BOOST: BOTTLENECK-OPTIMIZED SCALABLE TRAINING FRAMEWORK FOR LOW-RANK LARGE LANGUAGE MODELS

Zhengyang Wang [*1] **Ziyue Liu** [*1] **Ruijie Zhang** [1] **Avinash Maurya** [2] **Paul Hovland** [2] **Bogdan Nicolae** [2] **Franck Cappello** [2] **Zheng Zhang** [1]

ABSTRACT

The scale of transformer model pre-training is constrained by the increasing computation and communication cost. Low-rank bottleneck architectures offer a promising solution to significantly reduce the training time and memory footprint with minimum impact on accuracy. Despite algorithmic efficiency, bottleneck architectures scale poorly under standard tensor parallelism. Simply applying 3D parallelism designed for full-rank methods leads to excessive communication and poor GPU utilization. To address this limitation, we propose BOOST, an efficient training framework tailored for large-scale low-rank bottleneck architectures. BOOST introduces a novel Bottleneck-aware Tensor Parallelism, and combines optimizations such as online-RMSNorm, linear layer grouping, and low-rank activation checkpointing to achieve end-to-end training speedup. Evaluations on different low-rank bottleneck architectures demonstrate that BOOST achieves 1.46–1.91× speedup over full-rank model baselines and 1.87–2.27× speedup over low-rank model with naively integrated 3D parallelism, with improved GPU utilization and reduced communication overhead. Code available here.

1 INTRODUCTION

Large Language Models (LLMs) continue to evolve and are increasingly becoming an integral part of a wide range of industrial and scientific applications. More generally, modern transformer-based foundation models that combine multi-modal data, use domain-specific information, and being integrated into complex AI workflows are pushing the limits of scientific discovery. Such models become more useful as they grow larger and ingest more training data (Kaplan et al., 2020; Hoffmann et al., 2022). Unfortunately, this unprecedented scale of growth also poses a major challenge: pre-training such models is exceptionally costly. For example, GPT-3 (175B) was trained on roughly 300B tokens (Brown et al., 2020); BLOOM-176B required 1.08M A100 GPU-hours (Luccioni et al., 2023); Meta's LLaMA-3.1-405B model was trained on 15T tokens with 30.84M H100 GPU hours (Dubey et al., 2024). As the trend continues, it takes hundreds of millions of US dollars to develop such a state-of-the-art foundation model. For example, the cost of developing Grok-4 is estimated to be ∼$500M (Sanders et al., 2025). These realities motivate both academia and industry to actively develop efficient training techniques for foundation models.

[*]Equal contribution [1]University of California, Santa Barbara [2]Argonne National Laboratory. Correspondence to: Zheng Zhang <zhengzhang@ece.ucsb.edu>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026.

Motivation: Driven by the need of scalability, major state-of-the-art advancements have targeted both system-level and algorithmic considerations. From the system-level perspective, a key challenge is how to distribute the computations and data on multiple GPUs in order to parallelize computations and aggregate the GPU memory (and other memory tiers) while reducing coordination costs and communication overheads. In this regard, techniques such as 3D parallelism (a combination of data, tensor/sequence, and pipeline) (Shoeybi et al., 2019; Korthikanti et al., 2023; Huang et al., 2019; Narayanan et al., 2019; Qi et al., 2023) are indispensable in modern training frameworks such as DeepSpeed (Rasley et al., 2020), Nanotron (Tazi et al., 2025) and TorchTitan (Liang et al., 2024). Reduced communication costs and coordination overheads are often achieved by using high-performance communication libraries (e.g., NCCL and MPI (Walker & Dongarra, 1996; Gabriel et al., 2004)) and asynchronous I/O techniques that hide the cost of processing messages and GPU-host memory movements in the background. From the algorithmic perspective, many efforts have sought to improve model efficiency through different directions, including improved attention mechanisms (Wang et al., 2020; Shazeer, 2019; Ainslie et al., 2023; Yuan et al., 2025), quantization-based methods that preserve accuracy under reduced precision (Micikevicius et al., 2022; Peng et al., 2023; Chmiel et al., 2025; Abecassis et al., 2025; Xi et al., 2023), and sparsity-based approaches such as Mixture-of-Experts (MoE) (Jacobs et al., 1991; Jordan & Jacobs, 1994).

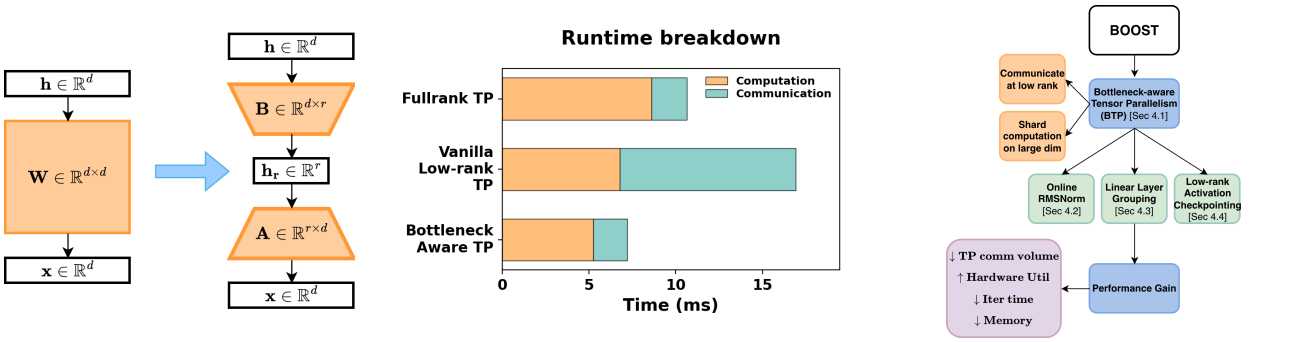

Figure 1. (Left) Linear layer in Bottleneck Architecture; (Middle) Decoder block runtime breakdown among different TP strategy; (Right) Overview of our framework BOOST.

Among these algorithmic approaches, low-rank methods that either project gradients/activations to low-dimensional space (Zhao et al., 2024; Zhu et al., 2024; Shamshoum et al., 2024; Chen et al., 2025) or have partial/entire weights factorized in low-rank matrix/tensor format (Lialin et al., 2023; Loeschcke et al., 2024; Han et al., 2024; Yang et al., 2024; Liu et al., 2025b; Zhanfeng Mo, 2025; Zhang et al., 2025; Kong et al., 2025; Li et al., 2025) are a particular class of algorithmic methods specifically targeted at balancing accuracy with performance and resource utilization. Thanks to a unified bottleneck architecture (Figure 1 left), examples such as CoLA (Liu et al., 2025b), LORO (Zhanfeng Mo, 2025) and LaX (Zhang et al., 2025) can simultaneously reduce parameter count, memory, compute and runtime compared with a full-rank architecture while preserving comparable accuracy. Unfortunately, a large part of these efforts have only been explored at small scale (e.g., below 7B parameters), in which case both the full-rank and the corresponding low-rank model can fit in the memory of a single GPU. Without a co-design with system-level approaches, the scalability potential of low-rank methods remains limited. Specifically, one difficult challenge is how to enable efficient tensor parallelism for low-rank methods, which, unlike data and pipeline parallelism, features a tightly coupled pattern with all-to-all dependencies. In this paper, we aim to address this challenge.

Limitations of state of the art: We cannot simply apply vanilla tensor parallelism techniques that were originally designed for full-rank methods to low-rank methods. As can be observed in Figure 1 (middle), a full-rank training iteration on four GPUs co-located on the same compute node (details in § 5.1) has less than 20% communication overhead, while an equivalent low-rank method experiences an explosion in the communication overhead. This is because the bottleneck architecture is based on smaller but deeper structures that feature more synchronization points, which increase the communication latency and volume. Additionally, the GPU utilization is also sub-optimal due to inefficient shapes of the weight matrices as partitioned by vanilla tensor parallelism onto individual devices, leading to insufficient low-rank computational speed-up that erodes algorithmic efficiency. Together, these limitations restrict the theoretical algorithmic scalability potential.

Contributions: To bridge this gap, we design and implement scalable tensor-parallel training strategy for efficient pre-training of low-rank (bottleneck) foundation models. We leverage the idea that synchronization points can either be avoided or placed strategically where the bottleneck architecture is narrow to reduce communication overheads. And we boost computation by sharding along the large dimensions to preserve healthy GEMM reduction sizes thus improving GPU utilization. We summarize our key contributions as follows:

- We present a theoretical analysis of bottleneck architectures in 3D parallelism, quantifying arithmetic intensity and communication volume in distributed training to reveal the scaling challenges of vanilla designs and the benefits of our approach.
- We propose a novel tensor parallelism strategy, namely, *Bottleneck-aware Tensor Parallelism* (BTP), which optimizes partitioning of low-rank weight matrices to facilitate efficient communication on low-rank activations. BTP also significantly increases the computational intensity of GEMM kernels, thus improving GPU utilization.
- We implement BOOST, a high-performance distributed training framework to take advantage of BTP in several ways: *Online-RMSNorm* to facilitate sharded-safe global normalization and reduce latency, *Layer Grouping* to improve computational intensity and lower the number of collective operations, and *Low-rank Activation Checkpointing* to reduce re-computation cost and eliminate additional collectives.
- We demonstrate a 1.46–1.91× speedup over full-rank baselines and a 1.87–2.27× speedup over vanilla tensor parallel implementations of bottleneck architectures. The ablation study further confirms efficiency gains on both the computation and communication axes.

2 Related Work

Low-rank methods have been extensively explored in the literature from different perspectives, the most relevant of which we discuss below.

Low-Rank Matrix Factorization. Classical low-rank training schemes (Khodak et al., 2021; Kamalakara et al., 2022) replace a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ with $\widehat{\mathbf{W}} \approx \mathbf{BA}$, where $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$, $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$, $r \ll \min(d_{\text{out}}, d_{\text{in}})$ determines the rank and thus the compression ratio. However, naively applying this low-rank matrix factorization often harms model accuracy in pre-training setting. Recent work proposed various modifications to improve model capacity. For example, ReLoRA (Lialin et al., 2023) adopted the LoRA (Hu et al., 2022) adapter with constantly merged and reinitialized low-rank factors. SLTrain (Han et al., 2024) combined an unstructured sparse matrix with low-rank factors to approximate $\mathbf{W}$. LORO (Zhanfeng Mo, 2025) preserved the classical low-rank formulation but perform some optimization steps directly on the Riemannian manifold.

