THUNDERSERVE: HIGH-PERFORMANCE AND COST-EFFICIENT LLM SERVING IN CLOUD ENVIRONMENTS

Youhe Jiang [*1] **Fangcheng Fu** [*2] **Xiaozhe Yao** [*3] **Taiyi Wang** [1] **Bin Cui** [2] **Ana Klimovic** [3] **Eiko Yoneki** [1]

ABSTRACT

Recent developments in large language models (LLMs) have demonstrated their remarkable proficiency in a range of tasks. Compared to in-house homogeneous GPU clusters, deploying LLMs in cloud environments with diverse types of GPUs is crucial for addressing the GPU shortage problem and being more cost-effective. However, the diversity of network environments and various GPU types on the cloud bring difficulties to achieving high-performance serving. In this work, we propose ThunderServe, a high-performance and cost-efficient LLM serving system for heterogeneous cloud environments. We introduce a *novel scheduling algorithm*, which optimizes the deployment plan of LLM serving to accommodate the heterogeneous resource and network bandwidth conditions in cloud environments. Furthermore, we propose a *lightweight re-scheduling* mechanism, designed to adapt to fluctuating online conditions (e.g., node failures, workload shifts) without the need for costly restarts of ongoing services. Empirical results in both heterogeneous cloud and homogeneous in-house environments reveal that ThunderServe delivers up to a $2.1\times$ and on average a $1.7\times$ increase in throughput and achieves up to a $2.5\times$ and on average a $1.5\times$ reduction in latency deadlines compared with state-of-the-art systems given the same price budget, suggesting opting for cloud services provides a more cost-efficient solution.

1 INTRODUCTION

Large Language Models (LLMs) such as GPT (Achiam et al., 2023), LLaMA (Touvron et al., 2023), OPT (Zhang et al., 2022) and Falcon (Institute, 2023) have demonstrated strong performance across a wide range of advanced applications. However, serving LLMs is cost-demanding, requiring a large amount of hardware accelerators like GPUs to satisfy efficiency requirements such as latency and throughput.

Mainstream LLM serving systems primarily focus on high-performance GPUs like NVIDIA A100 and H100 in homogeneous GPU clusters. However, it is difficult for many LLM service providers to get access to sufficient high-performance GPUs, either due to the well known GPU shortage problem (Yang et al., 2023; Strati et al., 2024) or the substantial fees. Meanwhile, with more and more advanced GPU architectures announced in the past few years, there are many less-performant GPUs in former generations remaining under-utilized. Thus, real-world cloud environments usually consist of heterogeneous GPUs and diverse prices. As shown in Table 1, cloud environments offer a wide range of hardware specifications and rental prices, providing users with diverse options to reduce the costs associated with LLM deployment and serving. Recent efforts (Jiang et al.; Mei et al., 2024; Griggs et al., 2024; Miao et al., 2023) have demonstrated that serving LLM with heterogeneous GPUs presents opportunities in reducing the serving cost. However, we find that these heterogeneous serving systems mainly address the heterogeneity in *hardwares* but fail to take account of the heterogeneity in *the computation and memory-access workloads* of different inference phases, which hinders the utilization of GPU resources. Such heterogeneity mainly comes from the distinct characteristics of LLM inference, and has raised a surge of research interests.

Table 1. GPU specifications and pricing

GPU Type	Memory Access Bandwidth	Peak FP16 FLOPS	Memory Limite	Price (per GPU)
A100	2 TB/s	312 TFLOPS	80 GB	$1.753/hr
A6000	768 GB/s	38.7 TFLOPS	48 GB	$0.483/hr
A5000	626.8 GB/s	27.8 TFLOPS	24 GB	$0.223/hr
A40	696 GB/s	149.7 TFLOPS	48 GB	$0.403/hr
3090Ti	1008 GB/s	71 TFLOPS	24 GB	$0.307/hr

Recent works (Patel et al., 2023; Hu et al., 2024; Qin et al., 2024; Jin et al., 2024) have designed phase splitting approaches to utilize different amount of computational resources for prefill and decoding phase in LLM inference, which involves partitioning the two phases onto separate devices and transmitting the intermediate results (primarily KV caches) between them. Many empirical evidences have

[*]Equal contribution [1]Department of Computer Science, University of Cambridge, Cambridgeshire, UK [2]Department of Computer Science, Peking University, Beijing, China [3]Department of Computer Science, ETH Zurich, Zurich, Switzerland. Correspondence to: Eiko Yoneki <eiko.yoneki@cl.cam.ac.uk>.

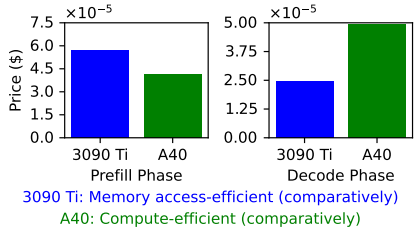

Figure 1. Prefill and decode prices for a single request with input and output lengths of 512 and 16 on 3090Ti and A40.

shown that such phase splitting approaches increase overall hardware utilization and system efficiency compared with the phase co-locating counterparts.

As the heterogeneity exists in both hardwares and workload characteristics (i.e., compute/memory-bound), we suggest that *the phase splitting approach fits the heterogeneous capabilities among GPUs in cloud environments well*. In particular, since the two phases differ in the workload characteristics, it is an intuitive idea to leverage different types of GPUs for the two phases. For instance, as illustrated in Figure 1, the A40 GPU with 149.7 TFLOPS is more cost-effective for the compute-intensive prefill phase, whereas the 3090Ti with 1008 GB/s memory bandwidth is better suited for the memory-bounded decode phase. Inspired by this, this work presents the first effort to integrate the phase splitting idea with the heterogeneity among GPUs, aiming to achieve high-performance and cost-effective LLM serving in cloud environments. Nevertheless, the unique attributes of cloud environments pose three key challenges:

Challenge 1: heterogeneous and limited resource pool. The available GPUs in cloud environments are usually in heterogeneous types, each with distinct specification (e.g., peak FLOPS, device memory bandwidth, and device memory limit), and the amount of each type is also restricted (Yang et al., 2023; Strati et al., 2024). As a result, to deploy multiple copies (a.k.a. model replicas) of the same LLM, we must consider how to organize the available GPUs from a global view — given the available resources of diverse types, we need to jointly consider which GPUs should be grouped together to serve one model replica, and whether this replica should serve as the prefill or decoding phase. To our knowledge, this is an unexplored problem so far.

Challenge 2: heterogeneous and low network bandwidth. The second essential characteristic of cloud environments is that GPUs are usually connected through low network bandwidth, typically, PCIe for intra-node and ethernet for inter-node communication. And the network bandwidth also exhibits a high level of heterogeneity across different pairs of GPUs due to the discrepancy in connectivity (different PCIe versions, node locality, etc.). Such a network condition raises a hurdle for efficient LLM serving. On the one hand, transmitting KV caches from prefill to decode replicas inevitably incurs significant communication volume. While prior works (Patel et al., 2023; Zhong et al., 2024) simply assume high-speed network connections (e.g., NVLink and Infini-Band) are available and overlook the communication overhead of KV caches (detailed in §2), which is impractical for clouds. On the other hand, due to the astonishing size of LLMs, model parallelism has been a cornerstone for LLM deployment. Thus, there expresses a need for designing heterogeneity-aware parallelization to facilitate phase splitting LLM serving on clouds.

Challenge 3: workload variability. Compared to in-house clusters, resources on clouds are more unstable (Miao et al., 2023; Duan et al., 2024; Yousif, 2018; Erben et al., 2024), and the distribution of requests (e.g., average arrival rate, input and output length) may change over time in online services in practice (Wang et al., 2024a). These factors exacerbate the variability of serving workloads in cloud environments. In order to adapt to such workload variations, prior works (Zhong et al., 2024) necessitate two steps: re-generating the deployment plan from scratch and re-loading the LLM parameters to adjust the model deployment. However, both steps are costly. Re-generating the deployment plan could take minutes to complete due to the complex hardware environments on cloud, and re-loading the LLM with a huge amount of parameters could be time-consuming. For instance, loading a 175B model with a disk bandwidth of 1.2 GBps takes over five minutes. Such expensive steps would lead to severe interruption to the online services.

To address these challenges, we develop ThunderServe, an efficient and robust LLM serving system on clouds. Our contributions are summarized as follows:

Contribution 1: We formulate the scheduling problem of LLM deployment and serving on cloud as a two-level hierarchical optimization problem, and develop a novel scheduling algorithm to optimize the deployment plan. In the upper-level, we develop a tabu search algorithm to partition the available GPUs of diverse types into model serving groups (with each group responsible for one model replica). In the lower-level, we determine the optimal parallel configuration for each group as well as the orchestration of prefill and decode replicas to optimize GPU and network usage.

Contribution 2: We design a lightweight re-scheduling mechanism, which only involves adjusting the phase designation and orchestration in real-time, accelerates the regeneration of deployment plan by a large extent, and does not need to re-load the LLM parameters. It enables our system to adapt to workload shifts at minimal cost, thereby enhancing the robustness of LLM serving on cloud.

Contribution 3: Based on these techniques, we implement ThunderServe, an efficient LLM serving system for clouds featuring phase splitting. ThunderServe allows the two

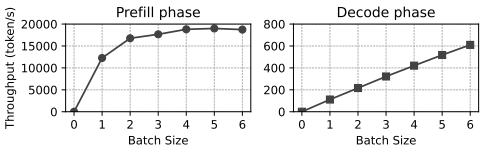

Figure 2. Effects of batching on different phases (LLaMA-7B with each input having a sequence length of 1024).

phases of LLM inference to be split onto separate GPUs with different resource allocations and parallel strategies. We further integrate a KV cache compression technique into our system, which performs a one-shot compression on the KV cache for efficient inter-phase communication on clouds while maintaining the model quality.

