DeInfer: Efficient Parallel Inferencing for Decomposed Large Language Models

You-Liang Huang
yliangh@bu.edu
Boston University
Boston, Massachusetts, USA
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Xinhao Huang
xhuang171@connect.hkust-gz.edu.cn
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Chengxi Liao
Zeyi Wen*
cliao118@connect.hkust-gz.edu.cn
wenzeyi@hkust-gz.edu.cn
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, Guangdong, China

Abstract

Existing works on large language model (LLM) decomposition mainly focus on improving performance on downstream tasks, but they ignore the poor parallel inference performance when trying to scale up the model size. To mitigate this important performance issue, this paper introduces DeInfer, a high-performance inference system dedicated to parallel inference of decomposed LLMs. It consists of multiple optimizations to maximize performance and be compatible with state-of-the-art optimization techniques. Extensive experiments are carried out to evaluate DeInfer's performance, where the results demonstrate its superiority, suggesting it can greatly facilitate the parallel inference of decomposed LLMs.

CCS Concepts

• **Computing methodologies** → **Neural networks**; **Parallel algorithms**; Natural language processing.

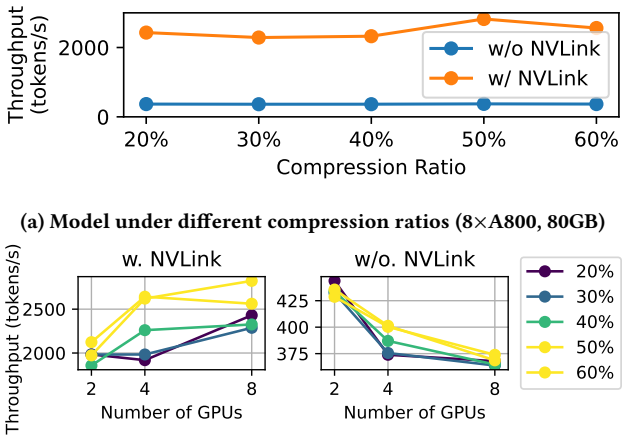
Keywords

Large language model (LLM) inference, parallel inference

ACM Reference Format:
You-Liang Huang, Xinhao Huang, Chengxi Liao, and Zeyi Wen. 2026. DeInfer: Efficient Parallel Inferencing for Decomposed Large Language Models. In *DAC '26: 63rd ACM/IEEE Design Automation Conference (DAC '26), July 26 – 29, 2026, Long Beach, California USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Following the scaling law [10], modern large language models (LLMs) usually have tens of billions of model parameters, which poses considerable challenges for model deployment and inference regarding memory footprint. To reduce the memory footprint, researchers have proposed a variety of model compression techniques, such as quantization [14], model pruning [4], and model decomposition [7, 18]. Among these, decomposition-based compression has received relatively less attention than its counterparts. One key reason is that it is not inference-friendly, especially in a parallel setting. The parallel inference performance of decomposed LLMs cannot scale with the increase of compression ratios, as shown in Fig. 1(a), nor the parallelism, as shown in Fig. 1(b).



(a) Model under different compression ratios (8×A800, 80GB)

(b) Model under different numbers of GPUs

Figure 1: Parallel inference throughput of decomposed LLaMA-3-70B on a cluster of 8×A800 (80GB).

To mitigate this performance issue, we propose DeInfer, a high-performance inference system dedicated to the parallel inference of decomposed LLMs. In this paper, we first identify three key performance bottlenecks and then elaborate on the details of DeInfer and how it is designed to eliminate these bottlenecks. After that, we carry out extensive and comprehensive experiments to evaluate the performance and scalability of DeInfer. To the best of our knowledge, there is no other work focusing on improving the performance of decomposed LLM parallel inference.

2 PRELIMINARIES OF LLM DECOMPOSITION

2.1 Background

Low-rank decomposition can reduce model parameters by approximating the weight matrix with several smaller factorized matrices. Unlike hardware-dependent compression techniques (*e.g.*, unstructured pruning), low-rank decomposition is emerging as a promising technique for LLM compression [1, 6, 7, 13, 15, 18, 21]. Taking truncated Singular Value Decomposition (SVD) as an example, it substitutes the weight matrix $W \in R^{m\times n}$ by the product of two smaller matrices $A \in R^{m\times k}$ and $B \in R^{k\times n}$ as follows.

$$W \approx AB \tag{1}$$

where $A = (U_k\sqrt{\Sigma_k})$, $B = (\sqrt{\Sigma_k}V_k^T)$, $U_k \in R^{m\times k}$ and $V_k^T \in R^{k\times n}$ are the top-k truncated matrices. $\sqrt{\Sigma_k} \in R^{k\times k}$ is a diagonal matrix by the square-roots of the corresponding top-k singular values in Σ.

*Corresponding Author

DAC '26, Long Beach, California USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

This low-rank decomposition reduces the number of parameters from $m \times n$ to $(m+n) \times k$, where users balance the trade-off between performance and model size by adjusting k.