Low-Rank Activation via Auto-Encoder. Observing that the activation function of every layer stays in a low-rank space, CoLA (Liu et al., 2025b) proposed to replace both an MLP layer $\sigma(\mathbf{W}\mathbf{x})$ and a linear projection layer $\mathbf{W}\mathbf{x}$ by an auto-encoder layer $\mathbf{B}\sigma(\mathbf{A}\mathbf{x})$. This method has achieved simultaneous reduction of perplexity, model size, memory cost and runtime in pre-training BERT and LLaMA. Recently LOST (Li et al., 2025) further modified CoLA by incorporating an unstructured sparse matrix $\mathbf{S}$, similar to the ones in SLTrain, as $\alpha\mathbf{B}\sigma(\mathbf{A}\mathbf{x}) + (1-\alpha)\mathbf{S}\mathbf{x}$.

Low-Rank Models Enhanced by Residual Connection. Another promising direction is to boost the accuracy of existing low-rank architecture with some plug-in residual blocks. LaX (Zhang et al., 2025) proposed to fuse cross-layer low-rank features to boost model capacity without increasing their physical rank, i.e., $\mathbf{B}_i(\mathbf{A}_i\mathbf{x}_i + \mathbf{A}_{i-1}\mathbf{x}_{i-1})$. This method can improve the accuracy of low-rank matrix/tensor-compressed models as well as CoLA with negligible parameter and computing overhead. CR-Net (Kong et al., 2025) proposed a similar approach from a different perspective: the difference of activations between adjacent layers exhibit low-rank property, thus $\mathbf{B}_i\mathbf{A}_i\mathbf{x}_i + \mathbf{B}_{i-1}\mathbf{A}_{i-1}\mathbf{x}_{i-1}$, with the exception of the first layer being full-rank.

Additionally, pre-training LLMs also requires sophisticated large-scale system-level optimizations. Two complementary paradigms dominate large-scale training:

Redundancy Elimination using Model/Optimizer State Sharding. Methods such as ZeRO (Rajbhandari et al., 2020), ZeRO-Offload (Ren et al., 2021), ZeRO-Infinity (Rajbhandari et al., 2021) and PyTorch FSDP (Zhao et al., 2023) partition optimizer states, gradients, and parameters across devices to reduce memory footprint and enable training larger models. Representative implementations include DeepSpeed ZeRO (Rasley et al., 2020) and FairScale's sharded optimizers (FairScale authors, 2021). Such techniques are complementary to our own work.

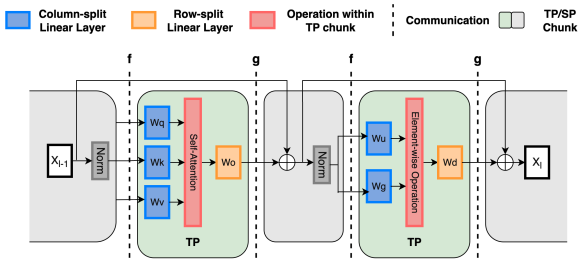

Figure 2. Megatron-LM style Tensor Parallelism. Here, f is identity in forward and all-reduce in backward, while g is all-reduce in forward and identity in backward.

3D Parallelism. Frameworks such as Megatron-LM and Megatron-DeepSpeed combine data (DP), tensor (TP), and pipeline (PP) parallelism to distribute memory and computation across devices for billion-parameter pre-training (Shoeybi et al., 2019; Narayanan et al., 2021). DP accelerates training by replicating the model across GPUs, processing different mini-batches in parallel, and synchronizing by *gradient all-reduce* once per iteration before the optimizer step. PP splits layers vertically into stages so micro-batches stream through as an assembly line, communicates activations/gradients at stage boundaries. Efficiency hinges on schedules such as GPipe/PipeDream (Huang et al., 2019; Narayanan et al., 2019) and on minimizing warm-up/drain bubbles. TP partitions the model horizontally, sharding individual weight matrices and corresponding activations across GPUs. Each device computes partial results that are later combined via tightly-coupled collective communication. TP builds on matrix/activation sharding (e.g., Mesh-TensorFlow, GShard, GSPMD) (Shazeer et al., 2018; Lepikhin et al., 2020; Xu et al., 2021). Megatron-LM adopts a standard *column–row* pattern: each TP *chunk* (two linears plus the intervening op such as SDPA/SwiGLU) issues one collective per pass. Here, f is identity in the forward pass and all-reduce in the backward pass, while g is all-reduce in the forward pass and identity in the backward pass (Figure 2) (Shoeybi et al., 2019).

Tensor Parallelism in Hybrid MoE Systems. Mixture-of-Experts (MoE) has become a major trend in modern LLM scaling. Although expert parallelism (EP) is a common scaling strategy in MoE systems, it introduces heavy token-level all-to-all communication and can suffer from severe load imbalance. By contrast, tensor parallelism (TP) relies on more regular collective communication, often within a node. Moreover, when experts become sufficiently large, EP may complement rather than replace TP, and hybrid EP+TP configurations can still be necessary (Singh et al., 2023; Jin

et al., 2025; Liu et al., 2025a). As a result, TP remains important in modern large-model training, including hybrid dense–MoE systems: large dense/shared components still rely on TP, and sufficiently large experts may also require TP in addition to EP.

Tensor Parallelism for Low-Rank Adapters. Prior work on low-rank methods has focused more on fine-tuning than on pre-training. S-LoRA (Sheng et al., 2024) proposes a tensor-parallel strategy for the low-rank adapter setting, where LoRA adapters are attached to a frozen dense backbone. While the adapter branch also introduces deeper layers with narrow intermediate activations, its output must be merged back into the frozen full-rank branch, so the communication and sharding design remains naturally coupled to the dense backbone. In contrast, our work focuses on pre-training pure low-rank or bottleneck backbone architectures without a frozen full-rank branch.

To the best of our knowledge, we are the first to tackle the problem of optimizing tensor parallelism for low-rank methods in pre-training. In this work, we focus on **dense** low-rank bottleneck architectures, which are the main setting considered by current low-rank **pre-training** methods, and design a general framework for unified bottleneck architectures such as CoLA, LORO, and LaX. In practice, this class of methods typically offers the largest training speedups and memory savings with minimal impact on accuracy. Furthermore, although outside the scope of this work, our framework is complementary to pipeline and data parallelism, as well as other techniques such as redundancy elimination; we discuss broader applicability to architectures such as MoE in Sec. 6.

3 PRELIMINARIES

As shown in Figure 1 (Middle), naively scaling a bottleneck architecture exhibits both *communication* and *computation* inefficiencies. We quantify system efficiency with two complementary metrics:

- **Communication Volume** V_{comm}: bytes moved by collectives, lower V_{comm} indicates reduced synchronization overhead and better scalability across devices;
- **Arithmetic Intensity (A.I.)**: FLOPs per byte moved by kernels, a proxy for how close execution is to compute-bound. A higher A.I. indicates a compute-bound operation capable of saturating GPU compute units, while a lower A.I. denotes a memory-bound kernel limited by memory bandwidth. For a matrix multiplication $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ with $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$, and $\mathbf{C} \in \mathbb{R}^{M\times N}$, the total number of FLOPs is $2MNK$. Assuming ideal memory reuse, the total memory traffic consists of reading $\mathbf{A}$ and $\mathbf{B}$ and writing $\mathbf{C}$, yielding:

$$\text{A.I.} = \frac{2MNK}{(MK + KN + MN) \cdot \text{bytes\_per\_element}}. \quad (1)$$

Table 1. Per-iteration communication volume (V_{comm}) under different parallelism strategies for full-rank and bottleneck architecture in LLaMA-like models.

STRATEGY	FULL-RANK	LOW-RANK	
		VANILLA	BOOST
DATA PARALLELISM	$O(d(d+d_{ff}))$	$O(r(d+d_{ff}))$	
PIPELINE PARALLELISM	$O(d)$	$O(d)$	
TENSOR PARALLELISM	$O(d)$	$O(d+d_{ff})$	$O(r)$

DP and PP in Bottleneck Architectures. Table 1 summarizes the communication volume across different parallelism strategy. Low-rank bottleneck architectures *intrinsically* benefit DP and PP due to smaller parameter size and low-rank activations. Because factorized weights yield smaller gradients, bottleneck models reduce gradient all-reduce traffic. In LLaMA with $r = \frac{d}{4}$, this corresponds to an $\sim 2.5\times$ reduction in communication volume. For pipeline parallelism, factorized layers lower per-stage FLOPs, which shortens per micro-batch compute and shrinks the pipeline bubble; the smaller footprint also permits larger micro-batches, improving arithmetic intensity and hardware utilization.

TP in Bottleneck Architectures. Contrary to being mostly model-agnostic in DP and PP, TP interacts directly with model structure: it shards activations and parameters in architecture-aware ways to balance compute intensity and communication. In the following section we will analyze why a naive low-rank TP degrades efficiency and motivates our bottleneck-aware TP design.

4 BOOST: BOTTLENECK-OPTIMIZED SCALABLE TRAINING

In this section, we introduce our BOOST framework and present the details of how it addresses scalability issues of low-rank (bottleneck) LLM training under 3D parallelism.

