

---

# TPLA: TENSOR PARALLEL LATENT ATTENTION FOR EFFICIENT DISAGGREGATED PREFILL & DECODE INFERENCE

---

Xiaojuan Tang<sup>1,3\*</sup>, Fanxu Meng<sup>1,3\*</sup>, Pingzhi Tang<sup>1</sup>, Yuxuan Wang<sup>1</sup>, Di Yin<sup>3</sup>, Xing Sun<sup>3</sup>, Muhan Zhang<sup>1,2†</sup>

<sup>1</sup>Institute for Artificial Intelligence, Peking University

<sup>2</sup>State Key Laboratory of General Artificial Intelligence, BIGAI

<sup>3</sup>Tencent YouTu Lab, Shanghai, China

<https://github.com/fxmeng/TransMLA>

## ABSTRACT

Multi-Head Latent Attention (MLA), introduced in DeepSeek-V2, compresses key–value states into a low-rank latent vector  $c^{KV}$ , caching only this vector to reduce memory. In tensor parallelism (TP), however, attention heads are computed across multiple devices, and each device must load the full  $c^{KV}$ , eroding the advantage of MLA over Grouped Query Attention (GQA). We propose Tensor-Parallel Latent Attention (TPLA): a scheme that partitions both the latent representation and each head’s input dimension across devices, performs attention independently per shard, and then combines results with an all-reduce. TPLA preserves the benefits of a compressed KV cache while unlocking TP efficiency. Unlike Grouped Latent Attention (GLA), every head in TPLA still leverages the full latent representation, maintaining stronger representational capacity. TPLA is drop-in compatible with models pre-trained using MLA: it supports MLA-style prefilling and enables efficient tensor-parallel decoding without retraining. Applying simple orthogonal transforms—e.g., the Hadamard transform or PCA—before TP slicing further mitigates cross-shard interference, yielding minimal accuracy degradation. By reducing the per-device KV cache for DeepSeek-V3 and Kimi-K2, we achieve  $1.79\times$  and  $1.93\times$  speedups, respectively, at a 32K-token context length while maintaining performance on commonsense and LongBench benchmarks. TPLA can be implemented with FlashAttention-3, enabling practical end-to-end acceleration.

## 1 Introduction

Currently, large language models (LLMs) [1, 2, 3, 4, 5] are typically memory-bound (limited by memory bandwidth) rather than compute-bound (limited by floating-point operations per second, FLOPs) during inference. To address this, KV cache compression [6, 7, 8, 9] and tensor parallelism [10, 11, 12, 13, 14] have emerged as two critical techniques for enabling efficient auto-regressive decoding in LLMs. KV cache compression methods prune/merge/share/quantize intermediate key–value pairs to reduce memory overhead. Tensor parallelism addresses memory and compute limitations by splitting large tensors—such as weight matrices—across multiple devices, enabling intra-layer parallel computation for models that cannot fit on a single GPU. GQA [15] inherently supports both KV cache compression and tensor parallelism by grouping query heads so that all heads within a group share a common set of key and value representations, which facilitates efficient distribution across multiple devices. Both theoretical analyses and empirical results demonstrate that the representational capacity of GQA is inferior to that of MLA [16, 17]. MLA introduces a pre-trained KV cache compression strategy that achieves an excellent trade-off between computational efficiency and model performance. However, when multiple attention heads are computed in parallel across multiple devices using tensor parallelism, MLA encounters a critical limitation: each device must load the full latent vector  $c_{KV}$ , undermining the memory savings that MLA offers over GQA. For example, in LLaMA-3-70B [18], the dimension of the KV cache per token is  $2 \times 8 \times 128 = 2048$ , and under tensor parallelism with  $TP = 4$ , each device holds a partitioned KV cache of size 512. In contrast, Deepseek-V3 [19] has a fixed KV cache dimension of  $64 + 512 = 576$ , which must be fully

---

\*Equal contribution.

†Corresponding author: muhan@pku.edu.cn

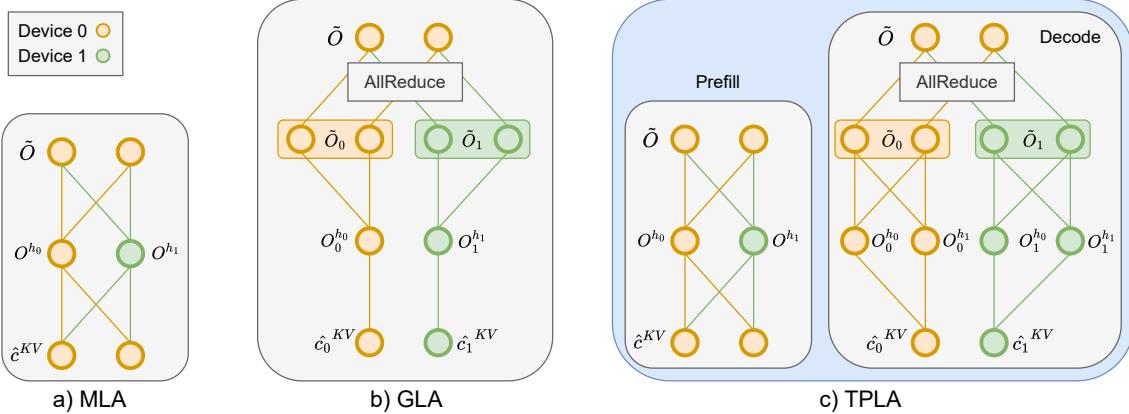


Figure 1: Comparison of MLA, GLA, and TPLA. In MLA, each device must load the entire KV cache. In GLA, each attention head only accesses the portion of the KV cache stored on its own device. In TPLA, the prefilling phase follows MLA for efficiency and accuracy, while during the decoding phase, attention heads are distributed across devices, each relying on the KV cache stored locally on its assigned device.

replicated on each device regardless of the parallelism degree. This results in a higher per-device KV cache memory footprint compared to GQA-based models under the same tensor parallel configuration.

GLA [20] was proposed to address the tensor parallelism limitations of MLA by dividing the attention heads and latent representations into  $g$  groups (typically  $g = 2$ ), such that each group of heads only loads its corresponding latent representation. However, this paper identifies two key limitations of GLA: (1) the reduction in KV cache size for single device comes at the cost of decreased representational capacity for each attention head; and (2) GLA requires training from scratch, which demands significant computational resources to validate its effectiveness.

To address these challenges, we propose Tensor Parallel Latent Attention (TPLA), a method that distributes the latent representations across multiple devices. Each attention head is split across devices, followed by an all-reduce operation on the output  $o$ . TPLA offers the following advantages: 1) Each attention head utilizes the full latent representation, preserving strong representational capacity; 2) Each device only loads a partition of the KV cache, improving inference speed under tensor parallelism; 3) TPLA can directly load pre-trained DeepSeek checkpoints, which incurs only a minor performance drop that is easily recovered; 4) We use reparameterized MLA for prefill and TPLA for decoding, reducing prefill latency while mitigating conversion-induced degradation. 4) TPLA can be viewed as a special case of GLA with more attention heads, making it compatible with FlashAttention-3.

## 2 Related Works

**Reducing KV-Cache Memory** Generative inference with large language models (LLMs) is often constrained by the memory footprint of the key-value (KV) cache, especially for long contexts. Several families of techniques have been explored to mitigate this burden: **token pruning/evicting** [21, 22, 23, 24, 25, 26] removes KV entries for low-importance tokens based on saliency or attention estimates; **token merging** [27, 28, 29] aggregates nearby or similar tokens into a single surrogate KV representation to eliminate redundancy while retaining context; **cross-layer KV sharing/fusion** [30, 31, 32, 33, 34] reuses one KV cache across adjacent layers to avoid per-layer storage; **low-rank KV compression** [35, 36, 37, 6] factorizes KV matrices into low-rank components (learned or SVD-based) to reduce dimensionality and memory; and **KV-cache quantization** [38, 39, 40, 41] stores K/V tensors at reduced numeric precision (e.g., int8 or int4), cutting memory and bandwidth with modest accuracy cost. Although effective, these approaches inevitably discard or alter information in the KV cache and can degrade model performance. In contrast, **TPLA** leaves the KV contents intact: it reduces the amount of cache each device must hold so the model retains full information while alleviating memory pressure. As a result, TPLA tends to preserve accuracy better than compression-based methods.

**Parallelism Strategies for Deployment** Current LLMs scale to billions of parameters; to cope with the resulting memory and compute demands, engineers adopt distributed deployment to reduce wall-clock latency and time costs. **Data parallelism** [42, 43] partitions input data across the sample or batch dimension while replicating model parameters across devices. However, for very large models full replication becomes impractical; moreover, variable sequence

lengths introduce load imbalance (“bubbles”) that waste compute resources. **Pipeline parallelism** [44, 45] partitions the model into contiguous blocks of layers, each placed on a different device. Intermediate activations and gradients are communicated between stages to complete the forward and backward passes, reducing cross-node traffic. This staging overlaps computation across devices to increase throughput, but pipeline bubbles can still leave some devices idle. **Tensor parallelism** [46, 47, 48] splits linear layers along their row or column dimensions, sharding tensors across devices and performing distributed matrix–matrix multiplication with collective communication. It achieves optimal performance on systems where GPUs are fully interconnected via NVLink. TPLA leverages the strengths of TP while addressing MLA’s inability to reduce the KV cache under TP. Long sequences inflate the memory footprint of intermediate activations; **sequence parallelism** [49] mitigates this by replicating the model across devices and splitting inputs along the sequence dimension so that each device processes only a subsequence. **Prefill/Decode Separation** [50, 51, 52] refines sequence parallelism for LLM inference: the prefilling phase is compute-intensive and thus compute-bound, whereas the decoding phase has low per-token compute but frequent memory accesses and is memory-bandwidth-bound. To match these characteristics, different machine counts and architectures are used across the two phases to improve latency and throughput. In TPLA, we further employ different model structures across phases—MLA during prefill to preserve accuracy while reducing computation (improving latency), and TPLA during decoding to reduce memory traffic and increase throughput.