Contribution 4: The performance of ThunderServe in the cloud environment is evaluated through comprehensive experiments. We compare its system and economic efficiency with state-of-the-art LLM serving systems, including HexGen in the same heterogeneous cloud environment, as well as DistServe and vLLM in a homogeneous in-house setting given the *same* budget in terms of cloud service fees. The empirical results demonstrate that ThunderServe achieves up to 2.1× and on average 1.7× increase in throughput and up to 2.5× and on average 1.5× reduction in latency compared with existing systems, showcasing the potential of cost-effective LLM serving over clouds.

2 Background and Related Works

Phases of LLM inference. Given an input prompt, the inference process of LLMs typically consists of two phases: *the prefill phase* processes the prompt to compute the key-value (KV) cache and generates the first token in a single step, and *the decode phase* takes the last generated token and KV cache as inputs to generate subsequent tokens. Different from the prefill phase, the decode phase is executed for several steps, with each step generating only one token, which makes the decode phase more memory bandwidth bounded than the computationally intensive prefill phase.

Performance metrics. There are three key metrics to evaluate LLM inference: *time to first token (TTFT)*, which measures the time of generating the first token; *time per output token (TPOT)*, which quantifies the average time of generating each token in the decode phase; *end-to-end (E2E) latency*, which assesses the overall processing time of a request (includes queuing, prefill, and decode costs). For LLM serving systems, there are also two key metrics: *Service Level Objective (SLO) attainment*, which represents the percentage (e.g., 99%) of requests that can be served within a predefined time frame set by the SLO, and the SLO is often scaled to different multiples of single device execution latency (denoted as SLO scale) to evaluate system performance under different levels of SLO stringency; *throughput*, which measures the number of requests a system can handle within a specified time period. Efficient LLM serving systems should optimize either of the metrics, and meanwhile meet the performance requirements of specific applications if necessary.

Batching. Due to the divergent workloads of the two phases, integrating batching strategies results in performance variations. In the prefill phase, a small batch size quickly saturates the GPU, yielding marginal benefits from further batching. Besides, execution latency increases linearly with batch size, rendering batching impractical when the TTFT constraints are strict. As shown in Figure 2, we conduct a small testbed with LLaMA-7B as an example, which reveals that when the total number of tokens in a batch exceeds 1024, GPU efficiency reaches a plateau rather than being further enhanced, showcasing the limited effect of batching on system performance during the prefill phase. Due to the token-by-token processing nature of the decode phase, batching is essential for preventing low GPU utilization and enhancing efficiency. As shown in Figure 2, increasing the batch size enhances GPU efficiency, particularly in the decode phase. In short, batching in the decode phase improves performance, even under stringent TPOT constraints.

Parallelism strategies. To parallelize the model over multiple GPUs, there are two prevalent forms of model parallelism, which are tensor model parallelism and pipeline model parallelism. *Tensor model parallelism (TP)* (Shoeybi et al., 2019; Nagrecha, 2021) divides model weights and computationally intensive operations such as matrix multiplication across various GPUs, thereby splitting data scanning and computation to minimize LLM inference latency, particularly the TTFT in the prefill phase. *Pipeline model parallelism (PP)* (Huang et al., 2019; Narayanan et al., 2019) divides the layers of a model into multiple stages. These stages are assigned to distinct GPUs for execution and they establish a pipeline. Only inter-layer activations are needed to be communicated between stages.

Phase splitting deployment. As the prefill and decode phases differ in workload characteristics (i.e., compute/memory-bound) significantly, recent efforts propose to utilize different hardware resources for the two phases in order to avoid performance interference (Patel et al., 2023; Zhong et al., 2024; Hu et al., 2024; Jin et al., 2024; Qin et al., 2024). In other words, there are two kinds of model replicas, one for prefill and the other for decode, respectively, and it is necessary to transmit the KV cache from the prefill replicas to the decode ones. Due to the substantial size of KV cache, existing efforts essentially require high communication bandwidth for the KV cache transfer — Zhong et al. (2024) proposed to colocate prefill and decode replicas on GPUs within the same node, facilitating fast KV cache transfer with NVLink, while Patel et al. (2023) and

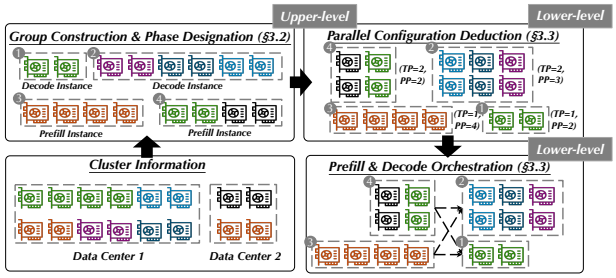

Figure 3. Workflow of our scheduling algorithm.

Hu et al. (2024) utilized high-speed Infini-Band connections for inter-node communication. However, in cloud environments, GPUs are connected via limited bandwidth (typically, PCIe for intra-node connections and Ethernet for inter-node connections) rather than high-speed connections (typically, NVLink and Infini-Band). Therefore, KV cache transfer would lead to a huge cost and it necessitates enhancement.

Heterogeneous GPU Computing. Recent research has investigated diverse approaches to deploy large models on heterogeneous GPU clusters. HexGen (Jiang et al.) proposes asymmetric partitioning and advanced scheduling to deploy generative inference in a decentralized and heterogeneous setting. Helix (Mei et al., 2024) formulates the heterogeneous GPUs and network connections as a maxflow problem and adopts a mixed integer linear programming algorithm to discover highly optimized strategies to serve LLMs. Our work has a similar objective, but is the first effort that integrates phase splitting with the heterogeneous GPUs to provide high-performance cloud serving for LLMs.

3 Scheduling in ThunderServe

This section introduces our scheduling algorithm, which aims to optimize the overall SLO attainment of the serving.

3.1 Overview and Problem Formulation

To describe the model deployment over the available resources of heterogeneous capabilities, the scheduling algorithm should produce four essential components: ① The *group construction*, i.e., how to partition the GPUs into multiple model serving groups, where each group is responsible for one model replica. ② The *phase designation* that indicates whether each group should serve as the prefill or decode phase. ③ The *parallel configuration* for each model replica. ④ The *orchestration of prefill and decode replicas* to guide how the requests should be routed. We term a solution to these four components as a ***deployment plan***.

As each deployment plan consists of four components, there is an extremely huge solution space. To ease the scheduling, we decouple the huge solution space into the Cartesian product of two sub spaces, by turning the derivation of deployment plan into a two-level hierarchical optimization problem as follows.

- **Upper-level:** Suppose there are G GPUs of T types in total, and G_t denotes the number of GPUs of type t. The objective of the upper-level problem is to find out the best combination of group construction and phase designation that maximizes the end-to-end SLO attainment.

- **Lower-level.** Given the group construction and phase designation, the objective of the lower-level problem is to determine the best parallel configuration for each group and how the prefill and decode replicas should be orchestrated to maximize the end-to-end SLO attainment.

Obviously, the Cartesian product of the solution spaces of the two problems completely covers all possible deployment plans, so finding the optimal deployment plan is equivalent to solving the hierarchical optimization problem by nature. Figure 3 shows the workflow of our scheduling algorithm. Given the target model and the available resources, we initiate the GPU construction and phase designation process, which involves a tabu search process that iteratively proposes solutions to the upper-level problem (§3.2). Then, for any possible solution to the upper-level problem, we solve the lower-level problem to obtain its performance, which involves the the parallel configuration deduction and phase orchestration process (§3.3).

3.2 Solving the Upper-level Problem

In cloud environments, there are various types of GPUs with heterogeneous capabilities, making the group construction a non-trivial problem. In addition, the phase-splitting design requires meticulous phase designation to the model serving groups for better performance, making the upper-level problem even more complex. Formally, in Appendix A, we show that the upper-level problem is essentially a job shop scheduling problem (JSSP), which is a notoriously difficult NP-hard problem in combinatorial optimization (Sotskov & Shakhlevich, 1995; Omar et al., 2006).

A well-known approach to solve JSSP is tabu search (Glover, 1990; Glover & Laguna, 1998; Gendreau & Potvin, 2005) and there have been many efforts applying tabu search to solve JSSP under various situations (Hurink et al., 1994; Dauzère-Pérès & Paulli, 1997; Zhang et al., 2007). Motivated by this, we adapt tabu search to the upper-level problem and design a brand new algorithm to identify the optimal deployment plan, which is demonstrated in Algorithm 1. In essence, it starts from an initial solution, and leverages an iterative neighborhood search process to improve the solution. Below we focus on how to determine the initial solution and how to construct neighbors given the current solution in our scenario.

