WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling

Jingqi Feng
Fudan University
Shanghai, Shanghai, China
jqfeng23@m.fudan.edu.cn

Yukai Huang
Fudan University
Shanghai, Shanghai, China
ykhuang20@fudan.edu.cn

Rui Zhang
Fudan University
Shanghai, Shanghai, China
zhangrui21@m.fudan.edu.cn

Sicheng Liang
Fudan University
Shanghai, Shanghai, China
scliang23@m.fudan.edu.cn

Ming Yan
Fudan University
Shanghai, Shanghai, China
myan@fudan.edu.cn

Jie Wu*
Fudan University
Shanghai, Shanghai, China
jwu@fudan.edu.cn

Abstract

Existing large language model (LLM) serving systems typically batch the compute-bound prefill and I/O-bound decoding phases together. This co-location approach not only leads to significant interference between the two phases but also limits resource allocation and placements. To address these limitations, recent work proposes disaggregating the prefill and decoding phases to enhance performance. However, these works often rely on coarse-grained static scheduling strategies, resulting in imbalanced and insufficient resource utilization. For instance, compute resources for the prefill phase may be overloaded while those for the decoding phase remain idle, resulting in performance bottlenecks.

In this paper, we propose **WindServe**, an efficient phase disaggregated LLM serving system that leverages stream-based, fine-grained dynamic scheduling to enhance resource utilization and performance. WindServe features a global scheduler that monitors compute and memory resource usage to dynamically orchestrate cross-phase jobs, effectively reducing queuing delay and KV cache swapping overhead. We also introduce a stall-free rescheduling strategy to saturate the memory resources while minimizing the scheduling overhead from KV cache transfers. Furthermore, we design a stream-based approach to mitigate interference between prefill and decoding jobs. Our evaluation demonstrates that WindServe achieves remarkable stability and SLO attainment under high-load scenarios, outperforming state-of-the-art phase-disaggregated LLM serving systems by delivering a 4.28× improvement in TTFT median latency and a 1.5× reduction in TPOT P99 latency.

CCS Concepts

• **Computing methodologies** → **Distributed artificial intelligence**; **Cooperation and coordination**.

*Corresponding author

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3730999

Keywords

Distributed LLM Inference, GPU Sharing, Scheduling Optimization

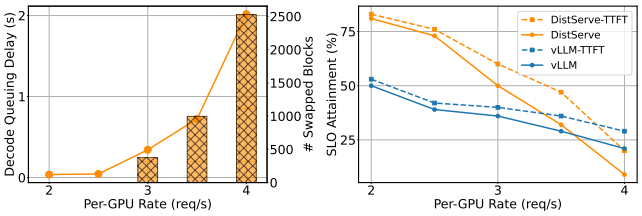
ACM Reference Format:
Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3695053.3730999

1 Introduction

Large Language Models (LLMs), such as GPT [6], GLM [11] and LLaMA [38], have gained considerable attention due to their impressive natural language processing capabilities. However, deploying LLMs demands substantial hardware acceleration, resulting in high operational costs. Specifically, the cost of processing LLM requests can be up to 10× greater than traditional keyword queries. Therefore, improving LLM inference efficiency is essential to reducing these costs in large-scale deployments. To achieve improved efficiency, many advances [1, 2, 13, 18, 29, 42, 45] have been proposed in recent years. Despite these advancements, efficient LLM inference remains a significant challenge.

In transformer-based [40] LLMs, an inference request consists of two phases: the *prefill* phase and the *decoding* phase. During the prefill phase, all prompt tokens are processed to generate the first output token, which is typically compute-intensive. Following this, the generally I/O-bound decoding phase begins and decodes iteratively until a special end-of-sequence (EOS) token is generated. Consequently, the performance of LLM inference is usually measured by two main metrics: time to first token (TTFT) , associated with the prefill phase, and time per output token (TPOT) , related to the decoding phase. TTFT measures the time from request issuance to the generation of the first output token, encompassing both queuing delays and prompt processing times. TPOT reflects the average time required to produce each subsequent token in the response sequence, excluding the first token. It is influenced by three factors: the decoding queuing delay, the elapsed time of the decoding phase, and the length of the output tokens.

Previous LLM serving systems either combine prefill and decoding phases of different requests into a hybrid batch [18, 42] or interleave multiple iterations of decoding jobs with prefill jobs [7].



(a) Decode queuing and swapping. **(b) SLO attainment.**

Figure 1: TPOT and TTFT degrade under high workloads when serving OPT-13B on the NVIDIA 80GB A800. DistServe [45] is phase-disaggregated (2-way tensor parallelism for both prefilling and decoding), while vLLM [18] is co-located.

However, prefill and decoding phases have distinct characteristics, making it challenging for these conventional approaches to optimize both TTFT and TPOT simultaneously. Concretely, a prefill-only jobs can be significantly longer, often dozens of times, than a decode-only job. As a result, batching prefill and decode jobs together frequently causes severe *prefill-decode* interference, leading to substantial Service Level Objective (SLO) violations. To address this issue, recent researches [13, 29, 32, 45] propose a novel ***Phase-Disaggregated (PD) architecture***, which splits the prefill and decoding phase into separate serving instances, transferring the KV cache generated by the prefill instance to the decoding instance. By decoupling resource allocation between these phases, this architecture also allows service providers to configure each instance independently according to workload demands.

However, our observations in Figure 1 indicate that the PD-architecture LLM serving system exhibits suboptimal performance under high request loads, even achieving a lower SLO attainment rate compared to co-located serving system. As the request rate increases, the limitations of LLM serving systems become more pronounced, particularly their significant underutilization of system resources. Firstly, existing PD architectures do not retain KV cache tensors in the prefill instance, thus, all active KV cache is stored in the decode instance, which leads to inefficient use of GPU global memory. As evidenced in Figure 1a, memory resources are not fully utilized, resulting in high decode queuing delays and frequent KV cache swapping between GPU global memory and CPU memory—a primary contributor to degraded TPOT latency. On the other hand, in practice, the decoding instances often requires redundant GPU resources to meet stringent TPOT SLOs. However, decoding jobs typically do not fully saturate the available computational capacity, resulting in resource underutilization and poor TTFT performance at high request rates, as shown in Figure 1b. These findings highlight the necessity of fine-grained dynamic scheduling mechanisms in PD-architecture systems to achieve balanced resource utilization. Nevertheless, implementing such mechanisms introduces three critical challenges:

Challenge 1: Overload Detection Mechanism Design & Scheduling Policy Optimization. Dynamic scheduling necessitates multi-dimensional overload detection mechanisms capable of adapting to heterogeneous workload patterns (e.g., variable prompt/output lengths) and distinct phase characteristics (prefill vs. decoding). Furthermore, designing efficient scheduling policies requires balancing conflicting objectives: minimizing TTFT queuing delays while preventing cascading interference that could impair TPOT performance. Suboptimal scheduling could intensify workload imbalance and induce cascading performance degradation.

Challenge 2: How to alleviate the overhead associated with the scheduling process? Dynamic scheduling involves additional overheads such as data transfers. For example, maximizing GPU memory utilization requires transferring KV cache from decoding instance to prefill instance. This transfer can block the inference process, potentially leading to increased latency and reduced system performance. Careful data flow management is essential to balance the benefits of dynamic scheduling against the risks of disrupting ongoing jobs.

Challenge 3: How to minimize cross-phase interference when jobs co-locate? To fully utilize resources and alleviate bottlenecks under high workloads, prefill and decoding jobs are necessary to co-locate within the same instance, leading to inevitable perfill-decode interference. Minimizing this interference and maintaining efficient executions is essential to improve system performance.

To address these challenges, we present WindServe, a latency-oriented LLM serving system based on the PD architecture. WindServe mitigates the inherent KV cache transfer overhead by overlapping transfers with prefill computations. It integrates a *Global Scheduler* that establishes specific overload detection indicators for different instances to generate precise and effective dynamic scheduling strategies, including *Dynamic Prefill Dispatch* and *Dynamic Rescheduling*. WindServe also implements a stall-free rescheduling method to reduce the rescheduling overhead. Finally, it employs *Stream-based Disaggregation* to mitigate the strong prefill-decode interference that occurs during dynamic scheduling under high workloads. Specifically, *Stream-based Disaggregation* allows decoding jobs and a few prefill jobs to run concurrently in separate CUDA streams within the decoding instance, thus minimizing cross-phase interference.