### 3 Preliminary

#### 3.1 Multi-Head Latent Attention

MLA is designed to reduce memory bandwidth overhead by compressing the Key-Value (KV) cache. Specifically, the multi-head keys and values are compressed into a single low-rank latent representation of dimension  $4d_h$ , denoted as  $\mathbf{c}^{\text{KV}}$ . Instead of reconstructing full-size keys and values from this latent representation, MLA adopts a more efficient decoding strategy. By isolating the Rotary Position Embedding (RoPE) operation, the up-projection matrix can be absorbed into the query activations, yielding  $Q$ . Similarly, the value projection is absorbed into the output projection matrix, resulting in  $W^{VO}$  (See Section 3.3). This allows for direct attention computation between  $Q$  and the normalized latent cache  $\hat{\mathbf{c}}^{\text{KV}}$ , followed by a projection through  $W^{VO}$  to produce the final output  $\tilde{O}$ . For simplicity in this initial description, we omit the RoPE components. The core computation is as follows:

$$\begin{aligned} \mathbf{c}^{\text{KV}} &\in \mathbb{R}^{B \times L \times 4d_h}, \quad \hat{\mathbf{c}}^{\text{KV}} = \text{RMSNorm}(\mathbf{c}^{\text{KV}}) \in \mathbb{R}^{B \times L \times 4d_h}, \\ Q &\in \mathbb{R}^{B \times 1 \times h_q \times 4d_h}, \quad W^{VO} \in \mathbb{R}^{(h_q \cdot 4d_h) \times D}, \\ O &= \text{softmax}\left(\frac{Q(\hat{\mathbf{c}}^{\text{KV}})^\top}{\sqrt{d_h}}\right) \hat{\mathbf{c}}^{\text{KV}} \in \mathbb{R}^{B \times 1 \times h_q \times 4d_h}, \end{aligned} \tag{1}$$

$$\tilde{O} = O W^{VO} \in \mathbb{R}^{B \times 1 \times D}. \tag{2}$$

#### 3.2 Grouped Latent Attention

During tensor-parallel decoding, MLA replicates its single latent head on every device, resulting in high KV-cache memory load across all devices. GLA avoids this replication by partitioning the latent KV cache itself. Consider a two-way tensor-parallel configuration. The latent KV cache is divided into two shards,  $\mathbf{c}_0^{\text{KV}}$  and  $\mathbf{c}_1^{\text{KV}}$ , each assigned to one GPU. Simultaneously, attention heads  $h_q$  are grouped such that the absorbed query projection matrix  $Q$  and output projection matrix  $W^{VO}$  are partitioned along both the head dimension ( $h_q$ ) and the feature dimension ( $4d_h$ ), yielding four groups. Thus, GPU 0 operates on  $(\mathbf{c}_0^{\text{KV}}, Q_{0,0}, W_{0,0}^{VO})$ , while GPU 1 operates on  $(\mathbf{c}_1^{\text{KV}}, Q_{1,1}, W_{1,1}^{VO})$ . Each GPU independently computes its local attention output, denoted  $\tilde{O}_0$  and  $\tilde{O}_1$ , respectively. The final output is obtained via an AllReduce operation that sums the local outputs across devices.

$$\begin{aligned} \mathbf{c}_0^{\text{KV}}, \mathbf{c}_1^{\text{KV}} &\in \mathbb{R}^{B \times L \times 2d_h}, \quad \begin{cases} \hat{\mathbf{c}}_0^{\text{KV}} = \text{RMSNorm}(\mathbf{c}_0^{\text{KV}}) \in \mathbb{R}^{B \times L \times 2d_h}, \\ \hat{\mathbf{c}}_1^{\text{KV}} = \text{RMSNorm}(\mathbf{c}_1^{\text{KV}}) \in \mathbb{R}^{B \times L \times 2d_h}, \end{cases} \\ Q_{i,j \in \{0,1\}} &\in \mathbb{R}^{B \times 1 \times \frac{h_q}{2} \times 2d_h}, \quad \begin{pmatrix} Q_{0,0}, Q_{0,1} \\ Q_{1,0}, Q_{1,1} \end{pmatrix} = Q, \\ W_{i,j \in \{0,1\}}^{VO} &\in \mathbb{R}^{\left(\frac{h_q}{2} \cdot 2d_h\right) \times D}, \quad W^{VO} = \begin{pmatrix} W_{0,0}^{VO}, W_{0,1}^{VO} \\ W_{1,0}^{VO}, W_{1,1}^{VO} \end{pmatrix}, \end{aligned}$$

$$\begin{aligned}
O_0 &= \text{softmax} \left( \frac{Q_{0,0} (\hat{\mathbf{c}}_0^{\text{KV}})^\top}{\sqrt{d_h}} \right) \hat{\mathbf{c}}_0^{\text{KV}} \in \mathbb{R}^{B \times 1 \times \frac{h_q}{2} \times 2d_h}, \\
O_1 &= \text{softmax} \left( \frac{Q_{1,1} (\hat{\mathbf{c}}_1^{\text{KV}})^\top}{\sqrt{d_h}} \right) \hat{\mathbf{c}}_1^{\text{KV}} \in \mathbb{R}^{B \times 1 \times \frac{h_q}{2} \times 2d_h}, \\
\tilde{O}_0 &= O_0 W_{0,0}^{VO} \in \mathbb{R}^{B \times 1 \times D}, \quad \tilde{O}_1 = O_1 W_{1,1}^{VO} \in \mathbb{R}^{B \times 1 \times D}, \\
O &= \text{AllReduce} \left( \tilde{O}_0 + \tilde{O}_1 \right) \in \mathbb{R}^{B \times 1 \times D}.
\end{aligned} \tag{3}$$

### 3.3 Matrix Absorption

Considering we apply orthogonal transformations  $U$  to reparameterize weight matrices, which involves matrix absorption. To make this process intuitive, we here present the complete calculation pipeline of MLA and show how the absorbed matrices from Section 3.1 are derived.

As stated in Section 3.1, MLA saves KV cache by multiplying the low-rank compress matrix  $W^{DKV} \in \mathbb{R}^{D \times 4d_h}$  with the input sequence  $X \in \mathbb{R}^{B \times L \times D}$  to obtain low-rank latent features  $\mathbf{c}^{\text{KV}}$ . Then, it uses the matrices  $W^{UK}, W^{UV} \in \mathbb{R}^{4d_h \times (h_q \cdot d_h)}$  to derive the full-heads key  $\mathbf{k}$  and value  $\mathbf{v}$ . Additionally, MLA also can decompose  $W^Q \in \mathbb{R}^{D \times (h_q \cdot d_h)}$  to  $W^{DQ} \in \mathbb{R}^{D \times r_q}$  and  $W^{UQ} \in \mathbb{R}^{r_q \times (h_q \cdot d_h)}$ , which reduces the activation memory during training. For positional embedding, MLA uses a decoupled RoPE strategy that uses additional multi-head queries  $\mathbf{q}^{\text{PE}}$  and a shared key  $\mathbf{k}^{\text{PE}}$ , generated by  $W^{QR} \in \mathbb{R}^{r_q \times (h_q \cdot d_r)}$  and  $W^{KR} \in \mathbb{R}^{D \times d_r}$ , to carry the rotary positional embeddings. The final attention output  $\tilde{O}$  is computed by separately combining the non-positional part ( $\mathbf{q} \mathbf{k}^\top$ ) and positional part ( $\mathbf{q}^{\text{PE}} (\mathbf{k}^{\text{PE}})^\top$ ), followed by projection with  $W^O \in \mathbb{R}^{(h_q \cdot d_h) \times D}$ .

$$\begin{aligned}
\mathbf{c}^{\text{KV}} &= X W^{DKV}, \quad \mathbf{c}^Q = X W^{DQ}, \quad \hat{\mathbf{c}}^{\text{KV}} = \text{RMSNorm}(\mathbf{c}^{\text{KV}}), \\
\mathbf{q} &= \mathbf{c}^Q W^{UQ}, \quad \mathbf{k} = \hat{\mathbf{c}}^{\text{KV}} W^{UK}, \quad \mathbf{v} = W^{UV} \hat{\mathbf{c}}^{\text{KV}}, \\
\mathbf{q}^{\text{PE}} &= \text{RoPE}(\mathbf{c}^Q W^{QR}), \quad \mathbf{k}^{\text{PE}} = \text{RoPE}(X W^{KR}), \\
O &= \text{softmax} \left( \frac{\mathbf{q} \mathbf{k}^\top + \mathbf{q}^{\text{PE}} (\mathbf{k}^{\text{PE}})^\top}{\sqrt{d_h + d_r}} \right) \mathbf{v}, \quad \tilde{O} = O W^O.
\end{aligned} \tag{4}$$

In Equation 4, the RoPE component is explicitly isolated, allowing us to restructure the attention computation using associativity of matrix multiplication. For clarity, we can temporarily omit positional encoding components and the scaling factor.

$$\begin{aligned}
O W^O &= \text{softmax}(\mathbf{q} \mathbf{k}^\top) \mathbf{v} W^O \\
&= \text{softmax}(\mathbf{q} (\hat{\mathbf{c}}^{\text{KV}} W^{UK})^\top) \hat{\mathbf{c}}^{\text{KV}} W^{UV} W^O \\
&= \text{softmax}(Q(\hat{\mathbf{c}}^{\text{KV}})^\top) \hat{\mathbf{c}}^{\text{KV}} W^{VO}.
\end{aligned} \tag{5}$$

Here, the matrix  $W^{UK}$  can be absorbed into  $\mathbf{q}$  to derive  $Q$  in Equation 2. Similarly, the matrix  $W^{UV}$  can be absorbed into  $W^O$ . In practice, however,  $W^{UV}$  is typically not absorbed into  $W^O$  to avoid generating an impractically large matrix.