4.1 Bottleneck-aware Tensor Parallelism

Tensor parallelism is non-trivial for bottleneck layers. Replacing a single $d\times d$ projection with consecutive $d\times r$ and $r\times d$ projections ($r \ll d$) *doubles* linear stages per block and raises two design questions: (i) *Where to place TP chunks in deeper block?* Since each TP chunk incurs one collective, adding stages increases the number of collectives per block. (ii) *How to shard efficiently given the low-rank structure?* Conventional TP reduces per-rank hidden width, but the bottleneck factorization already shrinks effective dimension, inefficient sharding can therefore harm hardware utilization

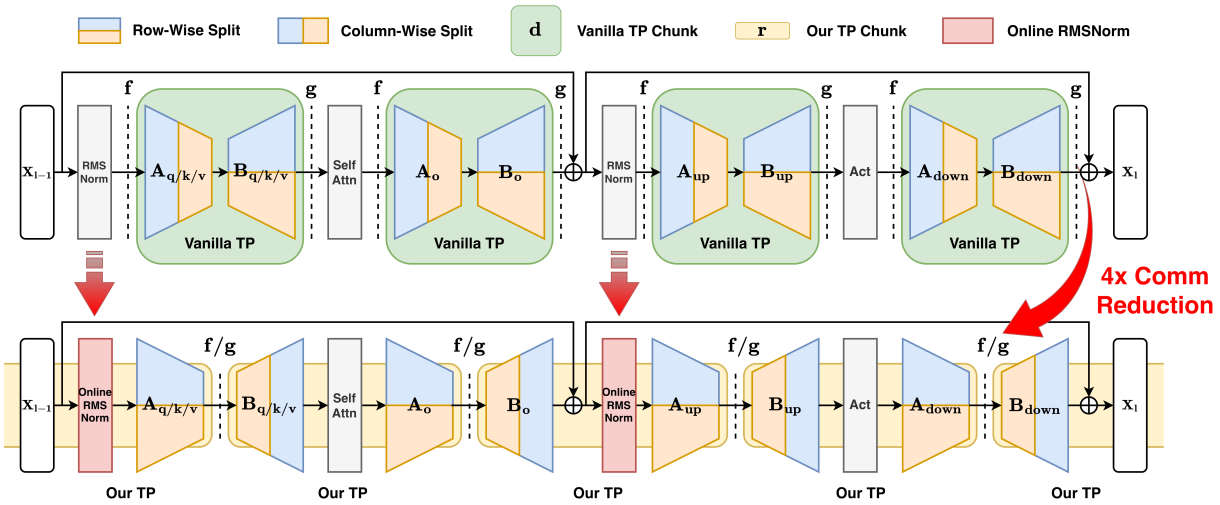

Figure 3. Modularized Tensor Parallelism Design for Bottleneck Architecture. Top: vanilla TP uses separate f and g communication boundaries for each TP chunk. Bottom: after shifting the TP chunk boundary to the low-rank bottleneck, the same low-dimensional boundary serves as a shared f/g point.

by having a lower arithmetic intensity.

Vanilla low-rank Tensor Parallelism. A straightforward baseline follows the Megatron-LM pattern: treat each pair of low-rank layers ($d\times r$ then $r\times d$) as a TP chunk: column-parallel for the down-projection, row-parallel for the up-projection. We refer to this as *vanilla TP* (top of Figure 3). However, this vanilla design introduces both communication and computation inefficiencies.

On the *communication* side, a full-rank Megatron style decoder block issues two activation-sized collectives of payload $[b, s, d]$, i.e., $V_{\text{comm}}^{\text{full}} = 2bsd$ (one in attention, one in MLP). In contrast, vanilla TP triggers a collective at *each* linear: $4\,bsd$ in the attention block and $bsd + 2\,bs\,d_{ff}$ in the MLP, for a total of

$$V_{\text{comm}}^{\text{vanilla}} = (5\,bsd + 2\,bs\,d_{ff}) = \left(\tfrac{5+2(d_{ff}/d)}{2}\right) \times V_{\text{comm}}^{\text{full}}. \tag{2}$$

Thus the per-block volume grows by $\mathbf{5}\times$ when $d_{ff}{\approx}2.5d$, and up to $\mathbf{6.5}\times$ when $d_{ff}{\approx}4d$. This explains the large communication time observed in the breakdowns for naively scaled bottleneck models.

On the computation side, bottleneck architecture already reduces the effective hidden dimension, and vanilla TP further shards along the low rank r, shrinking the GEMM reduction dimension. This results in kernels that move a lot of data but perform little compute, lowering arithmetic intensity and pushing execution into the memory-bound regime. Although vanilla low-rank TP reduces FLOPs, its A.I. remains very low due to disproportionate data movement; for example, in LLaMA-7B MLP blocks it attains only $0.2\times$ the A.I. of full-rank TP, leading to poor GPU utilization. This also explains why at scale the reduction in computation time is smaller than in the single-GPU setting.

Additionally, the up-projection materializes a full, unsharded activation even though each TP rank consumes only its local shard, leading to redundant allocation and unnecessary data movement.

Bottleneck-aware Tensor Parallelism (BTP). To address the inefficiencies of vanilla TP, we propose a bottleneck-aware sharding strategy that leverages the low-rank r-dimensional output of each pair of bottleneck layers to issue lightweight collectives. Concretely, this sharding strategy shifts the TP chunk boundary by exactly one individual bottleneck layer. The bottleneck-aware TP chunk starts with the *up-projection* layer ($r \times d$) being *column*-parallel and the *next down-projection* layer ($d \times r$) being *row*-parallel, while the intervened operations being executed on sharded activations. To shard along the large dimension and communicate at the low dimension, **BTP improves both the communication efficiency and computation efficiency.**

Effectively, BTP improves communication efficiency by reducing its workload. In BTP chunk, the collective operates on a *low-rank* activation of size $[b, s, r]$ rather than $[b, s, d]$. Per decoder block, this yields a total payload of

$$V_{\text{comm}}^{\text{BTP}} = 7\,bsr = \left(\tfrac{7r}{2d}\right) \times V_{\text{comm}}^{\text{full}}. \tag{3}$$

Thus BTP reduces communication volume by more than $5.7\times$ versus vanilla low-rank TP (when $r = \frac{d}{4}$) and by $1.14\times$ versus the full-rank TP baseline, substantially lowering communication time in practice.

From the computation side, BTP shards along the hidden dimension rather than the low-rank dimension r, producing kernels with a larger GEMM reduction dimension and thus

higher arithmetic intensity. Although BTP and vanilla TP perform the same FLOPs, BTP moves less data, yielding higher arithmetic intensity. For example, in LLaMA-7B MLP blocks, BTP achieves $2.5\times$ the A.I. of vanilla TP, translating into substantially better hardware utilization.

4.2 Online RMSNorm

Sharded-safe vs. sharded-unsafe operators. In tensor parallelism, we assume no extra collectives inside a TP chunk: a single all-reduce is issued at the end of each chunk. Under this assumption, any operation inside the chunk that can be correctly performed on sharded activations is **sharded-safe**, e.g., element-wise functions (activations, dropout) and attention heads when the head count is divisible by the TP degree. Conversely, ops that require cross-shard data are **sharded-unsafe**; a canonical example is normalization, which depends on global mean and variance computations and therefore cannot be placed inside a TP chunk without additional synchronization.

A simple design principle is to avoid sharded-unsafe operations inside the chunk. However, with BTP the RMSNorm [Eq. (4)] naturally falls in the middle of a TP chunk, so we must handle it explicitly. We therefore introduce and compare two variants: Sync RMSNorm (explicit in-place statistic synchronization) and Online RMSNorm (fused statistic exchange with recovery).

$$\text{RMSNorm}(\mathbf{X}) = \frac{\mathbf{X}\cdot\boldsymbol{\gamma}}{\text{RMS}(\mathbf{X})}, \quad \text{with} \quad \text{RMS}(\mathbf{X}) = \sqrt{\frac{1}{d}\sum_{j=1}^{d}\mathbf{X}_{:,j}^2 + \epsilon}. \tag{4}$$

Sync RMSNorm. A straightforward approach is to compute per-rank local statistics $\text{RMS}(\mathbf{X}_i)$ and use a collective to aggregate the global statistic before normalizing. Although the payload is tiny (the statistic exchange at $[b, s, 1]$), many such calls incur nontrivial launch overhead, are latency-dominated and achieve poor effective bandwidth, resulting in extra communication time.

Online RMSNorm. To eliminate the standalone small-payload collective, we compute normalization using local statistics and fuse the statistic exchange into the TP collective that follows the next GEMM. This mirrors the insight of online-softmax in FlashAttention (Dao et al., 2022): defer global normalization and recover the exact result afterward. In effect, the previously sharded-unsafe RMSNorm becomes a sharded-safe online RMSNorm.

The online-RMSNorm operator consists of three main steps (Alg. 1): **Step 1: Local norm computation.** Compute and record the local sum of squares on sharded activations (Line 1) and calculate local RMSNorm (Line 2 and 3). **Step 2: Row-split GEMM with fused all-reduce.** Apply the following row-parallel GEMM (Line 4), rescale the rank-dependent correction factor (Line 5), then all-reduce together the GEMM output and the local statistic (Line 6). **Step 3: Recovery.** Calculate correct global rms using global sum of squares (Line 7). And rescale the reduced GEMM output back by the global rms(Line 8).

Algorithm 1 Online RMSNorm

Input: $d_{\text{local}} = d/\text{TP\_size}$; sharded activation $X_i \in \mathbb{R}^{s\times b\times d_{\text{local}}}$ on TP rank i; row-split weight $W_i \in \mathbb{R}^{d_{\text{local}}\times r}$; scaling factor γ_i; numerical constant ϵ
Output: Final output $Y \in \mathbb{R}^{s\times b\times r}$

1: $S_{\text{local}} \leftarrow \sum_{j=1}^{d_{\text{local}}} X_i[:,:,j]^2$
2: $rms_{\text{local}} \leftarrow \sqrt{\frac{1}{d_{\text{local}}} S_{\text{local}} + \epsilon}$
3: $X_i \leftarrow \frac{X_i}{rms_{\text{local}}} \times \gamma_i$
4: $H_i \leftarrow X_i \cdot W_i$
5: $H_i \leftarrow H_i \cdot rms_{\text{local}}$
6: $[H_{\text{global}}, S_{global}] \leftarrow \text{AllReduce}([H_i, S_{local}])$
7: $rms_{\text{global}} \leftarrow \sqrt{\frac{1}{d} S_{\text{global}} + \epsilon}$
8: $Y \leftarrow H_{\text{global}} \cdot \frac{1}{rms_{\text{global}}}$
9: **return** Y

Online RMSNorm preserves **mathematical equivalence**: although the GEMM is performed using locally normalized inputs, the output can be rescaled by a per-row correction factor derived from the ratio of local to global statistics, making the result equivalent to standard RMSNorm. Specifically, we can show in Equation (5):

$$\sum_i \frac{\mathbf{W_i X_i}}{\text{RMS}(\mathbf{X})} = \Big(\sum_i \frac{\mathbf{W_i X_i}}{\text{RMS}(\mathbf{X_i})} \cdot \text{RMS}(\mathbf{X_i})\Big) \cdot \frac{1}{\text{RMS}(\mathbf{X})} \tag{5}$$

Moreover, Online RMSNorm helps maintain **numerical stability** in practice. By computing intermediate normalization statistics locally on sharded activations, it reduces the likelihood of extreme values that could otherwise lead to overflow or underflow. Online RMSNorm can closely match the $TP = 1$ baseline at both the kernel and training levels. As shown in Table 2, for $TP = 4$, Online RMSNorm followed by row-split linear yields only small numerical discrepancies from the $TP = 1$ baseline in both fp32 and bf16, within acceptable precision tolerances. Fig. 4 further shows that BTP with Online RMSNorm preserves the baseline training behavior.