We implement a prototype of WindServe, which supports various LLM series, ensuring versatility across a wide range of applications. We evaluate WindServe with the OPT [43] and LLaMA2 [39] models on realistic datasets [3, 36]. Compared to state-of-the-art solutions, DistServe [45], our experimental results show that WindServe achieves up to 1.65-4.28× reduction in TTFT median latency and it can also reduce the TPOT P99 latency by 1.5×. Moreover, due to its bottleneck-aware capabilities, WindServe performs competitively across diverse workloads. In general, this paper makes the following contributions.

- Reveal resource underutilization and imbalance in the Phase-Disaggregated architecture and propose fine-grained dynamic scheduling to address these issues.
- Design and implement WindServe with dynamic scheduling strategy. Furthermore, minimize the overhead associated with dynamic scheduling via *Stall-free Rescheduling* and

Stream-based Disaggregation.

- Conduct a comprehensive evaluation of WindServe, demonstrating its significant improvement in latency.

2 Background and Motivation

2.1 LLM Inference

Decoder-only LLMs (such as LLaMA [38], GPT [6] and OPT [43]) are composed of multiple Transformer [40] layers and demonstrate exceptional performance. When processing requests, these LLMs must forward through all these layers during both prefill phases and the decoding phases. In addition, due to the self-attention mechanisms, the last generated token, along with the key (**K**) and value (**V**) tensors of all previously generated tokens and the prompt tokens, are required as inputs during the decoding phase. Advanced LLM serving systems [7, 18, 42, 45] cache the key and value tensors, as known as KV cache [30], to avoid recomputation by trading storage for efficiency. Furthermore, various optimizations are commonly employed in LLM serving systems:

Batching. Batch processing is a widely adopted method to improve the efficiency in LLM inference. Each forward pass involves substantial computations that require transferring weights and intermediate variables from the GPU's high-bandwidth memory (HBM) to on-chip static random-access memory (SRAM), resulting in significant I/O overhead. LLM serving systems typically utilize batch processing to distribute this overhead across multiple requests. However, due to the unpredictable input and output lengths of the requests, naive batch processing will lead to serious barrel effects. Advanced LLM inference engines [18, 26, 42, 45] implement *continuous batching*, allowing requests to exit the batch processing as soon as they complete, while immediately joining ongoing batch processing when new requests arrive. In original vLLM [18] implementation, prefill and decoding jobs are batched together, increasing throughput but introducing prefill-decoding interference. However, in PD-architecture serving systems such as DistServe [45], prefill and decoding jobs are distributed to different serving instances to avoid interference, with KV cache transferring between instances.

Memory Optimization. In LLM serving systems, the KV cache could occupy a substantial amount of GPU global memory. Furthermore, as the output length of requests is hard to predict, previous LLM inference engines [7, 31, 42] reserve a large block of GPU memory equal to the maximum context length for each request. This results in significant memory fragmentation and reduced efficiency in batch processing. To address this, vLLM [18] introduces a novel approach known as *PagedAttention*, which mitigates internal fragmentation and improves the throughput of LLM inference. PagedAttention dynamically allocates KV cache tensors in blocks as the cache grows, eliminating the requirement to pre-allocate a large space for the KV cache. This method allows for a more flexible and efficient use of GPU memory. By integrating PagedAttention into WindServe, we aim to optimize GPU memory utilization, ensuring the LLM serving engine works efficiently.

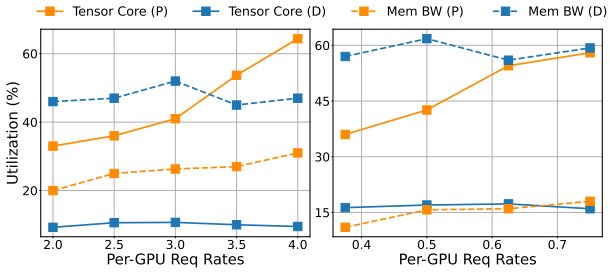


Figure 2: Mean resource utilization of prefill instances and decoding instances. *Tensor Core*(P) **denotes tensor core utilization in prefill instance, while** *Mem BW*(D) **represents memory bandwidth utilization during decoding. The left panel corresponds to OPT-13B, and the right panel depicts OPT-66B.**

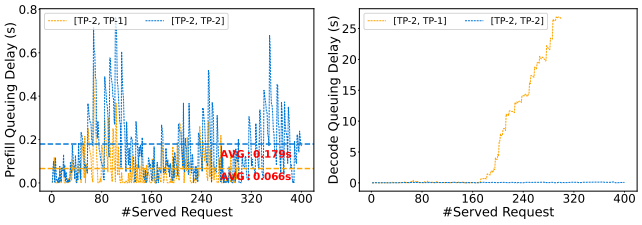
2.2 Phase-Disaggregated Architecture

High-throughput LLM serving systems [18, 42] typically integrate prefill and decoding jobs into hybrid batches, leading to interference between prefill and decoding jobs. Emerging PD-architecture LLM Serving systems [13, 29, 32, 45] address this interference through phase disaggregation. Moreover, this architecture offers an another advantage of enabling the decoupling of placement strategies and scales of prefill instances and decoding instances. However, these systems often struggle to maintain a high level of service quality when confronted with increased workload demands. The main issues are as follows:

KV cache migration overhead. Taking OPT-13B [43] as an example, for a request with 2048 tokens (the maximum supported context length), the KV cache to be transferred is approximately 1.5 GB when both the prefill and decoding instances are deployed on a single GPU respectively. This transmission overhead could be near-zero for devices with GPU high-speed interconnects (i.e., NVLink), but significant for PCIe devices. For instance, when GPUs are interconnected via PCIe Gen4 ×16 at 32 GB/s within the same NUMA node, transferring the KV cache for a single request may take ~65 ms when peer-to-peer (P2P) is enabled, which could be several times the duration of a single decoding iteration. Furthermore, the size of the KV cache to be transferred increases with average input length and arrival rate, and this transmission overhead is further magnified during inter-node transfers.

Lack of inter-instance coordination. Limited collaboration between serving instances prevents the PD-architecture LLM serving systems from effectively identifying performance bottlenecks. While DistServe [45] implements instance orchestration, including managing placement and instance sizing, it may still suffer from suboptimal resource utilization due to inflexible scheduling under diverse workloads. Each instance in the serving system requires a dynamic scheduling strategy that adapts to the actual workload. Without such adaptability, a bottleneck in any single instance can significantly degrade overall quality of service.

Insufficient and uneven resource utilization. Insufficient resource utilization manifests in two key aspects, computing and



(a) Prefill queuing delay (b) Decoding queuing delay

Figure 3: Queuing delays when serving an LLM with 13B parameters under Sharegpt [36] datasets at per-gpu rate = 4 req/s with different configures. [TP-2, TP-1] represents that the placement for the prefill instance's strategy is 2-way tensor parallelism without pipeline parallelism, and the decoding instance's is 1-way tensor parallelism without pipeline parallelism. For notational convenience, the placement strategies in the following section are represented according to this rule.

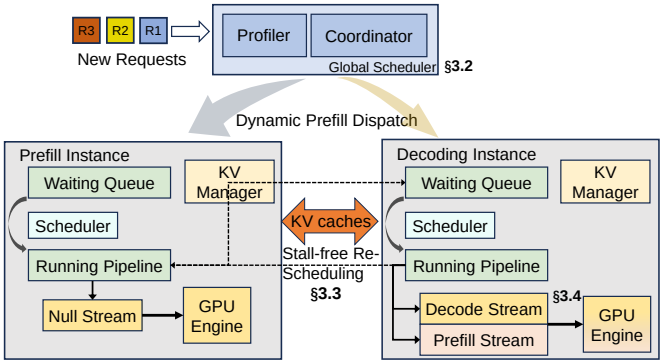


Figure 4: Overview of WindServe

memory resources. First, computing resources are often underutilized. As shown in Figure 2 and highlighted in recent studies [14, 15, 20, 41], I/O-bound AI/ML workloads often fail to fully utilize GPU computing resources, with decoding jobs being a typical I/O-bound workload. In scenarios with stringent TPOT SLOs, such as chatbot, it may be necessary to allocate additional GPUs to decoding instance for efficiency. However, this exacerbates the underutilization of computational resources. Even worse, in some cases, significant idle computing resources may exist in decoding instances while prefill instances suffer from insufficient computational capacity, resulting in increased queuing delay. In addition to computing resources, memory resources also demonstrate significant underutilization. The prefill instance is typically dedicated to prefill jobs, and once the KV cache transfer is completed, the KV cache is no longer retained. Consequently, the KV cache for all decoding jobs is stored in the decoding instance, meaning that the memory resources of the serving cluster are not fully utilized. This could lead to the overloading of the decoding instances, resulting in recomputations and excessive swapping of KV cache blocks between GPU and CPU memory.