## 4 Tensor Parallel Latent Attention (TPLA)

Motivated by the hardware efficiency of GLA, we retain its core principle of distributing latent KV across GPUs to mitigate memory wastage and communication overload. However, directly translating an existing MLA-based model to GLA incurs a significant performance penalty, as shown in Figure 2. This degradation stems from a key limitation in standard GLA: the latent vector within each group only accesses half of the query heads, restricting the model’s expressive power and leading to suboptimal performance. Moreover, training a new GLA model from scratch requires a substantial cost. To address this, we further propose Tensor-Parallel Latent Attention (TPLA). Unlike standard GLA, TPLA partitions latent vectors into two groups while preserving full query heads visibility. Specifically,

$$\begin{aligned} \mathbf{c}_0^{\text{KV}}, \mathbf{c}_1^{\text{KV}} &\in \mathbb{R}^{B \times L \times 2d_h}, \quad \left\{ \begin{array}{l} \hat{\mathbf{c}}_0^{\text{KV}} = \text{RMSNorm}(\mathbf{c}_0^{\text{KV}}) \in \mathbb{R}^{B \times L \times 2d_h}, \\ \hat{\mathbf{c}}_1^{\text{KV}} = \text{RMSNorm}(\mathbf{c}_1^{\text{KV}}) \in \mathbb{R}^{B \times L \times 2d_h}, \end{array} \right. \\ Q_0, Q_1 &\in \mathbb{R}^{B \times 1 \times h_q \times 2d_h}, \quad (Q_0, Q_1) = Q, \\ W_0^{VO}, W_1^{VO} &\in \mathbb{R}^{(h_q \cdot 2d_h) \times D}, \quad (W_0^{VO}, W_1^{VO}) = W^{VO}, \\ O_0 = \text{softmax} \left( \frac{Q_0 (\hat{\mathbf{c}}_0^{\text{KV}})^{\top}}{\sqrt{d_h}} \right) \hat{\mathbf{c}}_0^{\text{KV}} &\in \mathbb{R}^{B \times 1 \times h_q \times 2d_h}, \\ O_1 = \text{softmax} \left( \frac{Q_1 (\hat{\mathbf{c}}_1^{\text{KV}})^{\top}}{\sqrt{d_h}} \right) \hat{\mathbf{c}}_1^{\text{KV}} &\in \mathbb{R}^{B \times 1 \times h_q \times 2d_h}, \quad (6) \\ \tilde{O}_0 = O_0 W_0^{VO} &\in \mathbb{R}^{B \times 1 \times D}, \quad \tilde{O}_1 = O_1 W_1^{VO} \in \mathbb{R}^{B \times 1 \times D}, \\ O = \text{AllReduce} \left( \tilde{O}_0 + \tilde{O}_1 \right) &\in \mathbb{R}^{B \times 1 \times D}. \quad (7) \end{aligned}$$

This design ensures each latent vector attends to all query heads, mitigating the most performance loss. The residual performance loss now stems exclusively from tensor-parallel partitioning effects in RMSNorm and softmax operations. Through carefully designed mathematical reparameterization, TPLA can restore near-MLA performance. For illustration, we consider the case where the tensor-parallel degree of latent attention is 2, though the approach naturally scales to higher degrees.

### 4.1 RMSNorm Slicing

In MLA-like models, the “kv\_a\_layernorm” module normalizes input vectors using the Root Mean Square (RMS) value. Given an input vector  $\mathbf{x} \in \mathbb{R}^d$  (e.g.,  $d = 4d_h$ ), the RMSNorm is computed as:

$$\begin{aligned} \text{RMS}(\mathbf{x}) &= \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon} \\ &= \sqrt{\frac{1}{d} \|\mathbf{x}\|_2^2 + \epsilon}, \quad (8) \end{aligned}$$

$$\begin{aligned} \text{RMSNorm}(\gamma, \mathbf{x}) &= \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \odot \gamma \\ &= \text{RMSNorm}(\mathbf{1}, \mathbf{x}) \odot \gamma, \quad (9) \end{aligned}$$

where  $\epsilon$  is a small constant for numerical stability;  $\gamma \in \mathbb{R}^d$  is a learned scaling parameter and  $\odot$  denotes element-wise multiplication.

However, we face the following challenge when applying this to tensor-parallel processing of latent attention: When input latent vector  $\mathbf{x} \in \mathbb{R}^d$  is split into two partitions,  $\mathbf{x}^{(0)} \in \mathbb{R}^{d/2}$  and  $\mathbf{x}^{(1)} \in \mathbb{R}^{d/2}$ , across different devices, the RMS computation on each local device uses only half the original dimension ( $d/2$ ), while the true normalization requires the full  $\text{RMS}(\mathbf{x})$  over dimension  $d$ .

To resolve this discrepancy, we introduce an orthogonal transformation  $U \in \mathbb{R}^{d \times d}$  ( $U U^{\top} = \mathbf{I}$ ) to reparameterize this module. Before introducing the conditions that this transformation  $U$  need satisfy, we first establish that RMSNorm can, in principle, be realized in a mathematically equivalent form under any orthogonal transformation.

**Proposition 1.**

$$\text{RMSNorm}(\mathbf{1}, \mathbf{c}) = \text{RMSNorm}(\mathbf{1}, \mathbf{c} U) U^{\top} \quad (10)$$

*Proof.* we first represent the RMSNorm process as matrix multiplication. Let  $\mathbf{c} \in \mathbb{R}^{L \times d}$  be the input latent vector (for simplicity, we omit the batch size), we obtain:

$$\text{RMSNorm}(\gamma, \mathbf{c}) = \text{RMSNorm}(\mathbf{1}, \mathbf{c}) W_\gamma, \quad (11)$$

$$= D_c \mathbf{c} W_\gamma, \quad (12)$$

where  $D_c$  is a diagonal matrix of size  $L \times L$  with the reciprocal of the RMS values on the diagonal and  $W_\gamma$  is also a diagonal matrix of size  $d \times d$  with each learnable scaling parameter:

$$D_c = \text{diag}\left(\frac{1}{\text{RMS}(c_1)}, \frac{1}{\text{RMS}(c_2)}, \dots, \frac{1}{\text{RMS}(c_L)}\right), \quad (13)$$

$$W_\gamma = \text{diag}(\gamma_1, \gamma_2, \dots, \gamma_d). \quad (14)$$

Since the orthogonal transformation preserves the norm ( $\|\mathbf{c}U\|_2^2 = \|\mathbf{c}\|_2^2$ ), we easily have  $\text{RMS}(\mathbf{c}) = \text{RMS}(\mathbf{c}U)$ , i.e.,  $D_{cU} = D_c$ . Thus, we can have:

$$\text{RMSNorm}(\gamma, \mathbf{c}U) U^\top = D_c \mathbf{c}U W_\gamma U^\top. \quad (15)$$

Matrix multiplication does not satisfy the commutative property. Therefore, when and only when  $W_\gamma = \mathbf{I}$ , we can further prove:

$$\begin{aligned} \text{RMSNorm}(\mathbf{1}, \mathbf{c}U) U^\top &= D_c \mathbf{c}U \mathbf{I} U^\top \\ &= D_c \mathbf{c} \\ &= \text{RMSNorm}(\mathbf{1}, \mathbf{c}). \end{aligned} \quad (16)$$

□

Given by Equation 5, Equation 11 and Proposition 1, we can absorb  $W_\gamma$  into up-projection matrix  $W^{UKV} = (W^{UK}, W^{UV})$  to achieve the  $\gamma = 1$ , ensuring the orthogonal transformations  $U$  to  $\mathbf{c}$  with keeping the RMSNorm value no change. In addition, the  $U^\top$  can be further absorbed into  $W^{UKV}$ ;  $U$  can be absorbed into  $W^{DKV}$ , yielding the reparameterized weight matrix:

$$W_{new}^{UKV} = U^\top W_\gamma W^{UKV}, \quad W_{new}^{DKV} = W^{DKV} U. \quad (17)$$

We have proved that any transformation  $U$  can ensure the equivalence of RMSNorm. Now we will define some conditions that serve as the computational basis for  $U$ , deferring the specific calculation method to a later section.

**Condition 1** (RMSNorm Slicing Condition).

$$\alpha \|\mathbf{(c}U)_0\|_2^2 \approx \beta \|\mathbf{(c}U)_1\|_2^2 \approx \|\mathbf{c}U\|_2^2 = \|\mathbf{c}\|_2^2. \quad (18)$$

Here,  $\alpha$  and  $\beta$  are fixed constants, invariant to changes in the input data distribution (How to calculate their specific values is detailed in Section 4.3).  $(\mathbf{c}U)_0$  and  $(\mathbf{c}U)_1$  are the two partitions of the transformed  $\mathbf{c}$  split across devices. By satisfying this, the new RMS values computed from the two partitions are proportional to the global value, thereby providing an accurate approximation of the global RMSNorm:

$$\begin{aligned} \text{RMS}(\mathbf{c}) &= \sqrt{\frac{1}{d} \|\mathbf{c}\|_2^2 + \epsilon} \\ &\approx \sqrt{\frac{\alpha}{d} \|(\mathbf{c}U)_0\|_2^2 + \epsilon} \\ &\approx \sqrt{\frac{\alpha}{2}} \text{RMS}((\mathbf{c}U)_0) \approx \sqrt{\frac{\beta}{2}} \text{RMS}((\mathbf{c}U)_1). \end{aligned} \quad (19)$$

Thus, we can compute RMSNorm in a tensor-parallel manner while maintaining the integrity of the normalization process.