Algorithm 1 Routine of solving the upper-level problem based on tabu search, where N_{step} denotes the number of search steps, N_{nghb} denotes the number of neighbors to navigation in each step, N_{mem} denotes the maximum number of memorized solutions, and $f(\cdot)$ denotes the performance of a solution evaluated by solving the lower-level problem.

function TABUSEARCH(N_{step}=100, N_{nghb}=10, N_{mem}=5):
  Initialize the current solution x
  Initialize tabu list $T \leftarrow []$, best solution $x_{best} \leftarrow x$
  /* Iterative Neighborhood Search */
  for N_{step} search steps **do**
    Construct N_{nghb} neighbors of x and exclude those in T to form the neighborhood set $\mathcal{N}$ for navigation
    $x' \leftarrow \arg\max_{x'' \in \mathcal{N}} f(x'')$
    if $f(x') > f(x_{best})$ **then** $x_{best} \leftarrow x'$
    $T.\text{append}(x')$
    if $\text{len}(T) > N_{\text{mem}}$ **then** $T \leftarrow T[-N_{\text{mem}} :]$
    $x \leftarrow x'$
  end for
  return x_{best}
end function

Initialization. It is essential to have a good initial solution in tabu search in order to speedup the search process and escape from local optima. Thus, we utilize the Hierarchical Clustering method (Shetty & Singh, 2021) to cluster the GPUs according to their inter-connection bandwidth matrix, and subsequently treat each generated cluster as one model serving group at initialization. Intuitively, this makes the initial assignment of model serving groups strategically avoid connections with ultra-low communication bandwidth in the cloud environment. In addition, the phase designation of each group is randomly initialized.

Neighbor construction. In the iterative search process, tabu search evaluates a set of neighboring solutions given the current solution. Denote $g_{i,t}$ as the number of GPUs of type t in group i. We provide four approaches of to construct the neighboring solutions, as exemplified in Figure 4 and detailed below.

- *Flipping phase designation.* This approach randomly selects a group and flips its phase. In other word, if the group is originally designated to serve as a prefill replica, then it will be changed to a decode replica, and vice versa.
- *Splitting a group into two.* This approach randomly selects and splits a group $g_{s,t}$ into two based on a random ratio r, assigning $\lfloor g_{s,t} \times r \rfloor$ GPUs to the first new group $g'_{s_1,t}$ and the remainder to the second group $g'_{s_2,t}$, which is effective for exploring how dividing resources impacts performance, particularly when a group might be overly large or tasked beyond its efficient operating capacity, i.e.,

$$g'_{s_1,t} \leftarrow \lfloor g_{s,t} \times r \rfloor, g'_{s_2,t} \leftarrow g_{s,t} - g'_{s_1,t}, \forall t \in \{1, \ldots, T\}.$$

  The phase of the new groups will be randomly designated.
- *Merging two groups into one.* This approach randomly selects and merges two groups $g_{i,t}, g_{j,t}$ into one, which explores the potential benefits or drawbacks of resource centralization for individual model serving, i.e.,

$$g'_{\text{merged},t} \leftarrow g_{i,t} + g_{j,t}, \quad \forall t \in \{1, \ldots, T\}.$$

  The phase of the new group will be randomly designated.
- *Moving GPUs between groups.* This approach involves moving a certain number of (denoted as m_t) GPUs of type t from group i to group j. It is useful for exploring the effects of resource reallocation in scenarios where different groups may benefit from different GPU capabilities.

$$g'_{i,t} \leftarrow g_{i,t} - m_t, \quad g'_{j,t} \leftarrow g_{j,t} + m_t.$$

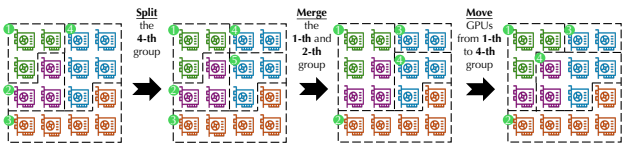

Figure 4. Examples of neighbor construction in tabu search (changes in phase designation are omitted for simplicity).

The adjustment in group construction and phase designation iteratively navigate the neighborhood space of the current solution, enabling tabu search to explore potential performance enhancements. Additionally, to expedite the search process, early checks are performed for each generated neighborhood. For instance, if the total GPU memory of any serving group after the moving or splitting operations is insufficient to hold even a single copy of the model parameters, then the constructed neighbor is eliminated from further evaluation.

3.3 Solving the Lower-level Problem

The goal of the lower-level problem is two-fold, i.e., finding the optimal parallel configuration for each model serving group and how to orchestrate all groups together. Fortunately, since the available resources and the designated phase of each group are given, we find that the deduction of optimal parallel configuration is independent to the orchestration. To be specific, suppose C' is a parallel configuration for an arbitrary group, and C'' is another configuration with higher performance, then it is obvious that we can always find out an orchestration with C'' that is at least as good as that with C'. As a result, we first deduce the optimal parallel configuration for each group individually, and then determine the orchestration, as introduced below.

Deduction of parallel configuration. Given the available resources and the designated phase of each group, we wish to deduce the optimal parallel configuration. As discussed in §2, the two phases differ in workload characteristics, so their desirable parallel configurations also vary. For groups

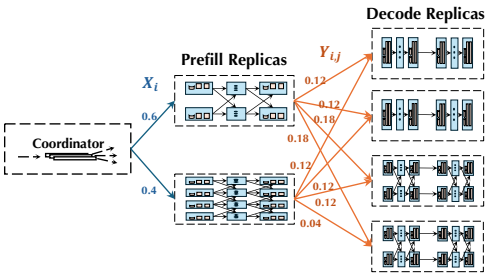

Figure 5. An example orchestration of prefill and decode replicas.

that serve as prefill replicas, we aim to deduce the latency-optimal parallel configurations, since the prefill phase is computation-intensive and batching does not help to enhance efficiency. In contrast, for groups that serve as decode replicas, we aim to deduce the throughput-optimal parallel configurations, since this memory bandwidth bounded phase benefits from batching.

Numerous studies (Jiang et al.; Zheng et al., 2022; Li et al., 2023; Miao et al., 2022; 2023; Wang et al., 2024b) have investigated how to deduce the optimal parallel configuration by meticulously enumerating a vast number of possible configurations. In this work, we further take the key characteristics of cloud environments into account and design heuristics to accelerate the enumeration process. We introduce the heuristics below and leave the details of the deduction routine in Appendix B.

- Typically, cloud services generally do not provide rapid links among nodes. Thus, we disallow tensor model parallelism to be deployed over GPUs across different nodes due to its demand of higher network bandwidth.
- Since different types of GPUs may differ in the memory capacity and computing ability, we support non-uniform pipeline layer partitioning for pipeline model parallelism.
- In response to the heterogeneity in inter-node communication, we employ a dynamic programming algorithm that aims to identify the path minimizing the cross-stage communication cost in pipeline model parallelism.

Orchestration of prefill and decode replicas. Due to the network heterogeneity in cloud environments, it is essential to identify the optimal orchestration of prefill and decode replicas within the cluster to minimize KV cache communication cost and optimize overall SLO attainment.

We adopt the inference task simulator from DistServe (Zhong et al., 2024), which estimates the SLO attainment according to workload information (e.g., input length, output length, etc.) and single request processing time. It is noteworthy that we integrate the KV cache communication cost into the simulator, since it is non-negligible in cloud environments. To be specific, suppose there are m prefill replicas and n decode replicas, our simulator enumerates every pair of them and estimates the SLO attainment by integrating the KV cache communication cost, which is analyzed via the alpha-beta model (Hockney, 1994):

$$T_{ij}^{(kv_comm)} = \alpha_{ij} + 2bsh\mathrm{N}_{\text{bytes}}/\beta_{ij}, \tag{1}$$

where b, s represent the batch size and sequence length for inference, h represents the hidden size of a Transformer block, $\mathrm{N}_{\text{bytes}}$ represents the byte size for KV cache communication, and α_{ij}, β_{ij} represent the network latency and bandwidth between the i-th prefill replica and the j-th decode replica. We evaluate the simulator and alpha-beta model accuracy in Appendix J.

Based on this, we estimate the SLO attainment of every pair of prefill and decode replicas. Formally, denote $D \in \mathbb{R}^{m\times n}$ as the SLO attainment matrix, where D_{ij} represents the estimated SLO attainment when requests are processed by the i-th prefill replica and the j-th decode replica. Then, we turn the optimization problem of overall system SLO attainment into a simple two-stage transportation problem (TSTP) (Santoso & Heryanto, 2022) as follows:

$$\begin{aligned}
&\underset{X\in\mathbb{R}^m, Y\in\mathbb{R}^{m\times n}}{\arg\max} \sum_{i=1}^{m}\sum_{j=1}^{n} X_i Y_{ij} D_{ij} \\
&\text{s.t.} \sum_{i=1}^{m} X_i = 1,\ \sum_{j=1}^{n} Y_{ij} \text{ for } \forall i,\ X_i \geq 0 \text{ for } \forall i,\ Y_{ij} \geq 0 \text{ for } \forall i, j,
\end{aligned}$$

where X_i denotes the portion of incoming requests that are assigned to the i-th prefill replica, and Y_{ij} denotes the portion of requests processed by the i-th prefill replica that are dispatched to the j-th decode replica. The TSTP can be solved by linear programming, and the optimal X^*, Y^* describe how requests are routed among the model serving groups, which also represent the orchestration of different replicas to maximize the overall system SLO attainment.

By combining the deduction of optimal parallel configuration and the orchestration of different replicas, we accomplish the solution to the lower-level problem, and the resulting system SLO attainment is returned to the tabu search process (i.e., $f(\cdot)$ in Algorithm 1).