We attribute the underutilizations and observed suboptimal performance to the coarse-grained GPU-based allocation method and the inflexible cross-instance scheduling strategy. Our evaluations follow a **linear scaling rule**, focusing on how service quality changes with *per-GPU Request Rate*, rather than *Total Request Rate*. Following the linear scaling rule, allocating more GPUs to prefill instances accelerates prefill computations, resulting in increased KV cache storage in decoding instances. This may lead to increased recomputation or even swapping of the KV cache. In contrast, if decoding instances are allocated relatively redundant resources, inadequate resource utilization can significantly increase the queuing delay of prefill jobs. As depicted in Figure 3, when the prefill instance is assigned to 2 GPUs and the decoding instance to 1 GPU ([TP-2, TP-1]), the prefill is redundant and the decoding instance limits the TPOT latency. When both instances are set to 2 GPUs ([TP-2, TP-2]), the decoding instance is redundant, and the prefill queuing delays increase. Although DistServe [45] suggests replanning the allocation strategy when the request pattern shifts significantly, the associated replanning overhead introduces non-negligible stagnation, rendering this approach suboptimal.

3 WindServe Design

In this paper, we present WindServe, which addresses the challenges (outlined in §2.2) in existing PD architecture through cost-efficient dynamic scheduling.

3.1 Overview

As shown in Figure 4, to efficiently coordinate pre-fill and decoding jobs for better utilization of resources, we introduce a *Global Scheduler* (§3.2). The *Global Scheduler* is composed of two components, a *Profiler* and a *Coordinator*. Based on the analysis and modeling of computational and I/O overheads in the LLM inference, we design the *Profiler* (§3.2.1). The *Profiler* characterizes the computational capability of each instance and predicts the completion time of a batch, enabling optimal scheduling. The *Coordinator* (§3.2.2) is responsible for monitoring the resource utilization and workload of each instance and collaborating with the *Profiler* to perform two dynamic scheduling strategies: ① *Dynamic Prefill Dispatch*: When a new request arrives, if the prefill instance is overloaded, then the *Coordinator* decides whether to dispatch its prefill tasks to the decode instance; ② *Dynamic Rescheduling*: if the decode instance is overloaded, WindServe reschedules some decoding requests to the prefill instance. Each instance of WindServe features a local scheduler responsible for scheduling requests from the waiting queue into the running pipeline following a *First-Come-First-Serve* (FCFS) order. Once requests complete the prefill phase in prefill instance and finish their KV transfer, they are pushed into the waiting queue of the decoding instance. Similar to previous advanced LLM serving systems [1, 18, 45], WindServe divides the KV cache space into blocks, and sets up a KV manager in each instance for KV block management. In addition, to avoid the KV cache transfer overhead from rescheduling, a *Stall-free Migration* method (§3.3) is introduced in WindServe that does not block decoding iterations during KV cache transfers. Finally, we propose *Stream-Based Disaggregation* (§3.4) technology to separate prefill and decoding jobs

Table 1: Per-layer overhead analysis of Attention and FFN. Take OPT family (FP16) for example. B, H, N **and** $\sum L$ **represent batch size, hidden size, the number of prefill input tokens and sum of context lengths respectively.**

Module	FLOPs		IO bytes	
	Prefill	Decode	Prefill	Decode
Attn	$8NH^2 + 4N^2H$	$8BH^2 + 4\sum LH$	$8H^2$	$8H^2 + 4\sum LH$
FFN	$16NH^2$	$16BH^2$	$16H^2$	$16H^2$

into different streams for asynchronous computations, reducing caused prefill-decode interference.

3.2 Global Scheduler

3.2.1 ***Profiler.*** We implement a *Profiler* in *Global Scheduler* to characterize the computing capabilities of the LLM serving system and to predict the time required for each iteration of the computation, allowing optimal dispatch of prefill jobs. Researches [1, 44] indicate that the Self-Attention mechanism and the FFN (Feed-Forward Network) are two of the most computationally intensive and time consuming modules in transformer-based LLMs. We list the FLOPs (floating-point operations) and IO overhead associated with these two modules in the Table 1. For example, in FFN, the two matrices multiplied at the first layer have the shapes $B \times H$ and $H \times 4H$, and the Matrix-Vector multiplication needs one multiply-add (2 FLOPs) per matrix element, so the FLOPs for the first layer are $B \times H \times 4H \times 2$.

For the prefill phase, its Attention and FFN module are both typically compute-bound. Therefore, prefill time-consumption is mainly limited by FLOPs ($8NH^2 + 4N^2H + 16NH^2$). Consequently, the estimated time cost for the prefill phase could be expressed as equation 1. While for the decoding phase, its Attention and FFN modules are both usually I/O-bound and the total IO overhead is $24H^2 + 4\sum LH$, proving that the time consumption of the decoding computation is mainly related to the total context length, as expressed in equation 2.

$$\hat{T}_{prefill} = a_p N + b_p N^2 + c_p \tag{1}$$

$$\hat{T}_{decode} = a_d \sum L + c_d \tag{2}$$

Based on the above analysis, the *Profiler* takes the number of prefill tokens and the total length of the decoding context as input to estimate the time taken for one batch of computations with a quadratic function, whose parameters are obtained by profiling and quadratic regression before runtime. It is worth noting that, due to certain optimizations in the attention mechanism, the attention elapsed time during the prefill phase is more **linearly** related to N.

3.2.2 ***Coordinator.*** The *Coordinator* collaborates with the *Profiler* for cross-instance dynamic scheduling. *Dynamic Prefill Dispatch* is triggered when the prefill instance is overloaded, while *Dynamic Rescheduling* is activated when decoding instance overwhelms.

Dynamic Prefill Dispatch. WindServe proposes *Dynamic Prefill Dispatch* to solve the high TTFT latency caused by overload of prefill instances. Algorithm 1 shows in detail how the *Coordinator* performs *Dynamic Prefill Dispatch* based on the overall workload.

Algorithm 1 Algorithm of Dynamic Prefill Dispatch

Input: Profiler P, Coordinator C, currently prefilling Batch B_p, prefill instance waiting queue Q_p, new request R_{new}, decoding instance KV block manager BM_d and decoding instance running pipeline $pipeline_d$

1: $TTFT_{pred} \leftarrow P$.PredictTime(B_p, Q_p, R_{new})
2: **if** *assistRequests* is $\emptyset$ **then**
3: $slots \leftarrow C$.CalculateAvailableSlots($pipeline_d, BM_d$)
4: **end if**
5: **if** $TTFT_{pred} > thrd$ **then**
6: **if** $slots \geq R_{new}.length$ **then**
7: *assistRequests*.insert(R_{new})
8: $slots \leftarrow slots - R_{new}.length$
9: **else**
10: Q_p.enqueue(R_{new})
11: **end if**
12: **else**
13: Q_p.enqueue(R_{new})
14: **end if**
15: **if** decoding instance is ready **then**
16: Add *assistRequests* to $pipeline_d$
17: *assistRequests* $\leftarrow \emptyset$
18: **end if**

When a new request arrives, the *Global Scheduler* decides on which instance to execute the prefill jobs of the request based on the prefill instance's waiting queue and the decoding instance's running pipeline. Initially, the *Profiler* estimates the prefill completion time ($TTFT_{pred}$) of the the new request if it is performed in the prefill instance. Indeed, we input the cumulative prefill tokens count for all requests in the waiting queue along with the new request to the Profiler. This input yields an estimated time, denoted as $TTFT^{-}_{pred}$. We then add the anticipatedremaining time ffor thethe currently prefilling batch to this estimate to derive tfinalal $TTFT_{pred}$. The $TTFT_{pred}$ is based on the number of tokens rather than the number of requests, thus its a more precise flag for determining if the prefill instance is overloaded. Further, if this flag exceeds a threshold ($thrd$), the decoding instance would be checked for available slots, which represents the number of prefill tokens that the decoding instance could help with. If sufficient slots are available, the prefill computation of a new request would be dispatched to the decoding instance.