## 4.2 Softmax Slicing

In common tensor-parallel techniques, matrices are typically split across devices to perform either row or column parallelism. For our TPLA attention score computation, row parallelism is employed, where the weight matrix  $A$  is split across devices according to its rows. To ensure a valid matrix multiplication, the input matrix  $X$  is correspondingly partitioned column-wise into  $X_1$  and  $X_2$ , such that

$$XA = (X_1 \quad X_2) \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = X_1 \cdot A_1 + X_2 \cdot A_2 = Y_1 + Y_2 = Y \quad (20)$$

where  $X_1$  and  $A_1$  are computed on GPU 0 to produce  $Y_1$ , and  $X_2$  and  $A_2$  are computed on GPU 1 to produce  $Y_2$ . The final output  $Y$  is then all-reduced by summing  $Y_1$  and  $Y_2$ .

In the context of softmax computation (Equation 4), TPLA partitions  $\hat{\mathbf{c}}^{\text{KV}}$  and ensures that the computation of positional components remains unaffected. Specifically, the shard of the key positional embedding  $\mathbf{k}^{\text{PE}}$  must be replicated across devices so that the local positional values remain consistent with the global values. As for non-positional parts ( $Q(\hat{\mathbf{c}}^{\text{KV}})^\top$ ), the latent vectors  $\mathbf{c}$  are split into two devices, and each device performs only its local computation, i.e., GPU 0 computes  $Q_0(\hat{\mathbf{c}}_0^{\text{KV}})^\top$ , while GPU 1 computes  $Q_1(\hat{\mathbf{c}}_1^{\text{KV}})^\top$ . However, in most cases,

$$\begin{aligned} & \text{softmax}(Q(\hat{\mathbf{c}}^{\text{KV}})^\top + \mathbf{q}^{\text{PE}}(\mathbf{k}^{\text{PE}})^\top) \\ &= \text{softmax}(Q_0(\hat{\mathbf{c}}_0^{\text{KV}})^\top + Q_1(\hat{\mathbf{c}}_1^{\text{KV}})^\top + \mathbf{q}^{\text{PE}}(\mathbf{k}^{\text{PE}})^\top) \\ &\neq \text{softmax}(Q_0(\hat{\mathbf{c}}_0^{\text{KV}})^\top + \mathbf{q}^{\text{PE}}(\mathbf{k}^{\text{PE}})^\top) \quad [\text{GPU 0}] \\ &\neq \text{softmax}(Q_1(\hat{\mathbf{c}}_1^{\text{KV}})^\top + \mathbf{q}^{\text{PE}}(\mathbf{k}^{\text{PE}})^\top) \quad [\text{GPU 1}] \end{aligned}$$

Thus, the challenge of TPLA is how to ensure the global value of  $Q(\hat{\mathbf{c}}^{\text{KV}})^\top$  can be approximated from local computations. We first show that applying any orthogonal transformation  $U$  does not alter the equivalence of the original softmax output. Based on Equation 5, we easily have:

$$\begin{aligned} Q(\hat{\mathbf{c}}^{\text{KV}})^\top &= QU(\hat{\mathbf{c}}^{\text{KV}}U)^\top = \mathbf{q}(U^\top W^{UK})^\top(\hat{\mathbf{c}}^{\text{KV}}U)^\top \\ &= Q'(\hat{\mathbf{c}}^{\text{KV}}U)^\top \end{aligned} \quad (21)$$

Analogous to Section 4.1, by absorbing  $U$  into  $W^{UKV}$  and  $W^{DKV}$  (equivalently, into  $Q$  to obtain  $Q'$ ), we can impose an orthogonal transformation  $U$ , which preserves the original softmax computation and must satisfy:

**Condition 2** (Softmax Slicing Condition).

$$Q'(\hat{\mathbf{c}}^{\text{KV}}U)^\top \approx \mu Q'_0(\hat{\mathbf{c}}^{\text{KV}}U)_0^\top \approx \nu Q'_1(\hat{\mathbf{c}}^{\text{KV}}U)_1^\top \quad (22)$$

Accordingly, by determining the coefficients  $\mu$  and  $\nu$ , each device can perform its local computation and scale by the factor, thereby approximating the global value.

### 4.3 Reparameterization Methods

From the derivation above, we need to find one orthogonal transformation matrix  $U$  applied to projection weights, ensuring that the transformation satisfies Condition 1 and Condition 2 — local computations can accurately approximate the global RMSNorm and softmax values. To achieve this, we explore two potential methods: Hadamard Matrix Transformation and Principal Component Analysis (PCA).

#### 4.3.1 Hadamard Matrix Transformation

Hadamard matrix is a special orthogonal matrix where each entry is either +1 or -1. It operates by balancing the numbers, thereby reducing extreme numerical deviations and promoting a more uniform distribution of data. In practice, we typically use the function `scipy.linalg.hadamard(d)` to generate a Sylvester-type Hadamard matrix (also known as a Walsh-Hadamard matrix)  $H_d \in \mathbb{R}^{d \times d}$ , constructed using a deterministic recursive rule:

$$H_{2n} = \begin{pmatrix} H_n & H_n \\ H_n & -H_n \end{pmatrix}, \quad H_1 = (1). \quad (23)$$

To increase robustness, a random diagonal matrix  $D$ , with entries drawn from  $\pm 1$  is multiplied with  $H_d$ , thereby breaking deterministic structure while preserving orthogonality. Since  $H_d H_d^\top = d \cdot \mathbf{I}$ , orthonormality is achieved by scaling  $H_d$  by  $\frac{1}{\sqrt{d}}$ , ensuring that normalization values are preserved.

Take an illustrative example. Consider a 4-dimensional vector  $\mathbf{c} = (100, 0, 0, 0)$  and the  $4 \times 4$  Hadamard matrix  $H_4$ . The transformed vector  $\mathbf{c}' = \mathbf{c}H_4$  is:

$$\mathbf{c}' = (100, 0, 0, 0) \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{pmatrix} = (50, 50, 50, 50). \quad (24)$$

When applied to an input vector  $\mathbf{c}$ :  $\mathbf{c}' = \mathbf{c} H_d$ ,  $d = 4d_h$ , we obtain:  $\frac{\|(\mathbf{c} H_d)_1\|_2^2}{d/2} \approx \frac{\|(\mathbf{c} H_d)_2\|_2^2}{d/2} \approx \frac{\|\mathbf{c} H_d\|_2^2}{d} = \frac{\|\mathbf{c}\|_2^2}{d}$ , satisfies our key Condition 1 and easily determine  $\alpha = 2$ . This uniformity minimizes approximation error in tensor-parallel RMSNorm, validated experimentally in Figure 2.

However, satisfying Condition 2 is more challenging. While the magnitudes of the Hadamard transformed vector elements are balanced, due to the presence of both positive and negative signs, this transformation does not guarantee that the multiplication of the two parts will be approximately clear. To illustrate this, consider the following example: let  $Q = (100, 0, 0, 0)$  and  $\mathbf{c} = (0, 0, 80, 0)$ . After applying the Hadamard transformation, we have:

$$\begin{aligned} Q' &= QH_d = (50, 50, 50, 50), \\ \mathbf{c}' &= \mathbf{c}H_d = (40, 40, -40, -40). \end{aligned}$$

The element-wise product is:  $Q \cdot \mathbf{c}' = (200, 200, -200, -200)$ . When this product is split into two parts, we get:  $400 \neq -400 \neq 0$ . This demonstrates that a standard Hadamard transformation cannot ensure Condition 2. A potential direction to address this issue is to search for an optimized Hadamard matrix via dimension permutations that minimizes the discrepancy between partitions. We leave the investigation of such optimized transformations for future work.

### 4.3.2 Principal Component Analysis (PCA)

PCA is a widely used technique in statistics and machine learning for dimensionality reduction, feature extraction, etc. It transforms a dataset into a new coordinate system such that the greatest variances of the data are captured along the new axes (principal components). Each subsequent component is orthogonal to (i.e., uncorrelated with) the preceding ones. In our context, we leverage this property to project data onto orthogonal dimensions, with the eigenvalues indicating the variance captured along each eigenvector and thus reflecting the statistical importance of each dimension. Moreover, for mean-centered features, the variance is equivalent to mean of the squared values, closely related to squared RMS value.

To implement this, we process a calibration dataset (e.g., Wikitext-2) to collect the KV latent cache (excluding position features) represented by  $F \in \mathbb{R}^{(B \cdot L) \times d}$ . We then compute the eigenvectors  $U$  and eigenvalues  $\Lambda$  by performing eigenvalue decomposition on the covariance matrix  $\Sigma_F = U\Lambda U^\top$ .

Based on Condition 1, we define  $\alpha$  as proportion of variance captured by the first  $d/2$  principal components. Similarly,  $\beta$  represents the proportion of variance captured by the remaining components. These ratios are as follows:

$$\alpha = \frac{\sum_{i=1}^{d/2} \lambda_i}{\sum_{i=1}^d \lambda_i}, \quad \beta = \frac{\sum_{i=d/2}^d \lambda_i}{\sum_{i=1}^d \lambda_i}. \quad (25)$$

For Condition 2, the metrics  $\mu$  and  $\nu$  are defined in the same manner, making them equivalent to  $\alpha$  and  $\beta$ , respectively.