Model decomposition techniques have two advantages in terms of reducing the memory footprint. First, it can directly reduce the size of model parameters by replacing original pre-trained weights with two smaller matrices. Second, the KV cache can be compressed as a by-product of compressing K and V matrices in the attention layers, where the low-rank intermediate results between the two matrix multiplications now act as KV caches.

However, these advantages in terms of reducing the memory footprint can turn out to be drawbacks in terms of inference efficiency. Additional matrix multiplications brought about by low-rank matrices and KV cache reconstruction inevitably cause computational overhead. More importantly, we notice that it would incur more significant performance degradation in a parallel setting. In the next subsection, we will elaborate on performance bottlenecks and problems that we observed in practice.

2.2 Performance Bottlenecks

2.2.1 Communication. **In low-rank decomposition, approximating one weight matrix with two smaller matrices introduces more *reduce-sum* operations in parallel inference.**

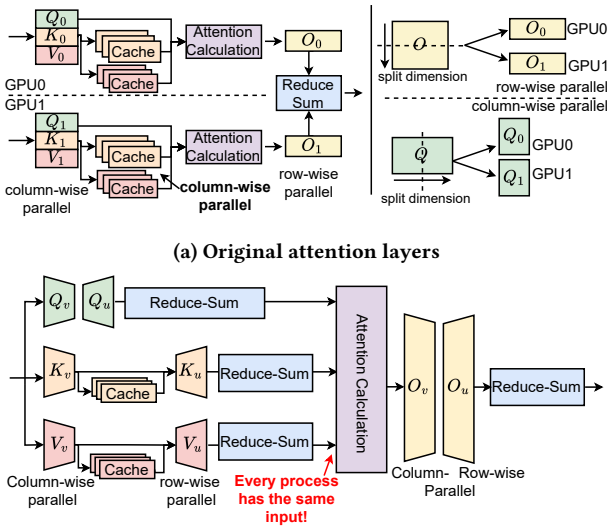
For example, in attention layers, as shown in Fig. 2(a), there are four matrices, Q, K, V, and O before the model decomposition. A common technique here for applying tensor parallelism is to split them into several shards. Every process holds a small chunk of matrices, where Q, K, and V are column-wise splits, and O is row-wise split. During the forward, there is only one *reduce-sum* operation to accumulate the partial results of multiplication between O and the result of self-attention. However, after model decomposition, as shown in Fig. 2(b), every matrix is replaced by two smaller matrices. For each paired low-rank matrices, there is a *reduce-sum* operation to keep the computation correct. In comparison, there are four *reduce-sum* operations in total instead of just one before model decomposition. More importantly, these additional operations are all *reduce-sum*, the collective communication primitive that has the highest cost in terms of bandwidth and latency. The situation is the same for the MLP layers.

2.2.2 Duplicate Self-Attention Computation. **In attention layers, self-attention computation cannot be parallelized.**

As depicted in Fig. 2(a), in the original model, with the column-wise parallel Q, K, and V, every process has a small fraction of data and can perform self-attention computation on their own data. After multiplying the row-wise parallel O, partial results will be accumulated with a *reduce-sum* operation, and each process has an identical copy. However, after model decomposition, as shown in Fig. 2(b), there are three *reduce-sum* operations after multiplying upward projection matrices. At this moment, every process has an identical copy of data. Therefore, each process would perform a duplicate self-attention computation, which is very costly and totally unnecessary.

2.2.3 Being incompatible with CUDA Graph. **CUDA Graph requires fixed arguments, but the results of the KV cache reconstruction have a dynamic shape.**

The latest LLM inference systems adopt paged KV cache management. Every process organizes a large chunk of GPU memory into a number of blocks, where a physical block can contain several slots of KV cache. Meanwhile, a block table is maintained to manage



(a) Original attention layers

(b) Decomposed attention layers

Figure 2: Self-attention computation is duplicated in parallel inference of decomposed LLMs.

the mapping between logical blocks and physical blocks, where requests can only access logical blocks, and the inference system decides the swap-in and swap-out of physical blocks. When doing self-attention computation, the Flash Attention kernel takes the block table and base address as input, using an internal loop to fetch KV cache scatter in the blocks to perform the calculation [2, 3]. For the original model, it is naturally compatible with a static computation graph, *i.e.*, CUDA Graph. However, for decomposed LLMs, things become totally different. During decoding, the number of KV cache is increasing, which means the size of KV cache reconstruction results are enlarging. The dynamic shape of the data conflicts with the requirement of building a static computation graph. Therefore, decomposed LLMs have to use a dynamic computation graph, which can cause considerable performance degradation.