The *thrd* setting presents a trade-off: a smaller threshold generally triggers dynamic prefill dispatch more frequently, resulting in lower TTFT but higher TPOT. However, if set too low, it can overwhelm the decoding instance with excessive prefill jobs, increasing TTFT due to queuing delay, as illustrated in Figure 5. Therefore, to avoid this problem and minimize the impact on TPOT, we set the threshold slightly below the TTFT SLO.

In line 3 of the algorithm 1, the *Coordinator* calculates the slots to assist with prefill jobs based on the running pipeline and KV block manager of the decoding instance. We establish a *budget* for assisting prefill jobs in the decoding instance, limiting the maximum number of prefill tokens that do not exceed the TPOT SLO in a single forward pass. WindServe determines the *budget* through

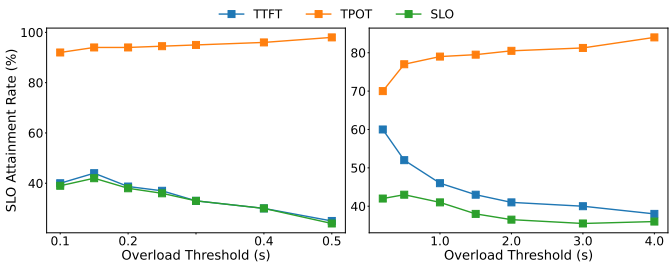


Figure 5: The impact of different threshold settings on the SLO attainment rate is shown on the left for the OPT-13B model using the Sharegpt [36] dataset at 4 per-GPU req/s, alongside results from the LLaMA2-13B with the Longbench [3] dataset at 1.5 per-GPU req/s.

simulation and profiling before runtime and identifies available slots during runtime based on the running pipeline of the decoding instance as calculated by the *Profiler*. Notably, if the KV blocks in the decoding instance are inadequate, the available slot is set to 0.

If some prefill jobs are executed in the decoding instance, the decoding instance would temporarily perform *Stream-based Disaggregation* to alleviate prefill-decode interference. Concretely, when pre-fill and decoding jobs coexist in the decoding instance, the local scheduler of the decoding instance puts them into one hybrid batch as does vLLM [18], but separates the computations into different CUDA streams during execution on GPU engine. We introduce *Stream-based Disaggregation* technology in detail in §3.4.

Dynamic Rescheduling. As mentioned in §2.2, the PD-architecture serving system suffers from underutilization of GPU memory. However, being able to utilize the GPU memory of the prefill instance while maintaining efficient decoding is challenging. One potential solution involves storing the KV cache from specific layers within the prefill instance and leveraging high-bandwidth GPU interconnection technologies such as NVLink to facilitate the swapping of KV caches between the prefill and decoding instances. Clearly, this approach is feasible due to the sequential ordering between layers. However, it should be noted that although this guarantees relatively high performance in most advanced devices such as the GB200 NVL72 [27], even the high bandwidth NVLink of the GB200 NVL72 is not sufficient to achieve fast swapping without compromising performance. Alternatively, we could consider migrating some KV cache to the prefill instance when the decoding instance runs low on KV cache blocks, allowing decoding jobs to proceed within the prefill instance. However, this method introduces migration overhead and can lead to prefill-decode interference in the prefill instance.

In comparison to the first method, the second method is more practical. The first method necessitates careful scheduling of KV cache transfers, and frequent swapping of KV cache across instances can disrupt the inherent KV cache transfer between prefill and decoding instances, resulting in unpredictable consequences, not to mention that existing GPU interconnect technologies cannot support such rapid transfers. Consequently, WindServe adopts the second approach to solve the memory underutilization problem by rescheduling certain requests to prefill instances when the KV

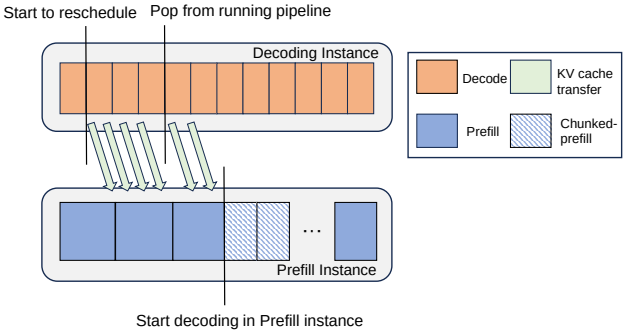


Figure 6: Stall-free Dynamic Rescheduling in WindServe

blocks of decoding instances are nearly exhausted. This strategy focuses on rescheduling long-context requests to free up more KV cache in the decoding instances.

3.3 Stall-free Rescheduling

To minimize the rescheduling overhead associated with KV cache transfers, we propose a *stall-free* request migration method, as illustrated in Figure 6. When the KV cache blocks in the decoding instance are nearing exhaustion, *Dynamic Rescheduling* is activated, prompting WindServe to transfer the KV cache of certain long-context requests to the prefill instance. During this transfer, these migrating requests continue their decoding iterations and generate new KV cache in the decoding instance without blocking. Once the remaining KV cache to be transferred falls below a certain threshold, the decoding instance pauses decoding for that request, and the Prefill instance resumes their decoding phases after the KV cache transfer is fully completed.

Additionally, WindServe bounds prefill-decode interference via *chunked-prefill* [1] in prefill instance. Specifically, if there are decoding jobs in the prefill instance, the prefill jobs in it would be converted to chunked-prefill fashion to avoid strong prefill-decode interference. The idea of *Stall-free Rescheduling* is similar to the multi-stage migration proposed by Llumnix [33], both aiming to prevent KV cache migration from blocking request decoding. However, Llumnix tends to migrate short-context requests to reduce migration overhead and fragmentation, while WindServe tends to migrate longer sequences in order to free up more space and decrease prefill-decode interference. Moreover, to minimize migration overheads, the prefill instance dynamically backs up the KV cache of some long-context requests when there is sufficient KV blocks and relatively limited KV blocks in decoding instance. These backups can reduce migration costs when the backed-up requests are later rescheduled to prefill instance.

3.4 Stream-based Disaggregation

Dynamic Prefill Dispatch results in the coexistence of prefill and decoding jobs within the decoding instance. Fortunately, the shared mechanisms in GPUs make it possible to mitigate the associating prefill-decode interference. On modern NVIDIA GPUs (Kepler architecture and later), Hyper-Q technique [5] supports 32 independent hardware working queues, enabling concurrent execution of CUDA kernels. Concurrent execution of CUDA kernels allows multiple

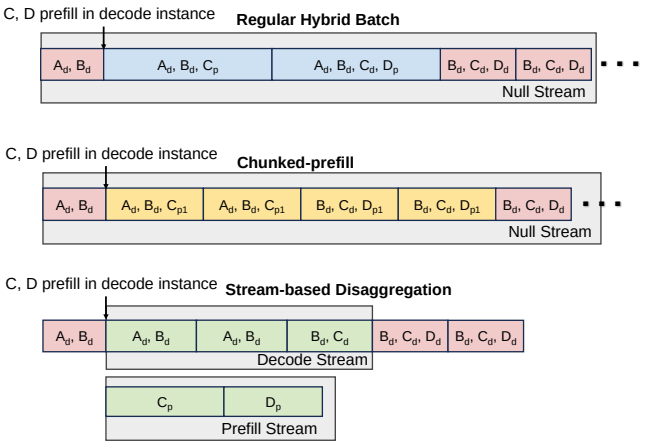


Figure 7: Comparison of chunked-prefill and Stream-based disaggregation. Subscript d represents a decoding iteration, p represents a prefill. A, B, C and D represent different requests. The figure shows that Stream-based disaggregation is friendly to both TTFT and TPOT, while chunked-prefill slows down prefill considerably.

