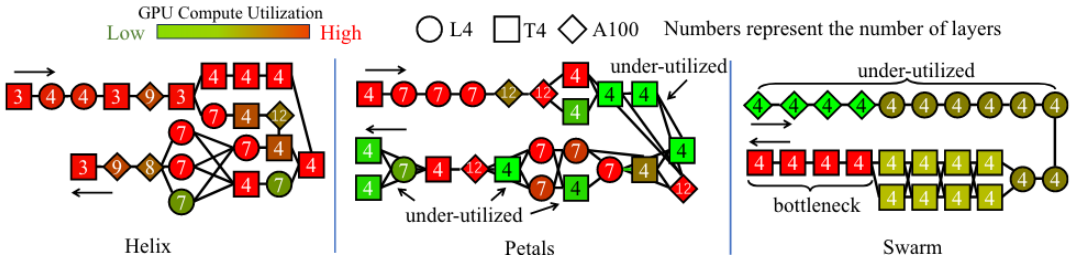


Fig. 3: Leveraging different types of computational resources in a heterogeneous GPU cluster.

for enabling high-speed, low-latency data transfer between nodes in a cluster. RDMA allows one computer's memory to be accessed directly by another without involving the operating system, significantly reducing communication overhead. GPUDirect-RDMA [20]further enhances this process by enabling direct communication between GPUs across different nodes, bypassing the CPU entirely. This is particularly advantageous for large-scale LLM training, where the rapid synchronization of model parameters and gradients is crucial. Two common RDMA technologies are InfiniBand and RDMA over Converged Ethernet (RoCE) [21]. InfiniBand, known for its low-latency, high-speed capabilities, is widely used in high-performance computing (HPC) environments, such as the Eagle supercomputer [22]. RoCE, on the other hand, takes advantage of existing Ethernet infrastructure to provide RDMA functionality.

- **Heterogeneous Computational Resources:** In contrast to traditional data center setups, which typically assume homogeneous clusters of nodes, modern LLM serving often requires heterogeneous computational resources. As LLMs scale to increasingly large sizes and the availability of the latest-generation GPUs becomes more limited, deploying models across heterogeneous devices has become an essential strategy. However, this approach introduces new challenges that were not fully addressed in earlier optimization methods. Figure 3 highlights several recent works [12] that explore the use of heterogeneous resources in LLM serving. These studies aim to better utilize the diverse hardware available in clusters, including CPUs, GPUs, and other accelerators, to achieve more efficient processing and resource allocation.
- **Memory Resources:** Memory resources play a pivotal role in the distributed training and serving of LLMs. Storage systems must be designed to match the computational power of the GPUs they support to avoid bottlenecks that could waste computational resources. Furthermore, these systems must be scalable, capable of storing both large-scale structured and unstructured training datasets, and able to handle the storage and retrieval of model checkpoints throughout training. In addition to meeting the performance requirements dictated by model size and training duration, the memory system must also satisfy enterprise-level needs such as data protection, high availability, and security.

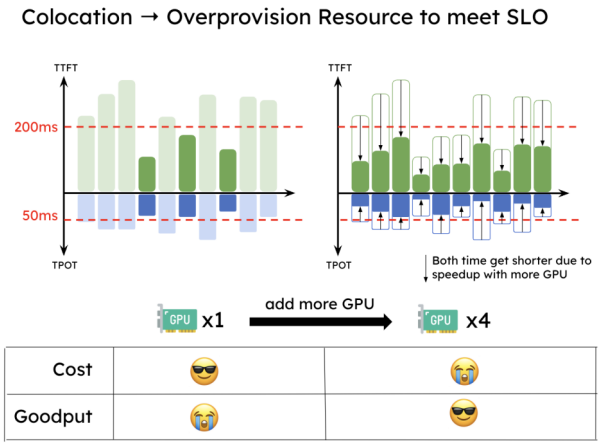


Fig. 4: Because of the interference between prefill and decoding under co-location, resources need to be over-provisioned to meet SLOs.

III. Analysis and Motivation

This section first discusses the reason people do disaggregation for prefill and decoding. After that, we address an intuitive implementation of disaggregation, and then introduce the tradeoff between data transfer overhead and throughput improvement.

A. Motivation for Disaggregating Prefill and Decoding

The motivation behind disaggregating the prefill and decoding stages of large language model (LLM) inference lies in addressing the inherent inefficiencies caused by the conventional approach of colocating these two stages. Prefill and decoding, while both integral to the inference process, exhibit distinct computational patterns and performance characteristics. This divergence introduces several challenges when both stages are executed together on the same infrastructure.

Firstly, the computation patterns of prefill and decoding differ significantly. The prefill stage typically involves the initialization and loading of context, requiring high throughput to efficiently process large amounts of input data in parallel. In contrast, the decoding stage, which generates predictions based on the context, involves sequential processing and more complex dependencies between tokens, leading to different resource demands. As a result, when these stages are colocated,

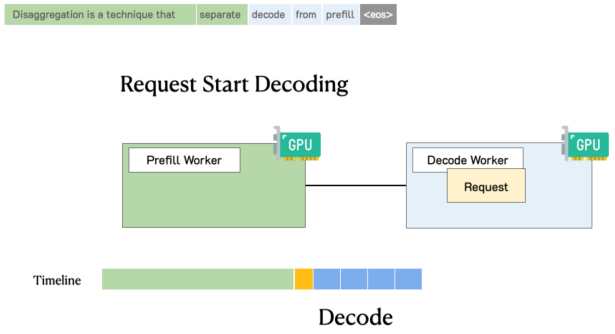


Fig. 5: Intuitive implementation of disaggregation.