3 RELATED WORKS

3.1 LLM Decomposition

LLM decomposition adopts low-rank decomposition techniques that approximate weight matrices through products of low-rank matrices. Singular Value Decomposition (SVD) is a classic technique [6], and the latest SVD-based methods introduce whitening techniques to mitigate the reconstruction error caused by outliers, which gives them superior performance in downstream tasks [18, 21]. Atomic Feature Mimicking (AFM) is another common technique in LLM decomposition. It applies principle component analysis for the feature-based low-rank factorization [20]. In addition to different matrix factorization, the search for better truncation positions now become another promising area for LLM decomposition [8, 9, 17]. At the same time, researchers are also active in exploring the potential of low-rank decomposition in LLM inference. Chang et al. [1] and Sun et al. [16] tried to reduce memory footprint by storing cache in a low-rank manner. Saxena et al. [15] operationalizes the attention mechanisms within low-dimensional eigenbases, achieving memory-efficient KV cache storage. Although much progress has been achieved in the development of LLM decomposition, there is still no attempt to explore what a decomposed

LLM would be like in a parallel setting when using the state-of-the-art LLM inference systems (*e.g.,* vLLM [11] and SGLang [23]).

3.2 DeInfer's Position

We need to emphasize that the primary goal and position of our work is completely different from the LLM decomposition work we discussed above. *Instead of providing a novel LLM decomposition technique that has better performance in downstream tasks, our work aims to improve the parallel inference performance of decomposed LLMs.* As for KV cache compression, storing the KV cache in a low-rank manner is a by-product of model decomposition and its reconstruction (on single GPU) is well-established in existing works. However, no attempts have been made to address the problems of KV cache reconstruction in a parallel setting (§2.2), and our DeInfer can address these problems.

4 PROPOSED WORK

In this section, our proposed system DeInfer will be elaborated. First, we will introduce a novel low-rank communication technique that can significantly reduce communication costs in decomposed LLMs. Then, we will give the details of other optimizations. In the end, we will demonstrate how our DeInfer can accommodate different LLM variants and model decomposition variants.

4.1 Highly Efficient Low-rank Communication

After model decomposition, we can obtain a downward projection matrix x_v and an upward projection matrix x_u from the original weight matrix x. We noticed that most attention layers and MLP layers in modern LLMs both are two sub-layers, thus there would be four sub-layers in total after model decomposition. Our key idea is that, *instead of having a reduce-sum operation every two sub-layers in the normal latent space, we can rearrange the current computation pipeline to let communication happen in the low rank latent space.* The rearranged computation pipelines are depicted in Fig. 3.

As shown in Fig. 3, all downward projection matrices x_v in the first sub-layer are in column-wise parallel (in a split manner, where all matrices are first concatenated and then evenly split), followed by an *all-gather* operation in the low-rank latent space. After *all-gather*, each process now has identical low-rank data. In attention layers, the low rank data for K and V (*i.e.,* k_{low_rank} and v_{low_rank}) will be stored as KV cache, whose reconstruction will be elaborated in the next subsection. The upward projection matrices x_u are also in column-parallel (but in a *shard* way, where each process has a shard of the same matrices). At this point, to keep correct computation, the forward performs a batched matrix multiplication for the low-rank results with different upward matrix shards x_u, respectively, *e.g.,* k_{low_rank} with k_u, and v_{low_rank} with v_u. Each process has their small chunk of data ready to proceed to perform self-attention or activation computation. *It's noted that the decomposed LLMs now no longer have duplicate self-attention computation.* In the second sub-layer, the downward projection matrix is in row-wise parallel, and upward projection matrix is identical in each process. Therefore, after multiplying the downward matrix, each process needs to have a *reduce-sum* operation to accumulate the partial results of all processes. Since every process has identical low-rank data and identical upward matrix, the output would also be identical.

To better understand how much bandwidth we can save through the communication, we analyze the bandwidth usage of LLaMA-3-70B in Table. 1, where we compare the unoptimized decomposed LLMs with our DeInfer under the compression ratio of 40%.

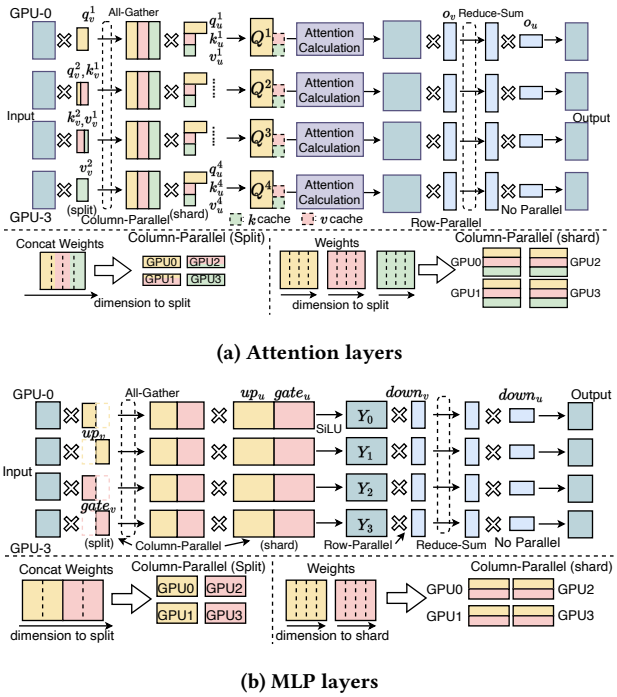


Figure 3: Low-rank communication design in decomposed LLaMA models.