## 4.4 TPLA as a Special Case of GLA

Tensor parallelism in GLA employs a two-dimensional sharding scheme, splitting both head axis  $h_q$  and the latent dimension axis  $4d_h$  across devices. For a query tensor  $Q \in \mathbb{R}^{B \times L \times h_q \times 4d_h}$ , this partitioning yields four logical sub-tensors:

$$Q = \begin{pmatrix} Q_{0,0} & Q_{0,1} \\ Q_{1,0} & Q_{1,1} \end{pmatrix}, \quad \text{where } Q_{i,j} \in \mathbb{R}^{B \times L \times \frac{h_q}{2} \times 2d_h}.$$

In standard GLA, they are distributed with only two devices. Thus, only two diagonal blocks can be materialized locally—one per device—without additional communication:

$$\begin{cases} \text{Device 0: } Q_{0,0} \in \mathbb{R}^{B \times L \times \frac{h_q}{2} \times 2d_h}, \\ \text{Device 1: } Q_{1,1} \in \mathbb{R}^{B \times L \times \frac{h_q}{2} \times 2d_h}. \end{cases}$$

Each latent slice (of width  $2d_h$ ) is paired with only half of the query heads ( $h_q/2$ ) and thus unable to access the off-diagonal head slices  $Q_{1,0}$  and  $Q_{0,1}$ . In effect, these parts do not contribute to the computation, resulting in significant performance degradation.

In contrast, TPLA overcomes this limitation by enabling each partitioned latent vector to attend to all query heads. It achieves this by reformulating the computation to be algebraically equivalent to a GLA system with double the number of heads. Concretely, define a conceptual query tensor  $Q'$  that duplicates the original query along the head dimension:

$$Q' = \begin{pmatrix} Q_{0,0} & Q_{0,1} \\ Q_{1,0} & Q_{1,1} \\ Q_{0,0} & Q_{0,1} \\ Q_{1,0} & Q_{1,1} \end{pmatrix} \in \mathbb{R}^{B \times L \times (2h_q) \times (4d_h)},$$

For  $h_q$  original heads, TPLA’s duplication additional creates  $h_q$  heads. This is algebraically equivalent to a GLA system with  $2h_q$  heads  $4d_h$  latent dimension. Thus, we can perfectly follow the same as TPLA sharding way. When split into two device, we have:

$$\begin{cases} \text{Device 0: } [Q_{0,0} \quad Q_{0,1}] \in \mathbb{R}^{B \times L \times h_q \times 4d_h} \\ \text{Device 1: } [Q_{1,0} \quad Q_{1,1}] \in \mathbb{R}^{B \times L \times h_q \times 4d_h} \end{cases}$$

Generalizing to  $k$  devices, let  $g$  denote the TPLA replication factor (number of latent-cache-slice groups). TPLA divides  $k$  devices into  $r$  group size of size  $k/g$ . Each group holds a disjoint slice of the latent axis of width  $4d_h/g$  and replicates the complete set of head parameters. Within each group, the head axis is sharded across the  $k/g$  devices. Consequently, each device processes  $\frac{h_q}{k/g}$  heads, and  $\frac{4d_h}{g}$  latent features. For  $g = 2$ ,  $k = 2$ , this recovers the two-device case above, where each device receives  $h_q$  heads and  $2d_h$  latent width. The computational complexity arising from parameter replication is analyzed in Section 4.5.

In summary, because TPLA preserves GLA’s sharding pattern—differing only by a constant-factor replication of head parameters—state-of-the-art attention optimizations (e.g., FlashAttention-3) can be applied to TPLA without substantial changes to the underlying framework.

#### 4.5 Prefill-Decode Separation

Large language model inference is usually into two phases with distinct performance characteristics: *prefill* and *decode*. The prefill phase processes the entire prompt in a single, parallel pass to compute the initial Key-Value (KV) cache. This large-batch computation is fundamentally compute-bound. The subsequent decode phase autoregressively generates one token at a time. Each generation step requires reading the entire, growing KV cache from high-bandwidth memory (HBM) to on-chip SRAM. As the context length increases, this large data transfer becomes the primary bottleneck, making the decode phase memory-bound.

Our proposed technique, TPLA, addresses this challenge by reducing the KV cache size on each device. This reduction effectively alleviates the memory bandwidth bottleneck at the cost of a minor increase in computation. In essence, TPLA shifts the decode phase from being memory-bound towards being more compute-bound. A detailed analysis is as below.

**Complexity Analysis of TPLA** To maximize the degree of tensor parallelism and enable acceleration methods compatible with GLA, TPLA requires the replication of head-specific parameters across latent attention groups, as discussed in Section 4.4. Specifically, let’s analyze the case with a tensor parallelism (TP) degree of 2. We consider a hidden state  $X \in \mathbb{R}^{L_q \times D}$  for a single-batch inference using the MLA-absorbing strategy. The dominant cost lies in the attention computation. For a KV cache of length  $S_{KV}$ , the complexity of the TPLA attention module (Equation 6) is approximately  $\mathcal{O}(L_q \times S_{KV} \times h_q \times 2d_h \times 2)$ . In comparsion, for MLA (Equation 1), with TP=2, the heads are split into two groups of  $\frac{h_q}{2}$ , leading to a complexity of  $\mathcal{O}(L_q \times S_{KV} \times \frac{h_q}{2} \times 4d_h \times 2)$ . These two complexities are arithmetically equivalent. (Strictly speaking, positional components introduce additional overhead, since TPLA doubles the number of heads without reducing the RoPE dimension, but this effect is relatively minor.) Similarly, the  $\hat{\mathcal{O}}$  computations are also equivalent.

Beyond the main attention computation, TPLA modifies other calculations: the computation of  $\mathbf{c}^{KV}$  in TPLA is distributed across two devices, reducing complexity by  $\mathcal{O}(L_q \times D \times 2d_h \times 2)$ , while the computation of  $Q$  increases by  $\mathcal{O}(L_q \times D \times 2h_q \times d_h)$ . However, as context length grows, the overall cost is increasingly dominated by the self-attention module (see Figure 4).

To mitigate the additional computational overhead of TPLA, we also strategically decouple the attention mechanisms: retaining standard MLA during compute-intensive prefilling to minimize computation and reduce loss caused by converting MLA to TPLA, while activating TPLA exclusively during memory-bound decoding to minimize KV cache footprint. This hybrid approach thereby further optimizes performance by matching each phase to its most suitable mechanism.

**Discussion** The above analysis focuses on MLA-absorbing computation. However, when constructing the full-size KV cache and performing attention, TPLA doubles the number of heads while maintaining the same head dimension  $d_h$ . As a result, the overall computational load increases, making training TPLA from scratch potentially costly. Designing effective and efficient training strategies for TPLA remains an open problem, which we leave for future work. Nevertheless, one practical pathway is to adopt the existing MLA design during training and then convert it to TPLA with only minimal loss. This approach allows us to retain training efficiency while still benefiting from TPLA’s advantages in inference scenarios.

Table 1: WikiText-2 Perplexity and Commonsense reasoning performance when converting the MLA to TPLA. The six benchmarks include MMLU, ARC (easy and challenge), PIQA, HellaSwag, OpenBookQA (OBQA), and Winogrande(WG).

| Model            | PPL $\downarrow$ | Avg. $\uparrow$ | MMLU  | ARC   | PIQA  | HellaSwag | WG    | OBQA  |
|------------------|------------------|-----------------|-------|-------|-------|-----------|-------|-------|
| DeepSeek-V2-Lite | 6.31             | 61.75           | 43.19 | 60.39 | 80.20 | 74.46     | 65.43 | 45.80 |
| - GLA            | 2212.            | 33.77           | 25.32 | 26.77 | 51.47 | 25.65     | 49.88 | 23.60 |
| - TPLA           | 7.24             | 54.33           | 37.67 | 51.50 | 75.46 | 63.56     | 59.19 | 38.60 |
| - TPLA (align)   | 6.51             | 61.52           | 42.72 | 62.58 | 79.82 | 73.32     | 65.90 | 44.80 |
| - TPLA (pd sep.) | 6.31             | 61.44           | 43.19 | 60.14 | 80.09 | 74.41     | 65.59 | 45.20 |
| DeepSeek-V2      | 3.89             | 68.32           | 51.91 | 69.09 | 83.13 | 82.17     | 74.03 | 49.60 |
| - TPLA           | 4.72             | 63.40           | 47.19 | 65.04 | 80.69 | 75.46     | 66.61 | 45.40 |
| DeepSeek-V3      | 3.24             | 72.10           | 60.85 | 77.16 | 85.58 | 85.41     | 75.22 | 48.40 |
| - TPLA           | 4.02             | 68.00           | 54.88 | 75.25 | 82.70 | 80.69     | 69.46 | 45.00 |
| Kimi-K2-Base     | 1.91             | 73.52           | 63.20 | 78.75 | 85.47 | 87.55     | 75.93 | 50.20 |
| - TPLA           | 2.44             | 70.49           | 57.64 | 76.00 | 83.79 | 83.53     | 72.38 | 49.60 |
| LLaMA-2-7B       | 5.47             | 59.85           | 41.43 | 59.24 | 78.40 | 73.29     | 64.96 | 41.80 |
| - TransMLA       | 5.88             | 58.95           | 40.38 | 57.64 | 78.18 | 70.59     | 62.90 | 44.00 |
| - TPLA           | 6.74             | 54.68           | 36.12 | 53.21 | 74.81 | 64.52     | 59.04 | 40.40 |

## 5 Experiment

An advantage of TPLA over GLA [20] is that TPLA can be applied without training a model from scratch. It allows direct loading of models originally trained with MLA (e.g., the DeepSeek series [53, 17, 19], Kimi-k2 [54], TransMLA [16]), and—through our proposed reparameterization method and Prefill/Decode Separation technology—mitigates performance degradation caused by changes in the attention mechanism.