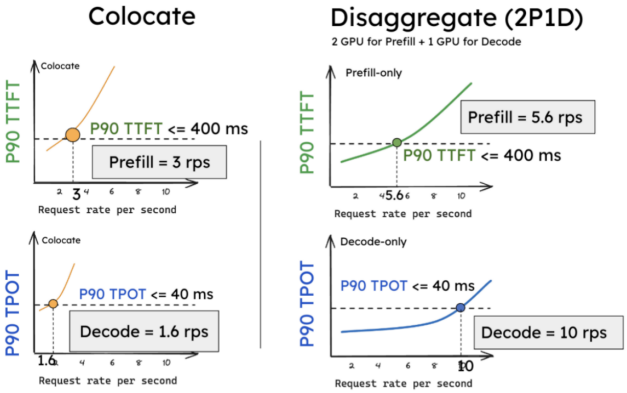


Fig. 6: KV cache transfer could be not a bottleneck.

interference occurs, as the resource allocation strategies optimized for one stage may negatively impact the performance of the other.

Secondly, prefill and decoding have different service-level objectives (SLOs). The prefill stage prioritizes throughput and fast data loading, aiming to handle a large volume of input efficiently. Decoding, on the other hand, is more latency-sensitive, as it involves generating output tokens one at a time, often requiring real-time responsiveness. The simultaneous execution of both stages under a unified resource management strategy may cause conflicts in optimizing these SLOs, as the system is forced to balance throughput and low-latency requirements, often to the detriment of one or the other, just as shown in Fig. 4.

Furthermore, the conventional practice of coupling prefill and decoding also leads to suboptimal resource allocation and parallelism strategies. The shared allocation of computational resources and parallelism mechanisms between the two stages prevents the system from tailoring its approach to the unique needs of each stage. This results in inefficient utilization of hardware resources and can significantly slow down the overall inference process.

By disaggregating the prefill and decoding stages, it becomes possible to optimize the resource allocation, parallelism strategies, and execution pipelines for each stage independently, thereby improving the overall performance and efficiency of LLM inference systems. This approach enables a more granular and adaptable optimization process, allowing for better utilization of hardware resources and a more balanced achievement of the performance goals for both stages.

B. Intuitive Implementation of Disaggregation

In Fig. 5 it shows an intuitive implementation of disaggregating prefill and decoding. To be specific, it deploys prefill stage on one GPU (called Prefill Worker), and decoding stage on another (called Decode Worker). In this way, these two stages that may interfere with each other are separated to different hardware. This intuitive implementation is beneficial for a better chance for separated optimization, but introduces extra overhead: transferring KV Cache between GPUs for the two stages.

C. Overhead-Throughput Tradeoff Analysis

We analyze from either side of the tradeoff to see how it works.

1) **The throughput gain.** Take Fig. 6 as an example. Assume the two SLOs (time-to-first-token (TTFT) for prefill, and time-per-output-token (TPOT) for decoding) are:

$$90\%\ TTFT \leq 400ms \tag{1}$$

$$90\%\ TPOT \leq 40ms \tag{2}$$

On the left of Fig. 6 shows the request rate per second (rps) for prefill and decode when these two stages are co-located, while on the right of Fig. 6 shows the rps for both when they are separated, with 2 GPUs serving prefill and 1 GPU serving decoding. In this case, we can calculate the max_system_rps following:

$$\begin{aligned} max\_system\_rps_{co} &= Min(Prefill, Decode) \\ &= 1.6\ rps/GPU \end{aligned} \tag{3}$$

$$\begin{aligned} max\_system\_rps_{disagg} &= \frac{Min(Prefill \times 2, Decode)}{3\ GPU} \\ &= 3.3\ rps/GPU \end{aligned} \tag{4}$$

From Equation 3 and Equation 4 we can tell that $max\_system\_rps_{co} \leq max\_system\_rps_{disagg}$, which represents an over 100% throughput enhancement after disaggregation.

2) **The KV cache transfer overhead.** Fig. 7 indicates that the introduced overhead of KV cache transferring is not always the bottleneck. We can infer that the overhead is acceptable if:

- The network between the nodes leverage high-speed networks (e.g., NVLink, PCI-e, IB, etc., or
- tasks are assigned to long sequences or large models, so that the overall computing time cost scales fast.

So the optimization goal of this kind of scenarios turns into: how to optimize throughput while maintaining low transfer overhead?

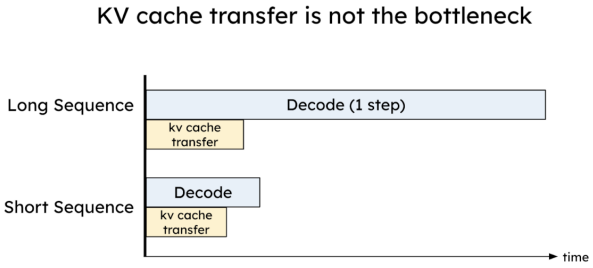


Fig. 7: Throughout comparison: co-location VS disaggregation.

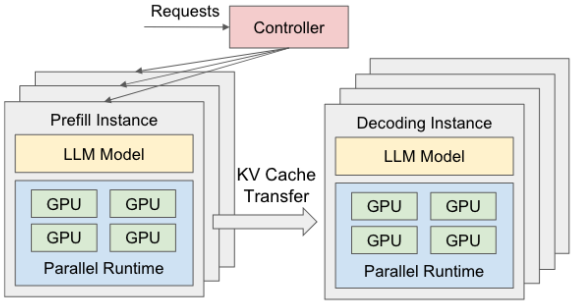


Fig. 8: DistServe Runtime System Architecture.