In Table. 1, l_X means the number of reduced dimension of the matrix X, h is the hidden dimension, h_{kv} is the dimension of the K and V matrices, and m is the intermediate dimension in MLP. And the bandwidth of *reduce-sum* is double of *all-gather*. As demonstrated, DeInfer significantly reduces bandwidth usage by 78% through rearranging the computation pipeline.

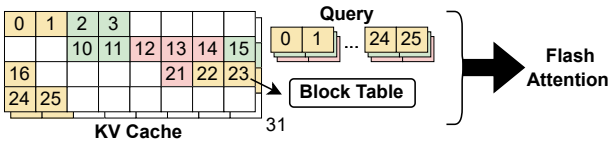
4.2 Integrating Paged Cache and CUDA Graph

As depicted in Fig. 4(a), Flash Attention kernel takes the KV cache (*i.e.,* base address of KV cache area), *Query* (*i.e.,* results of multiplying the Q matrix), and a block table as input. All input is fixed during the forward. However, after model decomposition, decoding can no longer use the static computation graph since the shape of KV cache reconstruction results is dynamic. Therefore, we redesign the computation of KV cache reconstruction to allow the decoding to be compatible with both paged cache and CUDA Graph. Our designed process for KV cache construction during decoding is shown in Fig. 4(b). It has two stages, the *preparation stage*, where we are allowed to do any CPU and GPU operations for the CUDA Graph execution, and the *graph replay stage*, where only captured

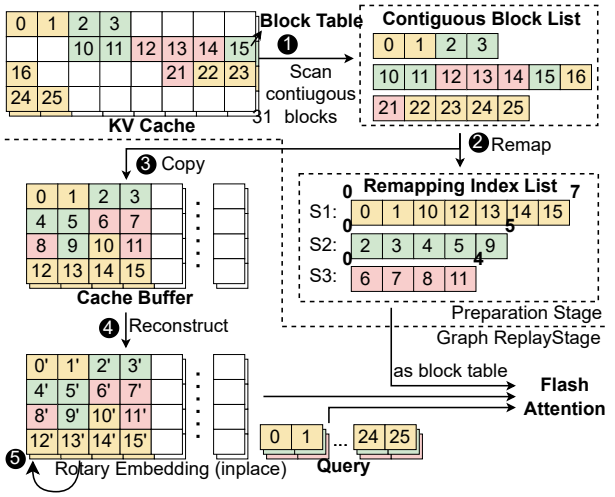
Layer	Status	Cost		Total Bandwidth
		all-gather, $O(n)$	*reduce-sum*, $2O(n)$	
Attention	Unoptimized	0	$2\times(2h+2h_{kv})$	$4h+4h_{kv}$
	DeInfer	$l_q+l_k+l_v$	$2h$	$l_q+l_k+l_v+2h$
MLP	Unoptimized	0	$2\times(2m+h)$	$4m+2h$
	DeInfer	$l_{up}+l_{gate}+l_{down}$	$2h$	$l_{up}+l_{gate}+l_{down}+2h$
LLaMA-3-70B†	Unoptimized	0	167,936	167,936 (100%)
	DeInfer	20,892	2×8192	37,276 (↓78%)

† : In LLaMA-3-70B, we have $h=8192$, $h_{kv}=1024$, $m=28672$. Under the compression ratio of 40% (evenly), $l_{up}=l_{down}=l_{gate}=l_q=60\%h=4916$ and $l_k=l_v=60\%h_{kv}=614$.

Table 1: Communication cost per token in a single Transformer block during forward pass (LLaMA-3-70B).



(a) Before model decomposition



(b) After model decomposition (DeInfer)

Figure 4: KV cache reconstruction process.

operations can be executed, and all operations must fulfill the requirements of CUDA Graph.

DeInfer introduces two buffers for squeezing and temporally storing the KV cache and their corresponding reconstruction results. These two buffers are allocated in the initialization of the inference system. We noticed that iteratively copying the scattered KV cache is extremely time-consuming, thus DeInfer tries to copy as many blocks as possible at one time. In the preparation stage, DeInfer performs a scan to find which logical block is physically contiguous. At the same time, DeInfer derives a new block table called *remapping index list*. It acts as a block table, indicating which blocks in the buffer a sequence owns. In the graph replay stage, the physically contiguous KV caches are sequentially copied to the buffer, where the KV cache in the buffer is compact. Then, the compact KV cache multiplies upward matrices to finish the reconstruction. It is noted here that the results have a dynamic shape but a fixed memory address and we only need to use the GEMM kernel that has a large size to adapt for CUDA Graph. Overhead here is inevitable, but it can bring larger overall efficiency improvement by eliminating all kernel launch overhead. At this point, we can apply our customized in-place rotary position embedding kernel to the reconstruction results if the model needs. When everything is done, the buffer act as the original KV cache and *remapping index list* as the block table. They and *Query* are used to call Flash Attention kernel to complete self-attention computation.

In addition to the highly efficient customized kernel, DeInfer also adopts other optimization techniques (*e.g.,* the communication-computation overlap and kernel fusion) to further improve the performance. We cannot introduce the technical details here due to page limit, but they can be found in DeInfer's implementation.

