Gyges: Dynamic Cross-Instance Parallelism Transformation for Efficient LLM Inference

Haoyu Chen
mike.chy@alibaba-inc.com
Fudan University and Alibaba Group
Shanghai, China

Xue Li
youli.lx@alibaba-inc.com
Alibaba Group
Hangzhou, China

Kun Qian
kunqian.qk@alibaba-inc.com
Alibaba Group
Hangzhou, China

Yu Guan
guanyu.guan@alibaba-inc.com
Alibaba Group
Hangzhou, China

Jin Zhao
jzhao@fudan.edu.cn
Fudan University
Shanghai, China

Xin Wang
xinw@fudan.edu.cn
Fudan University
Shanghai, China

Abstract

Efficiently processing the dynamics of requests, especially the context length variance, is important in Large Language Model (LLM) serving scenarios. However, there is an intrinsic trade-off: while leveraging parallelism strategies, such as Tensor Parallelism (TP), can coordinate multiple GPUs to accommodate larger context lengths, it inevitably results in degraded overall throughput.

In this paper, we propose Cross-Instance Parallelism Transformation (Gyges), which adaptively adjusts the parallelism strategies of running instances to align with the dynamics of incoming requests. We design (1) a page-friendly, header-centric layout to accelerate KV cache transformations; (2) dedicated weight padding to accelerate model weight transformations; and (3) a transformation-aware scheduler to cooperatively schedule requests and parallelism transformations, optimizing the overall performance. Evaluations using real-world traces show that Gyges improves throughput by 1.75×-6.57× compared to state-of-the-art solutions.

1 Introduction

As large language models (LLMs) play increasingly important roles across various domains, the demand for LLM inference services in cloud environments has surged dramatically. Delivering optimal performance for inference requests has become a paramount requirement. Significant efforts have been devoted to optimizing performance through instance-level acceleration mechanisms [13, 15, 16, 19, 23, 26, 28, 31] and global scheduling frameworks [5, 12, 17, 22, 25, 29, 30].

However, the intrinsic memory-intensive nature of inference services (i.e., both KV cache and model weight must be in GPU memory) introduces **a fundamental trade-off: extreme performance versus long context support**. When serving requests with extended sequence lengths, parallelized inference, like Tensor Parallelism (TP), becomes mandatory. As parallelism increases within a single instance, the overhead from computation partitioning and inter-worker communication escalates, causing significant performance degradation. Our practical measurements show that scaling from $4 \times (TP1)$ to $TP4$ can incur over 57% throughput loss. This dilemma remains unresolved in existing optimizations.

In production workloads, long-context requests occur from time to time. This tough nut forces us to maintain different deployments with varying TP configurations (e.g., $TP1$ for short requests and $TP4$ for long requests). However, these static TP settings compel workers to operate in an inefficient way even when long-context demands are absent.

Based on this observation, we propose a basic idea: **designing runtime cross-instance parallelism transformation to handle dynamic workloads**. For example, dynamically merging four $TP1$ instances into a single $TP4$ instance during the arrivals of long-context requests. This approach offers two key benefits: (1) When sudden long-context requests arrive, all workers are already operational, eliminating the need to wait for new instance provisioning. As long as we can achieve an efficient parallelism transformation, immediate service becomes possible, avoiding the minutes-long delays inherent in traditional scaling. (2) After long-context processing is complete, the $TP4$ instance can be elastically decomposed into four $TP1$ instances to maximize system throughput.

Achieving efficient parallelism transformation requires overcoming multiple challenges. The core reason $TP4$ can support longer contexts is its ability to save model weight memory for expanded KV cache storage. However, for performance optimization, mainstream inference engines pre-allocate memory for both model weights and KV cache, lacking dynamic resizing or cross-use capabilities. The most relevant work, vAttention [21], enables flexible KV cache management via dynamic page allocation. Since it does not consider massive KV cache/weight splitting and reconstruction in parallelism transformation, directly using vAttention in our scenario introduces notable performance issues.

First, the transformation of the KV cache requires redistributing the head-specific KV cache across workers, causing

Conference'17, July 2017, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

severe memory fragmentation. This fragmentation undermines the memory-saving benefits of the transformation, while defragmentation introduces prohibitive overhead. To resolve this, we design a page-friendly header-centric KV cache layout (§4.1) that reduces memory overhead by 91.6% and time cost by 86% during transformation.

Second, the transformation of the model weight requires the division of the weights between workers. Page-wise memory management uses a minimum unit of 2MB [1]. However, the split weights of many models do not align perfectly with this 2MB granularity. This misalignment results in additional memory allocation and data movement. By closely examining the splitting and calculation patterns, we overcome this challenge through parallelism-aware padding and a series of optimization techniques that enable in-place weight transformation (§4.2).

Finally, since parallelism transformation is a novel runtime inference serving paradigm, we develop a transformation-aware scheduler that tightly integrates parallelism adaptation with request dispatching (§5). This ensures optimal performance by dynamically balancing long-context demands and resource efficiency.

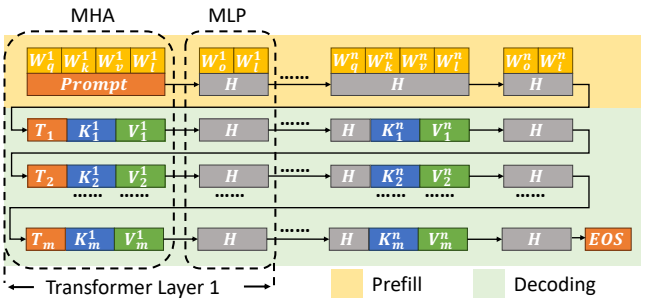
Our key contributions include:

- A thorough analysis of the trade-off between performance and long-context support in LLM inference, highlighting the requirement and challenges of cross-instance parallelism transformation (§3).
- The detailed design for the KV cache transformation, the weight transformation, and the layer-by-layer hybrid parallelism transformation (§4).
- The transformation-aware scheduler that dynamically adapts to workloads (§5).
- Evaluations demonstrate Gyges improving up to 6.57× throughput and decreasing 97% transformation cost compared with the state-of-the-art alternatives (§6).

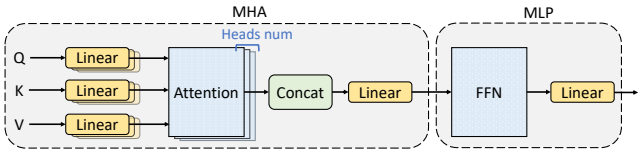
2 Background

Basic model serving. The basic LLM inference workflow is shown in Figure 1a. An LLM inference service receives requests from remote clients and generates responses in a token-by-token way. The entire serving process for each request consists of two main phases: prefill and decode. The prefill phase processes the entire input request to construct the KV cache and generate the first token. The decode phase recursively generates new tokens based on the KV cache of all previous tokens until it reaches the end-of-sequence (EOS) token or the maximum token length. A model is constructed by a series of transformer layers, and the generation of each token requires passing through all layers of the model.