IV. System Design

In this section, based on the previous analysis, we introduce three state-of-the-art works that dedicated to do system design for reaching the best tradeoff and enhance system total performance.

A. DistServe

DistServe [23] is a system designed to address the challenges associated with optimizing the serving of large language models (LLMs), particularly in the context of disaggregating the prefill and decoding stages. The approach taken by DistServe is centered around identifying the optimal placement of prefill and decoding instances within a cluster, ensuring that the system meets specific latency requirements and service level objectives (SLOs) while maximizing per-GPU goodput. The following section provides a detailed breakdown of the key contributions of DistServe, focusing on its placement algorithms, online scheduling strategies, and optimization techniques for both high and low node-affinity clusters.

1) Placement for High Node-Affinity Clusters: DistServe's placement algorithm for high node-affinity clusters, which are equipped with high-speed cross-node networks such as Infiniband, allows for the flexible deployment of prefill and decoding instances across multiple nodes. In such clusters, KV cache transmission overhead is negligible, allowing DistServe to optimize the parallelism strategies for prefill and decoding instances separately. The system employs a two-level placement algorithm, wherein it first optimizes the parallel configurations for each instance type to achieve phase-level optimal goodput, followed by replication to match the overall traffic rate. This process involves simulating different configurations using a workload simulator to estimate SLO attainment and identify the best configuration. The algorithm's complexity is manageable, and simulation results show that it can operate efficiently, even in large-scale settings.

2) Placement for Low Node-Affinity Clusters: In clusters with low node-affinity, where bandwidth between nodes is limited, DistServe addresses the challenge of KV cache transmission by colocating prefill and decoding instances on the same node. This approach leverages NVLINK, a high-bandwidth interconnect available inside GPU nodes, to facilitate efficient data transfer. However, for large models, this may require co-locating multiple instance segments to fit within the resource limits of the node. DistServe introduces a new method of partitioning the model into smaller instance segments, each corresponding to a specific inter-operation stage, and optimizes their placement within the node to minimize resource contention. This solution is implemented in Algorithm 2, which enumerates possible configurations for instance segments and uses simulation to identify the best parallelism strategy.

3) Online Scheduling Optimization: DistServe's runtime architecture is built around an online scheduling system (showed in Fig. 8) designed to optimize the use of prefill and decoding instances under real-world workload conditions. The system uses a simple FCFS (First-Come, First-Served) scheduling policy but includes several optimizations to address key challenges such as workload burstiness and non-uniform prompt lengths. The scheduler balances the execution time of batches to reduce pipeline bubbles and prevent memory overload by implementing a "pull" method for KV cache transmission, where decoding instances fetch the required data from prefill instances as needed. Additionally, DistServe supports periodic re-planning of resource allocations based on detected shifts in workload patterns, ensuring the system remains adaptable over time. Although not the primary focus of the system, DistServe also discusses potential future extensions, including preemption strategies and fault tolerance mechanisms to enhance system resilience.

This section encapsulates DistServe's key innovations in optimizing LLM serving through intelligent placement algorithms and scheduling strategies. The system's ability to adapt to varying hardware configurations and workload characteristics makes it a promising solution for large-scale, efficient LLM deployment.

B. Splitwise

1) Splitwise: A Technique for Phase Splitting in LLM Inference: Based on the characterization insights, the authors propose `Splitwise` [24], a technique that splits the prefill and decoding phases of large language model (LLM) inference onto separate machines. The overall architecture of `Splitwise` is depicted in Fig. 9. This approach maintains two distinct pools of machines for prompt and token processing. Additionally, a third mixed pool expands and contracts dynamically to meet workload demands. Each machine in the

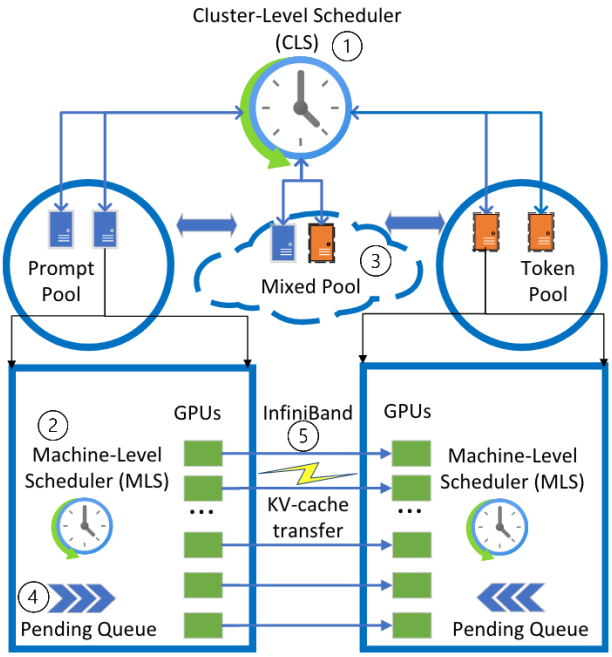


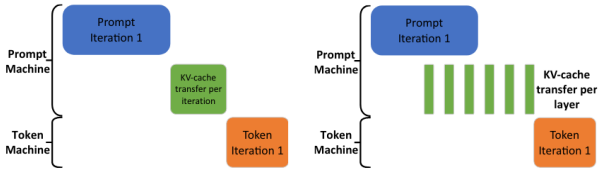
Fig. 9: High-level system diagram of Splitwise.

system is preloaded with the model of choice, ensuring that no time is wasted in loading the model during inference.