### 5.1 Performance on Commonsense Tasks

In this section, we evaluate TPLA by directly loading MLA checkpoints *without any additional training* on short-text commonsense tasks. Performance is measured with the LightEval framework on MMLU [55], ARC (Easy/Challenge) [56], PIQA [57], HellaSwag [58], OpenBookQA (OBQA) [59], and WinoGrande (WG) [60]. Results are reported in Table 1. For **GLA**, following the procedure in Section 3.2, we partition the attention heads into two groups, assigning each group half of the latent dimension. As shown in Table 1, discarding half of each head’s KV cache causes severe performance degradation—for example, WikiText-2 perplexity (ppl) increases from 6.31 with MLA to 2212 with GLA—whereas **TPLA**, which allows each attention head to use the full latent dimension across different devices, maintains a ppl of 7.24. This comparison indicates that TPLA preserves MLA’s representational capacity while reducing the per-device KV-cache footprint. We therefore expect that pretraining TPLA from scratch would outperform GLA. For **TPLA**, we first use WikiText-2 [61] as a calibration set and, following Sections 4.1 and 4.2, slice the MLA components (the  $KV_a$  RMSNorm and the softmax) to obtain TPLA weights. As shown in Table 1, this requires no fine-tuning and yields only minor accuracy degradation. The reparameterization method used here is the PCA-based approach described in Section 4.3.2. For **TPLA (align)**, we use the SmollLM-Corpus [62] for lightweight alignment. First, we match the layer-wise input/output features of TPLA to those of the original MLA model using 256 random samples of length 2,048 for 10 epochs, minimizing MSE with the Muon optimizer (initial learning rate  $1e-6$ ). Next, we align the end-to-end model outputs using 100M tokens, following the TransMLA setting (batch size = 32, learning rate =  $2e-5$ , warmup ratio = 0.03, cosine scheduler, max sequence length = 4096). Experiments are conducted on a node with  $8 \times$  GPUs (96 GB per GPU,  $\sim 148$  FP16 TFLOPS each). This small amount of alignment data is sufficient to recover the performance of the converted model. For **TPLA (PD-sep.)**, we use MLA in the prefilling stage with the same reparameterization but *without* slicing the RMSNorm or softmax; prefilling thus behaves identically to the original MLA, and the KV cache can be partially reused by TPLA during decoding. By avoiding slicing for most tokens, this prefill–decode separation achieves performance close to the original model *without any training*. For **LLaMA-2-7B**, we first apply TransMLA [16] to convert MHA/GQA to MLA (64 RoPE dimensions and 512 NoPE dimensions—corresponding to a pruning ratio of 92.97%.) and then fine-tune to recover performance. We subsequently convert the MLA checkpoint released by TransMLA directly into TPLA. With TransMLA as a bridge, TPLA can be applied to pretrained models that originally use MLA, GQA, or MHA.

Table 2: Longbench performance when converting the MLA to TPLA.

| Model                   | Avg.  | MultiQA | SingleQA | Summarize | Few-Shot | Code  | Synthetic |
|-------------------------|-------|---------|----------|-----------|----------|-------|-----------|
| DeepSeek-V2-Lite        | 28.90 | 12.43   | 20.04    | 16.74     | 62.59    | 57.86 | 3.77      |
| - <i>TPLA</i>           | 10.98 | 6.96    | 9.20     | 6.91      | 25.29    | 14.41 | 3.11      |
| - <i>TPLA (align)</i>   | 22.60 | 10.97   | 14.67    | 16.59     | 58.03    | 31.59 | 3.77      |
| - <i>TPLA (pd sep.)</i> | 24.44 | 13.95   | 15.07    | 8.62      | 59.10    | 46.67 | 3.23      |
| DeepSeek-V3             | 58.19 | 55.37   | 51.65    | 23.97     | 69.42    | 80.09 | 68.63     |
| - <i>TPLA</i>           | 44.52 | 35.02   | 38.53    | 12.55     | 53.01    | 61.20 | 66.83     |
| - <i>TPLA (pd sep.)</i> | 56.04 | 53.01   | 50.17    | 21.39     | 67.22    | 75.97 | 68.47     |

These experiments demonstrate that converting MLA-based models to TPLA can effectively preserve performance. Given the benefits of TPLA for tensor parallelism, this presents a promising approach for efficient model deployment and acceleration.

## 5.2 Performance on Longbench Tasks

As context length grows, memory traffic increases and the KV-cache size becomes a primary driver of latency and throughput. To assess how TPLA converted from MLA behaves on long inputs, we evaluate on *LongBench* [63], a bilingual (English/Chinese), multi-task benchmark for long-context understanding that comprises 21 tasks across six categories (e.g., question answering, summarization, and few-shot learning). Because long-text inference is slower, we report results only for DeepSeek-V2-Lite and DeepSeek-V3. Due to GPU memory constraints, the maximum input context length is set to 31,500 tokens for DeepSeek-V2-Lite and 127,500 tokens for DeepSeek-V3. For each task, the output length is kept the same as in the original paper. The outcomes are summarized in Table 2. We observe that slicing errors in RMSNorm and softmax accumulate with sequence length, leading to some degradation on LongBench. **TPLA (align)** follows the same alignment recipe as in the previous section, but its effectiveness is limited because the alignment corpus is formed by concatenating short texts. In contrast, **TPLA (PD-sep.)** adopts a prefill–decode separation: MLA is used unchanged in the prefill stage (no slicing of RMSNorm/softmax), and the resulting KV cache is partially reused by TPLA during decoding, which reduces first-token latency and accuracy loss. On DeepSeek-V2-Lite, the training-free **TPLA (PD-sep.)** surpasses the aligned variant, and on DeepSeek-V3 the model retains strong long-form reasoning with only a modest average drop of 2.15%. These small losses, compared with training from scratch, are likely recoverable with a small amount of additional training.

## 5.3 Ablation Study

### 5.3.1 Part 1

We highlight two structural differences. (i) *Per-head latent capacity*: GLA gives each attention head only half of the latent dimension, whereas TPLA preserves the full latent dimension per head. (ii) *Prefill–decode (PD) separation*: during the compute-intensive prefill stage we keep the reparameterized MLA form *without* splitting RMSNorm or softmax; during decoding, TPLA uses PD separation while consuming the prefill KV cache. We analyze the results in Table 1 to quantify these effects:

- 1) **MLA → GLA conversion.** Directly converting MLA to GLA forces each attention head to access only half of its original latent representation, causing substantial information loss and a marked accuracy drop across all benchmarks.
- 2) **Prefill–decode separation.** Avoiding RMSNorm/softmax partitioning in prefill reduces approximation error for the vast majority of tokens. Moreover, the MLA reparameterization enables the prefill KV cache to be used directly by TPLA at decode time, improving both quality and efficiency.

### 5.3.2 Part 2

In Section 4, we identified **RMSNorm** and **softmax** as the primary sources of error when converting MLA to TPLA. To mitigate this, we proposed two reparameterization strategies, **Hadamard-based** and **PCA-based**, to reduce the performance degradation introduced by parallelizing these components. This section presents ablation studies analyzing the impact of each reparameterization method on individual modules. Figure 2 presents ablation results analyzing the effectiveness of each method. The key findings are:

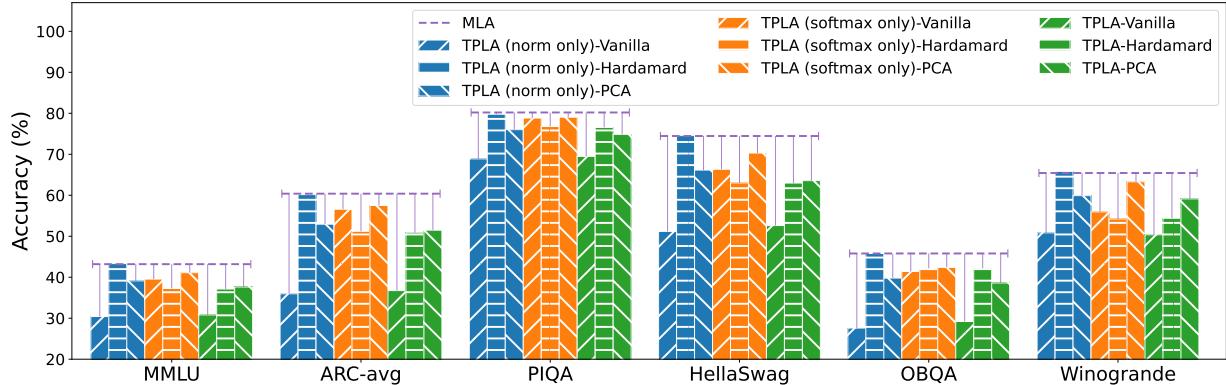


Figure 2: Accuracy across multiple benchmarks under different tensor-parallelism methods (indicated by colors) and reparameterization strategies (indicated by textures). The purple horizontal line marks the original DeepSeek-V2-Lite accuracy, and the vertical bars show each method’s accuracy drop relative to this MLA baseline. **TPLA (norm only)** parallelizes RMSNorm across two devices, followed by an allgather before the softmax. **TPLA (softmax only)** applies RMSNorm normally and parallelizes the softmax. **Original** splits parameters evenly; **Hadamard** balances parts prior to splitting; **PCA** concentrates information into earlier dimensions before splitting. To better visualize method-induced loss, TPLA results are reported without PD separation in this figure.