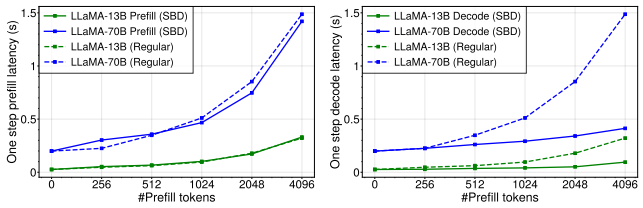


Figure 8: Comparison of single forward pass prefill and decoding time cost using regular batching (Regular) and stream-based disaggregation (SBD). We batch 16 decoding requests (context length = 2048) and different numbers of prefill tokens into one hybrid batch.

CPU threads or processes to simultaneously share the resources in a GPU, mitigating performance loss and resource waste.

NVIDIA provides five technologies to enhance GPUs utilization: *Streams*, *MPS* (Multi-Process Service), *Time-Slicing*, *MIG* (Multi-Instance GPU), and *vGPU* [16]. While *Time-Slicing*, *MIG* and *vGPU* technologies provide improved isolation between partitions, they face challenges such as inflexible partitioning, difficulty in dynamic reconfiguration, and limited inter-partition communication. Specifically, *MIG* has restricted partition specifications and disables P2P communication between GPUs, hindering distributed LLM inference. *Time-Slicing* is not suitable for latency-sensitive tasks, and *vGPU* lacks support for inter-partition communication and runtime reconfiguration. Although *MPS* offers greater flexibility and enables cross-partition operations via IPC (Inter-Process Communication), it can only be reconfigured at the launch of multiple processes.

In contrast, the *Stream-based* approach that we propose in this paper is well-suited for dynamic scheduling during service runtime due to its inherent flexibility. A CUDA stream is a software abstraction that represents a sequence of commands, which may include computation kernels, memory copies, and other operations that execute sequentially [16], potentially overlapping with operations in other streams. There are two types of streams in CUDA programming: **synchronous** streams and **asynchronous** streams. Synchronous streams block the execution of host-end (CPU) code, while asynchronous streams do not. By default, GPU operations execute in the *NULL stream*, a special synchronous stream. Additionally, asynchronous streams can be categorized as **blocking** and **non-blocking** streams. GPU executions in blocking streams are blocked by those in the *NULL stream*, whereas non-blocking streams continue execution independently. Since GPU resources (SM and Cache, etc.) are directly shared between different streams, the stream-based approach has poor isolation, but its flexibility and ease of management motivate us to propose it as a fine-grained scheduling strategy. For the other methods, they may lead to performance loss when only one type of jobs is executed. This happens because, in these methods, GPU resources can only be allocated separately for prefill and decode jobs prior to runtime. Consequently, when only one type of jobs is executed, it can only utilize the resources allocated for that specific job, leading to underutilization.

In this context, we propose *Stream-based Disaggregation* to alleviate the prefill-decode interference caused by dynamic prefill dispatch in the decoding instance. From an architectural perspective, Stream-based Disaggregation overcomes the inherent limitations of coarse-grained resource allocation, enhancing performance under high workloads. Figure 7 shows the timeline of proposed *Stream-based Disaggregation*. When prefill and decoding jobs coexist in the decoding instance and may interfere with each other significantly, WindServe separates them into different ***CUDA blocking streams***, and returns only the decoding result during computation, then fetches and synchronizes the prefill computation when it is about to complete (predicted by the profiler).

We recognize that limited GPU resources could reduce decoding efficiency when split computations into different streams. This is because different streams are directly sharing the whole GPU resources. For example, while kernel A is executing, kernel B can only start if any SM on the GPU has enough resources to execute a thread block in kernel B. However, compared with regular batch strategies or chunked-prefill, *Stream-based disaggregation* can effectively alleviate the interference of the prefill computation on the decoding computation while maintaining the efficiency of the prefill computation . Our following analysis of *Stream-based Disaggregation* demonstrates its advantages in detail.

Stream-Based Disaggregation Analysis. To better demonstrate the advantages of *Stream-based Disaggregation*, we evaluate this technology using multiple parameter size LLMs. Figure 8 indicates that *Stream-based Disaggregation* effectively mitigates prefill-decode interference. Compared to chunked-prefill [1], *Stream-based Disaggregation* provides benefits for both TPOT and TTFT. While chunked-prefill primarily exchanges increased prefill time for reduced TPOT, the data presented in Figure 8 reveal the limitations of this approach. Consider LLaMA2-70B with a 2048-token prefill job as a case study. Under the chunked-prefill approach with a chunk size of 512, the estimated prefill duration is approximately 1.4 s, which is four times the single-step decoding cost of 0.35 s. In contrast, employing *Stream-based Disaggregation* yields a prefill cost of

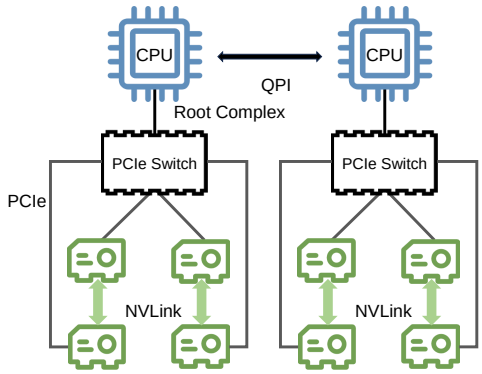


Figure 9: A basic schematic of topology in our testbed. Our machine has two NUMA nodes, with the GPUs within one NUMA connected via NVLink bridge or PCIe switch, and the cross-NUMA node going through the RC (Root Complex).

around 0.75 s, with each decoding iteration taking approximately 0.34 s. While reducing the chunk size may decrease single-step decoding costs, it furthur increases the prefill cost.

It is important to note that we do not adopt *Stream-based Disaggregation* in the Prefill instance, despite its significant advantages. If *Stream-based Disaggregation* is implemented in prefill instance, its scheduling policy would be highly dependent on the *Profiler*'s predicted completion time, leading to a lack of robustness. In the decoding instance, however, there are only a few prefill jobs that do not rely heavily on predictions of the *Profiler*.

4 Implementation

We implement WindServe on top of the open source implementation of DistServe [35, 45], with the aim of enhancing serving performance via flexible and cost-efficient cross-instance scheduling. The front-end, including algorithm module, global scheduler and asynchronous migration of WindServe are implemented with ~1.4K lines of Python code. Global Scheduler manages dynamic prefill dispatch and dynamic rescheduling to coordinate the workload across the serving system. And we made necessary extensions to SwiftTransformer [37] (as backend of WindServe), including asynchronous KV caches transfer, stream-based disaggregation and concurrent control, which were implemented in ~2K lines of C++/CUDA.

WindServe is an end-to-end distributed LLM inference system that uses NCCL [8] to implement communication in tensor parallelism and pipeline parallelism. When WindServe triggers *Stream-based Disaggregation*, each stream uses a separate NCCL communicator to avoid synchronization and blocking. In our implementation, each serving instance is backed by a parallel inference engine that employs Ray [25] actors to establish GPU workers. These workers are responsible for executing the LLM inference tasks and managing the KV Cache (organized as blocks) in a distributed fashion. In addition, in existing inference engine [45], GPU memories are required to be allocated to store intermediate variables such as input tokens and other projection buffers, which will implicitly synchronize the CUDA streams at runtime. To avoid such synchronizations, we allocate enough GPU memory to store them when initializing the inference engine and design a naive memory management mechanism, which could also avoid the overhead of repeated GPU memory allocation.