When a new inference request arrives, the scheduler allocates it to a pair of machines—one responsible for the prompt phase and the other for the token generation phase. The prompt machine processes the input query tokens and generates the initial token, creating the key-value (KV) cache that is necessary for the token generation phase. Once the KV-cache is generated, it is transmitted to the token machine, which continues generating the response. This separation allows for better resource utilization and a more flexible allocation of resources based on workload demands.

2) Cluster-Level Scheduling for Efficient Machine Pool Management: At the cluster level, `Splitwise` introduces a two-level scheduling mechanism, where the cluster-level scheduler (CLS) is responsible for managing the machine pools and routing inference requests. The machine pools are divided into three categories: the prompt pool, the token pool, and the mixed pool. The CLS ensures that machines are assigned to the appropriate pool based on the expected workload. If the load distribution deviates from initial assumptions, the CLS may dynamically move machines between the pools to balance the load and minimize fragmentation.

The two-level scheduling architecture is where the CLS uses a Join the Shortest Queue (JSQ) scheduling algorithm to assign requests to prompt and token machines. This approach minimizes the overall response time by selecting machines with the shortest pending queue. Additionally, the CLS is responsible for routing requests to the mixed pool when the queues for both prompt and token machines are excessively long. The mixed pool is designed to ensure that the performance of the system does not degrade under high load conditions.



(a) Serialized KV-cache transfer. (b) Optimized KV-cache transfer per-layer during prompt phase.

Fig. 10: Optimizing KV-cache transfer in Splitwise.

3) Machine-Level Scheduling for Token and Prompt Processing: On each individual machine, the machine-level scheduler (MLS) is responsible for managing the local queue, tracking memory utilization, and deciding how to batch incoming requests. For prompt machines, MLS employs a first-come-first-serve (FCFS) strategy to schedule requests. To ensure optimal performance, the MLS restricts the batching of multiple prompt requests to a total of 2048 tokens, beyond which throughput begins to degrade.

For token machines, the MLS also follows FCFS scheduling, but it dynamically adjusts the batch size based on available memory. As the machine's memory capacity is reached, the MLS starts queuing tokens to avoid memory overload. The MLS ensures that the memory is utilized efficiently and minimizes any idle time on the token machines.

4) Optimizing KV-Cache Transfer for Reduced Latency: The transfer of the KV-cache from the prompt machine to the token machine is a significant overhead in `Splitwise`. This transfer delay can be especially pronounced for larger prompts, as the time required for transfer increases with the size of the KV-cache and the bandwidth between the machines. To address this issue, `Splitwise` optimizes the KV-cache transfer by overlapping it with the ongoing computations in the prompt phase.

5) System Overview: Splitwise: Fig. 10 shows the process of KV-cache transfer in both serialized and asynchronous modes. In the serialized approach, the KV-cache transfer starts only after the prompt phase has completed, introducing a significant delay before the token generation phase can begin. In contrast, the asynchronous approach allows the KV-cache to be transferred in parallel with the ongoing computation in the prompt phase. As each layer in the model is computed, the corresponding KV-cache for that layer is transferred asynchronously to the token machine, significantly reducing the overall transfer time and allowing for the token generation phase to begin earlier.

6) Layer-Wise KV-Cache Transfer: Balancing Performance and Latency: To further optimize KV-cache transfer, `Splitwise` introduces a layer-wise transfer strategy. In this approach, the KV-cache corresponding to each layer is transferred immediately after that layer is computed, rather than waiting for the entire prompt phase to finish. This method allows for continuous transfer of the KV-cache while the next layer of the prompt computation is being processed.

As layer-wise KV-cache transfer works in parallel with prompt computation, this optimization reduces the time spent

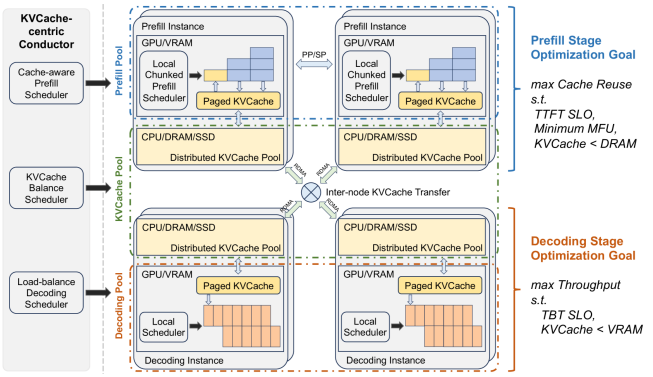


Fig. 11: Mooncake Architecture.

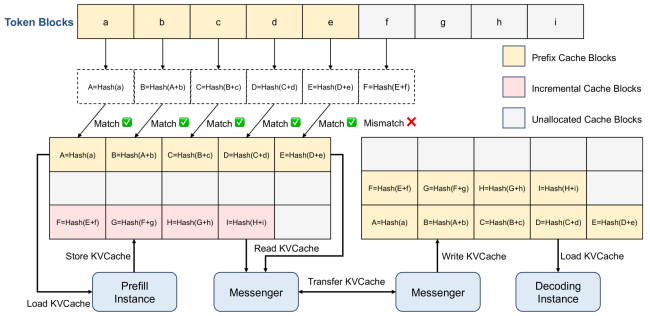


Fig. 12: The KVCache pool in CPU memory. Each block is attached with a hash value determined by both its own hash and its prefix for deduplication.

on KV-cache transfer and improves the overall throughput of the system. However, it may introduce some performance interference, particularly for smaller prompts, where the transfer overhead is less significant. To mitigate this, `Splitwise` dynamically selects between serialized and layer-wise transfer based on the size of the prompt, ensuring that the transfer method is best suited to the request size.