**1) Error ordering.** Empirically, slicing *RMSNorm* incurs the least loss, slicing *softmax* is worse, and slicing both is worst:

**2) TP on RMSNorm only:** The Hadamard-based method balances the norm computation across devices effectively, leading to performance comparable to the original MLA model on multiple tasks.

**3) TP on softmax only:** The PCA-based method concentrates information into the dimensions assigned to device 1, effectively preserving performance. In contrast, the Hadamard-based method fails to improve softmax accuracy. We hypothesize that the exponential nature of softmax makes it more sensitive to imbalance. Although Hadamard-based reparameterization achieves statistical balance across devices, small per-sample perturbations may result in significant asymmetries, adversely affecting final performance.

**4) TP over both RMSNorm and softmax.** When both components are parallelized, the PCA-based reparameterization consistently achieves the best performance. Consequently, we adopt this configuration for all experiments in the paper unless otherwise stated.

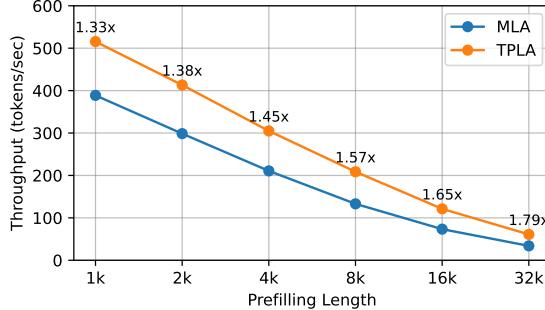
## 5.4 Inference Speedup with TPLA

### 5.4.1 Decoding Throughput

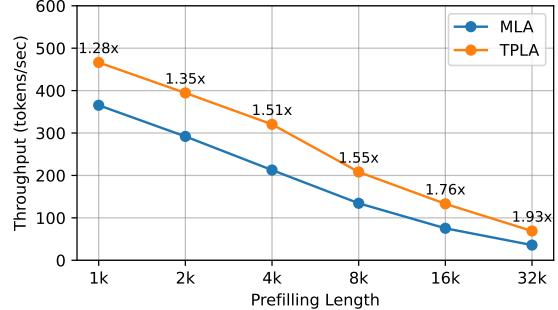
LLM decoding is often *memory-bandwidth bound*. TPLA splits each attention head’s input dimension across two devices, reducing per-device memory traffic and alleviating the bandwidth bottleneck. We evaluate the speedup of TPLA over MLA on two large models, **DeepSeek-V3-0324** (685B parameters) and **Kimi-K2-Base** (1T parameters). Because these models are extremely large and Mixture-of-Experts (MoE) routing can confound attention-speed effects, we remove MoE for timing. Both models are converted to **BF16**. All experiments use **FlashAttention-3** to ensure a fair comparison.

For **TPLA** with TP=2, the number of attention heads per device stays unchanged, while the latent dimension changes from  $64+512$  to  $(64+256) \times 2$ , so each device holds a 320-dimensional KV cache. For **MLA** with TP=2, the latent dimension is unchanged and heads are distributed across devices (e.g., DeepSeek-V3: 64 heads  $\times 2$ ; Kimi-K2: 32 heads  $\times 2$ ). For **MLA** with TP>2, we continue splitting along heads only. For **TPLA** with TP>2, we further split heads *in addition to* halving the latent dimension; for example, with TP=4 on Kimi-K2-TPLA, we use 32 heads  $\times 2$  per device with a 320-dimensional latent per head. In this setting, the per-device compute halves, while memory traffic matches TP=2; decoding remains memory-bound, so the speedup is similar to TP=2. Consequently, we report measurements on two GPUs.

Figure 3 configures the maximum batch size at each context length. At a decoding length of 4096, **TPLA** with  $2d_h$  achieves up to  $\sim 2\times$  the throughput of the single-head-latent **MLA** with  $4d_h$ , due to the smaller per-device KV cache. Our parallelization-friendly design raises peak throughput and is resilient under adverse serving loads. At a 32k context length, **DeepSeek-TPLA** is  $1.79\times$  faster than MLA, and **Kimi-K2-TPLA** is  $1.93\times$  faster.

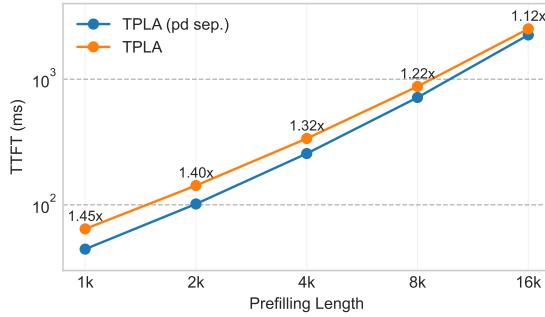


(a) DeepSeek-V3-0324

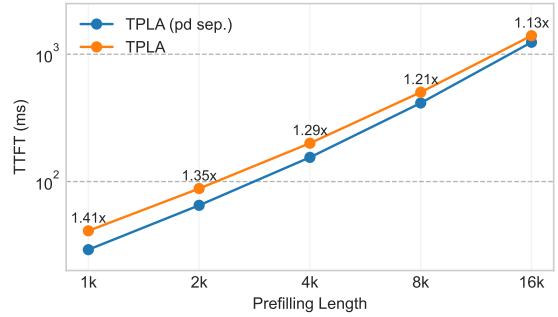


(b) Kimi-K2-Base

Figure 3: Throughput (Decoding) comparing MLA and TPLA.



(a) DeepSeek-V3-0324



(b) Kimi-K2-Base

Figure 4: Latency (TTFT) comparing TPLA and TPLA (pd sep.).

#### 5.4.2 Prefilling Latency

The *prefilling* stage of LLM inference is *compute-bound*. Under TPLA’s TP separation, each device retains the original number of heads, whereas MLA can reduce heads per device by splitting them across devices. As a result, the original TPLA is not ideal for the compute-bound prefill stage. To address this, we introduce **TPLA (sep.)**: it applies the same reparameterization to MLA but *does not* slice RMSNorm or softmax, thereby introducing no approximation error. During prefill, the structure matches MLA: under TP we do not change the latent dimension but partition heads across devices. This significantly reduces per-device compute and alleviates the compute bottleneck.

Figure 4 reports TTFT (Time to First Token) on two GPUs for MoE-removed DeepSeek-V3-0324 and Kimi-K2-Base. At a 1K prompt length, **TPLA (sep.)** is  $1.4\times$  faster than TPLA for both models. Given its accuracy-friendly design, this  $1.4\times$  gain is essentially a “free lunch.”

## 6 Conclusion, Limitation and Future Work

We introduce TPLA, which combines the KV cache compression efficiency of MLA with strong compatibility for Tensor Parallelism. It can directly inherit checkpoints from MLA-pretrained models. With two proposed reparameterization techniques, it substantially reduces the loss incurred by converting the attention formulation; combined with PD separation, the training-free conversion error can be driven to a very small level. We evaluate TPLA on commonsense reasoning tasks and the more challenging LongBench benchmark, finding that it preserves the original model’s performance well. Extensive ablations confirm the effectiveness of our TP slicing and reparameterization designs.

TPLA achieves up to  $2\times$  improvement in throughput, and PD separation delivers up to 29% latency reduction. Overall, TPLA shows strong potential as a powerful and efficient replacement for MLA.

**Limitation and Future Work.** Although PCA demonstrates better performance over Hadamard transform, it has inherent limitations. Specifically, PCA concentrates most of the data’s informative content in the first few dimensions, which provide a representative summary of the global structure. In contrast, the later dimensions primarily capture negligible noise and minor variations that contribute minimally to the overall representation. Consequently, TPLA with group-partitions  $g = 2$  can achieve good performance, but when  $g > 2$ , it probably fails to maintain effectiveness. By contrast, numerical-value balancing via orthogonal transforms, particularly the Hadamard transform, tends to be more effective when partitioning into multiple groups. Empirically, inserting a Hadamard transform into the RMSNorm slicing part yields almost no performance degradation. In future work, we will design and evaluate optimized Hadamard-like orthogonal matrices to balance softmax slicing, thereby improving both robustness and scalability. One advantage of TPLA is that it can directly inherit MLA checkpoints, but this also introduces some conversion errors. Our experiments fully validate TPLA’s expressive capacity and speed advantages. In future work, we will post-pretrain DeepSeek-V3, or train a TPLA-based model from scratch, to further demonstrate TPLA’s excellent expressiveness.

