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 $model\_parallel\_configurations$

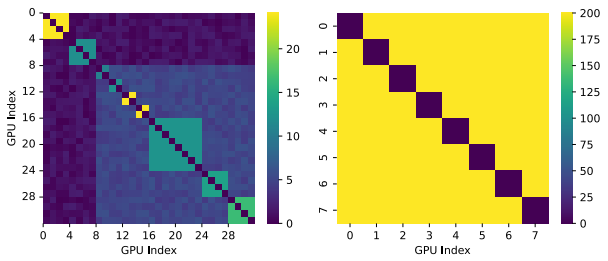

Figure 13. Heat map of inter-connection bandwidth matrix in the cloud (left) and in-house (right) settings.

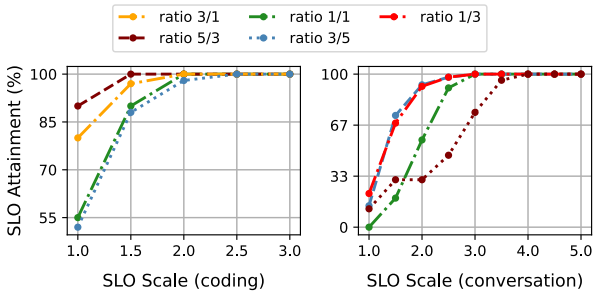

Figure 14. Impact of phase designation and orchestration on overall system SLO attainment. We experiment with LLaMA-13B on both coding and conversation workloads across 16 A5000 GPUs, with two GPUs serving one replica.

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., 2024), 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.

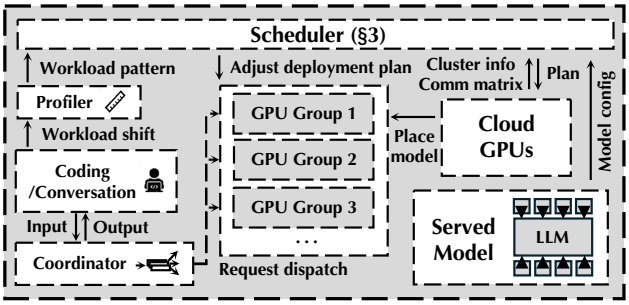

Figure 15. System overview of ThunderServe.

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

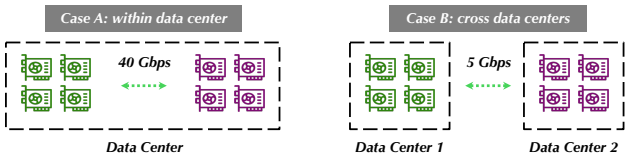

Figure 16. Two exampled network conditions on cloud.

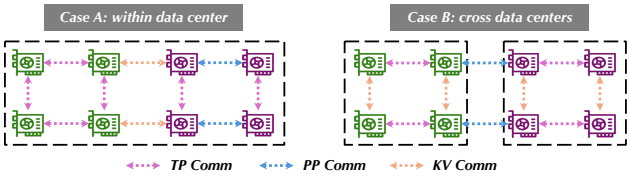

Figure 17. ThunderServe deployment plans on different cases.

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

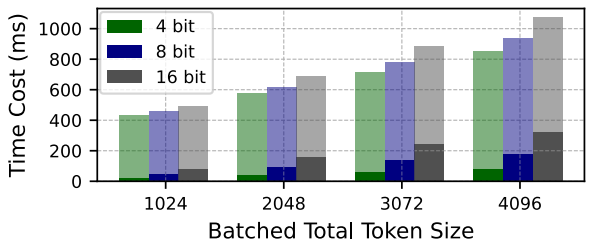

Figure 18. Impact of KV cache communication compression. (Non-transparent: time cost of KV cache communication. Transparent: end-to-end processing time.)

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

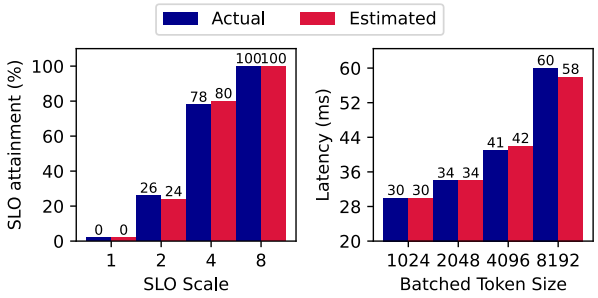

Figure 19. Comparison of benchmarked and estimated performance metrics for simulator (left) and alpha-beta model (right).

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.