Overall, the combination of these optimizations—dynamic scheduling, efficient KV-cache transfer, and parallelism—enables `Splitwise` to achieve both high throughput and low latency for large language model inference.

C. MoonCake

Despite from the previous two works, `MoonCake` [25] outperforms with a solid code base and real tests in an industrial cluster.

1) Disaggregated Architecture and KVCache Management: MoonCake adopts a disaggregated architecture (showed in Fig. 11)that separates the prefill and decoding nodes while integrating CPU, DRAM, SSD, and RDMA resources within the GPU cluster. This architecture enables the implementation of a disaggregated KVCache, which efficiently utilizes underutilized resources, offering substantial cache capacity and transfer bandwidth. The KVCache is stored as paged blocks in CPU memory, and these blocks can be managed using cache eviction algorithms, including Least Recently Used (LRU) and Least Frequently Used (LFU), based on the request patterns. The transfer of KVCache blocks across CPU and GPU nodes is managed by a dedicated RDMA-based component, named Messenger, which facilitates high-speed data movement. Additionally, MoonCake provides a context caching API to enable better KVCache reuse.

2) Conductor: The Global Scheduler: Central to MoonCake's operation is the Conductor, a global scheduler that dispatches requests based on the current distribution of KVCache and node workloads. Conductor is responsible for replicating or swapping KVCache blocks to improve future inference performance. As illustrated in Fig. 12, the typical workflow begins after tokenization is completed, with Conductor selecting prefill and decoding nodes. The workflow includes the following steps: KVCache Reuse, Incremental Prefill, KVCache Transfer, and Decoding. Each step is optimized to minimize latency and ensure that the Tail-to-Tail Firing Time (TTFT) Service Level Objective (SLO) is met.

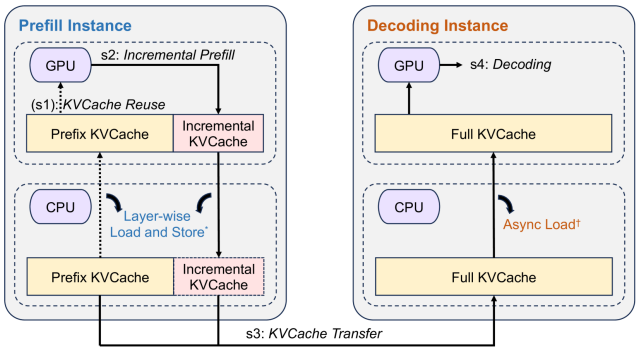
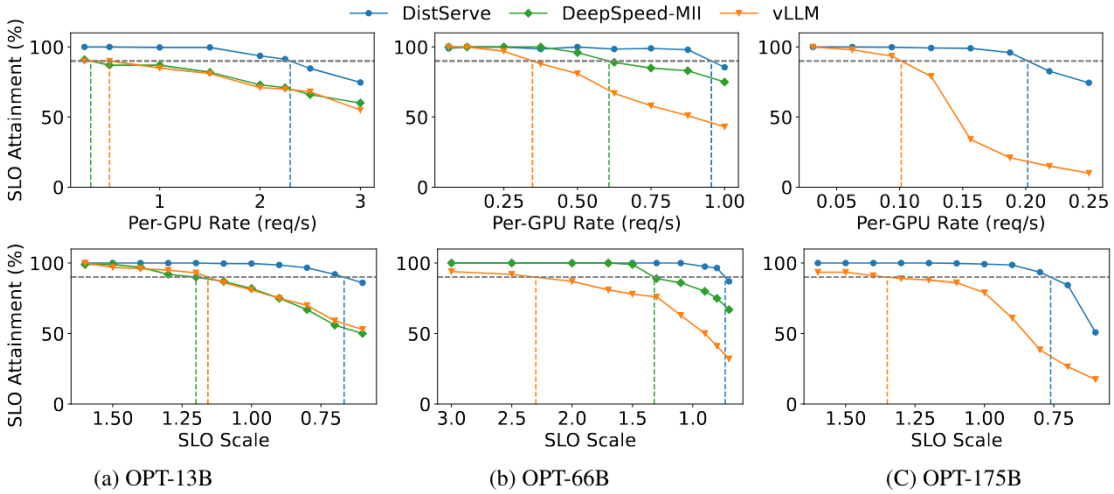


Fig. 13: Workflow of inference instances. For prefill instances, the load and store operations of the KVCache layer are performed layer-by-layer and in parallel with the prefill computation to mitigate transmission overhead (see §5.2). (†) For decoding instances, asynchronous loading is performed concurrently with GPU decoding to prevent GPU idle time.

3) KVCache Reuse and Incremental Prefill: The KVCache Reuse step involves loading relevant cached data into GPU memory, ensuring that as much KVCache as possible is reused while balancing node workloads and meeting TTFT SLO. In the Incremental Prefill step, the prefill node completes the prefix caching process and stores the newly generated KVCache back into CPU memory. For large inputs, the prefill process is chunked and executed in parallel, with each chunk processed across different nodes to fully utilize GPU computational resources.