4.3 Generalization

Revisiting the low-rank communication in §4.1, our proposed method does not place any constraint on the shape of the weight matrix. DeInfer supports all tensor parallelism configurations as long as one matrix can be appropriately partitioned and evenly distributed. Therefore, DeInfer can support decomposed LLMs with variadic compression ratios, *e.g.,* weights in different layers have different compression ratios, and even Q, K and V in the same attention layer have different compression ratios. Additionally, for varying model decomposition methods, we have observed that there is a fundamental principle in matrix factorization, *i.e.,* no non-linear operations are introduced during factorization. *This principle indicates that any more than two matrices derived from factorization can be absorbed into two matrices and then fit our DeInfer.*

As for transformer-based LLMs, even though there are MoE decomposition methods [12], our DeInfer now only supports non-MoE models. The supported LLM variants are reported in Table 2. As shown, most variants can be supported by DeInfer, which includes but is not limited to *LLaMA* [5], *OPT* [22], and *Qwen* [19] models.

Attn				MLP		Rotary Position Embed
MHA	MQA	GQA	MLA†	non-GLU	GLU-based	
✓	✓	✓		✓	✓	✓

† : Almost only be used in Deepseek's MoE models.

Table 2: Attention and MLP variants supported by DeInfer.

5 EXPERIMENTS

In this section, we evaluate our proposed DeInfer with the focus on the following three aspects: *throughput analysis*, *latency analysis*, and *system-level analysis*. Additionally, we conduct experiments to demonstrate the necessity of supporting CUDA Graph and how is the scalability of DeInfer. For demonstration, we implement DeInfer based on one of the state-of-the-art LLM inferencing systems, *vLLM*.

5.1 Setup

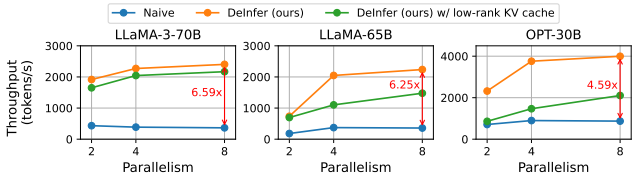
Our experiments are conducted on two hardware platforms: (1) 8×A800 (80GB) with 2×Intel Xeon-8358P and 2TB memory. (2) 8×A6000 (48GB) with 2×Intel Xeon-8358 and 512GB memory, where platform (1) has fully-connected NVLinks but platform (2) doesn't have NVLinks. To demonstrate generalizability, we select LLaMA-65B, LLaMA-3-70B, and OPT-30B as foundation models because they encompass MHA and GQA in attention variants, non-GLU and GLU-based MLPs, and rotary embedding. The test scripts that we adopt are from the vLLM benchmark suite[1] for fair and replicable comparison. *To the best of our knowledge, there is no other work on parallel inference of decomposed LLMs. Therefore, we implement a basic implementation of tensor parallelism (hereinafter referred to as Base).* Our experiments compare the performance of DeInfer and Base, which also include ablation studies and detailed analysis at the system level. Due to the inconsistent GPU memory size between our test platforms, we only keep our configurations consistent in each experiment.

5.2 Throughput Analysis

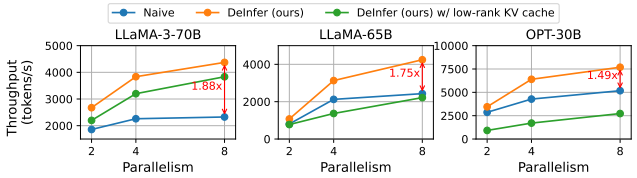
We prepare two different scenarios to evaluate DeInfer's throughput performance. In the first setting, there is a large batch size (512 for LLaMA-3-70B and LLaMA-65B, 768 for OPT-30B) but with a

[1] https://github.com/vllm-project/vllm/tree/main/benchmarks

relatively smaller maximal sequence length (32 pre-fill tokens and the maximal sequence length is 160), whereas in the second setting, it is vice versa. The batch size is set to 64 for LLaMA-3-70B and LLaMA-65B, 192 for OPT-30B, where pre-fill tokens is 64 and the maximal sequence length is 512. The configuration is kept the same for each experiment setup. The results are shown in Fig. 5 and Fig. 6, respectively for the first and the second setting. For the both scenarios, as shown in the figures, Base has poor performance and scalability on the three different models. In comparison, DeInfer shows a much better scalability and significantly outperforms Base by different extents.

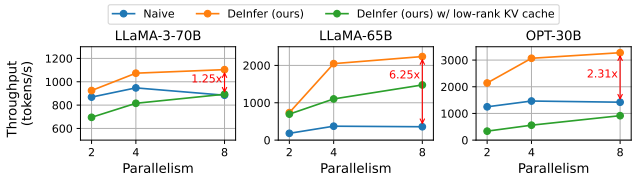


(a) Without NVLink

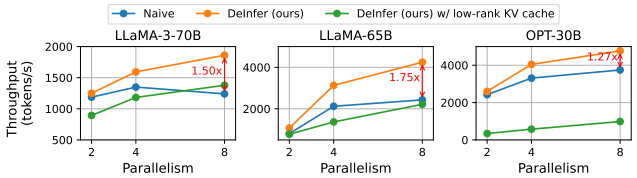


(b) With NVLink

Figure 5: Batched generation performance of different models under 40% compression ratio in the high load scenario on 8×A800 (80GB).