From a **performance standpoint**, local normalization reduces per-rank work via smaller dimension, and fusing the statistic exchange with the GEMM all-reduce removes a separate kernel launch and increases effective bandwidth by aggregating it into a larger message. The recovery is a cheap per-row scaling. Empirically, the recovery compute overhead is small relative to the latency-domiated statistic

Table 2. Kernel-level comparison between Online RMSNorm + row-split linear (TP=4) and the TP=1 baseline RMSNorm + linear output. We report the average maximum and mean absolute differences.

Precision	Avg. Max Abs. Diff.	Avg. Mean Abs. Diff.
FP32	7×10^{-7}	6×10^{-8}
BF16	3.125×10^{-2}	2.2×10^{-3}

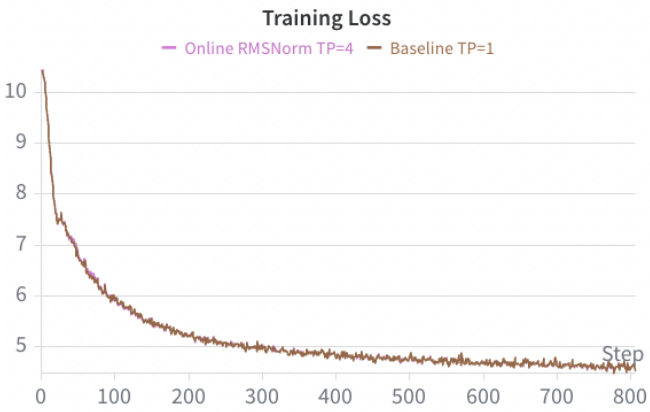

Figure 4. Training loss curves for a tiny LLaMA model, comparing BTP with Online RMSNorm against the TP=1 baseline. The two curves closely match, indicating that the corrected Online RMSNorm preserves stable training behavior.

collective in Sync RMSNorm, making the computation-communication trade-off favorable.

In practice, these two variants offer different trade-offs. **Sync RMSNorm** is the exact and conservative option: it explicitly synchronizes the scalar normalization statistic before applying RMSNorm, and therefore serves as a simple always-available fallback. **Online RMSNorm** uses a reordered computation to defer synchronization and recover the same global normalization factor after the following TP collective, making it the faster option when minimizing standalone small-payload communication is important.

4.3 Linear Layer Grouping

Although BTP outperforms vanilla TP, its extra sync point and small GEMMs still degrade efficiency. To translate BTP's analytical gains into consistent per-layer speedups, we apply linear-layer grouping optimization.

In a full-rank model, we group parallel linears into a single fused operation (e.g., QKV in attention; gate+up in MLP) to reduce kernel launches and enlarge GEMMs. However, grouping is more challenging for bottleneck layers because branch inputs differ: while down-projections share $\mathbf{X}$ and can be grouped by concatenating weights, up-projections use distinct inputs $(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3)$, so we use a batched GEMM over (input, weight) pairs in one fused kernel.

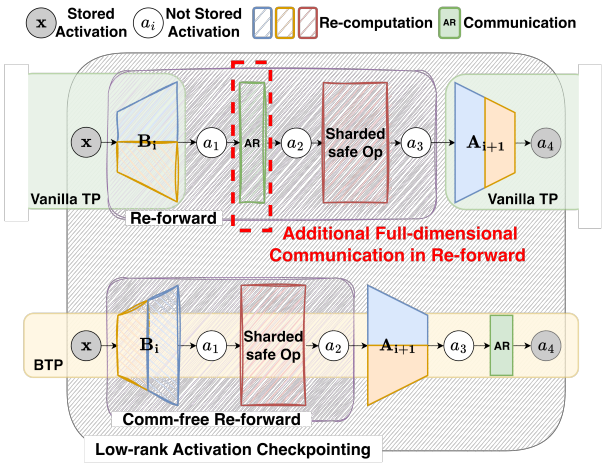

Figure 5. Low-rank efficient activation checkpointing under BTP

Under BTP, grouping delivers even larger benefits. On the compute path, it cuts kernel count and eliminates redundant reads of X, reducing total data movement, raising arithmetic intensity (A.I.) and hardware utilization. On the communication path, it lowers the number of collectives and improves effective bandwidth by increasing per-call payload, together producing a substantial boost in end-to-end efficiency.

4.4 Comm-free Low-rank Activation Checkpointing

As the model scales up, we use PP to span across nodes. However, standard 1F1B scheduling requires multiple microbatches in flight to fill pipeline bubbles, which inflates activation memory. Activation checkpointing is a common remedy, where intermediate results are partially saved in forward, and re-computed when needed in backward. When integrating with the bottleneck architecture, its low-rank activations provide a better choice for choosing the checkpointing interval that favors the tradeoff between memory and compute, as exemplified by CoLA-M (Liu et al., 2025b).

However, vanilla-TP would weaken such benefit by introducing additional communications. From Figure 5, we see that its re-forward path contains the collective operation that lives between two consecutive TP chunks. Effectively, this triggers an extra synchronization point during re-forward.

In contrast, our BTP amplifies the benefit of low-rank checkpointing by aligning its boundaries with the checkpoint interval: the re-forward stays *entirely within-chunk* and requires no additional communication. As shown in Figure 5, within each BTP chunk (yellow) we checkpoint only low-rank activations (x and a_4). In backward, we locally re-forward $x \to B_i \to a_1 \to$ sharded-safe $\to a_2$ to reconstruct the input needed for the gradient computation of the subsequent up-projection $\mathbf{A}_{i+1}$, resulting in a completely *communication-free* re-forward.

4.5 Implementation

Software Stack. We integrate BOOST into Nanotron (Tazi et al., 2025) pre-training framework as composable engines. Nanotron provides column/row-parallel linear APIs, a 3D (DP/TP/PP) execution engine, NCCL-based collectives that run on a dedicated communication stream, with built-in activation checkpointing. In our setup we operate in all-reduce mode (no sequence-parallel chunks), and Nanotron currently do not support fine-grained overlapped all-reduces.

Model Implementation. For full-rank baseline we use Nanotron's reference model. Low-rank variants factorize $d\times d \to d\times r,\ r\times d$ and select matching col/row-parallel linears. For BTP, we shard the embedding output to row-split the first down-projection layer; the final up-projection layer is replicated on every TP rank. Online RMSNorm computes on sharded activations, returns local RMS, and piggybacks local stats in the next TP collective via *all_reduce_coalesced*. Grouping adds (i) concatenated down-projections (shared X) and (ii) batched-GEMM up-projections (distinct inputs) to cut launches and traffic. Checkpointing stores only low-rank boundaries via Nanotron's decorator.

Configuration. Building on Nanotron's example configs, we expose minimal knobs to toggle BOOST components. When preconditions fail, we fall back to unfused operators or Sync RMSNorm; unsupported cases are auto-disabled safely.

Profiling. We use Nsight Systems for kernel time and Nanotron logs for step time, enabling end-to-end and micro-level attribution of speedups.

5 EVALUATION

5.1 Experimental Setup

Testbed. Our evaluations are conducted on the NERSC-Perlmutter supercomputer. Each node contains 192 CPU cores, 256 GB DDR4 memory, and 4×A100 GPUs, each containing 80GB HBM2 memory (320GB per node), with intra-node connection NVLink Gen3 and inter-node Slingshot 11. Following practice in BLOOM (Workshop et al., 2022), OPT (Zhang et al., 2022), and LLaMA (Dubey et al., 2024), we use tensor-parallelism for scaling within a node, and pipeline-parallelism for scaling across nodes. Because our optimizations mainly target TP and apply per pipeline stage, we report results on up to 4 nodes (16 GPUs) to highlight per-node performance and resource utilization use.

5.2 Methodology

Compared Approaches. Throughout our experiments, we compare three different tensor-parallellism strategies. *FullRank-TP* is representative of real-world TP deployments that do not perform low-rank compression, due to which they incur higher memory and runtime overheads. *Vanilla-TP* is a direct adaptation of the tensor-parallelism strategy proposed in Megatron-LM (Shoeybi et al., 2019) for low-rank architectures such as SVD, CoLA (Liu et al., 2025b), and LAX (Zhang et al., 2025), which show reduced memory consumption and faster runtimes compared to full-rank in single GPU case. Lastly, we compare our proposed approach– BOOST, which incorporates the design principles discussed in § 4 to accelerate the training of low-rank bottleneck LLM architectures using tensor-parallelism.

Models, Dataset, and Runtime Configuration. We evaluate 6 different models of sizes 1B, 3B, 7B, 13B, 30B and 40B parameters, all from the LLaMA-2 family under pure bf16 training. For system performance checks we use WikiText dataset with sequence length 4096, matching real-world settings (Lang et al., 2024). Unless stated, we use a batch size of 4, tensor-parallelism of degree 4 (equal to the number of GPUs per node), pipeline and data-parallelism of 1, and disable activation checkpointing for faster iterations. As a representative of bottleneck architecture LLM, we primarily use CoLA (Liu et al., 2025b) as its algorithmic effectiveness has been most thoroughly validated across different bottleneck architectures. Nonetheless, we demonstrate the generalizability of our proposed design principles to other low-rank approaches such as SVD (Zhanfeng Mo, 2025; Wang et al., 2025), and LAX (Zhang et al., 2025). For each experiment, we run 10 training iterations, of which the first 2 are considered as warm-up, and we report an average of the remainder 8 iterations.

Key Performance Metrics. Throughout our evaluations, we study the compared approaches using different metrics. First, we consider the *Average Iteration Time*, which captures the holistic performance impacts, and is representative of the end-to-end time taken throughout the training process. Next, we measure the *Kernel Execution Time* to break down the computational time taken by different GEMM operations, which also yields the corresponding FLOPS for understanding the kernel efficiency. Third, we study the *Hardware Utilization*, measured as a percentage of the peak computational capability of the GPU, to understand resource usage by different approaches. Fourth, to study TP communication overheads, we measure the *communication volume and time* of data transfers. Finally, we measure the *memory saving and runtime overheads* when enabling activation checkpointing.