4) KVCache Transfer with Messenger: MoonCake uses the Messenger service to manage and transfer KVCache blocks across nodes. Messenger is an independent process running within each inference instance, ensuring high-speed, asynchronous KVCache transfer. This transfer is overlapped with the incremental prefill to minimize waiting time, enhancing overall system performance.

5) Decoding Node Selection and Execution: After the KVCache is transferred to the decoding node, Conductor preselects a node based on current load and batch scheduling. This step is critical for meeting TTFT SLOs. However, the local scheduler re-evaluates this selection to ensure the node's



(a) OPT-13B (b) OPT-66B (C) OPT-175B

Fig. 14: Chatbot application with OPT models on the ShareGPT dataset.

load has not changed during the prefill stage, which might lead to request rejection if the SLO cannot be met.

6) Maintaining a Disaggregated Prefill Pool: While some studies suggest the benefits of a more integrated architecture, MoonCake maintains its disaggregated architecture for prefill operations. This decision is driven by the need to handle long-context requests efficiently and save VRAM by keeping prefill nodes separate from decoding nodes. Prefill is inlined into the decoding batch only when it meets TTFT requirements without chunking.

7) Multi-Node Prefill for Long Contexts: Given the rapid increase in the available context length for LLMs, MoonCake employs multi-node prefill to process long-context requests. Recent studies have proposed sequence parallelism (SP) to accelerate long-context processing. However, MoonCake utilizes chunked pipeline parallelism (CPP), which minimizes the need for cross-node communication and reduces the overhead of frequent dynamic adjustments to node partitioning. CPP allows MoonCake to process multiple chunks of a request simultaneously across different nodes, thus reducing TTFT and improving resource utilization.

8) Layer-wise Prefill and VRAM Optimization: MoonCake also optimizes VRAM usage by employing a layer-wise prefill strategy. In this approach, KVCache loading and storing are asynchronous, overlapping with computation to minimize VRAM occupation. By using this method, MoonCake can handle larger context lengths without significant performance degradation. The layer-wise prefill method, shown in Fig. 13, reduces latency for long-context requests by optimizing the transfer and computation overlap.

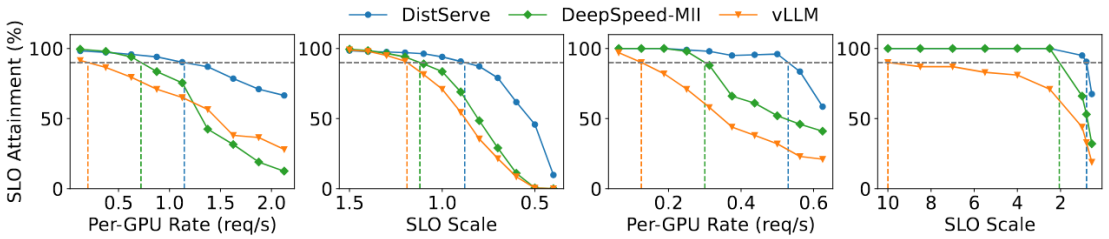
9) KVCache-Centric Scheduling: MoonCake's scheduler, Conductor, places significant emphasis on KVCache-centric scheduling. This section primarily discusses how Conductor handles scheduling under normal conditions, with overload scenarios addressed in subsequent sections. The KVCache-centric scheduling allows for more efficient resource utilization, balancing cache management with workload distribution, ensuring that the TTFT SLO is met even under varying system loads.

V. Performance Evaluation

In this section, we will address the experiments done in each work and do a performance evaluation based on the results.

A. DistServe

They evaluate the performance of DistServe on the chatbot application for all three OPT models. The first row of Fig. 14 illustrates that when they gradually increase the rate, more requests will violate the latency requirements and the SLO attainment decreases. The vertical line shows the maximum per-GPU rate the system can handle to meet latency requirements for over 90% of the requests. On the ShareGPT dataset, DistServe can sustain 2.0×4.6× higher request rate compared to vLLM. This is because DistLLM eliminates the prefill-decoding interference through disaggregation. Two phases can optimize their own objectives by allocating different resources and employing tailored parallelism strategies. Specifically, by analyzing the chosen placement strategy5 for 175B, they find the prefill instance has inter-op = 3, intra-op = 3; and the decoding instance has inter-op = 3, intra-op = 4. Under this placement, DistServe can effectively balance the load between the two instances on ShareGPT, meeting latency requirements at the lowest cost. This non-trivial placement strategy is challenging to manually find, proving the effectiveness of the algorithm. In the case of vLLM, collocating prefill and decoding greatly slows down the decoding phase, thereby significantly increasing TPOT. Due to the stringent TPOT requirements of chatbot applications, although vLLM meets the TTFT SLO for most requests, the overall SLO attainment is dragged down by a large number of requests that violate the TPOT SLO. Compared to DeepSpeed-MII, DistServe can sustain 1.6×–7.4× higher request rate. DeepSpeed-MII shows better performance on larger models because the prefill job is larger and chunkedprefill mitigates the interference to some extent. However, due to the reasons discussed in §2.3, chunked



(a) Code Completion (b) Summarization

Fig. 15: Code completion and summarization tasks with OPT-66B on HumanEval and LongBench datasets, respectively.

prefill is slower than full prefill, so it struggles to meet the TTFT SLO as a sacrifice for better TPOT.