(a) Without NVLink



(b) With NVLink

Figure 6: Batched generation performance of different models in the mild load scenario under 40% compression ratio on 8×A800 (80GB).

Additionally, when enabling low-rank KV cache, there is a noticeable performance gap between the GQA model (LLaMA-3-70B) and the MHA models (LLaMA-65B and OPT-30B). This is because MHA models have more attention heads (64 in LLaMA-65B vs. 8 in LLaMA-3-70B), which means higher KV cache reconstruction cost. This difference brings tremendous advantages for LLaMA-3-70B in rebuilding the KV cache. In comparison, LLaMA-65B suffers a much greater performance degradation when enabling low-rank KV cache. Similarly, the OPT-30B can also witness the same problem.

5.3 Latency Analysis

In this subsection, we adopt the vLLM serving benchmark to evaluate the latency performance of our system, where there are three metrics used for performance assessment: Time-to-First Token (TTFT) and Inter-token Latency (ITL). The request in this experiment has 128 prompts that have 32 pre-fill tokens, and the system is required to generate 256 tokens for each prompt. The configuration is kept the same across different platforms. Experiments are conducted on the A800 platform and A6000 platform, whose results are reported in Table 3 and Table 4, respectively.

Metrics	Methods	LLaMA-3-70B	LLaMA-65B	OPT-30B
TTFT (ms)	Base	21740	19341	6999
	DeInfer	(↓83%) 3633	(↓82%) 3489	(↓77%) 1594
	DeInfer†	(↓83%) 3703	(↓81%) 3626	(↓76%) 1704
ITL (ms)	Base	812	764	311
	DeInfer	(↓79%) 172	(↓76%) 181	(↓68%) 101
	DeInfer†	(↓74%) 211	(↓59%) 312	(↓39%) 190

† : w/ enabling low-rank KV cache.

Table 3: Serving performance of different models under a compression ratio of 40% on 8xA6000 (48GB) w/o. NVLink

As reported, our DeInfer achieves significant latency decreases in both test platforms and different NVLink settings. When enabling low-rank KV cache, the performance of DeInfer shows different patterns. Generally, DeInfer suffers more in the MHA models (*i.e.*, LLaMA-65B and OPT-30B) than the GQA model (LLaMA-3-70B). The reason remains the same: more attention heads in MHA models, which can also explain why enabling NVLink does not help. However, DeInfer on the A6000 platform (Table 3) shows a different performance pattern compared to it on the A800 platform, where on 8xA6000 we can observe a significant performance improvement when enabling low-rank KV cache. To explore the underlying reason, we profile the execution of Base and DeInfer, and the details are elaborated in the next subsection.

5.4 System Profiling

We report the breakdown of computation and communication profiled by NVIDIA Nsight Systems[2] in Table 5. As shown, Base's communication costs are predominant in experiments that do not enable NVLink, and things get worse as the parallelism increases. The exorbitant communication cost is thus the root cause of poor parallel inference performance. In comparison, DeInfer improves communication efficiency by reducing 80~90% communication cost. As parallelism increases, DeInfer shows incredible scalability. Moreover, since DeInfer eliminates duplicate self-attention computation, the computing time is reduced by 10~30%.

According to hardware specifications, NVIDIA RTX A800 (80GB)[3] has a much larger memory bus and bandwidth than NVIDIA RTX A6000[4]. As for the discussion at the end of the last subsection (*i.e.*, distinct performance on different platforms), inference performance

[2]https://developer.nvidia.com/nsight-systems
[3]https://www.techpowerup.com/gpu-specs/a800-sxm4-80-gb.c3966
[4]https://www.techpowerup.com/gpu-specs/rtx-a6000.c3686