5.3 Overall Training Performance

Scaling Model Sizes. In our first set of experiments, we evaluate the impact of weak scaling model sizes from 1B to 30B. For smaller models– 1B, 3B, 7B, that fit in a single node, we keep increasing the tensor-parallelism degree, and beyond the 7B model, we use pipeline parallelism of degree

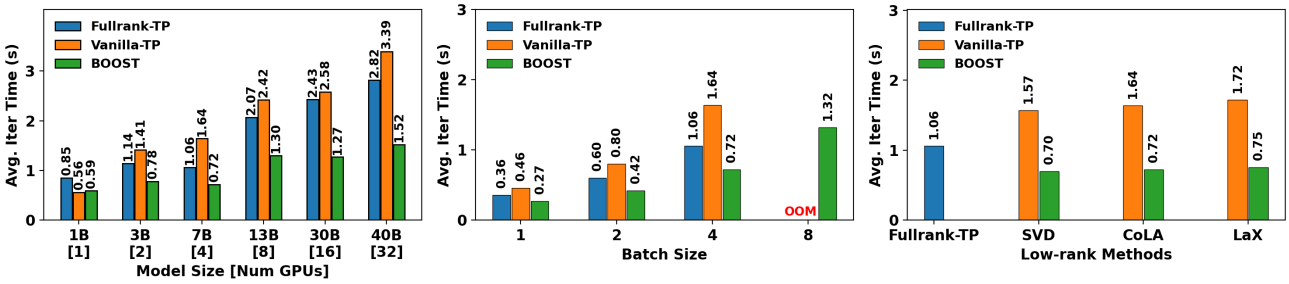

Figure 6. System-wide scalability and generality. (Left) Average iteration time on scaling model sizes with #GPUs; (Middle) Average iteration time on scaling micro-batch size; (Right) Average iteration time on different low-rank architecture.

2, 4 and 8 for the 13B, 30B and 40B models, respectively. We use the iteration time to study the scalability behavior for different model sizes.

As shown in Figure 6(left), the 1B model (with no TP) is $1.4\times$ faster with low rank, confirming baseline efficiency of bottleneck architectures for LLMs. On scaling to 3B model with 2 GPUs or higher, *FullRank-TP* outperforms *Vanilla-TP* by $\sim$25% because vanilla's extra collectives and low-A.I. kernels offsets FLOP savings (§4.1). In contrast, BOOST achieves up to $1.91\times$ over *FullRank-TP* and $2.28\times$ over *Vanilla-TP*. Even with cross-node PP at 13B/30B/40B, BOOST 's TP optimizations significantly reduce iteration time, demonstrating end-to-end scalability.

Scaling Batch Sizes. In the next set of experiments, we study the impact of increasing batch sizes on different approaches. We consider the 7B model running with a tensor-parallelism degree of 4, and scale the micro-batch size from $1\cdots 8$, until every approach runs out of memory (OOM). As depicted in Figure 6(middle), with increasing batch sizes of 1, 2, and 4, we see BOOST outperforms the *FullRank-TP* by $1.3\times$, $1.42\times$, and $1.48\times$, respectively, showing a linear acceleration in performance due to better hardware utilization of larger batches. Similar to scaling of models, *Vanilla-TP* shows the worst performance across the compared approaches because it collects full hidden states and holds redundant activations (§ 4.1), while BOOST avoids these memory overheads, preserving the memory efficiency of low-rank bottleneck architectures for tensor-parallel setups, resulting in accommodating larger batch sizes. Finally, we observe that BOOST's reduced memory footprint allows us to train larger batches while showing a superlinear scalability trend.

Extending Across Different Bottleneck Architectures. Next, we demonstrate the generality of our approach across different LLM bottleneck architectures- SVD, CoLA, and LaX. We use the 7B model with 4 GPUs and compare the average iteration time for all approaches. Given the diversity of factorization methods used for producing low-rank architectures, through this experiment, we study whether the different underlying bottleneck structure of transformer blocks impact training performance when scaling across tensor-parallel ranks. As seen in Figure 6(right), BOOST outperforms *Vanilla-TP* and *FullRank-TP* approaches by $1.5\times$ and $2.2\times$, respectively, for all low-rank approaches. The slight variations in the iteration times are a result of differences in their respective low-rank architectures– SVD runs fastest because of no intervening operations, CoLA is slower due to a nonlinearity between the low-rank linear layers, and LaX is slowest owing to an added residual path. These results underpin the versatility of the design formulations proposed in *FullRank-TP* to optimize the communication and computations across different bottleneck architectures, irrespective of how the layers were factorized.

5.4 Computation Efficiency

In the next series of experiments, we study the computation efficiency introduced by BOOST and quantify its impact on GEMM time and hardware utilization. We instrument each linear layer (attention QKV/out and MLP gate/up/down) in LLaMA-7B/13B and compare *FullRank-TP*, *Vanilla-TP*, and our approach BOOST. For each linear layer, we report (i) theoretical work (FLOPs) alongside measured GEMM time, and (ii) the resulting hardware utilization. This analysis explains why our design accelerates computation and verifies alignment with the predictions in § 4.1.

Figure 7 (left) shows that both low-rank designs, including *Vanilla-TP* and BOOST, perform substantially fewer FLOPs and achieve shorter GEMM times than *FullRank-TP*. The reductions are more pronounced in the MLP linears, where the factorization targets the larger d_{ff} dimension, yielding a bigger drop in work than in attention. This explains why low-rank methods are faster than fullrank in computation. Importantly, *Vanilla-TP* and BOOST have the same FLOP counts and do the same amount of math.

Figure 7 (middle) shows why BOOST finishes the GEMMs faster than *Vanilla-TP* despite their identical FLOP counts. From the figure we can see that BOOST sustains higher hardware utilization in both attention and MLP linears. The

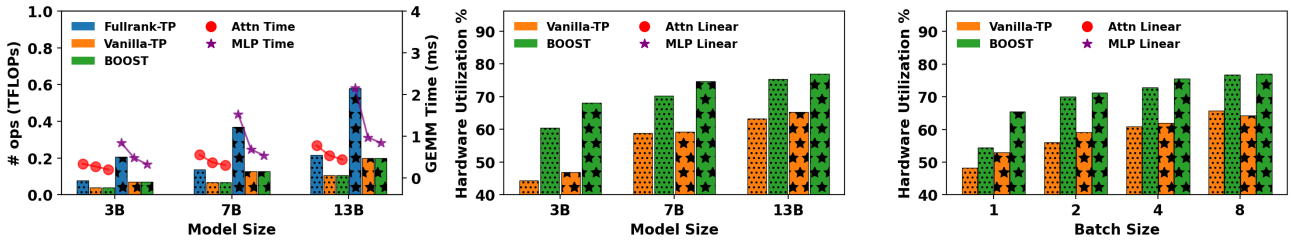

Figure 7. Computation Efficiency. (Left) Linear layer FLOPs and GEMM kernel time under different TP designs; (Middle) Hardware utilization of vanilla TP and BOOST for each linear layer; (Right) Hardware utilization on scaling micro-batch size for each linear layer.

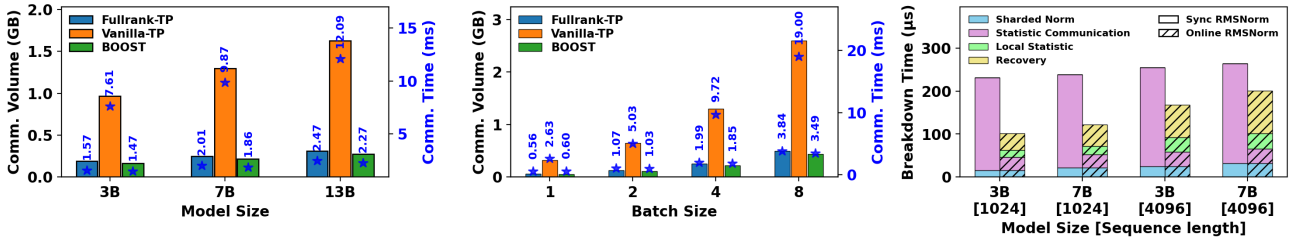

Figure 8. (Left) Communication volume and time on different TP strategy; (Middle) Communication volume and time on scaling micro-batch size; (Right) Online-RMSNorm breakdown.

improvement comes from a more hardware-friendly shard layout: BOOST splits along the larger dimension, yielding GEMMs with larger reduction dimension, which are more compute-bound and less memory-bound. This aligns well with our arithmetic intensity analysis in § 4.1.

From Figure 7 (right), we observe that as the micro-batch size increases, utilization rises for all methods because of more work per launch and better arithmetic intensity. BOOST remains consistently higher than *Vanilla-TP* across batches and layers, demonstrating the efficiency of its kernels to stay closer to the compute-bound regime and achieving end-to-end speedups.

Together, these results show that BOOST not only preserves the algorithmic efficiency of low-rank models, but also substantially improves hardware utilization—bridging the gap between theoretical compression and practical speedup.

5.5 Communication Efficiency

We instrument TP collectives and report the total communication volume per decoder block per pass and the corresponding wall-clock communication time. We compare *FullRank-TP*, *Vanilla-TP*, and BOOST across 7B/13B model sizes and varying micro-batch sizes.

From Figure 8 (left), we can see that *Vanilla-TP* communicates far more than *FullRank-TP* because each bottleneck block adds an extra TP chunk, making two activation all-reduces per block with payloads at the full hidden size d or even the intermediate size d_{ff} (§ 4.1). This inflates both communication volume and time. In contrast, BOOST also increases the number of collectives, but performs them on low-rank activations (bsr instead of bsd), leading to a communication volume smaller than *FullRank-TP* and orders of magnitude below *Vanilla-TP*. The measured time reflects the same reduction, up to 8% faster than *FullRank-TP* and 5.3× faster than *Vanilla-TP* in communication.

As the micro-batch size grows (Figure 8 (middle)), communication volume increases roughly linearly for all methods. The slope for *Vanilla-TP* is steep because of extra full-hidden activations. BOOST 's growth grows much gentler because it communicates on low-rank payloads, and it remains consistently faster than *FullRank-TP* across batch sizes.

These measurements show that our BOOST substantially reduces TP communication volume and time compared to *Vanilla-TP*, and keeps it below *FullRank-TP*. This communication efficiency is a key contributor to the end-to-end throughput gains observed as models and batch size scale.

5.6 Optimization Ablations

Online RMSNorm. Repositioning the TP chunk places sharded-unsafe RMSNorm inside the chunk, we therefore compare *Sync* vs. *Online* RMSNorm. Communication overhead for Online RMSNorm is measured as the difference between communication time with fusion and without fusion; per-step breakdowns are in Figure 8(right).

From this figure, we can observe that: 1) Normalization com-

Table 3. Per-decoder-block time (µs) for Default vs.Grouped linear-layer on CoLA LLaMA-7B model.

	bz=1			bz=4		
Block / Kernel	**Non.**	**Grouped**	**Speed**	**Non.**	**Grouped**	**Speed**
MLP1 Comp	355	292	1.22×	1115	1082	1.03×
MLP1 Comm	266	218	1.22×	620	580	1.07×
QKV Comp	391	255	1.53×	939	877	1.07×
QKV Comm	406	288	1.41×	981	806	1.22×
Layerwise Total	**2773**	**2395**	**1.16×**	**7577**	**7266**	**1.04×**

pute on sharded activations is effectively identical for both methods, the performance gap stems from communication, not arithmetic. 2) Sync RMSNorm is dominated by statistic communication, although its volume is tiny, it suffers high latency from kernel-launch overhead and poor effective bandwidth, so the pink bar is large and nearly flat across model sizes. With longer sequences, it grows modestly for the same reason, it is now launch/latency-bound instead of throughput-bound. 3) Online RMSNorm introduces minimal communication overhead because the statistic exchange is fused into the TP collective, adding little extra traffic when bandwidth is already saturated. Its runtime increase with model size/sequence length is driven primarily by the extra compute in recovery step, not by communication.

Overall, this highlights a clear communication–computation tradeoff: Online RMSNorm replaces the latency-dominated statistic exchange with a fused path, dramatically reducing communication time; the modest extra compute is more than offset by the lower communication cost.

Linear Layers Grouping. As shown in Table 3, grouping improves both compute and communication for the QKV projection and the MLP gate–up path. On compute, grouping amortizes activation reads and enlarges the effective GEMM, raising A.I. and utilization (e.g., 1.53× for QKV at bz=1, 1.07× at bz=4). On communication, it reduces kernel launches and increases per-call payload, improving effective bandwidth (e.g., 1.41× for QKV at bz=1, 1.22× at bz=4). Aggregated per block, this yields a 1.16× speedup at bz=1 and 1.04× at bz=4, confirming that grouping translates analytical gains into end-to-end savings across batch sizes.

Note that the gains are larger at bz=1 than at bz=4. With a smaller per-GPU workload, kernels are more memory-bound and exhibit lower hardware utilization, so grouping's effects in larger effective GEMMs and fewer launches will translate into a bigger speedup.

Efficient Activation Checkpointing. Table 4 provides a comparative per-TP-rank breakdown of memory utilization across methods. The choice of TP strategy does not affect the persistent memory used by weights, gradients, or optimizer states. Instead, *Vanilla-TP* incurs additional activation and communication-buffer memory due to redundant full-width activations materialized after the row-parallel all-reduce, together with larger buffers for full-rank collectives, whereas BOOST operates at the smaller bottleneck dimension r.

Table 4. Per-TP-rank memory breakdown on CoLA LLaMA-7B (bz=4, seq_len=4k). Memory is reported in GB.

Method	**Wgt.**	**Grad.**	**Opt.**	**Act.+ others**	**Total**
Vanilla TP	1.25	1.25	2.50	30.73	35.73
Ours	1.25	1.25	2.50	22.14	27.14

Table 5. Activation checkpointing efficiency on LLaMA-7B. ΔMem and $+$Time are measured with no ckpt. at the same batch size. $\mathrm{Eff}_{\mathrm{ckpt}} = \Delta\mathrm{Mem}/(+\mathrm{Time})$ (MB per ms; higher is better).

Method	**bz**	**ΔMem (MB)**	**+Time (ms)**	**$\mathrm{Eff}_{\mathrm{ckpt}}$ (MB/ms)**
Vanilla	4	26,022	229	113.7
Ours	4	17,414	90	**193.5**
Vanilla	8	52,280	460	113.6
Ours	8	35,392	200	**177.0**

Table 5 compares low-rank activation checkpointing under *Vanilla-TP* and BOOST. For each batch size, we report activation memory saved by checkpointing (ΔMem), extra re-forward time ($+$Time), and efficiency $\mathrm{Eff}_{\mathrm{ckpt}} = \Delta\mathrm{Mem}/(+\mathrm{Time})$. Larger values indicate more memory saved per unit of re-forward.

Our method consistently yields higher $\mathrm{Eff}_{\mathrm{ckpt}}$: for bz=4, 193.5 MB/ms for BOOST vs. 113.7 MB/ms for *Vanilla-TP* (**1.70**×); for bz=8, 177.0 for BOOST vs. 113.6 MB/ms for *Vanilla-TP* (**1.56**×). Although the absolute ΔMem of the BOOST is smaller because *Vanilla-TP* incurs extra activation memory (§ 4.1), its substantially lower re-forward cost ($+$Time) yields a higher $\mathrm{Eff}_{\mathrm{ckpt}}$.

This gain of $\mathrm{Eff}_{\mathrm{ckpt}}$ stems from BOOST eliminating extra communication in the re-forward path (§4.4), making activation checkpointing more efficient.

6 Discussion and Future Work

In this work, we focus on dense low-rank bottleneck architectures, since current low-rank pre-training methods primarily target this regime. However, the core motivation behind BTP is broader. At a high level, BTP applies whenever a submodule becomes deeper, introducing more tensor-parallel synchronization points, and contains a narrow bottleneck activation, allowing TP collectives to be shifted to the lowest-dimensional intermediate representation.

More generally, BTP does not require strictly paired down/up projections or a fixed global bottleneck rank. As

long as a module contains a well-defined narrow activation region, TP collectives can in principle be placed there, even in the presence of unpaired or non-uniform bottlenecks. This suggests that the framework may extend beyond the specific bottleneck layouts studied in this paper.

BTP can also be extended to architectures with grouped-query attention (GQA). In bottlenecked architectures, each linear transformation is replaced by a deeper sequence of low-rank linear layers. Under GQA, the required modification is simply to adjust the shapes of the Q, K, and V projections to respect the grouped-head layout. Importantly, these shape changes leave the core BTP mechanism and its communication-placement strategy unchanged.

Another promising direction is to extend BTP to mixture-of-experts (MoE) models. BTP is orthogonal to expert parallelism: EP routes tokens across experts, whereas BTP changes tensor-parallel boundary placement within a deeper, narrower bottlenecked submodule, provided the shifted TP chunk can still be executed correctly on sharded activations. In current MoE designs, experts are often made more fine-grained without introducing such bottlenecks, which can further increase token-routing all-to-all communication. However, if the experts themselves, or the dense/shared components, are bottlenecked or large enough to require tensor parallelism, BTP can still be applied to shift TP collectives to the narrow intermediate activation, thereby reducing TP communication while preserving higher arithmetic intensity. This is compatible with EP because token-routing all-to-all operates on sharded activations and is therefore orthogonal to TP boundary placement. Since TP remains common in MoE systems, often together with EP, understanding BTP–EP interactions under realistic workloads is an important direction for future work.

7 Conclusion

We have introduced BOOST, a framework for efficiently training large-scale low-rank bottleneck transformer models. By analyzing the computation and communication characteristics of bottleneck architectures, we identified key inefficiencies in naive tensor parallelism, including low arithmetic intensity and excessive communication volume. Our Bottleneck-aware Tensor Parallelism (BTP) design addresses these challenges by redefining TP chunk boundaries to raise GEMM efficiency and cut communication, while Online RMSNorm, linear grouping, and low-rank checkpointing further reduce launches and syncs. Across multiple low-rank architectures and model sizes, BOOST delivers consistent speedups over full-rank TP and vanilla low-rank TP, approaching high hardware utilization while preserving compression benefits, making it a practical, scalable solution for low-rank LLM pre-training.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their thoughtful and constructive feedback, which substantially improved this paper. We also thank Linxing Jiang for helpful discussions on the initial ideas and for insightful advice on framing of the paper.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Artificial Intelligence for Science program, under contracts DE-SC0025390 and DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0030039, as well as NERSC award ALCC-ERCAP0031379.

References

Abecassis, F., Agrusa, A., Ahn, D., Alben, J., Alborghetti, S., Andersch, M., Arayandi, S., Bjorlin, A., Blakeman, A., Briones, E., et al. Pretraining large language models with nvfp4. *arXiv preprint arXiv:2509.25149*, 2025.

Ainslie, J., Lee-Thorp, J., De Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, Y., Zhang, Y., Liu, Y., Yuan, K., and Wen, Z. A memory efficient randomized subspace optimization method for training large language models. *arXiv preprint arXiv:2502.07222*, 2025.

Chmiel, B., Fishman, M., Banner, R., and Soudry, D. Fp4 all the way: Fully quantized training of llms. *arXiv preprint arXiv:2505.19115*, 2025.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.

FairScale authors. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 97–104. Springer, 2004.

Han, A., Li, J., Huang, W., Hong, M., Takeda, A., Jawanpuria, P. K., and Mishra, B. Sltrain: a sparse plus low rank approach for parameter and memory efficient pretraining. *Advances in Neural Information Processing Systems*, 37: 118267–118295, 2024.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Jin, C., Jiang, Z., Bai, Z., Zhong, Z., Liu, J., Li, X., Zheng, N., Wang, X., Xie, C., Huang, Q., et al. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. *arXiv preprint arXiv:2505.11432*, 2025.

Jordan, M. I. and Jacobs, R. A. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2): 181–214, 1994.

Kamalakara, S. R., Locatelli, A., Venkitesh, B., Ba, J., Gal, Y., and Gomez, A. N. Exploring low rank training of deep neural networks. *arXiv preprint arXiv:2209.13569*, 2022.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Khodak, M., Tenenholtz, N., Mackey, L., and Fusi, N. Initialization and regularization of factorized neural layers. *arXiv preprint arXiv:2105.01029*, 2021.

Kong, B., Liang, J., Liu, Y., Deng, R., and Yuan, K. Cr-net: Scaling parameter-efficient training with cross-layer low-rank structure. *arXiv preprint arXiv:2509.18993*, 2025.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

Lang, J., Chen, F., Fan, H., Zhou, H., Wang, Y., and Huan, J. End-to-end llm training on instance clusters with over 100 nodes using aws trainium. Amazon Web Services Machine Learning Blog, 2024. Available online: https://aws.amazon.com/blogs/machine-learning/end-to-end-llm-training-on-instance-clusters-with-over-100-nodes-using-aws-trainium/ (accessed Oct 2025).

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

Li, J., Yin, L., Shen, L., Xu, J., Xu, L., Huang, T., Wang, W., Liu, S., and Wang, X. Lost: Low-rank and sparse pre-training for large language models. *arXiv preprint arXiv:2508.02668*, 2025.