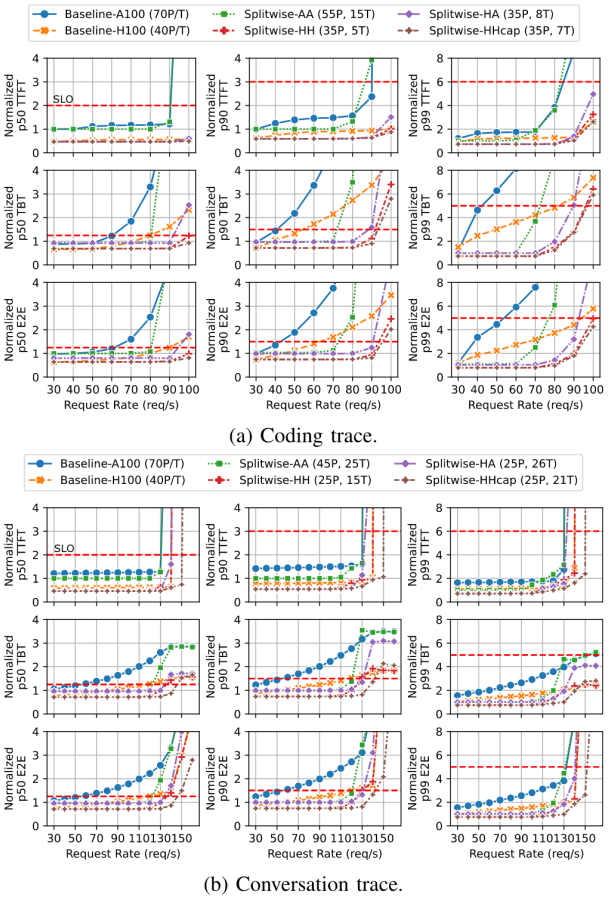
The second row of Fig. 14 indicates the robustness to the changing latency requirements of the two systems. They fix the rate and then linearly scale the two latency requirements in Table 1 simultaneously using a parameter called SLO Scale. As SLO Scale decreases, the latency requirement is more stringent. They aim to observe the most stringent SLO Scale that the system can withstand while still achieving the attainment target. Fig. 14 shows that DistServe can achieve 1.8×–3.2× more stringent SLO than vLLM and 1.7×–1.8× more stringent SLO than DeepSpeed-MII, thus providing more engaging service quality to the users.

Fig. 15(a) shows the performance of DistServe on the code completion task when serving OPT66B. DistServe can sustain 5.7× higher request rate and 1.4× more stringent SLO than vLLM. Compared to DeepSpeedMII, DistServe can sustain 1.6× higher request rate and 1.4× more stringent SLO. As a real-time coding assistant, the code completion task demands lower TTFT than chatbot, this leads to both systems ultimately being constrained by the TTFT requirement. However, in comparison, by eliminating the interference of the decoding jobs and automatically increasing intra-operation parallelism in prefill instances through the searching algorithm, DistServe reduces the average latency of the prefill jobs, thereby meeting the TTFT requirements of more requests.

Fig. 15(b) shows the performance of DistServe on the summarization task when serving OPT-66B. DistServe achieves 4.3× higher request rate and 12.6× more stringent SLO than vLLM. Compared to DeepSpeed-MII, DistServe achieves 1.8× higher request rate and 2.6× more stringent SLO. The requests sampled from LongBench dataset have long input lengths, which brings significant pressure to the prefill computation. However, due to the loose requirement of TTFT for the summarization task, the TPOT service quality becomes particularly important. Since vLLM collocates prefill and decoding phases, it experiences a greater slowdown in the decoding phase with long prefill jobs and fails to meet the TPOT requirement.

B. *Splitwise*

For Splitwise cluster designs under the coding trace, Splitwise-AA provisions 55 prompt machines and 15 for the token pool, denoted as (55P, 15P). Note that like BaselineA100, Splitwise-AA also provisions 75% more machines



(a) Coding trace.

(b) Conversation trace.

Fig. 16: Latency metrics across input loads for iso-power throughput optimized clusters. Dashed red lines indicate SLO.

than Baseline-H100. The legends in Fig. 16 show the different provisioning choices under coding and conversation workloads. Request size distributions reflect in the machine pool sizing. For example, we provision more prompt machines under SplitwiseHH (35P, 5T) for the coding trace, while we provision more token machines (25P, 15T) for the conversation trace. Latency and throughput. Fig. 16 shows a deep dive into all the latency metrics at different input load for each cluster design with the same power (i.e., iso-power). For the coding trace (Fig. 16a), Splitwise-HH, Splitwise-HHcap, and Splitwise-AA all perform better than Baseline-H100. As

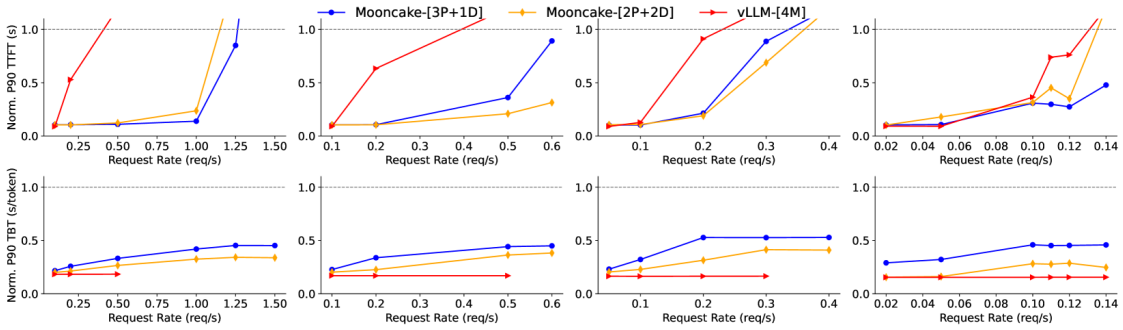


Fig. 17: End-to-end experiments of Mooncake and vLLM on simulated data.