w/o. NVLink										
Metrics	Methods	LLaMA-3-70B			LLaMA-65B			OPT-30B		
		TP=2	TP=4	TP=8	TP=2	TP=4	TP=8	TP=2	TP=4	TP=8
TTFT (ms)	Base	8950	10064	10764	8164	9359	10451	3126	3566	3845
	DeInfer	(↓76%) 2137	(↓79%) 2110	(↓79%) 2246	(↓73%) 2192	(↓78%) 2088	(↓83%) 1812	(↓69%) 960	(↓74%) 927	(↓75%) 944
	DeInfer†	(↓72%) 2484	(↓79%) 2102	(↓81%) 2093	(↓73%) 2216	(↓80%) 1840	(↓80%) 2085	(↓70%) 940	(↓75%) 878	(↓76%) 906
ITL (ms)	Base	156	122	96	206	103	101	54	47	47
	DeInfer	(↓31%) 108	(↓28%) 88	(↓9%) 87	(↓44%) 115	(↓12%) 91	(↓13%) 88	(↑6%) 57	(↓0%) 47	(↓2%) 46
	DeInfer†	(↓8%) 143	(↓6%) 115	(↑9%) 105	(↑220%) 660	(↑244%) 354	(↑114%) 216	(↑424%) 283	(↑228%) 154	(↑115%) 101
w/ NVLink										
Metrics	Methods	LLaMA-3-70B			LLaMA-65B			OPT-30B		
		TP=2	TP=4	TP=8	TP=2	TP=4	TP=8	TP=2	TP=4	TP=8
TTFT (ms)	Base	1662	1499	1605	1525	1417	1461	724	654	649
	DeInfer	(↓12%) 1463	(↓34%) 983	(↓50%) 797	(↓24%) 1162	(↓38%) 875	(↓49%) 749	(↓20%) 581	(↓32%) 448	(↓32%) 443
	DeInfer†	(↓17%) 1377	(↓37%) 948	(↓49%) 817	(↓20%) 1214	(↓36%) 901	(↓46%) 782	(↓12%) 640	(↓21%) 514	(↓30%) 457
ITL (ms)	Base	156	115	93	191	98	94	50	42	41
	DeInfer	(↓42%) 90	(↓34%) 76	(↓16%) 78	(↓52%) 92	(↓35%) 64	(↓40%) 56	(↓0%) 50	(↓12%) 37	(↓20%) 33
	DeInfer†	(↓19%) 127	(↓17%) 96	(↓0%) 93	(↑236%) 642	(↑237%) 330	(↑96%) 184	(↑452%) 276	(↑248%) 146	(↑110%) 86

† : w/ enabling low-rank KV cache.

Table 4: Serving performance of different models under compression ratio of 40% on A800 (80GB) platform

TP	w/o. NVLink			w/ NVLink		
	compute (ms)	comm. (ms)	total (ms)	compute (ms)	comm. (ms)	total (ms)
2	(16%) 2.352	(84%) 12.182	(100%) 14.534	(73%) 2.391	(27%) 0.891	(100%) 3.282
	(↓23%) 1.809	(↓92%) 0.977	(↓81%) 2.786	(↓25%) 1.792	(↓74%) 0.232	(↓39%) 2.014
4	(7%) 1.174	(93%) 14.853	(100%) 16.027	(49%) 1.176	(51%) 1.240	(100%) 2.416
	(↓12%) 1.038	(↓92%) 1.156	(↓86%) 2.194	(↓11%) 1.041	(↓78%) 0.267	(↓46%) 1.308
8	(6%) 1.112	(94%) 16.051	(100%) 17.163	(45%) 1.116	(55%) 1.369	(100%) 2.485
	(↓29%) 0.787	(↓92%) 1.285	(↓88%) 2.072	(↓30%) 0.782	(↓82%) 0.247	(↓59%) 1.029

Tag: Base , DeInfer . (X%) indicates the proportion of total cost. (↓X%) denotes time saved by DeInfer compared to Base.

Table 5: Breakdown of execution time in a LLaMA-3-70B transformer layer on A800 with compression ratio of 40%.

on the A6000 platform is not only limited to exorbitant communication cost but also to much smaller memory bandwidth. The advantage provided by DeInfer is so significant that it compensates more enough than the KV cache reconstruction cost. Therefore, we can witness a consistent performance improvement of DeInfer in the three models of different architectures on the A6000 platform.

5.5 Scalability Analysis

As demonstrated in Introduction (§1), one of the most important motivations of our work is the poor scalability of decomposed LLMs, which fails to scale up the performance as the compression ratio or the parallelism increases. In this experiment, we run benchmarks (*i.e.,* the high-load generation experiment in §5.2) on LLMs that are compressed to different extents to evaluate the scalability of DeInfer. The results are shown in Fig. 7, where DeInfer shows excellent scalability. When the parallelism is 8 and compression ratio is 60%, DeInfer achieves a speed up of 2.25x and 8.81x for w/ NVLink and w/o. NVLink, respectively. Moreover, as the compression ratio increases, the performance of DeInfer can also gradually improve.

5.6 Benefits of CUDA Graph

Base's decoding stage cannot support the static graph, which may cause significant performance degradation. To explore how much performance can be improved, we test two types of DeInfer, one with CUDA Graph and another without, in both throughput and serving experiments. The results are reported in Table 6.

Table 6 shows a general performance pattern. When NVLink is not enabled, the performance bottleneck remains the communication; therefore, supporting CUDA Graph does not help much.

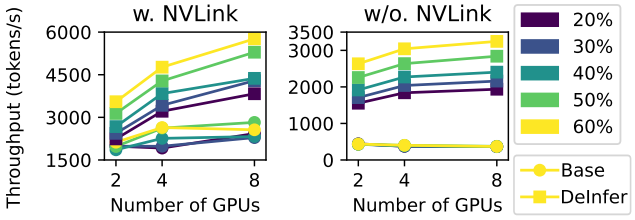


Figure 7: Generation performance of LLaMA-3-70B of different compression ratios on A800 platform.