Table 2: Datasets used to evaluate WindServe

Dataset	Prompt Tokens			Output Tokens		
	Avg	Median	P90	Avg	Median	P90
ShareGPT [36]	768.2	695	1556	195.9	87	518
LongBench [3]	2890.4	2887	3792	97.4	12	369

5 Evaluaion

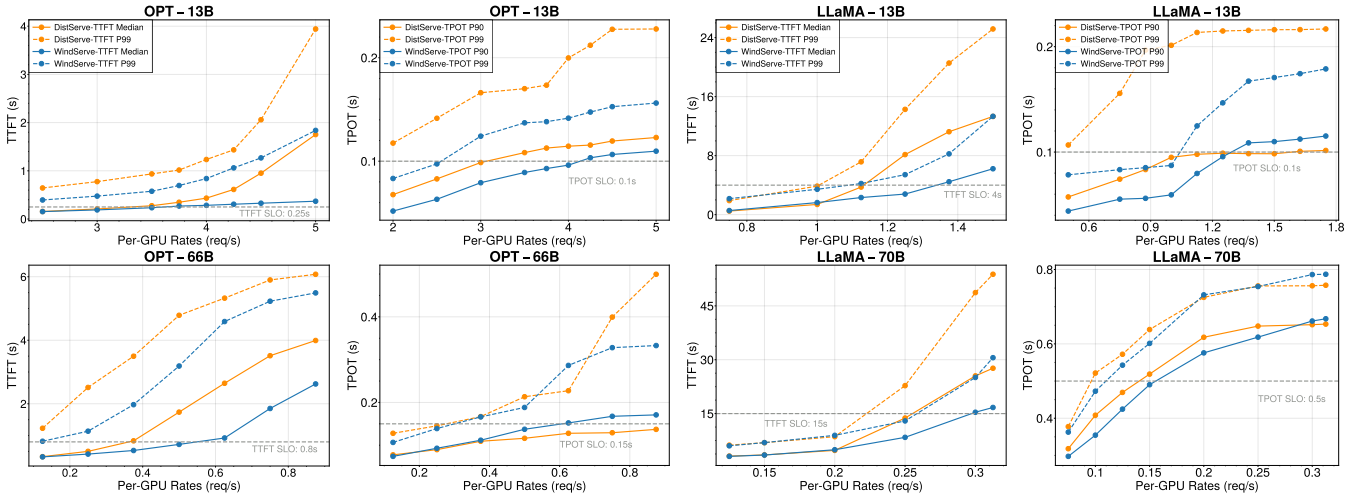
To more comprehensively evaluate the performance of WindServe, we evaluate WindServe with multiple LLMs of different parameter sizes and multi-scenario datasets, including chatbot [23] and summarization [3]. We choose ***DistServe*** [45] as one of the baselines, which is considered to be the *state-of-the-art* system under the Phase-Disaggregated architectures. Another baseline in our evaluation is ***vLLM*** [18], a highly active open-source LLM serving engine. The version utilized for comparison is v0.4.2. During the experiments, we enabled vLLM's *chunked-prefill* feature and implemented its recommended placement.

5.1 Experiments Setup

Testbed. We deployed WindServe in a node consisting of 8 NVIDIA PCIe A800-80 GB GPUs. One GPU is connected to another GPU via NVLink 400 GB/s (bidirectional) via NVLink bridge, and connected to other GPUs via PCIe Gen4 64 GB/s (bidirectional). In other words, our testbed only interconnects two by two via NVLink as shown in Figure 9. Moreover, the CPU is 2.90 GHz Intel Xeon Gold 6326 with 768 GB DRAM.

Models. We select the OPT [43] and LLaMA2 [39] series models as serving LLMs for evaluating WindServe because they are widely recognized representative families in the academic community. Both the OPT and LLaMA2 families include models with varying parameter sizes, ranging from 125M to 175B for OPT models. For our experiments, we choose the LLaMA2 model to evaluate WindServe's performance in the Summarization task, given its capability to handle longer contexts, with a token limit of 4K compared to OPT's 2K. Furthermore, we utilize the OPT-13B and OPT-66B models for evaluation in Chatbot scenarios. All model parameters used in our experiments are set to FP16 precision.

Workloads. We evaluated WindServe's capabilities using a variety of workloads across different scenarios. The specific dataset statistics used for the evaluation are listed in Table 2. To simulate the chatbot application scenario, we utilized the ***ShareGPT*** [36] dataset, which contains conversations between users and ChatGPT. This dataset is notable for its extensive range of input and output lengths, making it one of the most comprehensive available. In the summarization application—a popular LLM inference task—the prompts are typically longer while the outputs are shorter, aligning with the request distribution found in the ***Longbench*** [3] dataset.



(a) TTFT on ShareGPT dataset. **(b) TPOT on ShareGPT dataset.** **(c) TTFT on Longbench dataset.** **(d) TPOT on Longbench dataset.**

Figure 10: End-to-End performance of WindServe. 10a and 10b show WindServe's performance on the ShareGPT dataset; 10c and 10d show WindServe's performance on the Longbench dataset.

Table 3: Placement strategies (TP stands for tensor parallelism, while PP stands for pipeline parallelism)

Model	Prefill Placement	Decode Placement
OPT-13B	TP-2, PP-1	TP-2, PP-1
LLaMA2-13B	TP-2, PP-1	TP-2, PP-1
OPT-66B	TP-2, PP-2	TP-2, PP-2
LLaMA2-70B	TP-2, PP-2	TP-2, PP-2

Table 4: SLOs for different model size and scenarios

Model	Attention	TTFT	TPOT	Dataset
LLaMA2-13B	MHA	4s	0.1s	LongBench [3]
LLaMA2-70B	GQA	15s	0.5s	LongBench [3]
OPT-13B	MHA	0.25s	0.1s	ShareGPT [36]
OPT-66B	MHA	0.8s	0.15s	ShareGPT [36]

Additionally, during experiments, we employed a Poisson distribution to simulate the specified request rate and recorded timestamps at each stage for further analysis.

Placement Strategies. Based on the model, latency requirements and SLO attainment targets, DistServe [45] determines the placement of prefill and decoding instances by simulation. WindServe adopts the same method to establish its parallellism strategy, with the placement for our experiments listed in Table 3. However, as discussed in §2.2, the allocation of resources at GPU granularity is coarse-grained. In contrast, WindServe enables flexible scheduling at runtime, which we will explore further in §5.3.

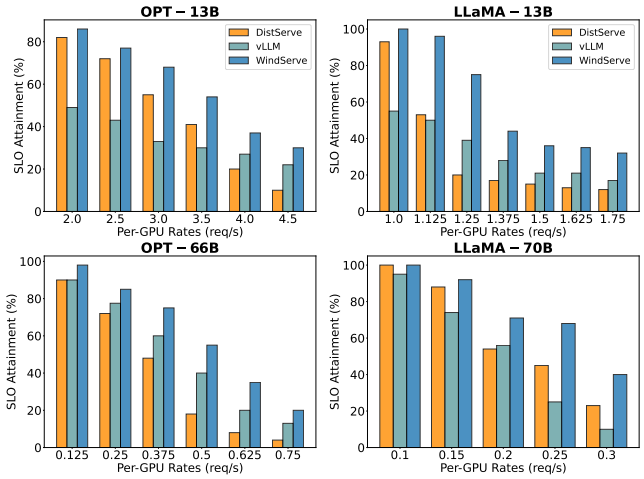
Metrics. TTFT and TPOT are two key metrics for evaluating quality of service in LLM inference. We first utilize the median (P50) and the 99th percentile (P99) of TTFT to provide a comprehensive assessment performance in terms of TTFT. For TPOT, we focus on the P90 and P99 latencies as key performance indicators. Additionally, we emphasize the *SLO attainment rate*, defined as the percentage of requests meeting both TTFT and TPOT SLOs, as a reflection of overall quality of service.

5.2 End-to-End Performance

We evaluate WindServe using the datasets listed in Table 2 across various LLMs (OPT and LLaMA2 models) and different request rates to demonstrate its serving capabilities under diverse workloads. The TPOT SLOs are established in a similar way as existing systems [1, 29]. specifically, we set the TPOT SLOs to be equal to ~4× the execution time of a decoding iteration for a request (with a context length equal to the average number of tokens in the dataset and a batch size of 16) running without prefill interference. We also empirically define varying TTFT SLOs for different scenarios. The absolute SLOs established in our evaluation are summarized in Table 4.

Chatbot. We evaluate the service performance of WindServe on OPT series models in a Chatbot application. The experimental results are shown in Figures 10a and 10b, with the upper row showing the results on the OPT-13B and the lower row on the OPT-66B.

The results indicate that WindServe can reduce the median TTFT latency to 4.28× and the TTFT P99 latency by 2.1× when serving OPT-13B, while also decreasing the TPOT P99 latency by 1.5× at high request rates. The significant improvement in the median TTFT is primarily attributed to *Dynamic Prefill Dispatch* and the wide distribution of input lengths of ShareGPT [36] dataset. *Dynamic Prefill Dispatch* enables the LLM serving system to fully utilize computational resources, unlike DistServe, which suffers from excessive prefill queuing delay. Additionally, the diverse prompt lengths in the ShareGPT dataset enhance WindServe's median latency performance. The improvement in TPOT latency is facilitated



(a) SLO on ShareGPT dataset. (b) SLO on Longbench dataset.