Transformer layer. As shown in Figure 1b, a typical transformer layer consists of two main parts: Multi-Head Attention (MHA) and Multi-Layer Perceptron (MLP). The MHA computation requires the context of all previous tokens. To



(a) The typical LLM inference workflow.



(b) Multi-head attention and Multi-Layer Perceptron.

Figure 1. LLM inference overview.

eliminate redundant calculations, KV cache is used to store the internal results of these prior tokens. In contrast, the MLP is primarily constructed from two General Matrix Multiplications (GEMM), which necessitate fixed-size model weights.

Parallelized model serving. To concurrently utilize multiple GPUs for a single LLM inference service (e.g., to accommodate larger KV cache by increasing GPU memory), a series of parallelism strategies is proposed.

The most fundamental and widely deployed parallelism strategy is TP. In each transformer layer, different headers in the MHA are distributed across various workers (each worker possesses a GPU), and the tensors in the MLP are divided into segments, each managed by a different worker. During the computation of each layer, each worker computes partial results and uses AllReduce among workers to generate the final results. With TP, the weights of MHA, MLP, and KV cache can be evenly distributed across all workers.

Besides TP, other parallelism strategies are also introduced in the inference scenario. Pipeline Parallelism (PP) splits different transformer layers among different workers to divide memory pressure across different GPUs. Sequence Parallelism (SP) splits the input sequence into multiple parts and serves them with different GPUs in sequence. In a $PP = N/SP = N$ case, during the serving of each request, only $1/N$ GPUs are activated in any time slot, leading to significantly low utilization [10]. For serving Mixtures-of-Experts (MoE) models, Expert Parallelism (EP) is specifically designed to optimize the use of more GPUs.

According to our production statistics (containing thousands of instances), TP is the dominant parallelism strategy. 91.7% of multi-GPU LLM serving instances employ TP. Only 3.2% of instances employ TP+EP, while none employ PP/SP.

Table 1. Performance of different parallelism strategies.

	TP1	TP2	TP4
Maximal supported sequence	3.75K	41.25K	120.5K
Single instance throughput	448 tps	670 tps	767 tps
Total throughput	1792 tps	1340 tps	767 tps

Therefore, in this paper, we focus mainly on solving performance challenges in the TP scenario.

3 Motivation and Basic Idea

3.1 Peak Throughput versus Large Context

While TP offers increased GPU memory for supporting a larger KV cache, it is not a free lunch: the overall throughput decreases accordingly. We conduct a representative experiment (serving Qwen2.5-32B with 4 H20 GPUs) to show the pros and cons introduced by TP. The model size of Qwen2.5-32B (BF16 datatype) is 62.34 GB, runtime activations take 14.3 GB, and the GPU memory size of H20 is 96 GB. Therefore, with 4 H20 GPUs, actually we have three typical deployment choices: $4\times(TP1)$, $2\times(TP2)$ and $TP4$. When deploying with $4\times(TP1)$, 64.9% GPU memory is used to maintain model weights. On the other hand, with $TP4$, only 16.2% is used to maintain model weights on each GPU. As shown in Table 1, $TP1$ can only support a maximum input sequence length of 3.75K, while $TP4$ can serve 32× larger. On the other hand, serving the same workloads (sequence length of 1K tokens), the performance varies according to different parallelism configurations. $4\times(TP1)$ can deliver 233% throughput compared to $TP4$, while keeping the target Service Level Objective (e.g., TTFT < 10s and TPOT < 100ms).

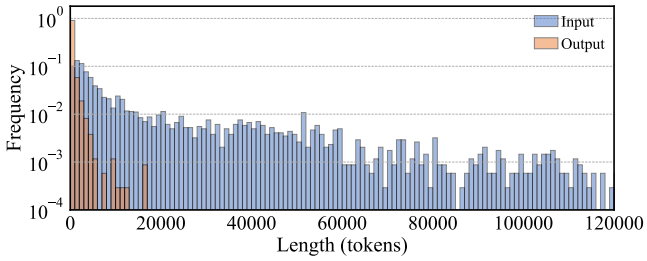
The throughput degradation in the $TP4$ scenario is caused by the extra TP communication. These results indicate that we need to carefully select parallelism strategies according to different workloads.

3.2 Dynamic Workload

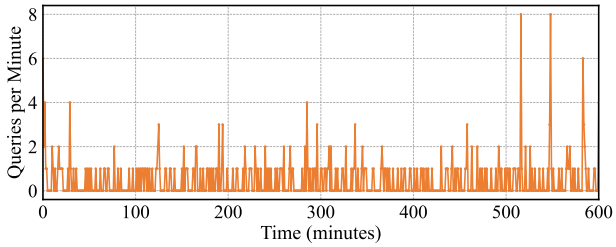
We investigate statistics related to user requests in our production deployment. Figure 2a shows the distribution of request input/output lengths for Qwen2.5-32B. While requests with short context lengths constitute the majority of the workload, the length distribution exhibits an extremely long-tail property, indicating that $TP4$ deployment is essential in production. Additionally, we analyze the arrival pattern of long requests (which exceed the maximum supported sequence length of $TP2$ deployment) in Figure 2b, demonstrating that the traffic exhibits significant dynamics.

3.3 Limitations of Existing Solution

The commonly used approach to handle dynamic workloads involves deploying instances of varying parallelisms (e.g., one $TP4$ and four $TP1$ instances on an 8-GPU host). While



(a) Input/output length distribution.



(b) Traffic pattern of long requests during 10 hours.

Figure 2. Dynamic workload in LLM serving.

this solution is used in our production, it faces critical limitations. Statistics in Figure 2b reveal that long requests occur sporadically. Therefore, reserving dedicated $TP4$ instances to accommodate these long requests is highly inefficient.

Seesaw [24] is the newest representation of a migration method based on CPU shared memory, which causes up to 41× time cost according to our evaluations (§6.2.3). KunServe [9] and LoongServe [27] are state-of-the-art solutions that try to provide dynamic parallelism based on dynamic PP/SP, respectively. However, both PP and SP are significantly inefficient (as discussed in §2). Our evaluations (§6.3) show KunServe and LoongServe cause 43.5% extra throughput degradation.

3.4 Basic Idea and Challenges

To overcome these challenges of instance-level hybrid TP deployment, we propose **cross-instance parallelism transformation**. The core idea is as follows: Under normal workloads, all instances operate with $TP1$ for optimal throughput. When a request exceeds the sequence length limit of $TP1$, multiple GPUs on the same host are dynamically aggregated into a $TP2$ or $TP4$ instance (Figure 3). By releasing memory used for model weights to accommodate more KV cache, we can handle longer requests. After serving these long requests, the system can revert to the normal state to maximize overall throughput. While the basic idea is concise, a series of challenges must be addressed in the detailed design.