3.4 Lightweight Rescheduling in Real Time

Numerous factors in cloud services affect the optimal deployment plan, with two primary factors being LLM inference workloads and GPU availability. On the one hand, LLM services usually exhibit significant variation in workload characteristics across different downstream tasks. For instance, coding workloads typically generate shorter responses than conversational workloads but usually have longer prompts (Patel et al., 2023). On the other hand, compared to in-house clusters, cloud resources are inherently more dynamic and unstable, necessitating a good support

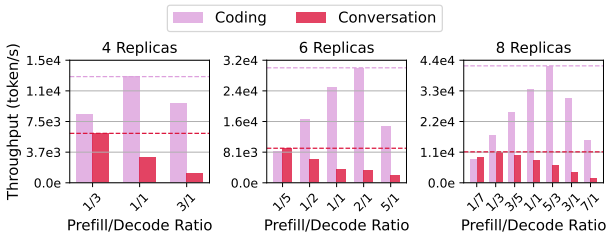

Figure 6. Throughput (token/s) by prefill-to-decode ratio. Impact of phase designation and orchestration on overall system throughput. We experiment with LLaMA-13B on both coding and conversation workloads across clusters with 8, 12, and 16 A5000 GPUs, respectively, with two GPUs serving one replica. The ratio represents the prefill-to-decode ratio (i.e., the ratio of # prefill replicas to # decode replicas). We have also provided SLO Attainment results in Figure 14 of Appendix D.

for cluster size adjustments in real-time. Consequently, rescheduling is essential for ThunderServe to adapt the deployment plan to varying online workloads and cluster size changes on cloud. However, altering the deployment plan is far from trivial in cloud environments. If we re-run the scheduling algorithm from scratch and reload the model parameters according to the updated deployment plan (take minutes to complete), it would lead to severe interruption to the online services. Therefore, we propose a *lightweight rescheduling* process that only adjusts the phase designation and orchestration in the deployment plan to accommodate varying workloads and cluster sizes.

The rationality behind our lightweight rescheduling is that the changes in workload generally influence the demands on the prefill and decode phases. To elaborate, we conduct comprehensive experiments to demonstrate the impact of phase designation and orchestration over diverse workloads and cluster sizes (with fixed group construction and parallel configuration). As shown in Figure 6, the coding workload, characterized by relatively longer input length and shorter output length, exhibits enhanced performance with more prefill replicas and fewer decode replicas. Conversely, the conversation workload, typified by relatively shorter prompts and longer responses, necessitates more decode replicas and fewer prefill replicas to prioritize resources to the long-running decoding, with the ideal prefill-to-decode ratio fluctuating as the cluster size varies. These findings underscore the critical importance of precise adjustments in the phase designation and orchestration to achieve optimal system performance and realize the ability to adapt to various workloads and cluster sizes.

The lightweight rescheduling is done by simplifying the routines introduced in §3.2 and §3.3:

- For the tabu search process, only the flipping phase designation approach is used to construct neighboring solutions, while the other approaches are not involved in the lightweight rescheduling.
- The deduction of parallel configuration is skipped and the orchestration problem will be solved using the unaltered parallel configurations and the newly designated phases.

Table 2. Impact of KV cache communication compression on the model accuracy on CoQA, TruthfulQA and GSM8K tasks.

Task		LLaMA-7B	LLaMA-13B
CoQA	16-bit	63.95	66.35
	4-bit	64.58	66.54
TruthfulQA	16-bit	30.64	29.68
	4-bit	30.13	29.34
GSM8K	16-bit	13.23	22.34
	4-bit	12.54	21.29

Although our lightweight rescheduling leads to sub-optimality, our experiments in §5.4 show that it achieves comparable performance against rescheduling from scratch in various scenarios. More importantly, by merely adjusting the phase designation and orchestration, there is no need to reload the parameters, and thus introduces almost zero overhead to the online services. Consequently, our lightweight rescheduling improves the flexibility and robustness of ThunderServe to a great extent.

4 Implementation

ThunderServe is a distributed LLM serving system designed to optimize online services in cloud environments, which develops a novel scheduling algorithm to partition the given cloud GPU resources into model serving groups, designate which phase each group should serve as, deduce the optimal parallel configuration for each group, and determine how the requests should be routed among groups. It is implemented using 20K lines of Python and C++/CUDA code. Besides, ThunderServe incorporates FlashAttention (Dao et al., 2022) and PagedAttention (Kwon et al., 2023) to accelerate LLM inference, and leverages the batching strategy proposed by Zhong et al. (2024) for LLM serving.

Overall routine. The overall routine of ThunderServe is as follows. ① To launch a serving process, the scheduling algorithm (§3.2 and §3.3) generates the deployment plan, which is then utilized to instantiate the model replicas over the cloud GPU resources. ② During the serving process, the incoming requests are dispatched across the prefill and decode replicas, and the generated responses are gathered. ③ At the same time, the inference workload is constantly monitored and reported to the scheduling algorithm. ④ Once a workload shift is detected, the scheduling algorithm triggers the lightweight re-scheduling process (§3.4) to adjust the deployment plan in response to the new workload. Due to the space constraint, we refer interested readers to Appendix E for more implementation details of ThunderServe.

KV cache compression technique. As discussed in §2, prior works rely on high-bandwidth connections (i.e., NVLINK or InfiniBand) for transferring KV cache in phase splitting deployment, which is impractical in cloud service scenarios characterized by heterogeneous network conditions among GPUs. To reduce the KV cache communication cost, we borrow the idea of low-precision quantization from KIVI (Liu et al., 2024) to quantize KV cache to fewer bits, so that the size of each element (i.e. N_{bytes} in Equation 1) is shrinked. However, unlike existing works in the field of KV cache quantization (Liu et al., 2024; Kang et al., 2024), our system does not retain low bitwidths when using the KV cache values for computation. Specifically, the KV cache values in the prefill replica are quantized and packed for communication, and then immediately unpacked and dequantized after they are received by the decode replica. Thus, both the prefill and decode phases are conducted using the 16-bit KV cache values rather than the quantized ones. By this means, we can significantly reduce the KV cache communication volume, without harming the model quality.

To elaborate, we conduct a small testbed with LLaMA-7B over two A5000 GPUs, which featured an inter-communication bandwidth of 40 Gbps — significantly lower than that of InfiniBand and NVLink. Quantizing the 16-bit elements to 4-bit significantly reduces KV cache communication costs from 16-30% to 4-9% of the total end-to-end inference costs, drastically improving the performance of the system. Besides, we demonstrate the accuracy results of LLaMA-7B and LLaMA-13B models on CoQA, TruthfulQA and GSM8K tasks with both 16-bit and 4-bit KV cache precision levels. As we do not retain low bitwidths when using the KV cache values for computation, our experiments in Table 2 consistently show that the accuracy drop when using 4-bit precision compared to 16-bit precision remains below 2% across all experimental scenarios, which confirms the validity of our approach. Due to the space constraint, we provide more evaluation results in Appendix I, including perplexity (PPL) and ROUGE-1/2/L on the WikiText2, PTB, and CBT datasets, and the end-to-end throughput comparisons between 16-bit and 4-bit precision.

5 EVALUATION

5.1 Experimental Setup

Hardware environments. We consider two types of hardware environments with almost the same price budgets.

- *Heterogeneous GPUs on the cloud.* We rent GPUs from Vast.ai, a GPU cloud service provider. We rent four types of instances with 32 GPUs in total: two 4×A6000 instances, two 4×A5000 instances, one 8×A40 instance and two 4×3090Ti instances, with a total price of $13.542/hour to represent the heterogeneous case.
- *Homogeneous GPUs in a in-house server.* For baseline systems that do not support heterogeneous GPUs, we use one in-house server equipped with 8×A100-80GB GPUs. According to Table 1, renting the same GPUs costs $14.024/hour, which is close to the aforementioned price budget on the cloud.

The GPU specifications are provided in Table 1, while the network bandwidth can be found in Appendix C.

Model and workloads. We deploy the popular open-source LLaMA-30B model across two real-world workloads, coding and conversation, from the Azure Conversation dataset (Patel et al., 2023). And we follow prior works (Li et al., 2023; Jiang et al.) to generate the inference workload using a Poisson process determined by the request rate, with consecutive requests (inter-arrival times) following an exponential distribution.

Evaluation metrics. Following prior works (Patel et al., 2023; Zhong et al., 2024), we focus on overall system SLO attainment and throughput when evaluating the performance. System SLO attainment indicates the percentage of requests completed within a predefined latency deadline. There are three types of SLO: TTFT, TPOT, and E2E SLO. We specifically measure system SLO attainment by the percentage of requests that meet the time criteria established by each SLO type. We scale the SLO to various multiples of the execution latency of A100 GPUs (SLO Scale in Figure 7), which allows us to evaluate system performance under different levels of operational stringency. For a target SLO attainment goal (e.g., 90% and 99%), we focus on the minimum latency deadline required to achieve the desired attainment.

Baselines. We consider state-of-the-art systems under both the cloud and in-house settings, respectively. Under the cloud setting, we consider HexGen (Jiang et al.), an LLM serving system for clouds featuring advanced scheduling and asymmetric parallelism. For the in-house scenario, we consider vLLM (Kwon et al., 2023), a prestigious LLM serving system, as well as DistServe (Zhong et al., 2024), an LLM serving system featuring phase splitting.

5.2 End-to-end Evaluation

In this section, we compare the end-to-end performance of ThunderServe against baselines on various workloads.

System SLO attainment comparisons on the cloud. We first evaluate the performance of ThunderServe on the cloud. As shown in Figure 7, typically higher average request arrival rates require higher SLO scales (i.e., longer latency deadlines) to meet the SLO attainment goal. ThunderServe consistently outperforms HexGen in terms of all TTFT, TPOT, and E2E SLO attainments.