## References

- [1] OpenAI. Hello GPT-4o, 2024.
- [2] Anthropic. Claude 3.5 sonnet, 2024.
- [3] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.
- [4] Alec Radford. Improving language understanding by generative pre-training. 2018.
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [6] Chang Chi-Chih, Lin Wei-Cheng, Lin Chien-Yu, Chen Chong-Yan, Hu Yu-Fang, Wang Pei-Shuo, Huang Ning-Chi, Ceze Luis, Abdelfattah Mohamed, S., and Wu and, Kai-Chiang. Palu: Compressing kv-cache with low-rank projection. *arXiv preprint arXiv:2407.21118*, 2024.
- [7] Chang Chi-Chih, Lin Chien-Yu, Akhauri Yash, Lin Wei-Cheng, Wu Kai-Chiang, Ceze Luis, and Abdelfattah Mohamed, S. xv: Cross-layer svd for kv-cache compression. *arXiv preprint arXiv:2503.18893*, 2025.
- [8] Oren Matanel, Hassid Michael, Yarden Nir, Adi Yossi, and Schwartz Roy. Transformers are multi-state rnns. *arXiv preprint arXiv:2401.06104*, 2024.
- [9] Liu Zirui, Yuan Jiayi, Jin Hongye, Zhong Shaochen, Xu Zhaozhuo, Braverman Vladimir, Chen Beidi, and Hu Xia. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.
- [10] Zhang Muru, Mishra Mayank, Zhou Zhongzhu, Brandon William, Wang Jue, Kim Yoon, Ragan-Kelley Jonathan, Song Shuaiwen, Leon, Athiwaratkun Ben, and Dao Tri. Ladder-residual: parallelism-aware architecture for accelerating large model inference with communication overlapping. *arXiv preprint arXiv:2501.06589*, 2025.
- [11] Kim Han-Byul, Hoang Duc, Kundu Arnav, Samragh Mohammad, and Cho Minsik. Spd: Sync-point drop for efficient tensor parallelism of large language models. *arXiv preprint arXiv:2502.20727*, 2025.
- [12] Lamprecht Itay, Karnieli Asaf, Hanani Yair, Giladi Niv, and Soudry Daniel. Tensor-parallelism with partially synchronized activations. *arXiv preprint arXiv:2506.19645v1*, 2025.
- [13] Li Qingyuan, Zhang Bo, Ye Liang, Zhang Yifan, Wu Wei, Sun Yerui, Ma Lin, and Xie Yuchen. Flash communication: Reducing tensor parallelization bottleneck for fast large language model inference. *arXiv preprint arXiv:2412.04964*, 2024.
- [14] Smith Shaden, Patwary Mostofa, Norick Brandon, LeGresley Patrick, Rajbhandari Samyam, Casper Jared, Liu Zhun, Prabhumoye Shrimai, Zerveas George, Korthikanti Vijay, Zhang Elton, Child Rewon, Aminabadi Reza, Yazdani, Bernauer Julie, Song Xia, Shoeybi Mohammad, He Yuxiong, Houston Michael, Tiwary Saurabh, and Catanzaro and, Bryan. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.
- [15] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

- [16] Fanxu Meng, Pingzhi Tang, Zengwei Yao, and Muhan Zhang. Transmla: Multi-head latent attention is all you need. *arXiv preprint arXiv:2502.07864*, 2025.
- [17] DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *CoRR*, abs/2405.04434, 2024.
- [18] AI@Meta. Llama 3 model card, 2024.
- [19] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [20] Zadouri Ted, Strauss Hubert, and Dao Tri. Hardware-efficient attention for fast decoding. *arXiv preprint arXiv:2505.21487v1*, 2025.
- [21] Ge Suyu, Zhang Yunan, Liu Liyuan, Zhang Minjia, Han Jiawei, and Gao Jianfeng. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.
- [22] Zhou Xiabin, Wang Wenbin, Zeng Minyan, Guo Jiaxian, Liu Xuebo, Shen Li, Zhang Min, and Ding Liang. Dynamickv: Task-aware adaptive kv cache compression for long context llms. *arXiv preprint arXiv:2412.14838*, 2024.
- [23] Lin Xiaolin, Wang Jingcun, Kondrateva Olga, Shi Yiyu, Li Bing, and Zhang Grace, Li. Compresskv: Semantic retrieval heads know what tokens are not important before generation. *arXiv preprint arXiv:2508.02401v1*, 2025.
- [24] Li Yuhong, Huang Yingbing, Yang Bowen, Venkitesh Bharat, Locatelli Acyr, Ye Hanchen, Cai Tianle, Lewis Patrick, and Chen Deming. Snapkv: Llm knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.
- [25] Fu Qichen, Cho Minsik, Merth Thomas, Mehta Sachin, Rastegari Mohammad, and Najibi Mahyar. Lazyllm: Dynamic token pruning for efficient long context lilm inference. *arXiv preprint arXiv:2407.14057*, 2024.
- [26] Zhang Zhenyu, Sheng Ying, Zhou Tianyi, Chen Tianlong, Zheng Lianmin, Cai Ruisi, Song Zhao, Tian Yuandong, Ré Christopher, Barrett Clark, Wang Zhangyang, and Chen Beidi. H<sub>2</sub>O: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023.
- [27] Wang Zheng, Jin Boxiao, Yu Zhongzhi, and Zhang Minjia. Model tells you where to merge: Adaptive kv cache merging for llms on long-context tasks. *arXiv preprint arXiv:2407.08454*, 2024.
- [28] Liu Xin, Liu Pei, and Tang Guoming. Zsmerge: Zero-shot kv cache compression for memory-efficient long-context llms. *arXiv preprint arXiv:2503.10714*, 2025.
- [29] Hu Jie, Wang Shengnan, He Yutong, Gong Ping, Yi Jiawei, Zhang Juncheng, Bai Youhui, Chen Renhai, Zhang Gong, Li Cheng, and Yuan Kun. Efficient long-context lilm inference via kv cache clustering. *arXiv preprint arXiv:2506.11418*, 2025.
- [30] Yang Yifei, Cao Zouying, Chen Qiguang, Qin Libo, Yang Dongjie, Zhao Hai, and Chen Zhi. Kvsharer: Efficient inference via layer-wise dissimilar kv cache sharing. *arXiv preprint arXiv:2410.18517*, 2024.
- [31] Rajput Shashank, Sheng Ying, Owen Sean, and Chiley Vitaliy. Inference-friendly models with mixattention. *arXiv preprint arXiv:2409.15012*, 2024.
- [32] Wu Haoyi and Tu Kewei. Layer-condensed kv cache for efficient inference of large language models. *arXiv preprint arXiv:2405.10637*, 2024.
- [33] Wu You, Wu Haoyi, and Tu Kewei. A systematic study of cross-layer kv sharing for efficient lilm inference. *arXiv preprint arXiv:2410.14442*, 2024.
- [34] Brandon William, Mishra Mayank, Nrusimha Aniruddha, Panda Rameswar, and Kelly Jonathan, Ragan. Reducing transformer key-value cache size with cross-layer attention. *arXiv preprint arXiv:2405.12981*, 2024.
- [35] Zhang Rongzhi, Wang Kuang, Liu Liyuan, Wang Shuohang, Cheng Hao, Zhang Chao, and Shen Yelong. Lorc: Low-rank compression for llms kv cache with a progressive compression strategy. *arXiv preprint arXiv:2410.03111*, 2024.
- [36] Lin Bokai, Zeng Zihao, Xiao Zipeng, Kou Siqi, Hou Tianqi, Gao Xiaofeng, Zhang Hao, and Deng Zhijie. Matryoshkakv: Adaptive kv compression via trainable orthogonal projection. *arXiv preprint arXiv:2410.14731*, 2024.
- [37] Yu Hao, Yang Zelan, Li Shen, Li Yong, and Wu Jianxin. Effectively compress kv heads for lilm. *arXiv preprint arXiv:2406.07056*, 2024.
- [38] Hooper Coleman, Kim Sehoon, Mohammadzadeh Hiva, Mahoney Michael, W., Shao Yakun, Sophia, Keutzer Kurt, and Gholami Amir. Kvquant: Towards 10 million context length lilm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

- [39] Wang Zongwu, Xu Peng, Liu Fangxin, Hu Yiwei, Sun Qingxiao, Li Gezi, Li Cheng, Wang Xuan, Jiang Li, and Guan Haibing. Million: Mastering long-context llm inference via outlier-immunized kv product quantization. *arXiv preprint arXiv:2504.03661*, 2025.
- [40] Dong Shichen, Cheng Wen, Qin Jiayu, and Wang Wei. Qaq: Quality adaptive quantization for llm kv cache. *arXiv preprint arXiv:2403.04643*, 2024.
- [41] Yao Dingyu, Shen Bowen, Lin Zheng, Liu Wei, Luan Jian, Wang Bin, and Wang Weiping. Tailorkv: A hybrid framework for long-context inference via tailored kv cache optimization. *arXiv preprint arXiv:2505.19586*, 2025.
- [42] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc’ aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.
- [43] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.
- [44] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- [45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.
- [46] Mohammad Shoeybi, Mostafa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [47] Xu Qifan, Li Shenggui, Gong Chaoyu, and You Yang. An efficient 2d method for training super-large deep learning models. *arXiv preprint arXiv:2104.05343*, 2021.
- [48] Bian Zhengda, Xu Qifan, Wang Boxiang, and You Yang. Maximizing parallelism in distributed training for huge neural networks. *arXiv preprint arXiv:2105.14450*, 2021.
- [49] Li Shenggui, Xue Fuzhao, Baranwal Chaitanya, Li Yongbin, and You Yang. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.
- [50] Amr Elmeleegy, Harry Kim, David Zier, Kyle Kranen, Neelay Shah, Ryan Olson, and Omri Kahalon. NVIDIA dynamo, a low-latency distributed inference framework for scaling reasoning ai models. NVIDIA Developer Blog, March 2025. Published March 18, 2025.
- [51] Juntao Zhao, Jiuru Li, and Chuan Wu. Sandwich: Separating prefill-decode compilation for efficient cpu llm serving. *arXiv preprint arXiv:2507.18454*, 2025.
- [52] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanze Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.
- [53] DeepSeek-AI. Deepseek LLM: scaling open-source language models with longtermism. *CoRR*, abs/2401.02954, 2024.
- [54] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.
- [55] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.
- [56] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the AI2 reasoning challenge. *CoRR*, abs/1803.05457, 2018.
- [57] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: reasoning about physical commonsense in natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 7432–7439. AAAI Press, 2020.
- [58] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In Anna Korhonen, David R. Traum, and Lluís Márquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4791–4800. Association for Computational Linguistics, 2019.

- [59] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? A new dataset for open book question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun’ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 2381–2391. Association for Computational Linguistics, 2018.
- [60] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, 2021.
- [61] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.
- [62] Loubna Ben Allal, Anton Lozhkov, Guilherme Penedo, Thomas Wolf, and Leandro von Werra. Smollm-corpus. 2024.
- [63] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.