the load increases, Baseline-H100 suffers from high TBT due to mixed batching with large prompt sizes. Although SplitwiseAA can support higher throughput, its TTFT is consistently higher than most designs. Splitwise-HA clearly bridges the gap by providing low TTFT and E2E at high throughput. The mixed machine pool in Splitwise becomes useful at higher loads to use all the available hardware without fragmentation. This benefit can be seen clearly in the P50 TBT chart for Splitwise-HA, where after 90 RPS, H100 machines jump into the mixed machine pool and help reduce TBT. For the conversation trace (Fig. 16b), Splitwise-HHcap clearly does better on all fronts, including latency. This is because its token generation phases typically run for much longer than in the coding trace, which is beneficial for the token machines.

C. Mooncake

Public datasets evaluates the performance of Mooncake and vLLM in end-to-end tests on public datasets using ArXiv Summarization and L-Eval. They 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 Fig. 17, 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.

VI. Conclusion

Prefill and decoding disaggregation enhances resource utilization and satisfies SLO at a lower cost. Prefill and decoding disaggregation decouple resource allocation and parallelism strategies such that optimization can be tailored for P and D separately. After disaggregation, it turns into a conventional distributed system problem, thus we use scheduling, batching, and caching to further optimize.

Acknowledgments

This should be a simple paragraph before the References to thank those individuals and institutions who have supported your work on this article.

References

[1] *GPT-4 Technical Report*. OpenAI, 2023. CoRR abs/2303.08774. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774. arXiv:2303.08774.

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. *Llama 2: Open foundation and fine-tuned chat models*. arXiv preprint arXiv:2307.09288, 2023. [Online]. Available: https://arxiv.org/abs/2307.09288.

[3] U. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, et al. *OPT: Open pre-trained transformer language models*. arXiv preprint arXiv:2205.01068, 2022. [Online]. Available: https://arxiv.org/abs/2205.01068.

[4] BigScience Workshop, T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, et al. *Bloom: A 176b-parameter open-access multilingual language model*. arXiv preprint arXiv:2211.05100, 2022. [Online]. Available: https://arxiv.org/abs/2211.05100.

[5] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. *Mistral 7B*. arXiv preprint arXiv:2310.06825, 2023. [Online]. Available: https://arxiv.org/abs/2310.06825.

[6] DeciAI Research Team. *DeciLM 6B*. 2023. [Online]. Available: https://huggingface.co/Deci/DeciLM-6b.

[7] Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, Fan Yang, et al. *Baichuan 2: Open large-scale language models*. 2023. arXiv preprint arXiv:2309.10305. [Online]. Available: https://arxiv.org/abs/2309.10305.

[8] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. *GLM-130B: An open bilingual pre-trained model*. 2022. arXiv preprint arXiv:2210.02414. [Online]. Available: https://arxiv.org/abs/2210.02414.

[9] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. *Bytetransformer: A high-performance transformer boosted for variable-length inputs*. In 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 344–355, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10109249.

[10] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. *Zeroquant: Efficient and affordable post-training quantization for large-scale transformers*. Advances in Neural Information Processing Systems 35, 27168–27183, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2022/hash/8c8d8a13d27934a3962b422da74f3033-Abstract.html.

[11] 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, 611–626, 2023. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3572780.3574049.

[12] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. arXiv preprint arXiv:1909.08053, 2019. [Online]. Available: https://arxiv.org/abs/1909.08053.

[13] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. *Memory-Efficient Pipeline Parallel DNN Training*. In *International Conference on Machine Learning*, PMLR, 7937–7946, 2021. [Online]. Available: https://proceedings.mlr.press/v139/narayanan21a.html.

[14] Hao Liu, Matei Zaharia, and Pieter Abbeel. *Ring Attention with Blockwise Transformers for Near-Infinite Context*. arXiv preprint arXiv:2310.01889, 2023. [Online]. Available: https://arxiv.org/abs/2310.01889.

[15] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. *Calculon: A Methodology and Tool for High-Level Co-Design of Systems and Large Language Models*. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–14, 2023.

[16] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. *AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving*. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 663–679, 2023.

[17] *FlexFlow-Serve*. 2023. Available: https://github.com/Flexflow/FlexFlow/tree/inference. Commit: 672cdad, Accessed on: 2023-11-25.

[18] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "SpotServe: Serving Generative Large Language Models on Preemptible Instances," in *Proceedings of the ASPLOS Conference*, 2024.

[19] RDMA Consortium, "Architectural Specifications for RDMA over TCP/IP," [Online]. Available: http://www.rdmaconsortium.org.

[20] G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier, "The development of Mellanox/NVIDIA GPUDirect over InfiniBand—a new model for GPU to GPU communications," *Computer Science-Research and Development*, vol. 26, pp. 267–273, 2011.

[21] Infiniband Trade Association, "Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16," pp. 1–17, 2010.

[22] A. C. Elster and T. A. Haugdahl, "Nvidia Hopper GPU and Grace CPU Highlights," *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.

[23] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving," *arXiv preprint arXiv:2401.09670*, 2024.

[24] P. Patel, E. Choukse, C. Zhang, A. Shah, I. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024.

[25] F. Asch, *Mooncake*, Simon and Schuster, 2014.