Lialin, V., Shivagunde, N., Muckatira, S., and Rumshisky, A. Relora: High-rank training through low-rank updates. *arXiv preprint arXiv:2307.05695*, 2023.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Liu, D., Yan, Z., Yao, X., Liu, T., Korthikanti, V., Wu, E., Fan, S., Deng, G., Bai, H., Chang, J., et al. Moe parallel folding: Heterogeneous parallelism mappings for efficient large-scale moe model training with megatron core. *arXiv preprint arXiv:2504.14960*, 2025a.

Liu, Z., Zhang, R., Wang, Z., Yang, Z., Hovland, P., Nicolae, B., Cappello, F., and Zhang, Z. Cola: Compute-efficient pre-training of llms via low-rank activation. *arXiv preprint arXiv:2502.10940*, 2025b.

Loeschcke, S., Toftrup, M., Kastoryano, M., Belongie, S., and Snæbjarnarson, V. Loqt: Low-rank adapters for quantized pretraining. *Advances in Neural Information Processing Systems*, 37:115282–115308, 2024.

Luccioni, A. S., Viguier, S., and Ligozat, A.-L. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of machine learning research*, 24(253): 1–15, 2023.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., et al. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–15, 2021.

Peng, H., Wu, K., Wei, Y., Zhao, G., Yang, Y., Liu, Z., Xiong, Y., Yang, Z., Ni, B., Hu, J., et al. Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313*, 2023.

Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–14, 2021.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3505–3506, 2020.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.

Sanders, J., Emberson, L., and Edelman, Y. What did it take to train grok 4?, 2025. URL https://epoch.ai/data-insights/grok-4-training-resources. Accessed: 2025-10-25.

Shamshoum, Y., Hodos, N., Sieradzki, Y., and Schuster, A. Compact: Compressed activations for memory-efficient llm training. *arXiv preprint arXiv:2410.15352*, 2024.

Shazeer, N. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., et al. Slora: Scalable serving of thousands of lora adapters. *Proceedings of Machine Learning and Systems*, 6:296–311, 2024.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Singh, S., Ruwase, O., Awan, A. A., Rajbhandari, S., He, Y., and Bhatele, A. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th International Conference on Supercomputing*, pp. 203–214, 2023.

Tazi, N., Mom, F., Zhao, H., Nguyen, P., Mekkouri, M., Werra, L., and Wolf, T. The ultra-scale playbook: Training llms on gpu clusters, 2025.

Walker, D. W. and Dongarra, J. J. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Wang, X., Zheng, Y., Wan, Z., and Zhang, M. SVD-LLM: Truncation-aware singular value decomposition for large language model compression. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=LNYIUouhdt.

Workshop, B., Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

Xi, H., Li, C., Chen, J., and Zhu, J. Training transformers with 4-bit integers. *Advances in Neural Information Processing Systems*, 36:49146–49168, 2023.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Yang, Z., Liu, Z., Choudhary, S., Xie, X., Gao, C., Kunzmann, S., and Zhang, Z. Comera: Computing-and memory-efficient training via rank-adaptive tensor optimization. *Advances in Neural Information Processing Systems*, 37:77200–77225, 2024.

Yuan, J., Gao, H., Dai, D., Luo, J., Zhao, L., Zhang, Z., Xie, Z., Wei, Y., Wang, L., Xiao, Z., et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Zhanfeng Mo, Long-kai Huang, S. J. P. Parameter and memory efficient pretraining via low-rank riemannian optimization. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=i0zzO7Hslk.

Zhang, R., Liu, Z., Wang, Z., and Zhang, Z. Lax: Boosting low-rank training of foundation models via latent crossing. *arXiv preprint arXiv:2505.21732*, 2025.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhao, J., Zhang, Z., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Zhu, H., Zhang, Z., Cong, W., Liu, X., Park, S., Chandra, V., Long, B., Pan, D. Z., Wang, Z., and Lee, J. Apollo: Sgd-like memory, adamw-level performance. *arXiv preprint arXiv:2412.05270*, 2024.

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation of BOOST, the training framework proposed in the MLSys 2026 paper "BOOST: Bottleneck-Optimized Scalable Training Framework for Low-Rank Large Language Models." The artifact includes the source code, experiment scripts, and configuration files used to reproduce the key experimental results in the paper.

The artifact implements the core components of BOOST, including Bottleneck-aware Tensor Parallelism (BTP) and Online RMSNorm integrated into a distributed training framework based on PyTorch and Nanotron. These components enable efficient distributed training of low-rank bottleneck architectures for large language models.

The artifact supports the main claims of the paper regarding improved training efficiency and scalability for low-rank LLM architectures. In particular, it demonstrates reduced communication overhead, improved GPU utilization, and faster training iteration times compared with both full-rank tensor-parallel baselines and naive tensor-parallel implementations of low-rank models. Minimal hardware requirements are 1–4 NVIDIA GPUs with CUDA support and a Linux environment with PyTorch and NCCL installed.

The artifact provides scripts to install dependencies, run small-scale distributed training experiments, and reproduce representative performance metrics such as iteration time, communication volume, and hardware flops utilization. Running the provided scripts generates logs and profiling outputs demonstrating the performance improvements of BOOST over baseline implementations.

A.2 Artifact check-list

- **Algorithm: Distributed training for low-rank LLMs; bottleneck-aware tensor parallelism**
- **Dataset: WikiText / synthetic short-run data for validation**
- **Run-time environment: Nanotron; Linux; CUDA; multi-GPU distributed training**
- **Hardware: 80GB A100 NVIDIA GPUs, 4 GPUs per node**
- **Execution: shell scripts; torchrun**
- **Metrics: iteration time; communication volume; hardware utilization; throughput**
- **Output: training logs; summary tables**
- **How much time is needed to prepare workflow (approximately)?: 10 minutes**
- **How much time is needed to complete experiments (approximately)?: 1 hour**

- **Publicly available: Yes.**
- **Code licenses: Apache-2.0**
- **GitHub link: https://github.com/Arcana-2236/BOOST**

A.3 Description

A.3.1 How delivered

The artifact is delivered as a public GitHub repository containing the BOOST implementation, example configurations, experiment scripts, and instructions for installation and evaluation. The repository includes:

- the BOOST implementation integrated into the Nanotron training framework,
- configuration files for full-rank, vanilla low-rank, and BOOST runs,
- scripts for lightweight validation experiments,

The public repository URL is provided in the submission form and in the README.

A.3.2 Hardware dependencies

The minimal supported setup is 1–4 NVIDIA GPUs with CUDA support. A single-node 4-GPU setup is sufficient for the main lightweight validation experiments. The main experiments in the paper were conducted on NVIDIA A100 GPUs (80GB) and larger multi-node settings, but the artifact includes smaller validation runs intended for AE reviewers.

Recommended:

- 4 NVIDIA A100 GPUs for the closest reproduction of the reported results;
- NVLink-enabled intra-node communication for tensor-parallel evaluation.

A.3.3 Software dependencies

The artifact requires:

- Linux,
- Python 3.10 or newer,
- CUDA 12.x,
- PyTorch with CUDA support,
- NCCL,
- Nanotron.

Optional dependencies:

- Nsight Systems for profiling,
- Conda or docker for environment management.

A.4 Installation

The artifact can be executed using the NVIDIA PyTorch container to ensure consistent dependencies across environments. Detailed installation instructions are provided in the GitHub repository README.

A.5 Experiment workflow

The repository provides scripts for running distributed training experiments comparing three configurations: the full-rank baseline, the low-rank vanilla tensor-parallel baseline, and the BOOST approach with Bottleneck-aware Tensor Parallelism (BTP).

To reproduce a direct comparison between the three approaches, the repository provides a script that sequentially runs the full-rank baseline, the low-rank vanilla tensor-parallel baseline, and the BTP implementation. The script records the average iteration time for each configuration, enabling a quick comparison of training performance.

The comparison experiment can be launched using the following command:

- bash run_iter_compare.sh

Alternatively, the configurations can be executed individually using the following commands:

- Full-rank baseline

  CUDA_DEVICE_MAX_CONNECTIONS=1 torchrun –nproc_per_node=4 run_train.py –config-file examples/config_tiny_llama.yaml

- CoLA model with BOOST (BTP)

  CUDA_DEVICE_MAX_CONNECTIONS=1 torchrun –nproc_per_node=4 examples/cola/train_cola.py –config-file examples/cola/config_tiny_cola_llama.yaml

A.6 Evaluation and expected result

After running the experiment workflow, logs and runtime statistics are generated in the logging directory.

Reviewers should observe the qualitative trends reported in the paper, including:

- BOOST achieves lower iteration time than naive tensor-parallel low-rank implementations.
- Communication overhead is reduced due to communication on low-rank activations.
- Hardware utilization improves compared with naive tensor-parallel baselines.

Exact numerical values may vary depending on GPU model and system configuration, but the relative performance trends should remain consistent with those reported in the paper.

The experiments reported in the paper were conducted on 80GB NVIDIA A100 GPUs. For reproduction on 40GB A100 GPUs, the batch size can be adjusted accordingly using the provided scripts.

A.7 Experiment customization

The artifact provides .yaml configuration files under the `examples/` directory that allow users to modify experimental parameters.

Users may customize:

- model size (TinyLLaMA, LLaMA-1B, LLaMA-7B, etc.)
- batch size and sequence length
- 3D parallelism configurations
- low-rank model variants and BOOST optimizations

These parameters can be modified by editing the configuration files before launching training runs. Additional instructions for running larger experiments and profiling workflows are provided in the GitHub README.

A.8 Methodology

Submission, reviewing, and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

B APPENDIX

B.1 Detailed Performance Analysis

Communication volume analysis. We report per-iteration communication volume V_{comm} for a LLaMA-style decoder in Table 6 with l layers, microbatch size b, sequence length s, hidden size d, MLP expansion d_{ff}, low-rank size $r \ll d$, and pipeline degree p. The counts assume Megatron-style implementations, no sequence parallelism, constants and embedding/norm traffic are omitted for clarity.

DP. Gradient synchronization volume scales with parameter size. For full rank, each layer's attention contributes $4d^2$ (Q,K,V,O) and MLP contributes $3dd_{ff}$ (gate, up, down), giving $l(4d^2 + 3dd_{ff})$. For low-rank variants (both Vanilla and Ours), replace $d^2 \to 2dr$ and $dd_{ff} \to dr + d_{ff}r$ consistent with factorized weights across the same 4+3 linear maps, yielding $l(11dr + 3d_{ff}r)$.

PP. Pipeline activations of size $[b, s, d]$ traverse stage boundaries in forward/backward. Abstracting per-edge factors and schedule details, volume scales as $pbsd$ (with p stages) per pass, the same for full rank and low rank because boundary tensors are at width d in this configuration.

TP. In Megatron's column–row pattern, each decoder block issues one activation-sized collective in attention and one in MLP per pass, i.e., $2 \times [b, s, d]$ per layer. Accounting for forward and backward gives $2l(2bsd)$. Vanilla Low-rank TP treats each low-rank linear separately increases collective count. Per pass, attention triggers $4bsd$ (one per $d{\times}r/r{\times}d$ linear) and MLP triggers $bsd + 2bs\,d_{ff}$ (down plus up), totaling $(5bsd + 2bsd_{ff})$; forward+backward yields $2l(5bsd + 2bsd_{ff})$. With BOOST, the TP chunk boundary is shifted so the single collective happens at low-rank tensor $[b, s, r]$. Summing over chunks gives $7\,bsr$ per pass and thus $2l(7bsr)$ per iteration. This replaces $O(d)$ activation-sized traffic with $O(r)$, explaining the large reduction relative to full rank and Vanilla low rank.

Arithmetic Intensity analysis. We report per-MLP–block arithmetic intensity (A.I. = FLOPs / data moved) under three TP designs (Table 7. Let hidden size be d, expansion $d_{ff} = \alpha d$, low-rank size r with $d = \beta r$ ($\beta = d/r$), microbatch b, sequence length s, and TP degree TP. We count forward only GEMMs and the dominant activation/weight traffic; constants, bias/elementwise ops, and embedding/norm terms are omitted.

The result separates *algorithmic* vs. *system* effects. First, both low-rank variants (Vanilla and BOOST) have strictly fewer FLOPs than Full-rank TP, this is the **algorithmic efficiency** of bottleneck factorization ($d \times d \to d \times r$, $r \times d$ with $r \ll d$). Second, *within* the low-rank family, **Vanilla and BOOST have the same FLOPs**: the TP design does not change the amount of math, only how it is scheduled and communicated. Third, BOOST's advantage comes from **lower data movement**: by sharding along the large dimension and communicating on the low-rank path, BOOST reduces activation traffic and yields higher arithmetic intensity (more FLOPs per byte). Hence, the performance gap between Vanilla and BOOST is a **system-level** gain (communication/memory traffic), not an algorithmic one.

B.2 Model Configuration

Table 8 lists the LLaMA-style model settings used in our experiments. For each size (1B–30B), we specify the number of transformer layers, hidden width d, and MLP expansion d_{ff}, alongside a canonical low rank $r = d/4$ for the bottleneck variants

Table 6. Per iteration communication volume (V_{comm}) under different parallelism strategies for full-rank and bottleneck architecture in Llama-like models.

STRATEGY	FULL-RANK	BOTTLENECK ARCHITECTURE	
		VANILLA	OURS
DATA PARALLEL (DP)	$l(4d^2 + 3dd_{ff})$	$l(11dr + 3d_{ff}r)$	
PIPELINE PARALLEL (PP)	$2pbsd$	$2pbsd$	
TENSOR PARALLEL (TP)	$2l(2bsd)$	$2l(5bsd + 2bsd_{ff})$	$2l(7bsr)$

Table 7. Per MLP block GEMM arithmetic intensity (A.I.) for fullrank TP, vanilla low-rank TP and bottleneck-aware TP (Assuming $d_{ff} = \alpha d$, $d = \beta r$).

TP DESIGN	FLOPS	DATA MOVEMENT	A.I.
FULLRANK TP	$4\alpha bsd^2/TP$	$4d(bs + \frac{\alpha(d+bs)}{TP})$	$\frac{\alpha bsd^2}{bsdTP+\alpha d(d+bs)}$
VANILLA LOW-RANK TP	$\frac{4(1+\alpha)bsd^2}{\beta TP}$	$4d((1+\alpha)bs + \frac{(1+\alpha)d+2bs}{\beta TP})$	$\frac{4bsd^2(1+\alpha)}{4bsd\,\beta TP(1+\alpha)+4d^2(1+\alpha)+8bsd}$
BOTTLENECK TP		$4d(\frac{(1+\alpha)(\beta bs+d)+2bsTP}{\beta TP})$	$\frac{4bsd^2(1+\alpha)}{4\beta bsd(1+\alpha)+4d^2(1+\alpha)+8bsd\,TP}$

Table 8. Model configuration (LLaMA-style). We use a canonical low rank $r = d/4$.

Model size	Layers	head	d	d_{ff}	r
1B	24	32	2048	5472	512
3B	28	24	3072	8192	768
7B	32	32	4096	11008	1024
13B	40	40	5120	13824	1280
30B	36	64	8192	22016	2048

(SVD/CoLA/LaX). These configurations follow common scaling rules while keeping r proportional to d so that accuracy remains comparable and the system effects of low-rank factorization (communication volume and arithmetic intensity) are isolated. We use these fixed shapes across all TP/PP settings to ensure that reported performance differences stem from the parallelism design rather than architectural changes.

B.3 Low-rank Training Algorithm

Common setup. Unless noted otherwise, we use pure **bf16** for training with **AdamW** and a single **fused** CUDA kernel that updates parameters and optimizer states.

SVD baseline (system baseline). To establish a system-performance lower bound with minimal algorithmic changes, we *mechanically* replace every full-rank linear W with two low-rank linears:

$$y \;=\; A\,(B\,x), \quad A \in \mathbb{R}^{d_{\text{out}} \times r},\; B \in \mathbb{R}^{r \times d_{\text{in}}}. \tag{6}$$

No nonlinearity is introduced between B and A. This variant isolates the effect of reduced arithmetic/memory while keeping the computation graph closest to the full-rank case.

CoLA variant (nonlinear bottleneck). For CoLA, we insert a nonlinearity in the low-rank pathway. Concretely, with *SwiGLU* as the nonlinearity, following the original paper setting,

$$y = A \cdot \mathrm{SwiGLU}(Bx). \tag{7}$$

LaX variant (residual low-rank). For LaX, we implement a residual low-rank path using an *Identity Gate* and inter-layer LaX (low-rank residual):

$$\begin{aligned} h_{i-1} &= A_{i-1}\,x_{i-1} \in \mathbb{R}^r, \\ h_i &= A_i\,x_i \in \mathbb{R}^r, \\ \tilde{y}_i &= B_i\,(h_i + h_{i-1}) \in \mathbb{R}^{d_{\text{out}}}. \end{aligned} \tag{8}$$

B.4 Linear Grouping Details

Figure 9 details our linear-grouping implementation. In BTP, the first down-projection is row-parallel; we therefore concatenate the weights along the rank dimension and execute a single GEMM to produce a wider activation, which is then split per path. The subsequent all-reduce remains unchanged. For the column-parallel up-projections, we stack weights/activations and use a batched GEMM (e.g., `einsum`) to accommodate the stacked shapes.

B.5 Framework Details

Process groups and topology. Nanotron exposes a *ParallelContext* that constructs and tracks all process groups for 3D parallelism (DP/TP/PP) alongside a world group. It provides rank mapping utilities (world_rank_matrix, local rank queries) and lets modules invoke collectives on the correct subgroup granularity.

Tensor parallelism (TP). Nanotron implements Megatron-style column/row-parallel linears, parallel vocabulary, cross-entropy over sharded logits, and distributed samplers. Different sharding patterns trade duplicated compute for reduced communication; engines select the appropriate collective (e.g., all-reduce or all-gather) over the TP group.

Pipeline parallelism (PP). Models are assembled from *PipelineBlocks*, a lightweight wrapper that pins a submodule to a

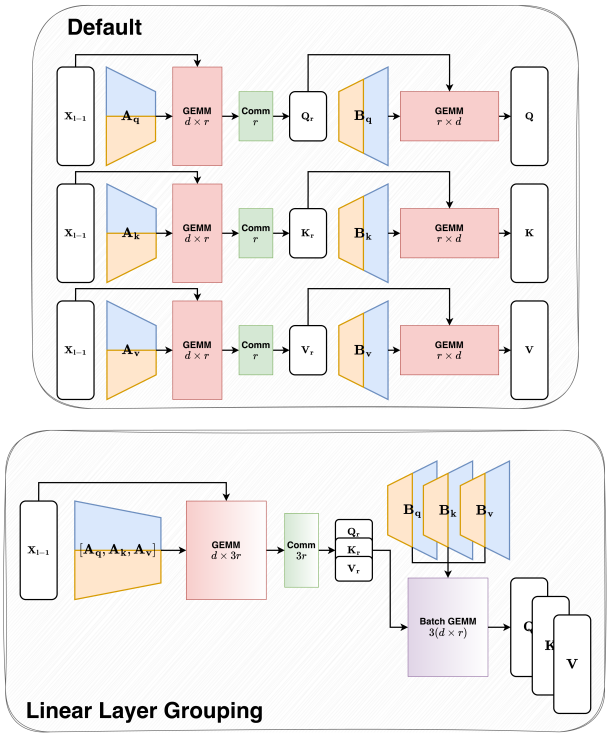

Figure 9. Linear layer grouping implementation

PP rank and routes tensors via `TensorPointer` when a rank is not responsible for compute. We use *One-Forward-One-Backward (1F1B)* scheduling.

Parameter metadata. To make sharding and tying explicit, Nanotron augments `nn.Parameter` with metadata: (i) *Sharded parameters* carry `ShardedInfo` (global ranks, local/global slices, and the unsharded shape), enabling precise slicing and regrouping; (ii) *Tied parameters* carry `TiedInfo` (a canonical name, scope root, involved ranks, and an optional `reduce_op` for gradient syncing). A parameter may be both sharded (across TP) and tied (across PP endpoints).

Recomputation utilities. A `@checkpoint_method` decorator enables activation recomputation on a per-module basis, reducing activation memory at the cost of controlled recompute. We implement the low-rank activation checkpointing with this.

On-device initialization. Context managers allow constructing modules directly on `cuda` (bf16) or on `meta` and then materializing on device, avoiding large CPU-side allocations and speeding up initialization.

Collectives. All DP/TP/PP communication uses NCCL collectives (e.g., all-reduce, all-gather, reduce-scatter) on a dedicated communication stream. Fine-grained overlap of all-reduce with compute is not currently supported in Nanotron.