On the coding workload, ThunderServe achieves up to 1.8×

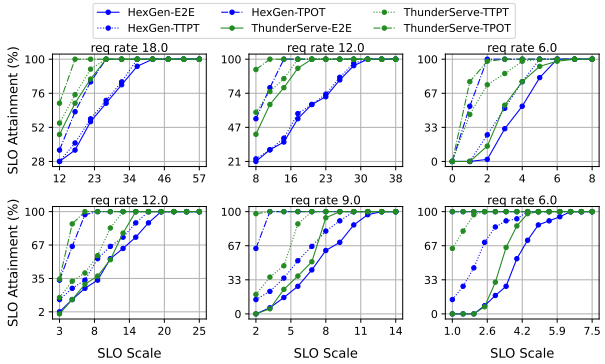

Figure 7. SLO attainment results on coding (top row) and conversation (bottom row) workloads.

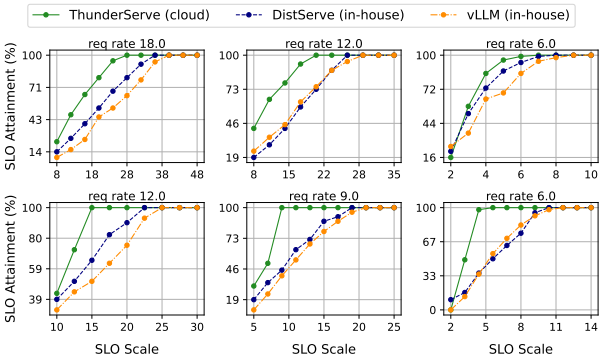

Figure 8. SLO attainment results on coding (top row) and conversation (bottom row) workloads.

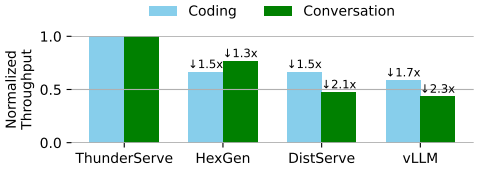

Figure 9. Throughput scaled by ThunderServe.

and on average 1.4× lower E2E latency deadlines compared with existing approaches. Specifically, as we will elaborate in §5.3, the coding workload makes our scheduling algorithm designate more prefill replicas than decode replicas, since the bottleneck is on prefilling given the relatively long input prompts. And the prefill-to-decode ratio decreases with the surge of the average request arrival rate, which matches our previous discussion in §3.4. On the conversation workload, ThunderServe achieves up to 1.4× and on average 1.3× lower E2E latency deadlines. The conversation workload makes our scheduling algorithm deploy more decode replicas than prefill replicas, since the bottleneck is on decoding given the relatively long output responses. The phase splitting technique significantly reduces prefill-decode interference during inference, leading to improved TTFT and TPOT SLO attainments in all cases.

Cost-efficiency of deploying LLM services on the cloud. To assess the cost-efficiency of deploying LLM services on the cloud, we compare ThunderServe in the cloud setting with DistServe and vLLM in the in-house setting, given the same price budget. As shown in Figure 8, ThunderServe significantly outperforms DistServe and vLLM, achieving up to 2.5× and on average 1.8× lower E2E latency deadlines. This advantage stems from ThunderServe's ability to deploy 3× more model replicas on the cloud than DistServe on the in-house server within the same price budget, which provides ThunderServe with superior parallel processing capability. And the scheduling algorithm of ThunderServe takes full advantage of the heterogeneity of cloud GPUs and improves system performance by allocating appropriate GPUs for prefilling and decoding, respectively. Thus, ThunderServe greatly improves the cost-efficiency of deploying LLM services on the cloud.

Table 3. Model deployment discovered by ThunderServe.

Workload	GPU Configuration	Strategy	Type of Replicas
Coding	8×A40	TP=2, PP=1	4 Prefill Replicas
	4×A5000	TP=4, PP=1	1 Prefill Replica
	4×A6000	TP=2, PP=1	2 Prefill Replicas
	2×A5000+2×3090Ti	TP=2, PP=2	1 Prefill Replica
	4×3090Ti	TP=2, PP=2	1 Decode Replica
	4×A6000	TP=1, PP=2	2 Decode Replicas
	2×A5000+2×3090Ti	TP=2, PP=2	1 Decode Replica
Conversation	6×A40	TP=2, PP=1	3 Prefill Replicas
	2×A5000+2×3090Ti	TP=2, PP=2	1 Prefill Replica
	4×3090Ti	TP=2, PP=2	1 Decode Replica
	2×A40	TP=1, PP=2	1 Decode Replica
	4×A5000	TP=2, PP=2	1 Decode Replica
	8×A6000	TP=1, PP=2	4 Decode Replicas
	2×A5000+2×3090Ti	TP=2, PP=2	1 Decode Replica

System throughput comparisons. We further compare the system throughput between ThunderServe and the baselines. As demonstrated in Figure 9, compared with HexGen, ThunderServe achieves 1.5 and 1.3× higher throughputs in coding and conversation workloads respectively. And compared with DistServe in the in-house setting, 1.5 and 2.1× higher throughputs are realized. These results demonstrate ThunderServe's ability to effectively manage larger loads.

5.3 Case Study of Scheduling

Table 3 presents the model deployment discovered by our scheduling algorithm. It can be seen that ThunderServe prioritizes the GPUs with better computing ability for prefilling (e.g., A40) and those with higher memory access bandwidth GPUs for decoding (e.g., 3090Ti). Additionally, ThunderServe automatically assigns more decode replicas to the conversation workload due to its longer output lengths, while more prefill replicas to the coding workload due to the longer prompt lengths. These results verify that ThunderServe is able to take the heterogeneity in both hardwares and serving workloads into account, generating model deployment that maximizes the system performance.

Compared to the in-house setting with an 8×A100 instance,

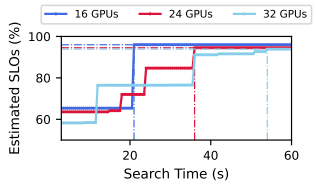

Figure 10. Convergence curves of scheduling from scratch for different cluster sizes.

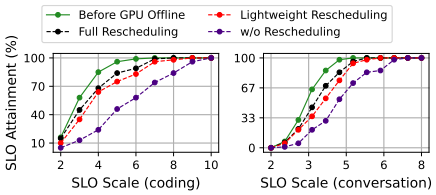

Figure 11. SLO attainments of before (indicated by the solid line) and after (indicated by the dotted lines) 4 out of 32 GPUs offline.

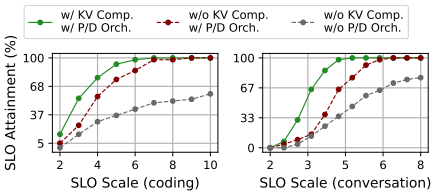

Figure 12. Impact of KV cache compression and prefill and decode orchestration on the SLO attainments.

where only 4 model replicas can be served, ThunderServe serves a maximum of 12 model replicas in the cloud setting within the same price budget. Although individual inference processes in the cloud setting may experience increased latency due to the lower hardware performance (i.e., the cloud GPUs are less performant than A100), the overall performance is improved due to the higher number of model replicas. Thus, ThunderServe conveys the ability of high-performance and cost-efficient LLM serving on clouds.

5.4 Effectiveness and Ablation Studies

Time cost of Scheduling. We evaluate the running time of our scheduling algorithm from scratch with cluster sizes of 16, 24 and 32 GPUs. Leveraging our effectively designed neighborhood construction method, the algorithm based on tabu search scales well with the number of GPUs, requiring approximately 21, 36 and 54 seconds to converge, as shown in Figure 10. This search process is executed once before the initial deployment of the system, rendering its time cost negligible given the hourly scale of online services.

Lightweight re-scheduling. To evaluate the effectiveness of our lightweight re-scheduling, we consider a scenario where 4 out of 32 GPUs become unavailable. Specifically, we remove two decode replicas and let ThunderServe re-schedule on the fly. We compare our lightweight re-scheduling with ① a full re-scheduling approach, which involves re-starting the services and reloading parameters, and ② a no re-scheduling approach, which does not make any changes to the deployment plan and keeps the services using the remaining GPUs. As shown in Figure 11, our lightweight re-scheduling achieves similar SLO attainment to the full re-scheduling approach and outperforms the no re-scheduling approach, showing the strengths of our lightweight re-scheduling. More importantly, our lightweight re-scheduling process finishes within seconds, without any overhead on parameter reloading, far exceeding the full re-scheduling approach. This is reasonable as tabu search is done locally and we only adjust the phase designation and orchestration to adapt to different cluster sizes. Therefore, ThunderServe is able to handle the dynamicity in cloud environments well.

KV cache compression and orchestration. There are two major techniques in ThunderServe to address the communication cost of KV cache transmission from prefill to decode replicas, which are the KV cache compression and orchestration method. We conduct experiments to assess their effects. As illustrated in Figure 12, ThunderServe exhibits a degraded performance in both coding and conversation scenarios without KV cache compression, incurring approximately 1.3× the overhead per single request. This undermines the benefits of phase splitting. If we further disable the orchestration method described in §3.3 and substitute it with a random dispatching, there is another 4× of performance degradation. In summary, ThunderServe chooses the parallel configurations and KV cache communication paths that optimize overall system performance given the high heterogeneity of communication bandwidth on the cloud.

Table 4. Overhead of Full and Lightweight Rescheduling.

	Rescheduling	Reloading	Overall
Full	54(±5)s	103(±10)s	157(±13)s
Lightweight	13(±2)s	0s	13(±2)s

More Experiment Results. We have also conducted more experiments to demonstrate the strengths of ThunderServe, including more case studies, the effect of KV cache compression on model quality, and the accuracy of our simulator. Due to the space constraint, we refer interested readers to our supplemental material for more details.

6 Conclusion

This paper explores the potential of deploying LLM services on clouds. Toward this end, we presented ThunderServe, a system that employs hybrid model parallelism and phase splitting to enhance LLM serving efficiency across heterogeneous cloud GPU clusters. With ThunderServe, we proposed a novel scheduling algorithm that co-optimizes resource allocation, phase designation, parallelism strategies, and the orchestration of both prefill and decode phases. Additionally, we proposed a lightweight re-scheduling mechanism to enhance ThunderServe performance in response to fluctuating online workloads for extremely fast adjustment on clouds. We conducted experiments on various workloads in both heterogeneous cloud and homogeneous in-house settings to demonstrate that ThunderServe outperforms state-of-the-art systems within the same price budget.

REFERENCES

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

Dauzère-Pérès, S. and Paulli, J. An integrated approach for modeling and solving the general multiprocessor job-shop scheduling problem using tabu search. *Annals of Operations Research*, 70(0):281–306, 1997.

Duan, J., Song, Z., Miao, X., Xi, X., Lin, D., Xu, H., Zhang, M., and Jia, Z. Parcae: Proactive,{Liveput-Optimized}{DNN} training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 1121–1139, 2024.

Erben, A., Mayer, R., and Jacobsen, H.-A. How can we train deep learning models across clouds and continents? an experimental study. *Proceedings of the VLDB Endowment*, 17(6):1214–1226, 2024.

Gendreau, M. and Potvin, J.-Y. Tabu search. *Search methodologies: introductory tutorials in optimization and decision support techniques*, pp. 165–186, 2005.

Glover, F. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

Glover, F. and Laguna, M. *Tabu search*. Springer, 1998.

Griggs, T., Liu, X., Yu, J., Kim, D., Chiang, W.-L., Cheung, A., and Stoica, I. Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity, 2024.

Hockney, R. W. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing*, 20(3): 389–398, 1994.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Hurink, J., Jurisch, B., and Thole, M. Tabu search for the job-shop scheduling problem with multi-purpose machines. *Operations-Research-Spektrum*, 15:205–215, 1994.

Institute, T. I. Falcon 180b, 2023. URL https://falconllm.tii.ae/falcon-180b.html.

Jiang, Y., Yan, R., Yao, X., Zhou, Y., Chen, B., and Yuan, B. Hexgen: Generative inference of large language model over heterogeneous environment. In *Forty-first International Conference on Machine Learning*.

Jiang, Y., Fu, F., Yao, X., He, G., Miao, X., Klimovic, A., Cui, B., Yuan, B., and Yoneki, E. Demystifying cost-efficiency in llm serving over heterogeneous gpus. *arXiv preprint arXiv:2502.00722*, 2025.

Jin, Y., Wang, T., Lin, H., Song, M., Li, P., Ma, Y., Shan, Y., Yuan, Z., Li, C., Sun, Y., et al. P/d-serve: Serving disaggregated large language model at scale. *arXiv preprint arXiv:2408.08147*, 2024.

Kang, H., Zhang, Q., Kundu, S., Jeong, G., Liu, Z., Krishna, T., and Zhao, T. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., and Stoica, I. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving, 2023.

LibP2P. A modular network stack, 2023. URL https://libp2p.io/.

Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

Mei, Y., Zhuang, Y., Miao, X., Yang, J., Jia, Z., and Vinayak, R. Helix: Distributed serving of large language models via max-flow on heterogeneous gpus. *arXiv preprint arXiv:2406.01566*, 2024.

Miao, X., Wang, Y., Jiang, Y., Shi, C., Nie, X., Zhang, H., and Cui, B. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proceedings of the VLDB Endowment*, 16(3):470–479, 2022.

Miao, X., Shi, C., Duan, J., Xi, X., Lin, D., Cui, B., and Jia, Z. Spotserve: Serving generative large language models on preemptible instances, 2023.

Nagrecha, K. Model-parallel model selection for deep learning systems. In *Proceedings of the 2021 international conference on management of data*, pp. 2929–2931, 2021.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

Omar, M., Baharum, A., and Hasan, Y. A. A job-shop scheduling problem (jssp) using genetic algorithm (ga). In *Proceedings of the 2nd im TG T Regional Conference on Mathematics, Statistics and Applications Universiti Sains Malaysia*, 2006.

Patel, P., Choukse, E., Zhang, C., Íñigo Goiri, Shah, A., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting, 2023.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

Santoso, S. and Heryanto, R. M. Development of two-stage transportation problem model with fixed cost for opening the distribution centers. *Jurnal Ilmiah Teknik Industri*, 21 (1):63–71, 2022.

Shetty, P. and Singh, S. Hierarchical clustering: a survey. *International Journal of Applied Research*, 7(4):178–181, 2021.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Sotskov, Y. N. and Shakhlevich, N. V. Np-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, 59(3):237–266, 1995.

Strati, F., Elvinger, P., Kerimoglu, T., and Klimovic, A. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pp. 107–116, 2024.

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

Wang, Y., Chen, Y., Li, Z., Kang, X., Tang, Z., He, X., Guo, R., Wang, X., Wang, Q., Zhou, A. C., et al. Burstgpt: A real-world workload dataset to optimize llm serving systems. 2024a.

Wang, Y., Jiang, Y., Miao, X., Fu, F., Zhu, S., Nie, X., Tu, Y., and Cui, B. Improving automatic parallel training via balanced memory workload optimization. *IEEE Transactions on Knowledge and Data Engineering*, 2024b.

Yang, Z., Wu, Z., Luo, M., Chiang, W.-L., Bhardwaj, R., Kwon, W., Zhuang, S., Luan, F. S., Mittal, G., Shenker, S., et al. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 437–455, 2023.

Yao, X. Open Compute Framework: Peer-to-Peer Task Queue for Foundation Model Inference Serving, September 2023. URL https://github.com/autoai-org/OpenComputeFramework.

Yousif, M. Cloud computing reliability—failure is an option. *IEEE Cloud Computing*, 5(3):4–5, 2018.

Zhang, C., Li, P., Guan, Z., and Rao, Y. A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers & Operations Research*, 34(11):3229–3242, 2007.

Zhang, J., Huang, H., Zhang, P., Wei, J., Zhu, J., and Chen, J. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread int4 quantization, 2024. URL https://arxiv.org/abs/2411.10958.

Zhang, J., Wei, J., Zhang, P., Zhu, J., and Chen, J. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. In *International Conference on Learning Representations (ICLR)*, 2025a.

Zhang, J., Xiang, C., Huang, H., Xi, H., Wei, J., Zhu, J., and Chen, J. Spargeattn: Accurate sparse attention accelerating any model inference, 2025b.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning, 2022.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

A NP-Hardness of Deployment Planning

Finding the optimal deployment plan that maximizes the overall SLO for deploying multiple models on a heterogeneous GPU cluster with variable interconnect topology and computational capabilities is non-trivial. In particular, we show that this problem is NP-hard by transforming it into the well-known NP-hard Job Shop Scheduling Problem (JSSP) (Sotskov & Shakhlevich, 1995; Omar et al., 2006).

Transformation of the deployment planning problem to JSSP. Each GPU in the heterogeneous cluster serves as a distinct machine in the JSSP. These GPUs exhibit differences in computation power, memory, and communication capabilities. Each model, or its components depending on the placement method, is considered a job within JSSP. The deployment of each model involves multiple tasks or operations, each corresponding to the deployment of a part of a model on one or more GPUs, accompanied by specific resource requirements and execution constraints. Sequential dependencies are evident in scenarios where the completion of one operation on a GPU is prerequisite for the initiation of the next on another GPU, characteristic of pipeline model parallelism. Concurrent dependencies arise when operations must occasionally synchronize across GPUs, reflecting interdependencies that require coordination akin to those in tensor model parallelism. In this context, maximizing SLO does not solely involve minimizing idle and wait times but also necessitates the optimization of the allocation and scheduling of operations to ensure continuous and efficient GPU utilization. Thus, this challenge can be viewed as a variant of JSSP where the objective shifts from minimizing makespan to maximizing SLO, analogous to maximizing the number of completed jobs or operations within certain latency deadlines. This requires managing both the sequence and concurrency of operations across heterogeneous resources and optimizing overall system efficiency to mitigate bottlenecks and reduce synchronization overheads.

Job shop scheduling is recognized as NP-hard due to the complexity inherent in managing dependencies and varying capabilities across machines. By formulating this problem as a variant of JSSP adapted for SLO, we establish that solving the model placement problem is at least as hard as solving the classic NP-hard JSSP, thus confirming the NP-hardness of the problem.

B Deduction of Parallel Configuration

Given the group formation and the designated phase, we need to deduce the optimal parallel configuration for each group. Algorithm 2 outlines the process. ① We enumerate all possible TP and PP combinations on each given group formation. Note that our first heuristic is to limit tensor model parallelism within single-type GPUs, so the TP degree should be smaller or equal to the minimum number of single-type GPUs in the group, which largely minimizes the search space. ② Dynamic programming algorithm is utilized to route the pipeline communication path. It optimizes communication routing in a network by using a bitmask to represent all possible subsets of stages, initializes each stage with a zero bandwidth and builds paths by calculating the potential bandwidth for each link between stages, updates the optimal path recursively if a higher bandwidth stage is found, and determines the maximum bandwidth path available by examining the states for the subset that includes all stages, ensuring the most efficient data transfer across the network. ③ We adjust the pipeline layer partition with respect to the memory capacity and computing ability of different GPU types. Specifically, the pipeline partition is adjusted in proportion to the total memory and computing capacity of the GPU set currently servicing this stage, while ensuring that the memory limits of individual GPUs are not exceeded. This heuristic has proven effective in determining an optimal pipeline partition. ④ For the compute-bound prefill replicas, we select the latency

Algorithm 2 Generate Model Parallel Configurations

Initialize: group formation: $G = \{G_1, G_2, ..., G_g\}$, minimum number of single-type GPUs in the group: $T = \{T_1, T_2, ..., T_g\}$, cluster information: I, model configuration: M
$model_parallel_configurations \leftarrow []$
for i in len(G) **do**
  $plan_list \leftarrow []$
  /* Limit TP within Single-type GPUs */
  for TP in $\{1, 2, ..., T_i\}$ **do**
    for PP in $\{1, 2, ..., \frac{G_i.num_gpus}{T_i}\}$ **do**
      /* Route Pipeline Communication */
      $plan \leftarrow$ `Dynamic_Programming`(I, TP, PP)
      /* Generate Pipeline Partition */
      $plan \leftarrow$ `Pipeline_Partition`$(M, plan)$
      if $G_i.type$ is prefill **then**
        $C \leftarrow$ latency$(plan)$
      else
        $C \leftarrow$ throughput$(plan)$
      end if
      $plan_list.append((C, plan))$
    end for
  end for
  if $G_i.type$ is prefill **then**
    /* Select Latency Optimal Plan */
    $plan \leftarrow \min(C)$ in $plan_list$
  else
    /* Select Throughput Optimal Plan */
    $plan \leftarrow \max(C)$ in $plan_list$
  end if
  $model_parallel_configurations.append(plan)$
end for
return $model_parallel_configurations$

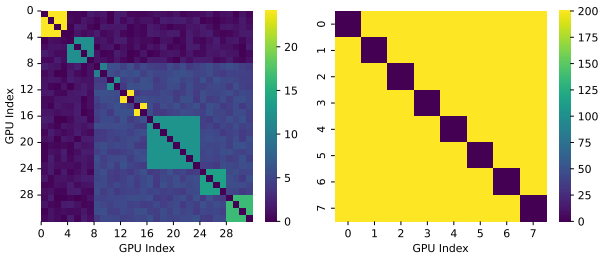

Figure 13. Heat map of inter-connection bandwidth matrix in the cloud (left) and in-house (right) settings.

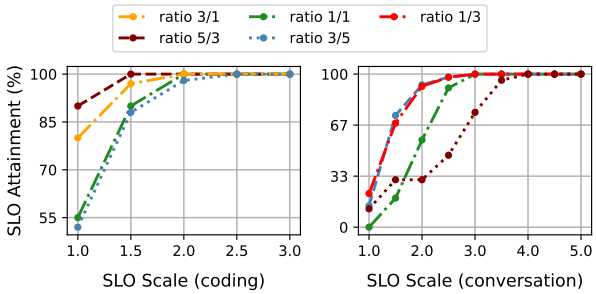

Figure 14. Impact of phase designation and orchestration on overall system SLO attainment. We experiment with LLaMA-13B on both coding and conversation workloads across 16 A5000 GPUs, with two GPUs serving one replica.

optimal plans, for the memory bandwidth-bound decode replicas, we select the throughput optimal plans. To estimate the latency and throughput of each plan, we employ the cost model proposed by HexGen (Jiang et al.), which directly provides us with the inference memory and latency costs for both prefill and decode phases, relative to different request batch sizes. We calculate the throughput by dividing the maximum total batched token size that the device group can handle by the decode latency. Note that the estimated latency information is also provided to our simulator for SLO estimation.

C Inter-connection Bandwidth Matrix

The bandwidth distributions exhibit significant variability in cloud and in-house environments. We measure the communication bandwidth between each pair of GPUs via NCCL for both environments described in §5.1. As shown in the left heatmap of Figure 13, the cloud environment demonstrates notable bandwidth heterogeneity, influenced by a range of GPU types and network configurations. This variability results in non-uniform connectivity patterns across the network. Conversely, the right heatmap showcases the in-house environment, characterized by a uniform GPU-to-GPU communication bandwidth, evidenced by consistently high connectivity values. These visualizations emphasize the distinctions between cloud and in-house environments.

D Ratio Impact on System SLO Attainment

We show the impact of phase designation and orchestration on overall system SLO attainment in Figure 14. The coding workload, characterized by relatively longer input length and shorter output length, exhibits enhanced performance with more prefill replicas and fewer decode replicas. A ratio of 5:3 yields the optimal results. Conversely, the conversation workload, typified by relatively shorter prompts and longer responses, necessitates more decode replicas and fewer prefill replicas to prioritize resources to the long-running decoding. Here, a ratio of 3:5 achieves the best performance.

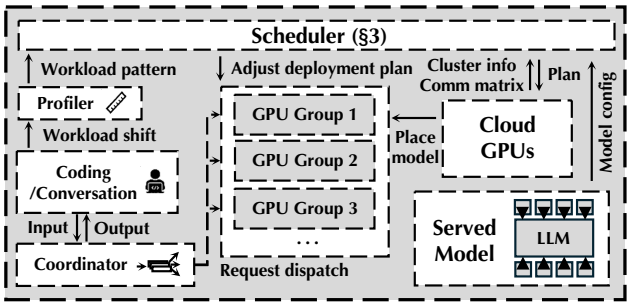

Figure 15. System overview of ThunderServe.

E Implementation Details

Overview of ThunderServe. The architecture overview of ThunderServe is shown in Figure 15. There are three major components, which are the scheduler, the workload profiler, and the task coordinator.

The *scheduler* is the core of ThunderServe for high-performance LLM serving in cloud environments. The scheduler takes as input the model configurations (e.g., hidden size and layer number), workload patterns obtained from the workload profiler, cluster information (e.g., available GPUs and their corresponding types), and communication bandwidth matrix among all GPUs. Then, it performs the scheduling algorithm introduced in §3 to provide the optimal deployment plan. Should there be a detected shift in workload, or a GPU heartbeat timeout that suggests a need for cluster size adjustment, the scheduler will perform the lightweight re-scheduling process and adjust the deployment plan to adapt to the new workload or cluster size.

The *workload profiler* monitors the real-time workload patterns, including the average prompt length of incoming requests and average output length of generated responses. These patterns are utilized to analyze the prefill and decode cost for each single request. For instance, in contemporary LLM services, common workload scenarios include coding and conversation (Patel et al., 2023), where both typically have a median prompt length exceeding 1000 tokens. However, the coding service produces much fewer output tokens, with a median of 13, while the conversation service generates a larger number of output tokens, with a median of 129. Undoubtedly, the overall system workload varies when the proportions of incoming requests for various services change in real-time. Once an obvious workload shift is detected, the workload profiler will notify the scheduler.

The *task coordinator* is in charge of the request dispatching among the prefill and decode replicas. Upon receiving a request, the task coordinator assigns the appropriate prefill replica and decode replica, respectively. The assignment is guided by the deployment plan generated by the scheduler. The task coordinator is mainly based on an open-source implementation of decentralized computation coordination (Yao, 2023) that utilizes libP2P (LibP2P, 2023) to establish connections among the work groups in a peer-to-peer network.

Based on these components, the overall routine of ThunderServe is as follows. ① To launch a serving process, the scheduler generates the deployment plan, which is then utilized to instantiate the model replicas over the cloud GPU resources. ② During the serving process, the coordinator dispatches the incoming requests across the prefill and decode replicas, and gathers the generated responses. ③ At the same time, the workload profiler consistently monitors the workload and reports to the scheduler. ④ Once a workload shift is detected, the scheduler triggers a lightweight re-scheduling process to adjust the deployment plan for better adaptation to the new workload.

Parallel communication groups. All communication primitives in ThunderServe are implemented using NVIDIA Collective Communication Library (NCCL). To circumvent the substantial overhead associated with constructing NCCL groups, ThunderServe preemptively establishes a global communication group pool containing all potentially required groups. For KV cache communication, we employ NCCL's asynchronous `SendRecv`/`CudaMemcpy` functions for KV cache communication to prevent GPU blocking and enable computation and communication overlapping during transmission. KV cache queues are maintained on the prefill replicas, and upon completion of a decoding round, the decode replicas retrieve KV caches from these queues, utilizing the GPU memory of the prefill replicas as queuing buffers.

F Case Study of Scheduling

We list the deployment plan generated by ThunderServe from coding workload to conversation workload in the heterogeneous setting. We use the following representation to describe the scheduled results. We use an array to specify one independent model replica, with two numbers representing the degrees of tensor model parallelism and pipeline model parallelism. For example, (2,2) indicates a model replica with tensor model parallel degree of 2 and pipeline model parallel degree of 2 (2 pipeline stages).

We also provide the instances we considered in §5.1 here for better readability: two 4×A6000 instances, two 4×A5000 instances, one 8×A40 instance and two 4×3090Ti instances, making up to be 32 GPUs in total.

Parallel configuration breakdown. In the coding workload, the 8×A40 instance employs a parallel strategy (2,1) to support four prefill replicas. One 4×A6000 instance uses a parallel strategy (2,1) to support two prefill replicas, while the other one 4×A6000 instance uses a parallel strategy (1,2) for two decode replicas. One 2×A5000 and one 2×3090Ti instances utilize a parallel strategy (2,2) to support one prefill replica, and the other one 2×A5000 and one 2×3090Ti instances utilize a parallel strategy (2,2) to support one decode replica. One 4×A5000 instance utilizes a parallel strategy (4,1) to support one prefill replica. One 4×3090Ti instance implements a parallel strategy (2,2) to support one decode replica.

