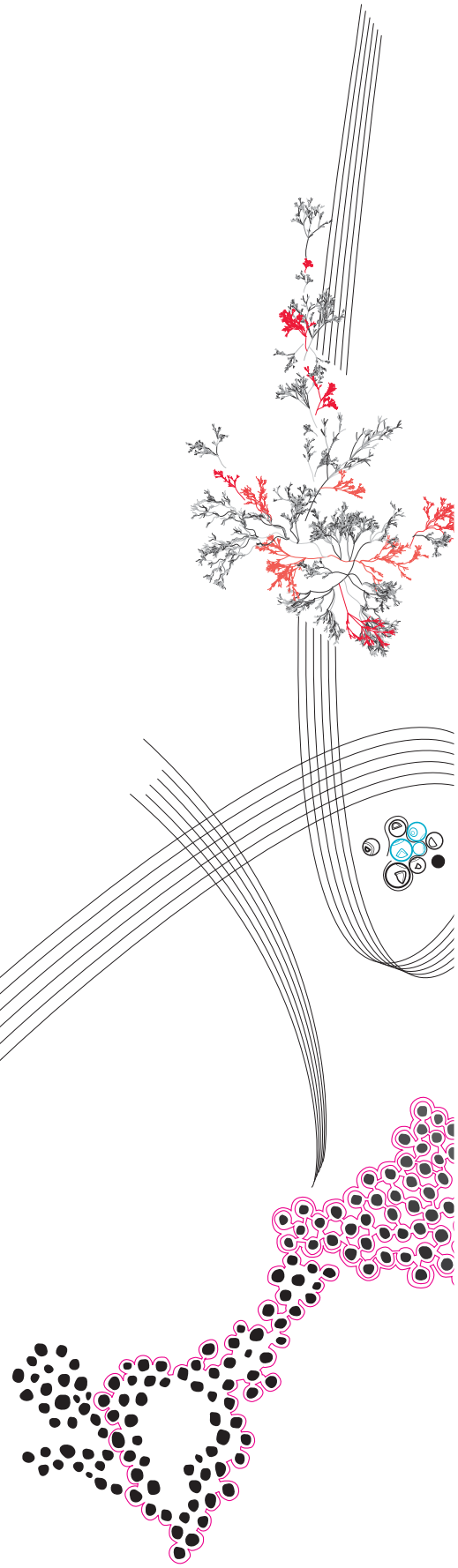


MSc Thesis Computer Science

Optimizing the Computational Efficiency of Fine-tuning and Inference for Large Language Models

Luat Gia Khoi Nguyen

Supervisor: Assoc. Prof. Alexandros Stergiou

June 26, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Acknowledgements

I would like to express my deepest gratitude to Assoc. Prof. Alexandros Stergiou at the University of Twente for his invaluable guidance and support throughout the formulation and writing process of this thesis. His insights and advice have been instrumental in shaping the direction of this work.

I am also sincerely grateful to Prof. Alexander Ilin at Aalto University and System 2 AI Ltd., as well as Dr. Harri Valpola at System 2 AI Ltd., for providing me with technical supervision and the opportunity to work on this project. Their expertise and mentorship have greatly enriched my understanding and contributed to the successful completion of this research.

Furthermore, I would like to extend my heartfelt thanks to Assoc. Prof. Pekka Marttinen for the resources and support provided by his research group at Aalto University, which enabled me to carry out this thesis work.

To all the individuals and institutions who have supported me during this journey, I am deeply appreciative of your contributions and encouragement.

Chapter 1

Introduction

1.1 Introduction

Fine-tuning Large Language Models (LLMs) for specific downstream tasks has become a critical step in leveraging their full potential. However, this process requires significant GPU resources, even with parameter-efficient methods like Low-Rank Adaptation (LoRA). The sheer size of LLMs poses a major challenge, as the models alone are often too large to fit into a single GPU, making efficient resource utilization a key priority during fine-tuning. This limitation necessitates advanced computational strategies to enable effective training and inference.