Challenge-1: Significant peak memory usage during parallelism transformation. In inference serving, model

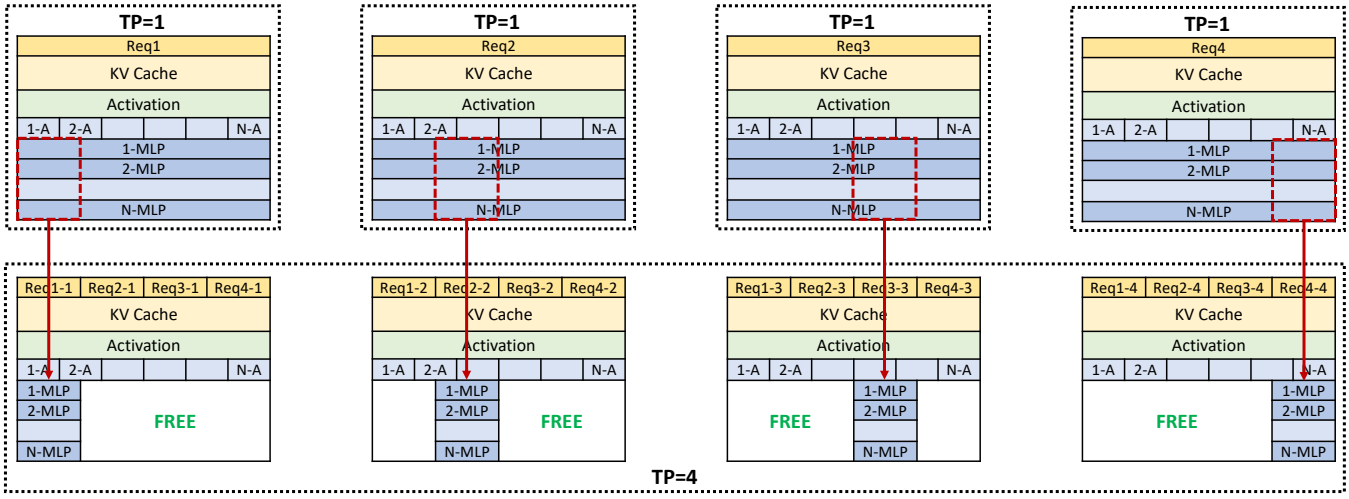


Figure 3. Parallelism transformation from $TP1$ to $TP4$.

weights are predetermined and loaded into GPU device memory as a single contiguous allocation. Consequently, mainstream inference engines (e.g., vLLM [8], SGLang [7]) statically reserve a dedicated memory region for them. Crucially, mainstream GPU programming toolkits [2, 3] lack native support for repartitioning memory allocations that have been committed to fixed patterns.

When performing parallelism transformations (e.g., transitioning from $4\times(TP1)$ to $TP4$), the only viable approach within the current implementation involves: (1) allocating an additional memory block sized at 25% of the model weights, (2) copying the required weight segments to this new allocation, and (3) releasing the original weight memory block. Using Qwen2.5-32B as a representative example, this transformation requires an additional 15.58 GB of GPU memory. Even worse, when transitioning from $TP4$ to $4\times(TP1)$, each worker must maintain 62.34 GB (64.9% of total GPU memory) of free space. This substantial memory overhead directly conflicts with our main objective.

Challenge-2: Performance degradation during parallelism transformation. Specifically, when transitioning from $4\times(TP1)$ to $TP4$, the entire set of KV cache needs to be transferred across GPUs and repositioned in the target device memory. Our unit tests reveal that this transformation requires between 522 ms (using 78 SMs) and 2240 ms (using a single SM) for Qwen2.5-32B. If service execution is suspended until the transformation is complete, it could lead to unacceptable request timeouts.

Challenge-3: Minimizing throughput loss under non-deterministic workloads. While the first two challenges focus on parallelism transformation itself, the entire cluster faces an additional issue. Current request scheduling mechanisms are unaware of parallelism transformation requirements. This results in two detrimental problems: (1)

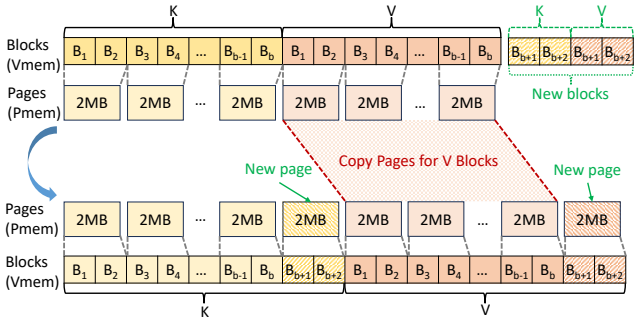


Figure 4. KV cache management with pages.

Long requests are uniformly distributed across service instances, triggering unnecessary parallelism transformations on multiple hosts. (2) Suboptimal request scheduling forces individual instances to frequently oscillate among different parallelism configurations. These inefficiencies result in substantial throughput degradation across the cluster.

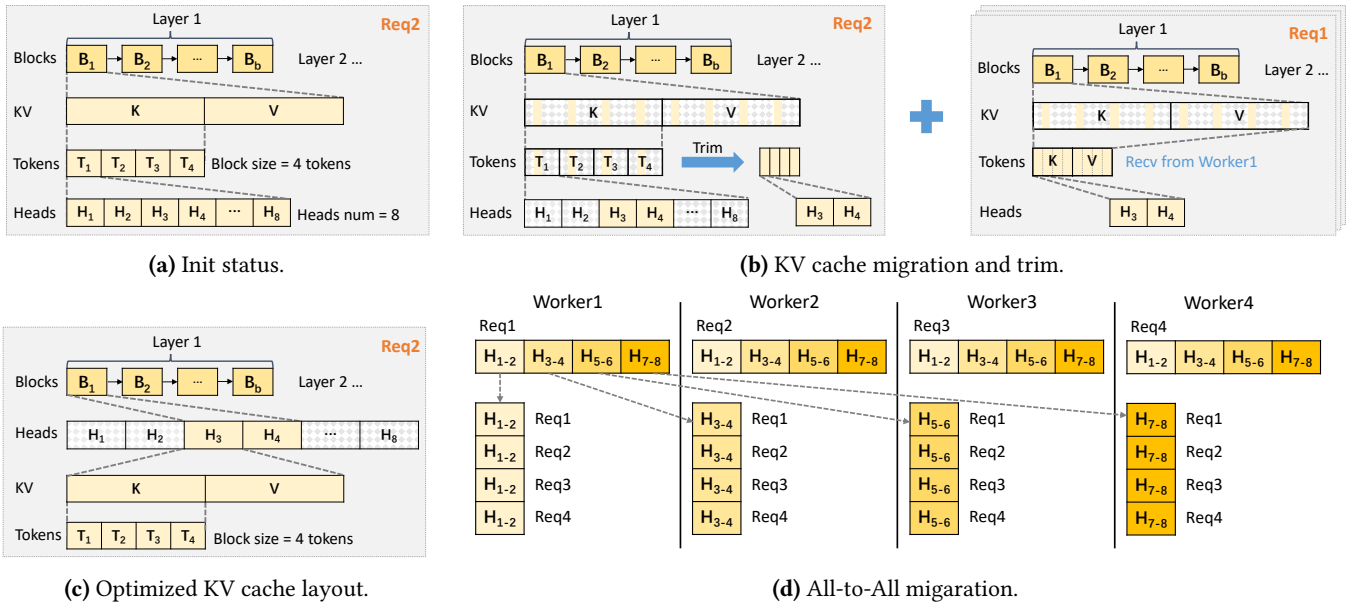
We address **Challenge-1** and **Challenge-2** via cross-instance parallelism transformation (§4) and resolve **Challenge-3** with a transformation-aware scheduler (§5).