Figure 11: SLO attainment of WindServe. 11a show WindServe's performance on the ShareGPT dataset; 11b show WindServe's performance on the Longbench dataset.

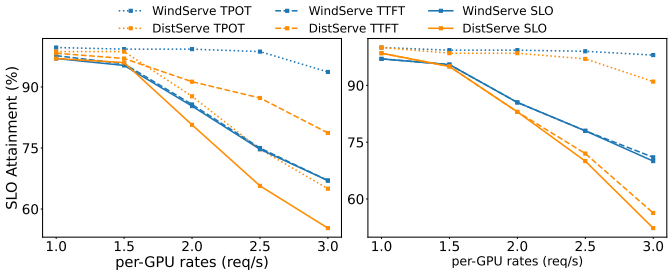


Figure 12: SLO attainment rates for different resource allocations when serving OPT-13B with Sharegpt [36] dataset. The left subgraph is configured as [TP-2, TP-1] and the right as [TP-2, TP-2].

by WindServe's overlapping KV cache transfers and prefill computations. As anticipated, *Stream-based Disaggregation* alleviates prefill interference on TPOT caused by *Dynamic Prefill Dispatch*. Notably, the increase in TPOT P99 latency for DistServe at high request rates is due to the transmission overhead of the KV cache and longer decoding queuing delay.

We also evaluated WindServe's serving capabilities on the OPT-66B, which has a larger parameter size. For these evaluations, WindServe employs the [TP-2, PP-2, TP-2, PP-2] placement, identified as the optimal strategy in our evaluation system. The experimental results are shown in the Figure 10a and 10b (bottom). In the ShareGPT dataset, WindServe improves the latency of TPOT P99 by 1.52× and reduces the median latency of TTFT by 1.54× compared to DistServe. However, these enhancements come with a slight increase in TPOT P90 latency, which is due to decreased decoding efficiency associated with *Stream-based Disaggregation*. During this evaluation, DistServe experiences a surge in TPOT P99 latency due to swapping in/out of KV cache at high request rates, while WindServe mitigates this via *Dynamic Rescheduling*.

Additionally, the SLO attainment rates in Figure 11a also demonstrate that WindServe outperforms both DistServe [45] and vLLM [18], achieving a higher SLO attainment rate, despite vLLM enabling the *Chunked-prefill* technology. Overall, WindServe surpasses both DistServe and vLLM in TTFT and TPOT latency in this application, showcasing superior stability and SLO attainment rates under high load conditions. This indicates that WindServe is more efficient and reliable for processing concurrent requests.

Summarization. In the summarization application, we evaluate the performance of WindServe using LLaMA2 [39] series models on the Longbench [3] dataset. This is because LLaMA2 models are capable of supporting a context length of up to 4K, while the OPT models have a maximum context length of only 2K. The experimental results, presented in Figures 10c and 10d, show that WindServe significantly outperforms DistServe, reducing the median latency of TTFT by 1.65-2.1× and P99 latency by 1.55-1.76×, with minimal impact on TPOT.

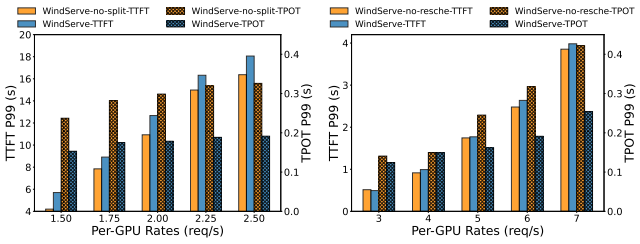
As expected, WindServe demonstrates superior performance in TPOT under relatively low workloads, as shown in Figure 10d. This performance advantage is attributed to the fact that WindServe asynchronously transfers KV cache during the prefill phase. However, asynchronous migration also causes WindServe to experience a slight increase in TTFT latency here compared to DistServe. Nevertheless, WindServe could successfully achieve TTFT SLOs, as indicated by the SLO attainment rate shown in Figure 11b. Further, as request rate increases, WindServe exhibits enhanced TTFT performance, attributed to its *Dynamic Prefill Dispatch*.

The asynchronous KV cache transfer provides significant advantages in the TPOT comparison of LLaMA2-13b (Figure 10d upper), where long context requests increase the KV transfer overhead via PCIe. However, this advantage is less pronounced for LLaMA20-70b (Figure 10d down), which utilizes Group Query Attention (GQA) instead of the Multi-Head Attention mechanism used in other evaluated models. The implementation of GQA reduces the size of the KV cache tensors, thereby decreasing the transmission overhead of the KV cache.

As request rates increase, TPOT performance declines due to the overhead of *Stream-based Disaggregation*. Nevertheless, the overhead remains acceptable and does not significantly hinder performance. As shown in Figure 11b, WindServe outperforms baseline systems in SLO attainment rates, also demonstrating that the impact of *Stream-based Disaggregation* on TPOT latency is acceptable. Specifically, our evaluation shows that WindServe can improve SLO attainment by at least 1.5× at high request rates.

5.3 Bottleneck-aware Ability

Insufficient dynamic coordination between prefill and decoding instances can result in significant TTFT and TPOT SLO violations, negatively impacting overall quality of service. In contrast, WindServe is bottleneck-aware; when TTFT limits quality of service (as identified by the *Profiler* monitoring the waiting queue), it dynamically dispatches certain prefill jobs to relieve pressure on the prefill instance. Conversely, when TPOT becomes a bottleneck, i.e., the



(a) P99 Latency in *WindServe-no-split* on Longbench dataset **(b) P99 Latency in *WindServe-no-resche* on ShareGPT dataset**

Figure 13: Ablation Studies

KV cache blocks in the decoding instance are nearly exhausted, WindServe reschedules certain long-context requests from the decoding instance to the prefill instance, freeing up KV cache blocks in the decoding instance and decoding these requests stall-freely.

Figure 12 illustrates WindServe's capabilities in adaptively addressing specific performance challenges. In the left subfigure, the SLO attainment rate for DistServe is mainly limited by TPOT, which WindServe mitigates through *Dynamic Rescheduling*. In contrast, as evidenced by the right subfigure, TTFT serves as the main bottleneck for DistServe, while WindServe saturates the decoding instance and enhances performance via *Dynamic Prefill Dispatch*. In more extreme cases, such as when the resources of the prefill instance are significantly smaller than those of the decoding instance, the *avaiblable slots* in Algorithm 1 for the decoding instance is automatically large, depending on the configuration of the instance and SLOs.

5.4 Ablation Studies

In this section, we mainly study the effects of the proposed *Stream-based Disaggregation* and *Dynamic Rescheduling* technologies on the performance of our LLM serving system. We use ***WindServe-no-split*** to denote WindServe without *Stream-Based Disaggregation*, and ***WindServe-no-resche*** to represent WindServe without *Dynamic Rescheduling*. We compare the performance of these two variants with WindServe while serving the OPT-13B. The experimental results indicate that these two techniques significantly enhance the performance of the LLM serving system.

As shown in the Figure 13a, our ablation study on the Longbench [3] dataset reveals that *Stream-Based Disaggregation* effectively mitigates the prefill-decode interference caused by *Dynamic Prefill Dispatch*. Additionally, the comparison in the Figure 13b indicates that *Dynamic Rescheduling* also effectively reduces TPOT latency. This improvement is primarily due to the technique's ability to alleviate decode queuing delay and to minimize unnecessary KV cache block swapping I/O. Notably, both technologies have minimal impact on TTFT. For *Stream-based Disaggregation*, the impact arises from the reduced CUDA kernel parallelism due to limited resources. Meanwhile, *Dynamic Rescheduling* causes temporary chunked-prefill in the prefill instance, which moderately increases TTFT.