Parallelism techniques have emerged as essential tools for improving the computational efficiency of fine-tuning LoRA-infused models. Among these, Fully Sharded Data Parallelism (FSDP) and Tensor Parallelism (TP) have shown significant promise in optimizing memory utilization and throughput for large-scale models. These techniques enable training on longer context lengths and larger batch sizes while maintaining high throughput and lower peak memory consumption, which is critical when fine-tuning with LoRA. FSDP distributes model parameters, optimizer state, and gradients while offering competitive throughput thanks to communication-computation overlapping. However, as it is still indeed a data parallel (DP) approach at its core, the minimum batch size is limited to the number of GPUs, with no possibility to reduce it further, consequently limiting the maximum trainable sequence length of the input batch. TP, on the other hand, offers distribution of computation, enabling sub-unit batch size and longer trainable sequence length per GPU. However, its high communication overhead greatly affects throughput. As a result, hybrid parallelism approaches that combine FSDP and TP offer a powerful solution by balancing inter-GPU communication and memory utility, addressing the trade-offs inherent in standalone parallelism strategies. Although the benefits of hybrid approaches are theoretically well-established, comprehensive comparisons between these three configurations in general, and the practical superior performance that hybrid approaches offer specifically, have not been thoroughly studied, particularly for fine-tuning LoRA-infused models on single-digit GPU node resources.

This thesis investigates the computational trade-offs and performance of FSDP-TP hybrid approaches applied to LoRA-infused models. Specifically, we evaluate their impact on maximum trainable sequence length (MTSL), throughput, and processing time per sample (PTS) across different configurations and scenarios, including single-node and multi-node setups. In order to apply TP to LoRA adapters, we propose a TP paradigm for LoRA adapters, in addition to the original paradigm for transformer model weights, enabling compatibility of LoRA computed logits with those produced by original model weights.

We also propose an efficient model loading method under FSDP-TP to address critical bottlenecks such as CPU memory allocation and CPU-GPU communication overheads.

The contributions of this work are fourfold:

1. A proposed TP paradigm for LoRA adapters, enabling integration with existing TP paradigms for original transformer model weights.

2. A proposed method for efficient model loading under FSDP-TP, addressing critical bottlenecks such as CPU memory allocation and CPU-GPU communication overheads.

3. A comprehensive analysis of the computational performance of FSDP, TP, and their FSDP-TP hybrid approaches for LoRA-infused models, highlighting their advantages and limitations in terms of maximum trainable sequence length (MTSL), throughput, and processing time per sample (PTS).

4. Experimental validation of parallelism configurations across single-node and multi-node setups, providing practical insights into optimal parallelism strategies for various training scenarios, enabling effective scaling.

By bridging the gap between parallelism strategies and parameter-efficient fine-tuning, this thesis contributes to the broader goal of enabling scalable and efficient training and inference for large-scale LLMs. The findings and methodologies presented herein aim to serve as a foundation for future research in the domain of computational optimization for fine-tuning LLMs with LoRA.

This thesis is organized as follows. Chapter 2 reviews relevant literature on LLM architectures, efficiency techniques, and fine-tuning approaches. Chapter 3 details our methodology, including tensor parallelism for LoRA modules and our efficient model loading method. Chapter 4 presents experimental results for both single-node and multi-node configurations, focusing on three defined metrics: maximum trainable sequence length (MTSL), throughput, and processing time per sample. Chapter 5 discusses our findings and outlines future research directions. Supplementary materials and additional experimental data are provided in the appendices. In this document, we occasionally use "DP" to refer to "FSDP" since FSDP is a data parallel (DP) technique, and no plain DP is involved in this study; therefore, this should not cause any confusion.

Chapter 2

Literature Review

2.1 Introduction to Large Language Models

This section overviews the core architecture, training approaches, and applications of modern Large Language Models (LLMs). In section 2.1.1, we examine the fundamental Transformer architecture and its key variants - encoder-only, decoder-only, and encoder-decoder models - which form the backbone of contemporary LLMs. Subsequently, section 2.1.2 explores essential techniques for scaling LLMs by increasing parameters, depth, and width, along with methods to address the resulting computational challenges.

2.1.1 Modern LLM Architecture

Transformers [61] are the foundational blocks for modern LLMs and rely entirely on attention.

Attention allows the model to weigh the importance of different parts of the input sequence when generating a representation for each element in the sequence. This mechanism enables the model to capture long-range dependencies and relationships within the data more effectively than RNN-based models.

Key Components Transformers typically comprise an encoder and a decoder (Figure 2.1). The encoder processes the input sequence and generates a contextualized representation of it. The decoder then uses this representation, along with its own internal state, to generate an output sequence. Both the encoder and decoder are made up of stacked layers, each containing multi-head attention mechanisms and position-wise feed-forward networks (Figure 2.1).

Architectural variations

This section will discuss three main attention-based architectural variations: encoder-only, decoder-only, and encoder-decoder models.

Encoder-only models like BERT [15] utilize an encoder-only architecture for tasks like sentence classification, question answering, and natural language inference. BERT's pre-training approach involves two key objectives: masked language modeling, which trains the model to predict randomly masked tokens in a sentence, and next sentence prediction, which enables the model to determine if two sentences logically follow each other. These objectives help BERT learn rich contextual representations from large amounts of

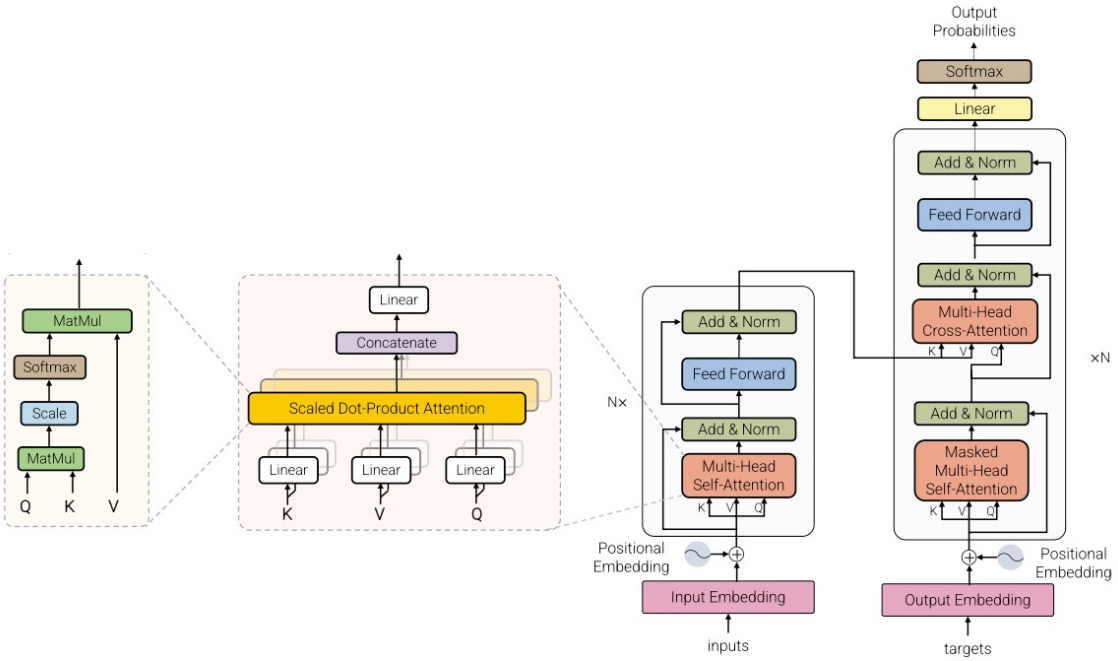


FIGURE 2.1: Architecture of the standard Transformer. Excerpt from [55].

unlabeled text data, resulting in significant performance improvements compared to previous non-contextual embedding methods (e.g., Word2Vec, GloVe) due to its ability to capture contextual nuances in language. However, the computational cost of BERT can be a limiting factor for certain applications, particularly in resource-constrained scenarios or real-time systems, as BERT's large size and complexity demand significant computational resources. Specifically, BERT's bidirectional architecture necessitates simultaneous processing of the entire input sequence, increasing computational overhead. This limitation has prompted research into more efficient variants, such as DistilBERT and TinyBERT.

Decoder-only models. such as the GPT family [41, 42, 7] exemplifies the decoder-only approach. These models focus on autoregressive text generation with causal attention. GPT models generate the next token in a sequence by conditioning on the preceding context, using a unidirectional (causal) transformer architecture. This unidirectional approach has shown remarkable performance in tasks like machine translation [56], text summarization [68, 1], and dialogue generation [57, 49]. In addition, this architecture has proven particularly effective for in-context learning [7, 29] and chain-of-thought reasoning [64, 63]. The development of larger GPT models, such as GPT-3 [7], has further pushed the boundaries of language generation, showcasing the potential of decoder-only architectures in capturing long-range dependencies and generating human-quality text. Notably, the principles underlying these models extend beyond text processing, as the decoder architecture can effectively process encodings from various modalities, including vision and audio inputs [3, 16].

Encoder-decoder models like T5 [44], utilizes both an encoder and a decoder. This structure is well-suited for sequence-to-sequence tasks, such as machine translation [5, 65, 61], text summarization [50, 68, 33], and question answering [47, 30, 8], where the model needs to capture complex dependencies between the input and output sequences.

Memory Usage of Multi-head Attention (MHA) and Multi-layer Perceptron (MLP) Blocks

The memory usage of an MHA block, according to Li *et al.* [34], can be represented as:

$$\frac{16AZH}{N} + \frac{4BLZA}{N} + \frac{BZL^2}{N} + BLH \tag{2.1}$$

where B is batch size, L is sequence length, H is hidden size of linear layers, A is attention head size, Z is number of attention heads, and N is number of GPUs.

On the other hand, the memory usage of an MLP block is:

$$\frac{32H^2}{N} + \frac{4BLH}{N} + BLH \tag{2.2}$$

2.1.2 Scaling Large Language Models

This section examines the critical dimensions of scale in large language models and associated techniques.

Scaling techniques (depth, width, and parameter count) Scaling LLMs involves increasing the number of parameters, layers (depth), or hidden units (width). Simply increasing the size of LLMs can lead to significant computational challenges. Techniques such as recurrence mechanisms (e.g., Transformer-XL [12]), memory-based approaches (e.g., Compressive Transformer [43]), low-rank methods (e.g., Linformer [62]), and fixed patterns (e.g., Big Bird [67]) have been used to reduce computational overheads. Additionally, sparse architectures (e.g., Sparse Transformer [9]) and conditional computation (e.g., Switch Transformer [18]) offer promising avenues for scaling Transformers while maintaining efficiency.

Impact of scaling Scaling has demonstrated significant benefits across various applications. Models like Llama 2 [60] have shown that increased scale, combined with careful fine-tuning on high-quality conversation data, can produce models that rival or exceed the performance of specialized dialogue systems. Similarly, unified approaches like T5 [44] have effectively leveraged scale to transfer learning across diverse NLP tasks.

2.2 Efficiency in LLMs

This section examines the key challenges and solutions in making Large Language Models (LLMs) more computationally efficient. In Section 2.2.1, we analyze the fundamental computational challenges facing modern LLMs, including memory requirements, training costs, and computational complexity. Section 2.2.2 explores various model compression techniques such as quantization, pruning, and knowledge distillation. Section 2.2.3 discusses efficient architecture design approaches, particularly focusing on attention mechanisms and mixture of experts. Finally, in section 2.2.4, we detail different parallelization strategies for memory-efficient training, including data, tensor, pipeline, and hybrid parallelism.

2.2.1 Computational Challenges in LLMs

Model Complexity and Capacity. The parameter count in modern LLMs has undergone a dramatic surge, escalating from BERT's 340 million parameters [15] to 540 billion parameters in PaLM [11]. This exponential growth translates directly into substantial memory requirements and computational overhead. In mixed-precision training, as detailed by Rajbhandari et al. [45], a single parameter requires 16 bytes of memory, leading to significant hardware demands. For instance, training GPT-3 with its 175 billion parameters cost over $4.6 million using Tesla V100 cloud instances [31], highlighting both the financial and computational intensity of developing these models.

Measures of Compute. The computational demands of LLMs can be quantified through several key metrics. Floating-point operations (FLOPs) measure the basic arithmetic operations during inference, with the self-attention mechanism exhibiting quadratic complexity $O(n^2d)$ relative to sequence length n and model dimension d [61]. Another metric is memory footprint. Memory footprint refers to the RAM required during model operation. It encompasses both model states and residual states, where even inference can exceed single-GPU capacity [45]. To measure inference performance, latency (response time) and throughput (tokens per second) are common metrics, which are especially crucial for real-time applications. These computational challenges have driven innovations in model compression, efficient architectures, and distributed training strategies like Tensor Parallelism (TP) [52], which forms a key technique in this research.

2.2.2 Model Compression Techniques

Quantization. Quantization compresses LLMs by converting model weights and/or activations from high-precision data types to low-precision ones. There are several approaches to quantization. Post-training quantization (PTQ) applies quantization after model training, with methods like LLM.int8() achieving significant memory reduction while maintaining model performance [14]. GPTQ enables compression to 3 or 4 bits with minimal accuracy loss [20]. Quantization-aware training (QAT) incorporates quantization during the training process, with methods like QuantGPT achieving 14.4× compression rates while maintaining performance [54]. Mixed-precision training enhances efficiency by using low-precision models for forward and backward propagation while converting gradients to high precision for weight updates. Notable implementations include Automatic Mixed Precision (AMP) [38] and BFLOAT16 [28].

Pruning. Pruning reduces model size by removing redundant or less important parameters. Structured pruning focuses on eliminating structured patterns like rows or columns in weight matrices. For example, LLM-Pruner uses gradient information to selectively remove non-essential interconnected structures [37], while Sheared LLaMA achieves superior compression by pruning layers, heads, and dimensions in an end-to-end manner [66]. Unstructured pruning removes individual weights, offering more flexibility but potentially creating irregular sparsification patterns. SparseGPT demonstrates that models like OPT-135B can reach about 60% unstructured sparsity with only slight performance degradation [19]. Wanda achieves competitive performance by pruning based on weight magnitudes and their respective input activations [53].

Knowledge Distillation. Knowledge distillation transfers knowledge from a large teacher model to a smaller student model, and can be categorized into white-box and black-box

approaches. White-box KD utilizes the teacher model's parameters or logits in the distillation process, with methods like Baby LLaMA demonstrating successful distillation through training an ensemble of smaller models [59]. MiniLLM improves conventional KD by employing policy gradient techniques to minimize reverse KLD, achieving better accuracy than traditional KD [21]. TED enhances performance through layer-specific task distillation using specially designed filters to align internal states of both models [35]. In contrast, black-box KD only uses the teacher model's output generations, making it more flexible. For instance, Lion introduces an adversarial distillation architecture that incrementally improves the student model's skill level through imitation, discrimination, and generation [27].

2.2.3 Efficient Architecture Design Techniques

Efficient Attention. The quadratic time and space complexity of attention modules significantly impacts the efficiency of LLMs in pre-training, inference, and fine-tuning. Several approaches have been proposed to optimize attention mechanisms. Sharing-based attention techniques like multi-query attention (MQA) [51] and grouped-query attention (GQA) [2] accelerate inference through KV heads sharing. Another approach involves kernelization or low-rank techniques, adopted by models such as Sumformer [4] and Performer [10], which enhance efficiency by utilizing low-rank representations of the self-attention matrix. Hardware-assisted attention techniques, exemplified by FlashAttention [13], optimize memory access between GPU high-bandwidth memory and on-chip SRAM during attention computation.

Mixture of Experts (MoE). MoE represents a sparse approach that segments tasks into sub-tasks handled by specialized smaller models called experts. This architecture enhances model capacity while managing computational and memory requirements efficiently. Notable implementations include GShard [32], which offers refined parallel computation frameworks, and Switch Transformer [17], which introduces a switch routing algorithm supporting up to one trillion parameters divided among 2,048 experts. Recent developments include Mixtral 8x7B [26], which outperforms larger models like LLaMA-2 70B on various benchmarks while using only 12.9B parameters per token for inference, demonstrating a 6x faster inference speed.

2.2.4 Parallelism for Memory Efficiency

Data Parallelism. Data parallelism is a fundamental approach to scaling deep learning training across multiple devices where each worker maintains a complete copy of the model while the input dataset is sharded across workers [40]. In this strategy, each worker processes different subsets of data in parallel, computes gradients locally, and then aggregates these gradients periodically through collective communication operations to ensure all workers maintain consistent model versions. The standard implementation uses an AllReduce operation to average gradients across workers before applying optimizer updates. While simple and effective for models that fit on a single device, basic data parallelism becomes inefficient for very large models due to the memory overhead of replicating the entire model, gradients, and optimizer states on each device [69]. Additionally, beyond a certain point, the per-GPU batch size becomes too small, reducing GPU utilization and increasing communication costs relative to computation. The maximum number of devices that can be effectively used is also limited by the global batch size, as each worker needs at least one sample to process [40].

Fully Sharded Data Parallel. Fully Sharded Data Parallel (FSDP) addresses the memory limitations of standard data parallelism by sharding model parameters, gradients, and optimizer states across data-parallel workers [69]. FSDP decomposes the model into smaller units and manages each unit independently, only materializing unsharded parameters and gradients of one unit at a time while keeping other units sharded. During forward and backward passes, parameters are gathered on-demand before computations and then immediately resharded afterward. FSDP offers various sharding strategies ranging from fully replicated to fully sharded, with hybrid approaches in between. The sharding factor F determines how many ranks share parameters, with $F = 1$ being equivalent to standard data parallelism and $F = world\_size$ providing maximum memory savings [69]. Key optimizations include deferred initialization for efficient model creation, communication scheduling to overlap with computation, and rate limiting to manage GPU memory fragmentation. FSDP can be combined with other parallelism techniques and has been shown to achieve comparable performance to standard data parallelism while enabling training of significantly larger models [45].

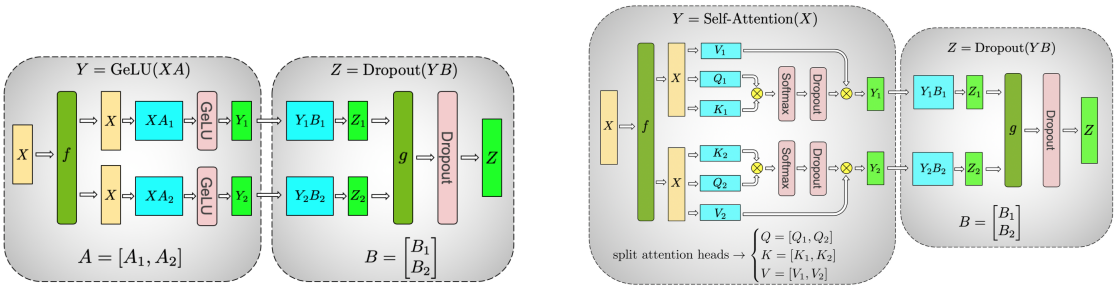
Pipeline Parallelism. Pipeline parallelism partitions model layers across multiple devices, with each device responsible for a subset of consecutive layers. The input batch is split into smaller micro-batches that flow through the pipeline stages sequentially. Two main scheduling approaches exist: GPipe-style scheduling [25] where all forward passes are executed followed by all backward passes, and 1F1B scheduling [39] where forward and backward passes are interleaved. A key challenge is the pipeline bubble - idle time at the start and end of processing each batch when devices wait for activations to propagate through the pipeline. The bubble size is proportional to (p-1)/m where p is the number of pipeline stages and m is the number of micro-batches. An innovative interleaved pipeline schedule can reduce this bubble by assigning multiple chunks of layers to each device, though this comes with increased communication overhead [40]. While effective for scaling very large models, pipeline parallelism in isolation can only scale to the number of layers in the model. Therefore, pipeline parallelism is often combined with other parallelism techniques, such as tensor parallelism, to leverage additional advantages like sub-unit weight sharding.

Tensor Parallelism. First introduced in the Megatron-LM paper [52], tensor parallelism (TP) partitions individual neural network layers across multiple devices to enable training of very large models by dividing operations within each layer. Figure 2.2a and 2.2b illustrate how TP is implemented in transformer-based language models, showing both the matrix partitioning schemes and the required communication patterns.

For transformer-based language models, TP exploits the inherent parallelism in both the multi-layer perceptron (MLP) and self-attention blocks. The approach splits weight matrices along their width and distributes the sharded weights across devices in a way that minimizes communication while maintaining computational efficiency. In the MLP blocks, the weights are partitioned to allow independent application of activation functions across devices. Similarly, for self-attention blocks, the key, query, and value matrices are strategically split to reduce communication overhead. The communication pattern in both blocks is implemented using complementary operators that alternate between identity operations and all-reduce operations in the forward and backward passes.

This partitioning scheme requires only two all-reduce operations in both forward and backward passes for synchronization. However, tensor parallelism necessitates expensive all-reduce communication between devices, making it primarily suitable within a single

node with high-bandwidth interconnects like NVLink [52]. The degree of tensor parallelism is thus typically limited to the number of GPUs within a server node (e.g., 8 for NVIDIA DGX systems).



(A) MLP (B) MHA

FIGURE 2.2: Transformer blocks with tensor parallelism implementation. The operations f and g form a conjugate pair where f performs identity in forward propagation and all-reduce in backward propagation, while g performs the opposite pattern: all-reduce in forward propagation and identity in backward propagation.