In the conversation workload, the 8×A40 instance employs parallel strategies (2,1) and (1,2) to support three prefill replicas and one decode replica, respectively. The two 4×A6000 instances utilize a parallel strategy (1,2) to support four decode replicas. One 2×A5000 and one 2×3090Ti instances utilize a parallel strategy (2,2) to support one prefill replica, and the other one 2×A5000 and one 2×3090Ti instances utilize a parallel strategy (2,2) to support one decode replica. One 4×A5000 instance utilizes a parallel strategy (2,2) to support one decode replica. One 4×3090Ti instance implements a parallel strategy (2,2) to support one decode replica.

Insights. In the in-house setting, the 8×A100 instance can only serve 4 model replicas, while in the cloud setting, the 32 cloud GPUs with various types can serve a maximum of 12 model replicas with various parallel configuration within the same price budget. In this case, although individual inference tasks in the cloud setting may experience increased latency due to the lower hardware performance (e.g., GPU flops and bandwidth), the overall system performance is improved due to the higher number of model replicas. Additionally, our scheduling algorithm prioritizes GPUs with high peak fp16 flops for prefilling (e.g., A40) and high memory bandwidth GPUs for decoding (e.g., 3090Ti), and

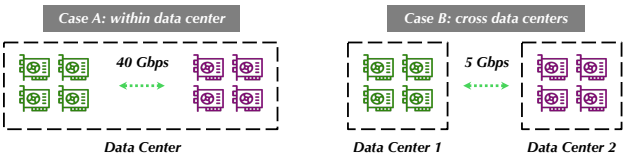

Figure 16. Two exampled network conditions on cloud.

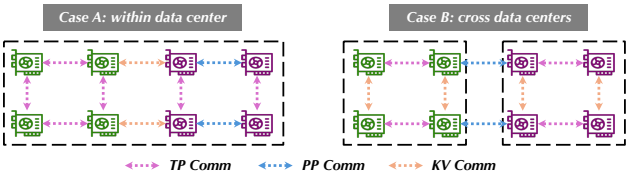

Figure 17. ThunderServe deployment plans on different cases.

selects the most suitable model parallel configuration for each phase to optimize the overall system performance. And although KV cache compression can linearly mitigate communication overhead, significant disparities in bandwidth across different cloud environments render extremely low bandwidth scenarios—such as those experienced between data centers—unsuitable for effective KV cache communication. Thanks to our scheduling and orchestration algorithms, ThunderServe automatically identifies KV cache transmission paths that maintain overall performance.

G Case Study of Lightweight Rescheduling

We list the deployment plan change during lightweight rescheduling with 4 out of 32 GPUs (one 4×A6000 instance that support two decode replicas) become unavailable.

The deployment plan for the coding workload, detailed in Appendix F, initially includes 8 prefill and 4 decode replicas. After the offline of 4 GPUs, there are 8 prefill and 2 decode replicas remaining. A subsequent lightweight rescheduling converts one prefill replica, which uses 4 A5000 GPUs with a (4,1) strategy, into a decode replica. The adjustment is reasonable as this group of GPUs exhibits the highest overall memory bandwidth among the prefill replicas. The deployment plan for the conversation workload initially includes 4 prefill and 8 decode replicas. After the offline of 4 GPUs, there are 4 prefill and 6 decode replicas remaining. A subsequent lightweight rescheduling converts one prefill replica, which uses 2 A40 GPUs with a (2,1) strategy, into a decode replica.

H Case Study of Network Effect on Phase Splitting

Table 5. Benchmarks of non-disaggregation baseline vs. ThunderServe under high inter-instance communication bandwidth vs. ThunderServe under low inter-instance communication bandwidth.

Configuration	Prefill	KV Comm	Decode	E2E Throughput
Baseline	884 ms	0 ms	1689 ms	1610 tokens/s
ThunderServe (High)	698 ms	133 ms	1126 ms	3292 tokens/s
ThunderServe (Low)	964 ms	41 ms	1846 ms	2196 tokens/s

Consider use ThunderServe to serve LLaMA-30B model in a heterogeneous environment featuring two GPU instances: the first instance equipped with 4×A40 GPUs, and the second with 4×3090Ti GPUs. We conducted tests on the inference throughput of this setup by feeding it continuous input sequences of length 1024 under two different inter-instance communication bandwidths: 40 Gbps and 5 Gbps, as demonstrated in Figure 16.

We established a non-disaggregating baseline that utilizes 4×A40 GPUs to support one model replica and 4×3090Ti GPUs to support another. By comparing the baseline with ThunderServe under different network conditions, we observed some interesting results: With a bandwidth of 40 Gbps, ThunderServe leverages the 4×A40 GPUs with higher peak flops to support one prefill replica, and the 4×3090Ti GPUs with higher memory access bandwidth to support one decode replica. This configuration optimizes system performance, achieving a 2× performance gain over the non-disaggregating baseline. However, at a lower bandwidth of 5 Gbps, the inter-instance communication bandwidth is insufficient for efficient KV cache communication. Consequently, ThunderServe allocates 2×A40 GPUs and

Table 6. Impact of KV cache communication compression on the perplexity results on WikiText2, PTB and CBT datasets.

Dataset		LLaMA-7B	LLaMA-30B
WikiText2	16-bit	3.53	2.73
	4-bit	3.55	2.75
PTB	16-bit	7.46	6.49
	4-bit	7.42	6.55
CBT	16-bit	7.66	6.31
	4-bit	7.70	6.30

Table 7. LLaMA rouge results (using 16-bit outputs as the ground truth and the 4-bit outputs as the prediction) on WikiText2, PTB and CBT datasets.

Dataset		LLaMA-7B	LLaMA-30B
WikiText2	ROUGE-1	0.962	0.942
	ROUGE-2	0.941	0.928
	ROUGE-L	0.955	0.941
PTB	ROUGE-1	0.975	0.928
	ROUGE-2	0.950	0.911
	ROUGE-L	0.971	0.928
CBT	ROUGE-1	0.925	0.946
	ROUGE-2	0.912	0.931
	ROUGE-L	0.925	0.937

Table 8. Benchmarks of ThunderServe with 16-bit vs. 4-bit communications.

Configuration	Prefill	KV Comm	Decode	E2E Throughput
16-bit	684 ms	584 ms	1108 ms	2450 tokens/s
4-bit	698 ms	133 ms	1126 ms	3292 tokens/s

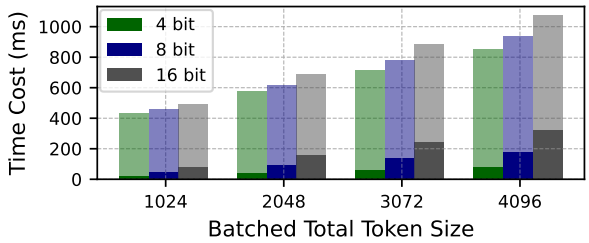

Figure 18. Impact of KV cache communication compression. (Non-transparent: time cost of KV cache communication. Transparent: end-to-end processing time.)

2×3090Ti GPUs to both prefill and decode replica, which utilizes intra-instance high network bandwidth for KV cache communication and inter-instance low network bandwidth for pipeline communication, resulting in a 1.4× improvement over the non-disaggregating baseline. The illustration of deployment plans are demonstrated in Figure 17, the single request prefill/decode/KV cache communication time and overall system throughputs are demonstrated in Table 5.

I PPL AND ROUGE RESULTS ON KV CACHE COMPRESSION

We list the PPL and ROUGE results of LLaMA-7B and LLaMA-30B models on WikiText2, PTB and CBT datasets with both 16-bit and 4-bit KV cache precision levels, as shown in Table 6 and Table 7. Experimental results have demonstrated that the PPL between 16-bit precision and 4-bit precision is within 1% across all experimental scenarios, and the ROUGE-1, ROUGE-2 and ROUGE-L scores are around 0.95 across all cases, which confirms the validity of our approach. We also demonstrate the the benchmarks of ThunderServe with 16-bit vs. 4-bit communications in Table 8 with the same experimental setups as mentioned in Appendix H, and benchmarks in Figure 18, with two A5000 GPUs serving a LLaMA-7B model.

J SIMULATOR AND ALPHA-BETA MODEL ACCURACY

To assess the accuracy of the simulator and alpha-beta model for KV cache communication, we conducted a series of micro-benchmarks using the LLaMA-30B model. These benchmarks varied in SLO scales and batched token sizes

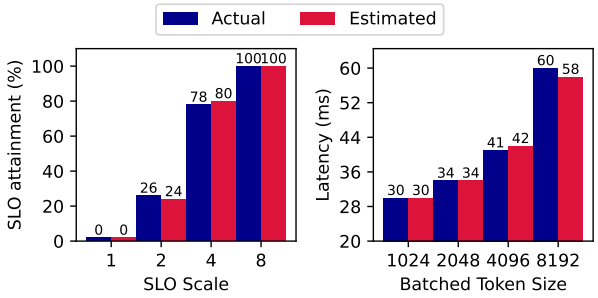

Figure 19. Comparison of benchmarked and estimated performance metrics for simulator (left) and alpha-beta model (right).

to evaluate our estimation outputs against actual execution metrics, specifically SLO attainment and latency. The results, detailed in Figure 19, indicate that the simulator and alpha-beta model closely correspond with actual execution performance.