6 Related Work

LLM serving systems. In the domain of LLM serving systems, there is a significant body of advancing work focused on enhancing the efficiency and performance of large language model inference [7, 18, 24, 26, 28, 29, 31, 33, 34, 42, 45]. Orca [42] schedules the execution of the engine at the iteration level, enabling batching of operations for better hardware utilization. vLLM [18] is a system designed to improve the throughput and efficiency of large language model inference. It introduces Paged-Attention to manage the memory footprint of key-value caches efficiently. SARATHI [1] aims to improve LLM inference efficiency with chunked-prefills, which helps to saturate GPU compute resources and reduce pipeline bubbles in distributed inference. FlexFlow-Serve [24] supports speculative decoding and tree-based parallel decoding kernels for accelerating inference. Llumnix [33] could reacts to such heterogeneous and unpredictable requests by runtime rescheduling across multiple model instances. In contrast to the above work, DistServe [45], Splitwise [29], DéjàVu [32] and TetriInfer [13] propose to disaggregate the prefill and decoding jobs to different instances, and in this paper we introduced and addressed some of the limitations of this architecture.

Attention Optimizations. The attention module is crucial for LLM inference, making its optimization a key focus in inference enhancement. FlashAttention [10] is an IO-aware attention algorithm that employs tiling to minimize memory reads/writes between GPU high bandwidth memory (HBM) and on-chip SRAM. FlashAttention-2 [9] improves FlashAttention with better work partitioning for better performance. And we incorporate FlashAttention-2 into WindServe for prefill phase. A concurrent work, POD-Attention [15] is a GPU kernel that improves the efficiency of attention computation for hybrid batches. It optimizes compute and memory bandwidth by effectively allocating GPU resources, enabling concurrent prefill and decoding jobs on the same multiprocessor.

In addition, recent efforts [4, 12, 17, 19, 21, 22, 44] have utilized sparse attention to improve the efficiency of long context request inference. However, we believe that these approximations lack competitive accuracy in LLM inference. For example, Longformer [4] constructs the sparse attention using a fixed-size sliding window on the most recent local tokens. ALISA [44] emphasizes tokens crucial for generating new tokens using a Sparse Window Attention algorithm, which enhances sparsity in attention layers and minimizes KV caching memory usage with minimal accuracy loss.

7 Discussion

Limitations. Firstly, due to constraints in the experimental environment, we were unable to evaluate our WindServe in a multi-node setting. It is anticipated that inter-node communication would necessitate additional tuning efforts. Obviously, it is essential to avoid inter-node paths in tensor-parallel collective communication due to their high overhead and potential unreliability. Cross-node GPUs typically communicate via GDR (GPU Direct RDMA) technique, whereas GDR cannot be finely controlled in NCCL. Therefore, additional communication code needs to be implemented to enable fine-grained control of the communication. Secondly, although stream-based disaggregation allows for the asynchronous execution of prefill and decoding jobs to reduce interference, the

independent execution of kernels doubles the model's I/O overhead. Finally, the GPU sharing based on streams remains coarse-grained, and the transparent nature of the CTA scheduler somewhat hinders the higher performance of stream-based disaggregation.

Future Work. We advocate for the strategic allocation of GPU resources based on specific workloads. Heterogeneous GPU clusters are increasingly prevalent in computing due to the high expense of GPUs. Different types of GPUs exhibit significant variations in computational performance, storage, and power consumption. The PD architecture offers a viable framework for organizing these heterogeneous GPUs to achieve high efficiency while minimizing costs. High computing-resource GPUs with lower memory bandwidth, such as the NVIDIA RTX 4090, are well-suited for prefill jobs. Additionally, despite not supporting NVLink, the RTX 4090 offers significant savings compared to expensive datacenter GPUs, highlighting the advantages of the PD architecture. We regard the performance optimization of the PD architecture in the heterogeneous hardware clusters as our future work. In addition, there are still many pressing issues to be addressed in large-scale deployment, such as load balancing across instances and the exploration of fine-grained and efficient autoscaling strategies. We will explore these practical issues in the future.

8 Conclusion

We present WindServe, a latency-oriented LLM serving system based on the PD architecture. WindServe addresses the critical issues of resource imbalance in existing Phase-Disaggregated architecture serving systems. It minimizes the overhead and interference associated with dynamic scheduling through Stall-free Rescheduling and Stream-based Disaggregation. WindServe achieves remarkable improvements in latency and SLO attainment, demonstrating up to 1.65-4.28× enhancements in TTFT median latency and a 1.5× reduction in TPOT P99 latency compared to state-of-the-art systems.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG] https://arxiv.org/abs/2308.16369

[3] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 3119–3137. https://aclanthology.org/2024.acl-long.172

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL] https://arxiv.org/abs/2004.05150

[5] T. Bradley. 2024. Hyper-Q Example. https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.

[7] NVIDIA Corporation. 2019. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.

[8] NVIDIA Corporation. 2023. Nvidia collective communications library (NCCL). https://developer.nvidia.com/nccl.

[9] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*.

[10] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[11] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 320–335. https://doi.org/10.18653/v1/2022.acl-long.26

[12] Insu Han, Rajesh Jayaram, Amin Karbasi, Vahab Mirrokni, David P. Woodruff, and Amir Zandieh. 2023. HyperAttention: Long-context Attention in Near-Linear Time. arXiv:2310.05869 [cs.LG] https://arxiv.org/abs/2310.05869

[13] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[14] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale GPU datacenters *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 104, 15 pages. https://doi.org/10.1145/3458817.3476223

[15] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2024. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. arXiv:2410.18038 [cs.LG] https://arxiv.org/abs/2410.18038

[16] Kyrylo Perelygin Kevin Klues and Pramod Ramarao. 2022. Improving GPU Utilization in Kubernetes. https://developer.nvidia.com/blog/improving-gpu-utilization-in-kubernetes/.

[17] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkgNKkHtvB

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[19] Heejun Lee, Geon Park, Youngwan Lee, Jaduk Suh, Jina Kim, Wonyoung Jeong, Bumsik Kim, Hyemin Lee, Myeongjae Jeon, and Sung Ju Hwang. 2024. A Training-free Sub-quadratic Cost Transformer Model Serving Framework With Hierarchically Pruned Attention. arXiv:2406.09827 [cs.CL] https://arxiv.org/abs/2406.09827

[20] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. *Proceedings of the 13th Symposium on Cloud Computing* (2022). https://api.semanticscholar.org/CorpusID:252762247

[21] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. arXiv:2404.14469 [cs.CL] https://arxiv.org/abs/2404.14469

[22] Evan Lucas, Dylan Kangas, and Timothy C Havens. 2024. Extra Global Attention Designation Using Keyword Detection in Sparse Transformer Architectures. arXiv:2410.08971 [cs.CL] https://arxiv.org/abs/2410.08971

[23] Meta. 2024. chatgpt. https://openai.com/index/chatgpt/.

[24] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 932–949. https://doi.org/10.1145/3620666.3651335

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications.

In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[26] NVIDIA. 2024. TensorRt-llm: A tensorrt toolbox for optimized large language model inference. https://github.com/NVIDIA/TensorRT-LLM.

[27] NVIDIA. 2025. GB200 NVL72. https://www.nvidia.com/en-us/data-center/gb200-nvl72/.

[28] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 369–384. https://doi.org/10.1145/3620665.3640383

[29] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[30] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 606–624. https://proceedings.mlsys.org/paper_files/paper/2023/file/c4be71ab8d24cdfb45e3d06dbfca2780-Paper-mlsys2023.pdf

[31] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[32] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 46745–46771. https://proceedings.mlr.press/v235/strati24a.html

[33] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[34] DeepSpeed team. 2022. DeepSpeed Inference. https://github.com/microsoft/DeepSpeed.

[35] Distserve teams. 2024. Distserve. https://github.com/LLMServe/DistServe

[36] Sharegpt teams. 2023. ShareGPT. https://sharegpt.com/.

[37] SwiftTransformer teams. 2024. Swiftransformer. https://github.com/LLMServe/SwiftTransformer.

[38] 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. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[39] Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Cantón Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony S. Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel M. Kloumann, A. V. Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, R. Subramanian, Xia Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zhengxu Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *ArXiv* abs/2307.09288 (2023). https://api.semanticscholar.org/CorpusID:259950998

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[41] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[42] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[43] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL] https://arxiv.org/abs/2205.01068

[44] Youpeng Zhao, Di Wu, and Jun Wang. 2024. ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1005–1017. https://doi.org/10.1109/ISCA59077.2024.00077

[45] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin