Amoeba: Runtime Tensor Parallel Transformation for LLM Inference Services

Haoyu Chen[1], Xue Li[2], Kun Qian[2,*], Yu Guan[2], Jin Zhao[1,3,*], Xin Wang[1]
[1]Fudan University China, [2]Alibaba Group China, [3] Songshan Lab China,
[*] Corresponding Authors: kunqian.qk@alibaba-inc.com, jzhao@fudan.edu.cn

Abstract

In Large Language Model (LLM) inference services, it is challenging to make a parallelism strategy configuration, to efficiently process the requests of variance context lengths. Requests of long context require high degree of parallelism to provide more memory for Key-Value (KV) Cache, while requests of short context prefer low degree of parallelism to increase concurrency, thus improving throughput.

To maintain high throughput while supporting large context lengths on demand, we propose Amoeba, a runtime Tensor Parallel (TP) transformation for online LLM inference services, which adaptively adjusts the TP of running instances to align with the dynamics of incoming requests. Evaluations using real-world traces show that Amoeba 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, 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 [10–12, 14, 16, 19, 21, 24] and global scheduling frameworks [3, 9, 13, 15, 18, 22, 23].

Finding an optimal parallel strategy is one of the challenges in LLM inference services. In production workloads, requests are of variance context length. Requests with long context length require more GPU memory for KV Cache, so parallelized inference, like Tensor Parallel (TP), becomes mandatory to provide enough memory from multiple GPUs. However, parallelized inference increases the overhead from computation partitioning and inter-worker communication, leading to GPU utilization loss (e.g., 57% reduction from TP=1 to TP=4, based on our practical measurements). As a result, requests with short context length prefer no parallelism, which can serve more requests concurrently to improve the throughput.

A naïve solution is to provide multiple LLM serving instances with different parallelism strategy configurations (e.g., 80% instances with TP=1 and 20% with TP=4). However, in production, the distribution of requests' context length is jittery and unpredictable (§2.4). Such a static TP setting compels workers to operate in an inefficient way when long-context requests are absent, or fails to serve some requests in the burst of long context requests.

Under dynamic request workloads, runtime parallel transformation is a promising technique to provide the ability to serve long-context requests while maintaining high throughput for short-context requests. KunServe extends Pipeline Parallel (PP) on demand without recomputing existing requests, and LoongServer extends Sequence Parallel (SP). Unfortunately, in productive online LLM inference services, due to serious performance degradation, only 3.2% of instances apply PP or SP. Most instances (91.7%) only apply TP, so they can not benefit from above solutions.

In this paper, we propose Amoeba, the first runtime tensor parallel transformation for online LLM inference services. Amoeba adapts the degree of TP on demand. For example, when long-context requests arrive, it merges four $TP1$ instances into a single $TP4$ instance, without recomputation of running requests. After long-context processing completes, the $TP4$ instance can be decomposed into four $TP1$ instances to maximize throughput.

Different from other parallel strategies (e.g., PP and SP), enabling runtime TP transformation is more challenging:

- *Challenge 1*: In TP transformation, KV Cache requires redistributing the KV Cache across GPUs in kv-head dimension, causing severe memory fragmentation which undermines the memory-saving benefits of the transformation. De-fragmentation introduces prohibitive overhead.
- *Challenge 2*: In TP transformation, the model weight requires the division of the weights among GPUs. Page-wise memory management uses a minimum unit of 2MB [1]. However, the weight partitions of most models do not align perfectly with this 2MB granularity. This misalignment results in additional memory allocation and data movement.
- *Challenge 3*: How to schedule the TP transformation in dynamic workload?

To solve the above challenges, we (1) design a page-friendly header-centric KV Cache layout (§3.1) that reduces memory overhead by 91.6% and time cost by 86% during KV Cache transformation, (2) by closely examining the partition and calculation patterns, we propose a parallelism-aware padding technique that allows in-place weight transformation (§3.2), and (3) we develop a TP transformation-aware scheduler that tightly integrates parallelism adaptation with request dispatching (§3.3).

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 TP parallelism transformation (§2).
- The detailed design for the KV Cache transformation, the model weight transformation, layer-by-layer optimizations and the transformation aware scheduler (§3).
- Experiments demonstrate Amoeba improving up to 6.57× throughput and decreasing 97% transformation cost compared with the state-of-the-art alternatives (§4).

2 Motivation

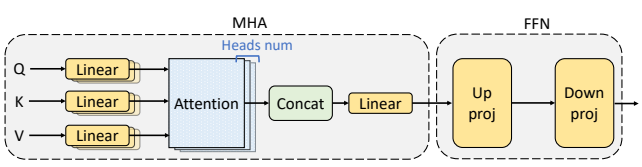


Figure 1: Transformer layer overview.

2.1 LLM Serving

An LLM inference service receives requests from remote clients and generates responses in a token-by-token way. An large language

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803989

model is constructed by a series of transformer layers, and the generation of each token requires passing through all layers. As shown in Figure 1, a typical transformer layer consists of two main parts: Multi-Head Attention (MHA)[1] and Feed-Forward Network (FFN). The MHA computation requires the context of all previous tokens. To eliminate redundant calculations, KV Cache is used to store the internal results of these prior tokens. In contrast, the FFN is primarily constructed from two General Matrix Multiplications (GEMM), which necessitate fixed-size model weights.

2.2 Supporting Long Context by Parallelism

In LLM inference services, enlarging parallelism is the dominant technique to support long context without accuracy loss, because it utilizes multiple GPUs for a single service instance (e.g., to accommodate larger KV Cache by increasing GPU memory). A series of parallelism strategies has been proposed.

The most common parallelism strategy is Tensor Parallel (TP). In each transformer layer, different headers in the MHA are distributed across various workers (each worker possesses a GPU), and the weights in the FFN 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, FFN, and KV Cache are evenly distributed across all workers.

Besides TP, other parallelism strategies are also introduced in the inference scenario. Pipeline Parallel (PP) splits different transformer layers among different workers to divide memory pressure across different GPUs. Sequence Parallel (SP) splits the input sequence into multiple parts and serves them with different GPUs. For serving Mixtures-of-Experts (MoE) models, Expert Parallel (EP) is designed to optimize the use of more GPUs.

2.3 Dilemma between Peak Throughput and Large Context

While parallelism offers increased GPU memory for supporting long context, it is not a free lunch: the overall inference throughput decreases accordingly.

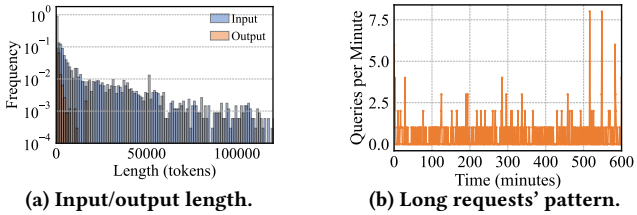
Taking TP for example, we conduct an experiment (serving Qwen2.5-32B with 4 H20 GPUs) to show the pros and cons introduced by TP. The model size of Qwen2.5-32B (BF16 datatype) is 62.34 GB, runtime activations take 14.3 GB, and the GPU memory size of H20 is 96 GB. Therefore, with 4 H20 GPUs, actually 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 context length of 3.75K, while $TP4$ can serve 32× larger. On the other hand, serving the same workloads (context 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 (SLO), e.g., Time to first token (TTFT) < 10s and Time per output token (TPOT) < 100ms. The reason is that high degree of TP requires extra TP communication.

Similarly, the throughput degradation also happens in other parallelism implementations (e.g., EP, SP, and PP). This indicates that the optimal parallelism varies for different context lengths.

[1]Group-Query Attention (GQA) is a variance of MHA, and we use MHA to refer to this family of attention mechanisms.

Table 1: Performance of different TP strategies.

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



(a) Input/output length. **(b) Long requests' pattern.**

Figure 2: Dynamic workload in LLM serving.

2.4 Existing Solutions and Limitations

Deploying instances of different parallelisms. The straightforward solution is to deploy multiple instances of different parallelisms (e.g., one $TP4$ and four $TP1$ instances on an 8-GPU host). While this solution is used in our production, it leads to inefficient utilization of GPUs, reducing the overall throughput.

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. The above results reveal that long requests occur sporadically. Therefore, reserving dedicated $TP4$ instances to accommodate these long requests is highly inefficient.

Runtime parallel transformation. KunServe [7] and LoongServe [20] provide dynamic PP and SP transformation for dynamic workloads. However, both PP and SP severely reduce GPU utilization in online LLM inference. When $PP = N$, during the serving of each request, only $1/N$ GPUs are activated in any time slot, leading to low GPU utilization [8]. As for $SP = N$, each worker hosts the entire model and more communication is required during MHA [20]. Due to this drawback, TP is the dominant parallelism strategy in production LLM inference. According to our statistics (containing thousands of instances), 91.7% of multi-GPU LLM serving instances employ TP. 3.2% of instances employ TP+EP, while none employ PP/SP. As a result, most LLM services can not benefit from these solutions.

Seesaw [17] try to migrate running requests to CPU shared memory, but memory offloading and loading cause up to 41× time cost according to our evaluations (§4.2.3).

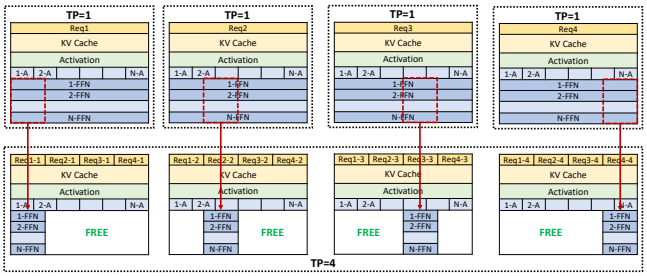


Figure 3: Parallelism transformation overview from $TP1$ to $TP4$.

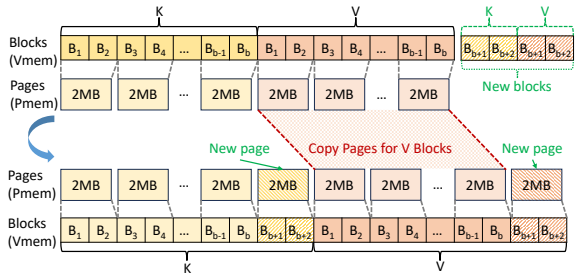


Figure 4: Expand KV Cache with page-wise memory management.

3 Amoeba Design

We present Amoeba, the first runtime tensor parallel transformation for online LLM inference. The core idea is as follows: By default, all instances operate with $TP1$ for optimal throughput. When a request exceeds the context length limit of $TP1$, multiple GPUs on the same host are dynamically aggregated into a $TP2$ or $TP4$ instance (Figure 3) to transfer memory space from FFN to KV Cache. After serving these long requests, the system can revert to $TP1$ to maximize overall throughput. Since TP is dominant in parallel strategies of LLM serving, Amoeba can be easily deployed in production.

Amoeba includes three design choices: §3.1 proposes a page-friendly header-centric KV Cache layout to reduce memory overhead and cudaMemcopy operations in KV Cache transformation (*Challenge 1* in §1). §3.2 proposes a parallelism-aware padding technique that allows in-place weight transformation to avoid weight migration (*Challenge 2*). §3.3 proposes a scheduler to control aggregation and split of TP (*Challenge 3*).

3.1 Transformation of KV Cache

Transformation of KV Cache introduces huge overload of memory management. The overload primarily consists of two parts.

The first overload lies in cudaMemcopy operations introduced by page-wise memory management. To enable flexible KV Cache expansion during transformation, we utilize page-wise memory management [1] to store the KV Cache. However, the standard KV Cache layout in mainstream inference engines [5, 6] is incompatible with flexible expansion, because it requires K and V values to reside in two contiguous blocks. As illustrated in Figure 4, adding new pages necessitates additional shifting operations to V values (via copy or memory unmap+remap operations) to maintain contiguous allocation of the entire K and V.

The second overload lies in utilizing the memory with bubbles after transformation. Figure 5 shows an example of KV Cache migration. Each worker starts at $TP1$, and maintains the full KV Cache for requests it serves, we call them *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 makes a KV Cache "full of bubbles", creating significant challenges for reusing the released memory space.

Base solution: Migration and defragmentation based on page-friendly KV Cache layout. We redisign the KV Cache architecture to a page-friendly one by reorganizing the hierarchical relationship between K/V tensors and memory blocks. The key innovation is arranging KV Cache for consecutive tokens to occupy adjacent blocks, aligning with the page allocation patterns. Thus, each new page can be directly attached to the end of the existing KV Cache with no shifting overhead.

To fully utilize memory bubbles after transformation, we utilize the released KV Cache space by de-fragmenting the local KV Cache to a more compact style after migration. During migration, worker W_i selectively remains headers in the range $(H/TP)*(i-1)+1 \sim (H/TP)*(i-1)+H/TP$ for each locally stored token, 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.

Improvement 1: In-place migration with header-centric KV Cache layout. Since the KV Cache for *local requests* (e.g., *Req*2 in Figure 5b) is full of bubbles, we need to de-fragment the KV Cache to utilize the memory. This de-fragmentation operation requires extensive copying, which significantly affects performance. Our evaluations show that this leads to 12× memory usage and 2.6× processing time. To avoid the KV Cache de-fragmentation, we find that non-contiguous memory spaces are caused by the token-first KV Cache layout (i.e., token-level contiguous storage of different K/V pairs). Since increasing TP configuration splits 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 6), allowing memory bubbles to be directly reused through block reshaping.

To enable the existing libraries to adapt to these novel KV Cache layouts, we developed a `kv_stride_order()` function for the attention kernel, which automatically maps different layouts to corresponding element offsets and strides for the kernel. Thus, from the attention kernel's perspective, the input remains unchanged, eliminating the need for further modifications.

Improvement 2: Pipelining KV Cache transformation with existing GPU kernel computations. In KV Cache transformation, main GPU driver calls are `cuMemMap` and `cuMemSetAccess`. All of them are GPU driver functions that can run in parallel with GPU kernels. KV Cache transformation also require All-to-all communication which needs GPU SMs to achieve good performance. It 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.

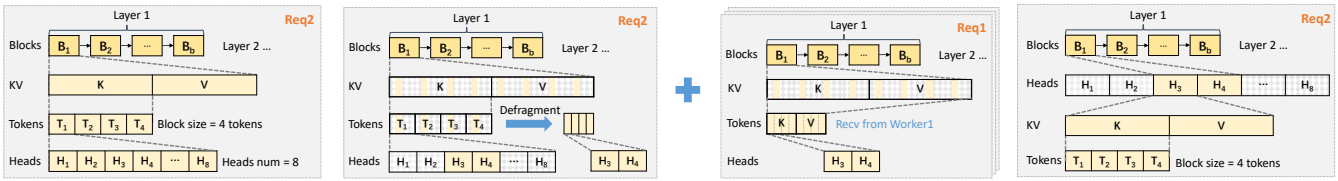
Table 2: FFN weight size in different models. Decimals mean unaligned placements of tensors.

Model	Model Structure $[H, I, \#Exp]$	#Pages/Tensor ($TP1$)	#Pages/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	63.281/126.563
Llama-3.1-70B	[8192, 28672, -]	224	56
Qwen2.5-32B	[5120, 27648, -]	135	33.75

3.2 Transformation of Model Weights

In TP transformation of model weights, we focus on FFN weights, which constitute 88% of the overall model weight, while keeping other weights duplicated for implementation simplicity.

Transforming FFN weights from $TP1$ to $TP4$ may generate fragmented memory pages which cannot be efficiently utilized. As demonstrated in Figure 7a, a blue block refers to the minimum memory allocation page of 2MB enforced by CUDA's native memory management. While the red lines indicates the vertical partition boundaries of the FFN weight during transformation. In many cases, the weight partition boundary is not in align with page boundary.



(a) Init status. (b) KV Cache migration and de-fragmentation. (c) Optimized KV Cache layout.

Figure 5: KV Cache migration solutions.

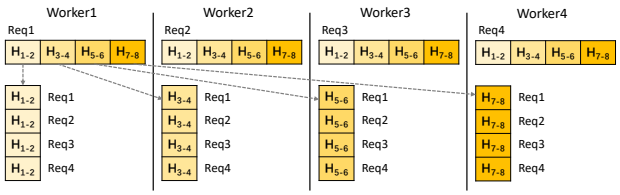
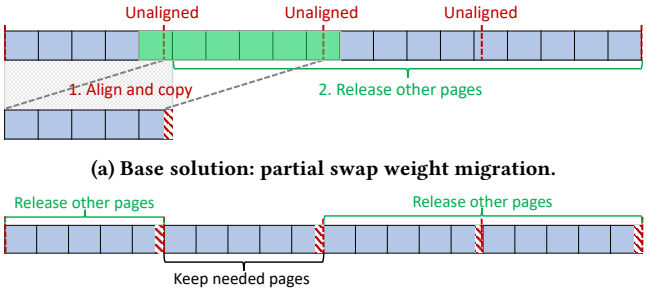


Figure 6: All-to-all KV Cache migration in head dimension.



(a) Base solution: partial swap weight migration.

(b) Improvement 1: in-place migration with padding.

Figure 7: One layer's model weight layout and migration solution.

If we directly remain the pages containing the weight partition (green part in Figure 7a) and free others. The FFN cannot even been executed as the calcultion of GEMM requires weights stored in an aligned way. Unfortunately, our analysis of several representative open-source models, as shown in Table 2, reveals that more than half of the models encounter this fragmentation issue.

Base solution: Partial swap for model weight. To handle this problem, we conduct a two phase transformation as illustrated in Figure 7a, (1) we align the needed weight partition by moving it to the beginning of current layer's weight space through GPU driver operation `cudaMemcopy` and (2) we release other unused pages through `cuMemUnmap` operation. However, the first phase results in extra weight copying that will block the CUDA stream.

Improvement 1: 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 2) 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 *pre-alignment optimization strategy*: by introducing small padding to the 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 pre-set padding, parallelism transformation can directly release unused pages, completely eliminating the weight copying overhead.

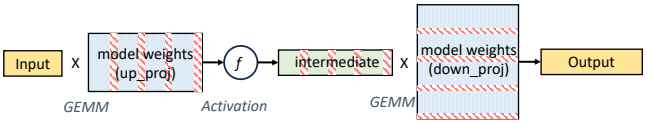


Figure 8: FFN main workflow with padding.

Now we prove that such a padding does not influence the computation result. Figure 8 illustrated the workflow of FFN implementation. The raw FFN 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 padded $U' = [U_1, 0, U_2, 0, U_3, 0, U_4, 0]$, and $D' = [D_1^T, 0, D_2^T, 0, D_3^T, 0, D_4^T, 0]^T$. After padding, the computation result FFN' is:

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

Since $FFN = FFN'$, with our proposed even column-wise padding, we do not need further tensor movement or reshaping.

Improvement 2: Pipelining CUDA Driver API Calls with computation kernels. During the parallelism scale-up, each worker only needs to free the memory pages through cuMemUnmap, which can be completely overlapped. During the parallelism scale-down, All-to-algl communication is needed. Being similar to §3.1, we utilize an independent communication stream to launch All-to-all kernels when enough SMs are available, thereby minimizing overhead.

Discussion: Supporting EP transformation. The above design can also smoothly support EP transformation with minor change. In MoE models, the FFN is stored in an expert-wise layout, allowing us to directly free unused expert weights. We only need to determine how to release weights according to the LLM types. In §4.2, we involve Qwen3-30B-A3B to evaluate EP transformation.

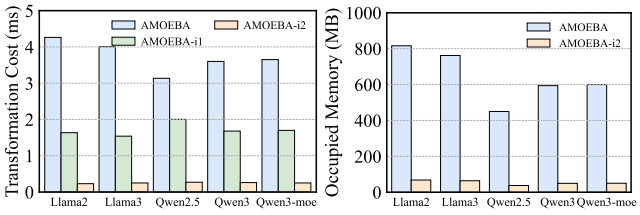
3.3 Transformation-Aware Scheduler

We present a scheduler to control whether to execute TP transformation scale-up or scale-down based on dynamic requests. We also present the order of KV Cache and FFN to make transformation.

Transformation scale-up. Once a new request arrival, the scheduler first identifies candidate instances capable of serving this request based on its input length and the current load (e.g., KV Cache capacity) of each instance. If any available instance can serve this request, this request is scheduled to the available least-loaded instance. If no available instance can support the input length, a parallelism scale-up is triggered on the least-loaded instances with same parallelism configuration. The scale-up step size is set to ×2 (e.g., from $TP1$ to $TP2$ or from $TP2$ to $TP4$), which delivers optimal performance in practice. When consecutive long requests occur,

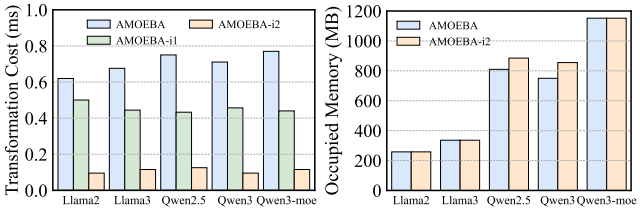
Table 3: Details of selected models.

Model	Parallelism	Weights Size	GPU Type
Llama2-7B (BF16)	TP	15.67 GB	A100 (40GB)
Llama3-8B (BF16)	TP	16.66 GB	A100 (40GB)
Qwen2.5-32B (BF16)	TP	62.34 GB	H20 (96GB)
Qwen3-32B (BF16)	TP	62.34 GB	H20 (96GB)
Qwen3-30B-A3B (FP16)	TP+EP	56.87 GB	H20 (96GB)



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

Figure 9: KV Cache transformation.



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

Figure 10: Model weights transformation.

the scheduler prioritizes instances already operating in higher parallelism degree to minimize the number of transformations.

During scale-up, Amoeba first transforms FFN to free the memory occupation, and then for KV Cache.

Transformation scale-down. Once long requests are completed and if any instances with $TP > 1$ remain, the scheduler sets these instances as temporally unavailable for scheduling. Once sufficient free KV Cache capacity is available (50% in practice), the scheduler will trigger a parallelism scale-down to maximize overall throughput. Then these instances are set back to being available for scheduling. We do not involve additional complex mechanisms for predicting the arrival of future long requests, and real-trace experiments (§4.3) validate the current scheduler is efficient in practice.

During scale-down, Amoeba first makes parallelisam transformation for KV Cache to free the memory occupation, then FFN is able to gather the entire model weights for the next decoding step.

4 Evaluation

In this section we show that:

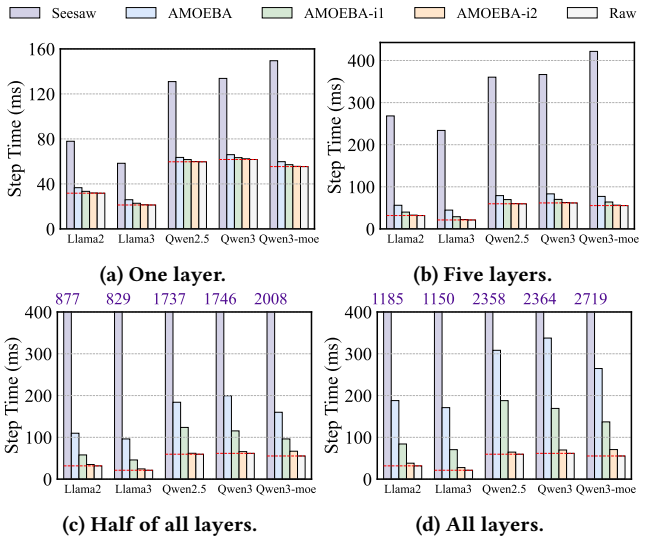
• All techniques introduced by Amoeba-i1 lead to less memory overhead and faster transformation speed (§4.2).

• In real-world request workload, Amoeba-i2 outperforms state-of-the-art solutions by 1.75x-6.57x in throughput (§4.3).

4.1 Evaluation Settings

Testbed. We have instances of two hardware configurations: H20 and A100. 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 [4] and Llama [2] families. The detailed information of selected models is shown in Table 3. The model size ranges from 7B to 32B, which is the dominant and representative size deployed in practical model inference



(a) One layer. (b) Five layers.

(c) Half of all layers. (d) All layers.

Figure 11: Overall transformation cost.

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.

Methods in Comparison. We present Amoeba, along with two improved version, Amoeba-i1 and Amoeba-i2, for evaluation. Amoeba refers to the naïve implementation of KV Cache (Migration and Defragmentation) and weight transformation (Partial Swap). Amoeba-i1 integrates our in-place migration techniques (improvement 1 in §3.2 and §3.3). Amoeba-i2 further integrates the pipelining techniques (improvement 2 in §3.2 and §3.3). KunServe [7] and LoongServe [20] are also included in end-to-end performance comparison as the state-of-the art methods. As KunServe has not open sourced yet, we reproduce it with the online parameter dropping and Network-based KV Cache exchange proposed in the paper.

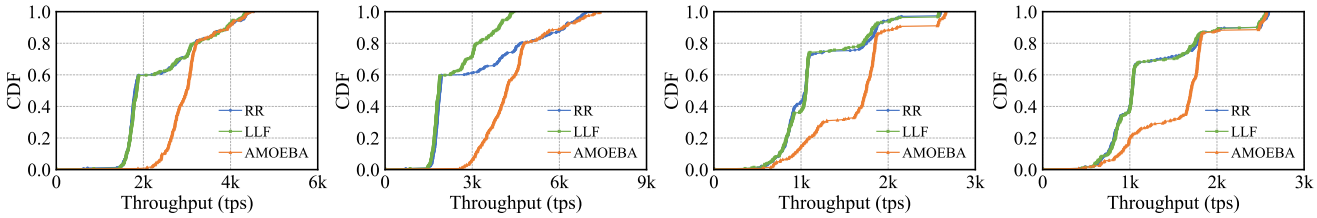
4.2 Microbenchmark

4.2.1 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. We use the $4\times(TP1) \rightarrow TP4$ as a representative example, which is also the most common transformation for supporting the occurrence of long context requests. Specifically, for the MoE model (Qwen3-30B-A3B), its transformation is $4 \times (TP1EP1) \rightarrow TP4EP4$. Considering that in this scale-up transformation, the KV Cache already reaches high utilization, 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 in Figure 9a. Amoeba introduces 3.15 - 4 ms extra time cost while serving different LLMs, while Amoeba-i1 decreases it by up to 61%. These results show the effectiveness of the page-friendly header-centric KV Cache layout. With pipelining, Amoeba-i2 further decreases the cost by 86%.

GPU memory saving. We further quantify the GPU memory cost and the results are shown in Figure 9b. The memory utilized by Amoeba-i1 is 91.6% lower than that of Amoeba. With Amoeba-i1, we consistently maintain additional memory usage below 70 MB, allowing for transformation even under high load scenarios.

4.2.2 Model weights transformation. Similarly, we illustrate the performance and overhead of different solutions under a single time of model weights transformation.



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

Figure 12: Throughput performance with different scheduling strategies on four distinct models.

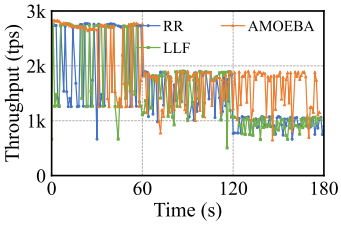


Figure 13: Throughput Trends

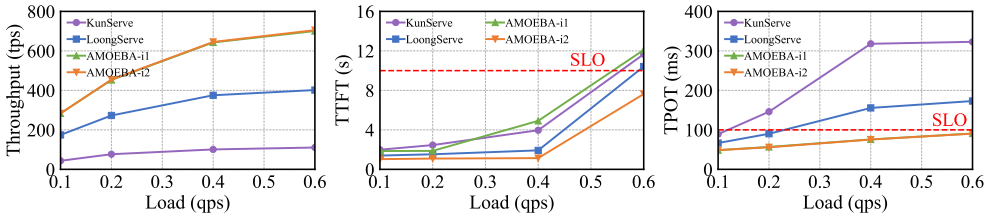


Figure 14: End-to-end performance comparison on throughput, TTFT and TPOT.

Transformation time. The transformation time per layer is shown in Figure 10a, evaluated on different LLMs. The time of Amoeba varies from 620 ms to 770 ms, which is mainly caused by the extra migration. With the weights padding mechanism, Amoeba-i1 completely eliminates unnecessary memory operations, decreasing the cost by 18.9% - 42.2%. With further overlapping, Amoeba-i2 decreases the cost by up to 85% compared with the Amoeba.

Padding overhead. We evaluate the extra memory cost and the effect on GEMM computation speed of the extra padding technique in Amoeba-i1 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 the computation-friendly padding in Amoeba-i2.

4.2.3 Overall transformation cost. We further evaluate the performance of Amoeba-i1 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 make a comparison between Seesaw [17], Amoeba, Amoeba-i1, and Amoeba-i2. The results are presented in Figure 11. Raw refers to the original step time without any transformations. As the number of transformed layers increases, Amoeba-i1 consistently maintains an overhead of less than 1%. In scenarios where we aim to transform all layers in a single step, Amoeba-i1 reduces the extra cost by 97.2% compared to Seesaw.

4.2.4 Transformation-aware scheduler. In this part, 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 new request to the least-load instance. 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 sample the throughput periodically during LLM serving and Figure 12a-12d shows the overall CDF of the system throughput performance. Amoeba improves average throughput by 26.1%-39.2% compared to both the RR and LLF.

Figure 13 captures a representative duration of this experiment. RR and LLF suffer from excessive transformation of $TP1$ instances when a new long request arrives in the system, while Amoeba delivers higher throughput by better scheduling.

4.3 End-to-End Performance

We apply the real trace captured in production to compare the end-to-end performance of Amoeba-i2 with KunServe and LoongServe under different query-per-second (QPS) settings. We do not include Seesaw [17] since its transformation cost (10s TTFT and 100% increased TPOT, Figure 11) does not satisfy the SLO requirement. Figure 14 shows the result of throughput, TTFT, and TPOT, on Qwen2.5-32B. Compared with alternatives, Amoeba-i2 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. Our pipelining optimization (Improvement 2 in §3) further decreases 26.7% TTFT. This optimization plays an important role under the 0.6 QPS load to keep TTFT lower than 10 seconds. It enables the system to serve more requests without violating SLO.

5 Conclusion

To handle the dynamic context lengths and request arrival patterns in LLM serving, we design Amoeba to conduct runtime tensor parallel transformation. 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 Amoeba improves throughput by 1.75×-6.57× compared to state-of-the-art solutions.

Acknowledgments

This work was supported in part by the Songshan Laboratory Fund (Grant ZZK202402010), the National Natural Science Foundation of China (Project No. 62472101), the Science and Technology Commission of Shanghai Municipality (Project No. 25511107502), and a research grant from Alibaba Group through the Alibaba Innovative Research Program. We thank the anonymous reviewers for their thoughtful feedback.

References

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

[2] 2025. Llama: Industry Leading, Open-Source AI. https://www.llama.com/.

[3] 2025. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[4] 2025. Qwen: Qwickly forging AGI, enhancing intelligence. https://qwenlm.github.io/.

[5] 2025. SGLang is a fast serving framework for large language models and vision language models. https://github.com/sgl-project/sglang.

[6] 2025. Welcome to vLLM: Easy, fast, and cheap LLM serving for everyone. https://docs.vllm.ai/en/latest/.

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

[8] 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. doi:10.1109/SCW63240.2024.00178

[9] 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

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

[11] 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

[12] 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. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[13] 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

[14] 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 (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 624–639. doi:10.1145/3694715.3695978

[15] 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 (Huntsville, Ontario, Canada) (SOSP '19). Association for Computing Machinery, New York, NY, USA, 322–337. doi:10.1145/3341301.3359658

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

[17] 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. arXiv:2503.06433 [cs.DC] https://arxiv.org/abs/2503.06433

[18] 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 (Santa Clara, CA, USA) (OSDI'24). USENIX Association, USA, Article 10, 19 pages.

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

[20] 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 (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[21] 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. arXiv:2305.05920 [cs.LG] https://arxiv.org/abs/2305.05920

[22] 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

[23] 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

[24] 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