4 Cross-Instance Parallelism Transformation

This section presents a detailed design for the transformation of KV cache (§4.1) and model weights (§4.2), respectively. Furthermore, we minimize the transformation overhead through layer-by-layer hybrid parallelism transformation (§4.3).

4.1 Transformation of KV Cache

4.1.1 Page-wise memory management. To reduce peak memory consumption during transformation (**Challenge-1**), the critical cornerstone is managing KV cache storage at a



(a) Init status.

(b) KV cache migration and trim.

(c) Optimized KV cache layout.

(d) All-to-All migaration.

Figure 5. KV cache migration solutions.

fine granularity rather than a whole bulk. The most closely related work is vAttention [21], which proposes page-based virtualization for KV cache management. However, in the context of parallelism transformation, we need not only to dynamically manage KV cache pages but also to **support efficient migration of these pages among different GPUs**. Achieving this efficient migration is non-trivial.

The primary challenge we identify is the incompatibility between page-wise memory management and the standard KV cache layout in mainstream inference engines [7, 8]. While page-wise management allows incremental expansion of the KV cache by appending new pages for newly generated tokens, it introduces significant overhead due to the conventional requirement that K and V values must reside in contiguous blocks. As illustrated in Figure 4, adding each new page necessitates additional shifting operations (via copy or memory unmap+remap operations) to maintain contiguous allocation of the entire K and V.

Page-friendly KV cache layout. While classic mitigation strategies (e.g., copy acceleration or overlap techniques) can help alleviate side effects, we aim to eradicate the root cause through architectural redesign. The fundamental issue stems from the limitations of KV cache management, specifically the legacy requirement for contiguous memory blocks. To preserve compatibility with existing engine implementations and minimize extensive modifications, we reorganize the hierarchical relationship between K/V tensors and memory blocks. Our key innovation is arranging KV cache for consecutive tokens to occupy adjacent blocks, aligning with the page allocation patterns. Thus, this design could eliminate the shifting overhead.

The additional challenge lies in the mismatch between this new layout and the attention kernel's input, Specifically, any change in the attention kernel's input requires significant modifications to its implementation. We address this by implementing a general `kv_stride_order()` function that automatically maps between different layouts. We then use `permute(*stride_order)` to reshape the stored KV cache to the expected layout. Thus, from the attention kernel's perspective, the input remains unchanged, eliminating the need for further modifications.

4.1.2 KV cache migration. In this part, we illustrate the key aspects of KV cache migration during parallelism transformation. Figure 5 shows a representative example. Each worker starts at $TP1$, and maintains the full KV cache for requests it serves, referred to as local requests (Figure 5a). When transforming from $4 \times (TP1)$ to $TP4$, for each layer, the KV cache for each token must be divided and distributed among workers according to the number of headers in MHA. As illustrated in Figure 5b, this method results in a KV cache that is "full of holes", creating significant challenges for reusing the released space.

Basic solution: migration and trim. The straightforward solution involves executing KV cache migration followed by trimming local KV cache to a more compact style. For locally stored token, worker W_i selectively remains headers in the range $(H/TP)*(i-1)+1 \sim (H/TP)*(i-1)+H/TP$, where H is the number of headers in MHA and TP is the target TP configuration size. Each worker sends the remaining headers to the corresponding workers. In Figure 5b, assuming $H=8$ and a $TP4$ configuration, W_2 retains $H3$ and $H4$, and sends $(H1+H2)/(H5+H6)/(H7+H8)$ to $W_1/W_3/W_4$, respectively.

Simultaneously, each worker prepares reserved pages to store the KV cache received from remote requests.

In addition, since the KV cache for local requests (e.g., *Req2*) is "full of holes", we need to further trim these KV cache, to make the unused memory really released. This trimming operation requires extensive copying, which significantly affects performance during transformation. Our evaluations show that this leads to a 12× increase in extra memory usage and 2.6× increase in extra processing time.

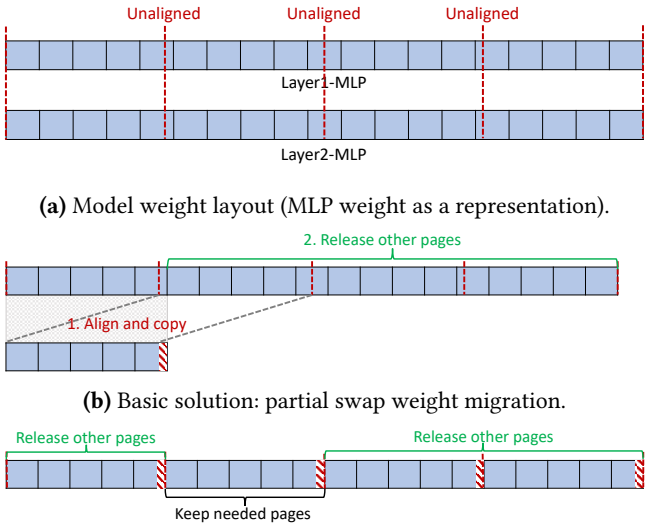
Optimized solution: in-place migration with header-centric KV cache layout. The ideal scheme would allow for direct reuse of freed memory space without additional trimming. We identify that the current generation of non-contiguous memory spaces arises from the token-first KV cache layout (i.e., token-level contiguous storage of different K/V pairs). Since increasing TP configuration partitions attention heads across workers (e.g., 8 heads evenly distributed across 4 workers in *TP4*), we propose a *header-centric layout* that organizes K/V storage in the order of [Block, Header, K/V, Token]. As illustrated in Figure 5c, this reorganization allows each block to generate compact memory segments during migration (Figure 5d), allowing unused portions to be directly repurposed through block reshaping.

While trimming eventually reduces the size of the local KV cache, peak memory usage before trimming remains significant (Figure 5b). So, can we prioritize the GPU memory regions released by local requests to store remote KV cache from other workers? With the header-centric KV cache layout, the freed memory space from the local KV cache can be directly reused. To leverage this characteristic, we introduce *phased KV cache migration*.

The core idea involves decomposing KV cache migration into multiple all-to-all communication stages (Figure 5d). In each stage, alongside data transfer for KV migration, workers also exchange metadata about GPU memory addresses that will become available once the stage is finished. This metadata exchange enables workers to immediately utilize the freed memory space in subsequent communication stages, significantly reducing peak memory overhead.

Table 2 concludes our two main contributions in KV cache transformation. The page-friendly layout avoids unnecessary block shifts when adding new pages. The extension to a page-friendly header-centric layout further minimizes memory trimming in the KV cache parallelism transformation.

Overlapping. In KV cache transformation, main GPU driver calls are `cuMemUnmap`, `cuMemMap` and `cuMemSetAccess`. All of them are GPU driver functions that can run in parallel with GPU kernels. All-to-all communication needs GPU SMs to achieve good performance, which would conflict with normal kernel executions. Our strategy is to launch all-to-all on an independent communication stream, which will actually be executed when enough free SMs are available.



(a) Model weight layout (MLP weight as a representation).

(b) Basic solution: partial swap weight migration.

(c) Optimized solution: in-place migration with padding.

Figure 6. Model weight layout and migration solution.

4.2 Transformation of Model Weights

There is a significant difference between typical dynamic KV cache management and memory management during parallelism transformation. As mentioned in §2, our focus is on transforming MLP weights, which constitute a major component of the overall model weight (88%), while keeping other weights duplicated for implementation simplicity. In Figure 6a, we illustrate the need to vertically partition the originally contiguous weight memory of each transformer layer (e.g., equally dividing it into four parts). The fundamental challenge arises from CUDA's native memory management, which enforces a minimum allocation unit of 2 MB [1]. Direct partitioning would inevitably lead to fragmented memory pages that cannot be efficiently utilized (Figure 6a). Our analysis of several representative open-source models, as shown in Table 3, reveals that more than half of the models encounter this fragmentation issue.

Basic solution: partial swap for model weight. To handle this problem, a basic solution is to conduct GPU driver operation `cudaMemcpy` or employ a customized GPU kernel to swap unaligned model weights for aligned ones (Figure 6b). However, this procedure results in extra weight copying.

Optimized solution: in-place migration with weight padding. In parallelism transformation, performing any additional memory copies or movements on model weights leads to substantial time overhead. However, our analysis (Table 3) reveals that significant copying and shifting cost is triggered by small alignment deviations (e.g., less than 0.7% of the total weight size).

This observation motivates our *amortization-based optimization strategy*: by introducing small padding to the

Table 2. Benefits from different KV cache layout optimizations.

KV Cache Layout	Detailed Hierarchy	Benefits
Raw layout	[K/V, Block, Token, Header]	–
Page-friendly layout	[Block, K/V, Token, Header]	Eliminating memory shifting when adding new pages. $O(\#KV\ cache\ pages) \rightarrow 0$
Page-friendly header-centric layout	[Block, Header, K/V, Token]	Minimizing memory trimming in parallelism transformation. $O(\#Local\ tokens) \rightarrow O(1)$

Table 3. MLP weight size in different models. Decimals mean unaligned placements of tensors.

Model	Model Structure [$Hidden\_Size, Inter\_Size, \#Experts$]	#Pages Per Tensor ($TP1$)	#Pages Per Tensor ($TP4$)
GPT-OSS-120B	[2880, 2880, 128]	1012.5 / 2025	253.125 / 506.25
GPT-OSS-20B	[2880, 2880, 32]	253.125 / 506.25	63.28125 / 126.5625
Llama-3.1-70B	[8192, 28672, -]	224	56
Qwen2.5-32B	[5120, 27648, -]	135	33.75

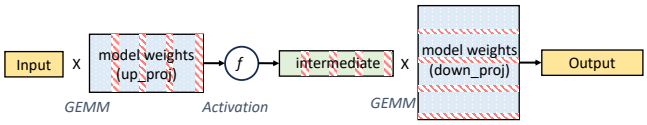


Figure 7. FFN workflow with padding.

weights, we can eliminate expensive runtime redistribution during transformation. Specifically, we proactively add padding at potential partitioning boundaries to ensure that the subsequent weights align with CUDA allocation granularity. Since the set of possible TP configurations (e.g., $TP1/2/4$) is fixed for a given model, these partitioning boundaries can be predetermined during model loading. With this preset padding, parallelism transformation can directly release unused pages, thereby completely eliminating the weight copying overhead identified in **Challenge-2**.

With padding applied, the next challenge is to enable MLP computations that use padded weights as inputs.

Padding-compatible MLP computation. As shown in Figure 1b, the implementation of MLP consists of an FFN and a linear operation. We further dive into the FFN implementation, illustrated in Figure 7. The raw MLP calculation can be presented as follows:

$$FFN = f(I \times U) \times D \tag{1}$$

where I is the input tensor, U is the `up_proj`, D is the `down_proj`, and $f()$ is the activation function.

We introduce even column-wise padding in U and row-wise padding in D. The modified $U' = [U_1, 0, U_2, 0, U_3, 0, U_4, 0]$, and $D' = [D_1^T, 0, D_2^T, 0, D_3^T, 0, D_4^T, 0]^T$. The padding-compatible computation is denoted as FFN'. We employ a series of equivalent variations to demonstrate that FFN' is equal to FFN.

$$\begin{aligned} FFN' &= f(I \times U') \times D' \\ &= f([I \times U_1, 0, I \times U_2, 0, I \times U_3, 0, I \times U_4, 0]) \times D' \\ &= [f(I \times U_1), 0, f(I \times U_2), 0, f(I \times U_3), 0, f(I \times U_4), 0] \\ &\times [D_1^T, 0, D_2^T, 0, D_3^T, 0, D_4^T, 0]^T \\ &= f(I \times U) \times D = FFN \end{aligned} \tag{2}$$

With this padding approach, while the intermediate tensor is extended, the final result remains the same as the raw FFN output. This allows us to maintain consistent results without introducing any further tensor trimming or reshaping.

Overlapping. During the parallelism scale-up, each worker only needs to unleash parts of pages, which can be completely overlapped. During the parallelism scale-down, all-to-all communication is needed. Similarly, we utilize an independent communication stream to schedule its launch when enough SMs are available, thereby minimizing overhead.

4.3 Hybrid Parallelism Transformation

Through an analysis of TP execution patterns, we find that the entire model does not execute fully in parallel. At the end of each computation module (MHA and MLP), a TP group must perform an all-reduce to aggregate computation results across workers before proceeding to the next execution stage. This indicates that the finest granularity of TP execution and parallelism transformation operates at the computation module level.

MLP-first transformation. During scale-up, the KV cache in each MHA requires shuffling across workers. Assuming a balanced load on different workers, parallelism scale-up does not introduce additional GPU memory usage. In contrast, there is significant memory release in MLP weights during

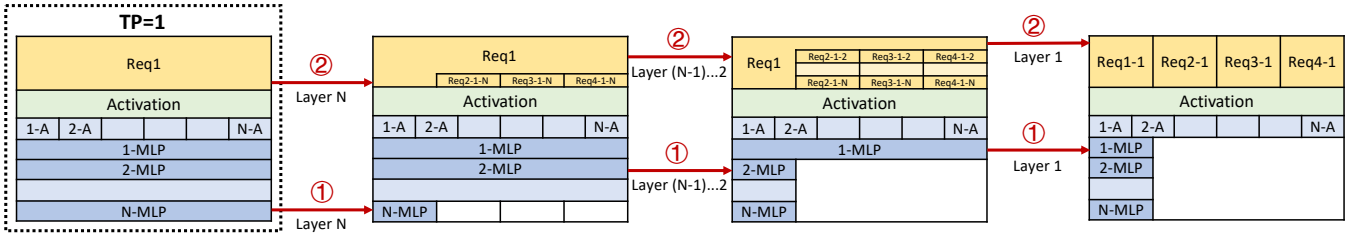


Figure 8. Layer-staggered transformation.

Algorithm 1 schedule_request

```
1:  function SCHEDULE_REQUEST(request)
2:      t_load ← MAX; t_instance ← NULL
3:      for all instance do
4:          if no_long_req() then
5:              #Long-context-aware scheduling
6:              res ← check_reserve(instance)
7:              if res == True then
8:                  continue
9:          check_and_update(instance, t_load, t_instance)
10:     if valid(t_load, t_instance) then
11:         #Directly serve current request
12:         return target_instance
13:     else
14:         #Parallelism scale-up
15:         instance = execute_scale_up()
16:         return instance
```

Algorithm 2 schedule_parallelism

```
1:  function SCHEDULE_PARALLELISM(instance_id)
2:      cur_tp_size ← get_tp_size(instance_id)
3:      if cur_tp_size > 1 and no_long_req() then
4:          cur_load ← get_load(instance_id)
5:          #Safe parallelism scale-down
6:          if cur_load < THRESHOLD then
7:              instances = execute_scale_down(instance_id)
8:              update_instance_list(instances)
9:      update_reserve()
```

scale-up, which is essential for supporting longer request lengths. Therefore, our strategy prioritizes completing MLP transformations from $TP1$ to $TP4$ as early as possible. As shown in Figure 8, the MLP transformation is executed first ①, followed by the KV cache transformation ②.

Layer-staggered transformation. On the other hand, during scale-down, MLP weights see a considerable increase in memory consumption. To minimize sudden spikes in memory allocation pressure, we stagger MLP transformations across layers, as shown in Figure 8. This manner also splits the entire transformation procedure into multiple independent components, thus minimizing overall side effects.

Reversed transformation. We start from the last layer and traverse the model architecture in reverse order to the first layer. This design enables active requests to continue operating under the previous parallelism configuration until they reach the transformation boundary, requiring only one switch in parallelism to complete the following layers. Moreover, by independently controlling the transformation timing for each layer, we maximize the overlap between the parallelism adaptation process and ongoing computations.

5 Transformation-Aware Scheduler

With the foundation established in §4, our system can handle dynamic request arrivals using fixed GPU resources. However, if the request scheduler remains unchanged (e.g., round-robin or load-minimum-first scheduling), it can easily cause frequent parallelism transformations within the cluster, causing significant performance degradation. Therefore, a parallelism-aware global scheduler is necessary to actively manage request distribution and instance configuration transitions. When designing the scheduler, several key factors must be considered:

Long sequence requests. The primary trigger for scaling up parallelism is the need to serve long sequence requests. As shown in Figure 2a, the sequence length is primarily determined by the input, with the output contributing only 10.3% to the total length. Thus, the scheduler must consider input length when making routing decisions.

Throughput degradation after parallelism scale-up. As indicated in Table 1, under stable workload, scaling up to $TP4$ results in a 57% decrease in overall throughput compared to $4\times(TP1)$. The scheduler must take this performance impact into account when making transformation decisions.

Timing for parallelism scale-down. While larger parallelism increases KV cache capacity, if the cache is consistently full, instances cannot execute parallelism scale-down without risking out-of-memory errors. This results in suboptimal resource utilization, as the system cannot revert to higher-throughput mode. Based on these considerations, we designed a scheduler that collaboratively manages request scheduling and parallelism transformation, with its pseudocode shown in Algorithm 1 and Algorithm 2.

Upon request arrival (Algorithm 1), the scheduler first identifies candidate instances capable of serving the request

Table 4. Details of selected models.

Model	Weights Size	GPU Type
Llama2-7B (BF16)	15.67 GB	A100 (40GB)
Llama3-8B (BF16)	16.66 GB	A100 (40GB)
Qwen2.5-32B (BF16)	62.34 GB	H20 (96GB)
Qwen3-32B (BF16)	62.34 GB	H20 (96GB)

based on its input length and the current load of each instance. Rather than simply selecting the least loaded instance, our scheduler evaluates all potential transformation scenarios, calculates the expected performance after scaling up parallelism, and reserves sufficient resources to complete the transformation if necessary. This ensures service continuity during transformation periods. When consecutive long requests occur, the scheduler prioritizes instances already operating in higher TP configurations to minimize the number of required transformations.

As indicated in Algorithm 2, once long requests are completed and if any instances with $TP > 1$ remain, the scheduler reduces the request rate to these instances to facilitate scaling down. When sufficient cache space is available, the scheduler will proactively initiate a parallelism scale-down to maximize overall system throughput.

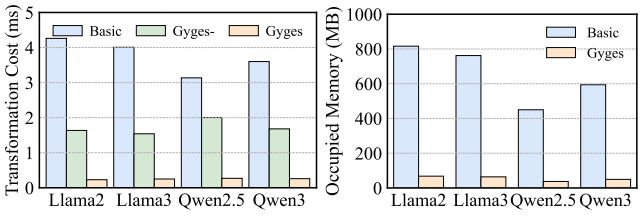
6 Evaluation

In this section, we first conduct a series of microbenchmarks to validate the effectiveness of KV cache transformation (§6.2.1), model weights transformation (§6.2.2), overall cost (§6.2.3) and transformation-aware scheduler (§6.2.4), respectively. Furthermore, we conduct comprehensive end-to-end experiments, with real-world traces, to evaluate the performance of Gyges compared with other state-of-the-art alternatives (§6.3).

6.1 Evaluation Settings

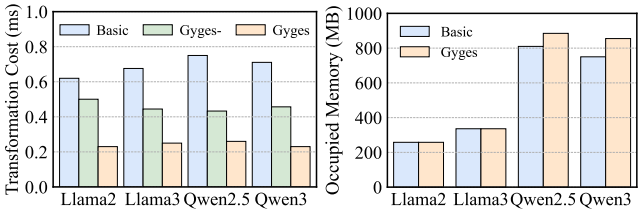
Testbed. In our testbed, we have two types of nodes (H20 instances and A100 instances). Each H20 instance is equipped with eight NVIDIA H20 (96GB) GPUs connected via NVLINK, 2 TB of DDR5 memory, and 128 Intel Xeon Platinum 8469C CPUs. Each A100 instance is equipped with eight NVIDIA A100 (40GB) GPUs connected via NVLINK, 2 TB of DDR5 memory, and 128 Intel Xeon Platinum 8369B CPUs.

Models. We employ a mainstream LLMs from Qwen [6] and Llama [4] families. The detailed information of selected models is shown in Table 4. The model size ranges from 7B to 32B, which is the dominant and representative size deployed in practical model inference scenarios. The selection of GPU instance type for serving each model aligns with the online configuration. The fundamental selection strategy is to ensure that a single GPU can accommodate the entire model to achieve optimal performance.



(a) Transformation cost. **(b)** Occupied memory.

Figure 9. KV cache transformation.



(a) Transformation cost. **(b)** Occupied memory.

Figure 10. Model weights transformation.

6.2 Microbenchmark

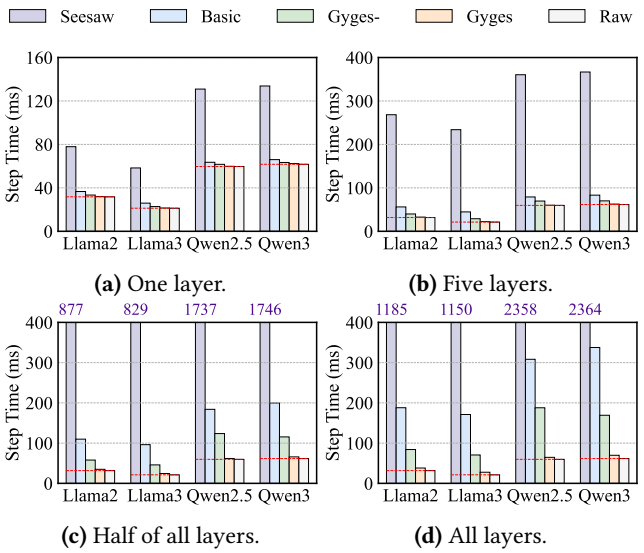
6.2.1 KV cache transformation. In this part, we demonstrate the effectiveness of KV cache transformation. To clearly compare different methods, we focus on the procedure of a single KV cache transformation. Basic is the basic KV transformation solution presented in §4.1.2. We use the $4\times$ $(TP1) \rightarrow TP4$ as a representative example, which is also the most important transformation for supporting the occurrence of long context requests. Considering that in this scale-up transformation, the KV cache utilization would already be very high, we set the overall KV cache utilization to be 90% in these experiments.

Transformation time. We test the time cost of KV cache transformation with different solutions. Basic solution introduces 3.15 - 4 ms extra time cost while serving different LLMs, while Gyges- (without overlapping) decreases it by up to 61%. These results show the effectiveness of the page-friendly header-centric KV cache layout. With overlapping, Gyges further decreases the cost by 86%.

GPU memory saving. We further quantify the GPU memory cost with different solutions. The results are shown in Figure 9b. The memory utilized by PT is 91.6% lower than that of the Basic solution. With Gyges, we consistently maintain additional memory usage below 70 MB, allowing for transformation even under extremely high load scenarios.

6.2.2 Model weights transformation. Similarly, we illustrate the performance and overhead of different solutions under a single time of model weights transformation. Partial Swap presented in §4.2 represents the basic solution.

Transformation time. The transformation time per layer is shown in Figure 10a, evaluated on different LLMs. The



(a) One layer. **(b)** Five layers. **(c)** Half of all layers. **(d)** All layers.

Figure 11. Overall transformation cost.

time of Partial Swap varies from 611 ms to 696 ms, which is mainly caused by the extra migration. With the weights padding mechanism, Gyges- completely eliminates unnecessary memory operations, decreasing the transformation cost by 18.9% - 42.2%. With further overlapping, Gyges decreases the cost by up to 67.6% compared with the Basic solution.

Padding overhead. The side-effect introduced by Gyges is that we need extra padding in model weights. Therefore, we further evaluate the padding overhead. We record the memory occupied for model weights in different LLMs, and the results are shown in Figure 10b. In various LLMs, the memory overhead ranges from 0% to 14%. Furthermore, we record the corresponding computation time of FFN, before and after padding, for different models. The extra computation cost is negligible (<0.1%), validating the effectiveness of our computation-friendly padding.

6.2.3 Overall transformation cost.

We further evaluate the performance of Gyges under practical execution conditions. The number of layers transformed in each inference step is progressively increased from one to the total number of layers in various models. We compare our method with Seesaw [24], Basic, and Gyges- (the version of Gyges without overlapping). The results are presented in Figure 11. Raw refers to the original step time without any transformations. As the number of transformed layers increases, Gyges consistently maintains an overhead of less than 1%. In scenarios where we aim to transform all layers in a single step, Gyges reduces the extra cost by 97.2% compared to Seesaw.

6.2.4 Transformation-aware scheduler.

We compare our transformation-aware scheduler with several widely used global scheduling strategies: (1) the Round-Robin (RR) scheduler, which distributes each new request to instances in a round-robin manner; (2) the Least-Load-First (LLF) scheduler, which directs each new request to the instance with the least load. When an instance is unable to handle a new incoming long request, it collaborates with neighboring instances to implement a scale-up parallelism transformation.

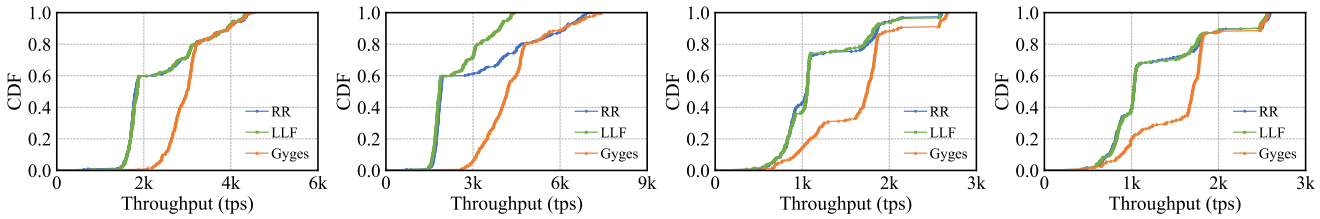
In these experiments, we establish a hybrid workload consisting of both short and long requests. Short requests (1K input length) arrive at a rate of 60 queries per minute, creating the foundational background traffic. Long requests (50K input length) arrive at a rate of one query per minute, consistent with the general observations in Figure 2b. At initialization, we deploy 8 $TP1$ instances to handle the overall workload. We conduct experiments using different models. Figure 12a-12d shows the overall system throughput. Gyges improves average throughput by 26.1%-39.2% compared to both the RR and LLF. To highlight the key differences in scheduling, Figure 13 captures a representative duration from this experiment. At the 120-second time, a $TP4$ instance and $4 \times (TP1)$ instances exist, and a new long request arrives. Since the $TP4$ instance is already serving a long request, it experiences a heavier load. As a result, both RR and LLF are more likely to assign the new long request to a $TP1$ instance, prompting another parallelism scale-up and leading to significant performance degradation. In contrast, Gyges correctly schedules long requests to the existing $TP4$ instance.

6.3 End-to-End Performance

We further use the real trace captured in production to evaluate the end-to-end performance of Gyges. We do not include Seesaw [24] in this evaluation due to its unsatisfactory performance. KunServe [9] and LoongServe [27] are further employed as our main comparisons. We record throughput, TTFT, and TPOT. Owing to limited space, we select Qwen2.5-32B as a representation, and the results are shown in Figure 14. Compared with alternatives, Gyges increases throughput by 1.75×-6.57×, and TTFT and TPOT are decreased by up to 53% and 74%, respectively. One important gain contributor is that our main design choice (leveraging TP transformation) delivers better performance than PP/SP transformation. Overlapping optimization further decreases 26.7% TTFT. This optimization plays an important role under the 0.6 QPS load to keep TTFT lower than 10 s. It enables the system to serve more requests without violating SLO.

7 Related Work

Serving long sequence requests. Infinite-LLM [18] serves long requests through managing a distributed pooled GPU memory. It, however, introduces frequent and non-neligible cross-instance communication. LoongServe [27] and KunServe [9] are the most relevant and state-of-the-art work that proposes dynamic SP and PP. However, intrinsically, both SP and PP cannot fully utilize the computational resources of GPUs, leading to significant performance degradation according to our evaluations.



(a) Llama2. **(b)** Llama3. **(c)** Qwen2.5. **(d)** Qwen3.

Figure 12. Performance with different scheduling strategies.

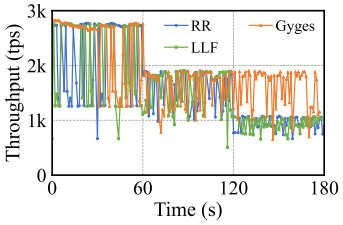


Figure 13. TPS trends.

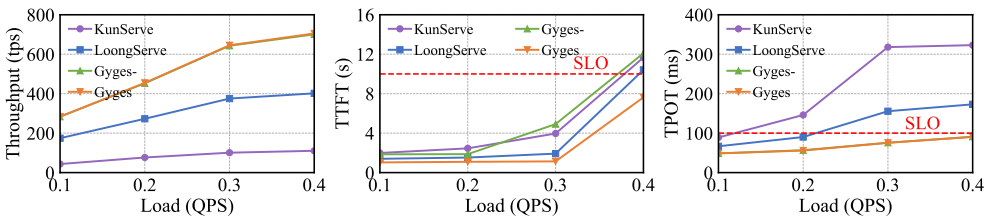


Figure 14. End-to-end performance on throughput, TTFT and TPOT.

Inference performance optimization. There are many efforts devoted to optimizing the performance of individual inference instances, including kernel/cache/batch/procedure optimizations [13, 15, 16, 19, 23, 26, 28, 31] and disaggregated serving [14, 20, 32]. These solutions focus on delivering extreme performance with a static parallelism configuration and are orthogonal to parallelism transformation; they can be utilized alongside our solution.

Request and resource scheduling. A series of schedulers have been proposed in both DNN model serving and LLM serving scenarios [5, 11, 12, 17, 22, 25, 28–30]. However, these solutions schedule requests and resources based on the static parallelism configuration of all instances.

8 Conclusion

To handle the dynamic context lengths and request arrival patterns in LLM serving, we design Gyges. It proposes (1) a page-friendly, header-centric KV cache layout; (2) dedicated weight padding; and (3) a transformation-aware scheduler to comprehensively optimize the performance of serving instances under dynamic workloads. Evaluations using real-world traces show that Gyges improves throughput by 1.75×-6.57× compared to state-of-the-art solutions.

References

[1] 2024. Virtual memory management minimum granularity. https://forums.developer.nvidia.com/t/virtual-memory-management-minimum-granularity/268699. (2024).

[2] 2025. AMD ROCM Software. https://www.amd.com/en/products/software/rocm.html. (2025).

[3] 2025. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. (2025).

[4] 2025. Llama: Industry Leading, Open-Source AI. https://www.llama.com/. (2025).

[5] 2025. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. (2025).

[6] 2025. Qwen: Qwickly forging AGI, enhancing intelligence. https://qwenlm.github.io/. (2025).

[7] 2025. SGLang is a fast serving framework for large language models and vision language models. https://github.com/sgl-project/sglang. (2025).

[8] 2025. Welcome to vLLM: Easy, fast, and cheap LLM serving for everyone. https://docs.vllm.ai/en/latest/. (2025).

[9] Rongxin Cheng, Yuxin Lai, Xingda Wei, Rong Chen, and Haibo Chen. 2025. KunServe: Efficient Parameter-centric Memory Management for LLM Serving. (2025). arXiv:cs.DC/2412.18169 https://arxiv.org/abs/2412.18169

[10] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. 2024. LLM-Inference-Bench: Inference Benchmarking of Large Language Models on AI Accelerators . In SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, Los Alamitos, CA, USA, 1362–1379. https://doi.org/10.1109/SCW63240.2024.00178

[11] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[12] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In 2022 USENIX Annual Technical Conference (USENIX ATC 22). USENIX Association, Carlsbad, CA, 183–198. https://www.usenix.org/conference/atc22/presentation/cui

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. (2022). arXiv:cs.LG/2205.14135 https://arxiv.org/abs/2205.14135

[14] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling. In Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25). Association for Computing Machinery, New York, NY, USA, 1283–1295. https://doi.org/10.1145/3695053.3730999

[15] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In Proceedings of Machine Learning and Systems, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 148–161. https://proceedings.mlsys.org/paper_files/paper/2024/file/5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. (2023). arXiv:cs.LG/2309.06180 https://arxiv.org/abs/2309.06180

[17] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[18] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. (2024). arXiv:cs.DC/2401.02669 https://arxiv.org/abs/2401.02669

[19] Anand Padmanabha Iyer, Mingyu Guan, Yinwei Dai, Rui Pan, Swapnil Gandhi, and Ravi Netravali. 2024. Improving DNN Inference Throughput Using Practical, Per-Input Compute Adaptation. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24). Association for Computing Machinery, New York, NY, USA, 624–639. https://doi.org/10.1145/3694715.3695978

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

[21] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 1133–1150. https://doi.org/10.1145/3669940.3707256

[22] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19). Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

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

[24] Qidong Su, Wei Zhao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, Zhanda Zhu, Kevin Song, Christina Giannoula, and Gennady Pekhimenko. 2025. Seesaw: High-throughput LLM Inference via Model Re-sharding. (2025). arXiv:cs.DC/2503.06433 https://arxiv.org/abs/2503.06433

[25] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: dynamic scheduling for large language model serving. In Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI'24). USENIX Association, USA, Article 10, 19 pages.

[26] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. LightSeq: A High Performance Inference Library for Transformers. (2021). arXiv:cs.MS/2010.13887 https://arxiv.org/abs/2010.13887

[27] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24). Association for Computing Machinery, New York, NY, USA, 640–654. https://doi.org/10.1145/3694715.3695948

[28] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Distributed Inference Serving for Large Language Models. (2024). arXiv:cs.LG/2305.05920 https://arxiv.org/abs/2305.05920

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

[30] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[31] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[32] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI'24). USENIX Association, USA, Article 11, 18 pages.