Experiments	Configurations	LLaMA-3-70B		LLaMA-65B	
		w/o. NVLink	w/ NVLink	w/o NVLink	w/ NVLink
Throughput	s1 (tokens/s)	2068	3488	1435	2011
		(↑5%) 2168	(↑10%) 3834	(↑3%) 1477	(↑9%) 2217
	s2 (tokens/s)	816	831	552	660
		(↑20%) 978	(↑66%) 1379	(↑6%) 587	(↑8%) 713
Serving	ITL (ms)	118	97	279	249
		(↓3%) 114	(↓5%) 93	(↓28%) 202	(↓31%) 173

Tag: Base , DeInfer . Table reports performance of DeInfer that enables low-rank KV cache.

Table 6: Performance improvement brought by static kernel execution graph, on 8xA800.

However, when NVLink is enabled and the performance bottleneck becomes the computation, we can notice a significant performance improvement in most scenarios. This is because the overhead of CUDA kernel launch is eliminated, which saves tremendous time and thus brings such a performance gain.

6 CONCLUSION

To mitigate the performance issue of decomposed LLM parallel inference, this work proposes DeInfer, a high-performance inference system dedicated to parallel inference of decomposed LLMs. It consists of a novel low-rank communication technique and other optimizations that can significantly improve the parallel inference performance. We integrate it into one of the state-of-the-art inference systems, *vLLM*. Extensive experiments are carried out, where results demonstrate the excellent scalability and efficiency of DeInfer, suggesting the practicality of DeInfer in parallel inference of decomposed LLMs.

References

[1] Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S Abdelfattah, and Kai-Chiang Wu. 2025. Palu: KV-Cache Compression with Low-Rank Projection. In *The Thirteenth International Conference on Learning Representations*.

[2] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations*, Vol. 2024. 35549–35562.

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[4] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *ICML (Proceedings of Machine Learning Research, Vol. 202)*. PMLR, 10323–10337.

[5] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *arXiv:2407.21783* (2024).

[6] Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. 2022. Language model compression with weighted low-rank factorization. In *ICLR*.

[7] Xinhao Huang, You-Liang Huang, and Zeyi Wen. 2025. SoLA: Leveraging Soft Activation Sparsity and Low-Rank Decomposition for Large Language Model Compression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 17494–17502.

[8] You-Liang Huang, Xinhao Huang, Xuemei Peng, and Zeyi Wen. 2024. Automatic Truncation Position Selection in Singular Value Decomposition for Large Language Models. *Openreview* (2024).

[9] Yixin Ji, Yang Xiang, Juntao Li, Qingrong Xia, Zi Ye, Xinyu Duan, Zhefeng Wang, Kehai Chen, and Min Zhang. 2024. Adaptive feature-based low-rank compression of large language models via bayesian optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 4152–4168.

[10] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv:2001.08361* (2020).

[11] 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. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[12] Wei Li, Lujun Li, Hao Gu, You-Liang Huang, Mark G. Lee, Shengjie Sun, Wei Xue, and Yike Guo. 2025. MoE-SVD: Structured Mixture-of-Experts LLMs Compression via Singular Value Decomposition. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 35209–35230.

[13] Chi-Heng Lin, Shangqian Gao, James Smith, Abhishek Patel, Shikhar Tuli, Yilin Shen, Hongxia Jin, and Yen-Chang Hsu. 2025. MoDeGPT: Modular Decomposition for Large Language Model Compression. In *International Conference on Learning Representations*, Vol. 2025. 101355–101390.

[14] Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. 2024. Duquant: Distributing outliers via dual transformation makes stronger quantized llms. *Advances in Neural Information Processing Systems* 37 (2024), 87766–87800.

[15] Utkarsh Saxena, Gobinda Saha, Sakshi Choudhary, and Kaushik Roy. 2024. Eigen Attention: Attention in Low-Rank Space for KV Cache Compression. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 15332–15344.

[16] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2024. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. *arXiv:2410.21465* (2024).

[17] Xin Wang, Samiul Alam, Zhongwei Wan, Hui Shen, and Mi Zhang. 2025. SVD-LLM V2: Optimizing Singular Value Truncation for Large Language Model Compression. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 4287–4296.

[18] Xin Wang, Yu Zheng, Zhongwei Wan, and Mi Zhang. 2025. SVD-LLM: Truncation-aware Singular Value Decomposition for Large Language Model Compression. In *International Conference on Learning Representations*, Vol. 2025. 19299–19319.

[19] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv:2505.09388* (2025).

[20] Hao Yu and Jianxin Wu. 2023. Compressing Transformers: Features Are Low-Rank, but Weights Are Not!. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 11007–11015.

[21] Zhihang Yuan, Yuzhang Shang, Yue Song, Qiang Wu, Yan Yan, and Guangyu Sun. 2023. ASVD: Activation-aware Singular Value Decomposition for Compressing Large Language Models. *CoRR* abs/2312.05821 (2023).

[22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv:2205.01068* (2022).

[23] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009