Hybrid Parallelism. Hybrid parallelism combines multiple forms of parallelism to overcome the limitations of each individual approach. Common combinations include 3D parallelism (pipeline, tensor, and data parallelism) and 2D parallelism, e.g., tensor parallelism with fully sharded data parallelism or tensor parallelism with pipeline parallelism [40]. For LLMs, tensor parallelism is typically used within a single node to maximize use of high-bandwidth intra-node connections, while pipeline parallelism spans across nodes using cheaper point-to-point communication. The optimal configuration depends on model size, hardware topology, and communication bandwidth. When using FSDP instead of standard data parallelism, memory usage can be further optimized by sharding parameters across data-parallel workers [46]. However, the interaction between different parallelism strategies must be carefully managed - for instance, tensor parallel all-reduce operations must be coordinated with pipeline schedules, and data parallel gradient synchronization must account for both tensor and pipeline parallel communication patterns [40]. The trade-offs include memory efficiency, communication overhead, and computational utilization, which must be balanced based on specific hardware configurations and model architectures.

2.3 Fine-tuning

This section explores the evolution of fine-tuning approaches for language models, beginning with Traditional Fine-tuning (Section 2.3.1), which becomes impractical as models grow larger due to the computational burden of updating billions of parameters. The chapter then introduces Parameter-Efficient Fine-Tuning (PEFT) methods (Section 2.3.2) that solve these limitations by updating only a small subset of parameters, categorizing them into addition-based, reparameterization-based, and selection-based techniques. The discussion concludes with a detailed examination of Low-Rank Adaptation (LoRA) (Section 2.3.3), a breakthrough PEFT method that introduces trainable low-rank decomposition matrices alongside frozen pre-trained weights, offering significant advantages in parameter efficiency, memory usage, inference performance, and task-switching capabilities while maintaining comparable performance to traditional fine-tuning across various model scales.

2.3.1 Traditional Fine-tuning

Traditional fine-tuning involves updating all parameters of a pre-trained model to adapt it to downstream tasks [15]. This approach, while effective, becomes increasingly impractical as models grow in size. For instance, fine-tuning GPT-3 175B requires maintaining separate copies of 175 billion parameters for each task [24], making it computationally expensive and memory-intensive. The storage requirements alone can be prohibitive, as each fine-tuned model requires hundreds of gigabytes of storage space.

2.3.2 Parameter-Efficient Fine-Tuning (PEFT)

Parameter-Efficient Fine-Tuning (PEFT) methods address computational limitations of traditional layer-based fine-tuning by introducing and updating a small subset of parameters [23]. These approaches can be broadly categorized into several types:

Addition-based methods. These techniques introduce new trainable parameters to the model while keeping the original parameters frozen. Examples include adapter layers [48] and soft prompts [58].

Reparameterization-based methods. These approaches reparameterize the weight updates using low-rank decompositions, such as LoRA [24].

Selection-based methods. These methods selectively update only certain parameters of the original model, such as tuning only the biases [6], or using structured pruning approaches [22].

The key advantage of PEFT methods is their ability to achieve performance comparable to full fine-tuning while training orders of magnitude fewer parameters. For instance, on the GLUE benchmark, adapter-based methods can match within 0.4% of full fine-tuning performance while adding only 3.6% parameters per task [23].

2.3.3 LoRA: Low-Rank Adaptation of Large Language Models

Low-Rank Adaptation (LoRA) [24] represents a significant advancement in parameter-efficient fine-tuning. The method is based on the hypothesis that the weight updates during model adaptation have a low "intrinsic rank." Instead of directly updating the model's weights, LoRA introduces low-rank decomposition matrices that are trained alongside the frozen pre-trained weights.

LoRA offers several key advantages. It significantly reduces the number of trainable parameters. For example, when applied to GPT-3 175B, LoRA reduces the number of trained parameters by a factor of 10,000 while maintaining model quality. LoRA also improves memory efficiency. The method reduces GPU memory requirements by up to 2/3 during training since optimizer states are needed only for the smaller matrices. Furthermore, LoRA introduces zero inference overhead. During inference, the low-rank matrices can be merged with the original weights, introducing no additional latency. Finally, LoRA facilitates task switching. Multiple tasks can be efficiently handled by storing different sets of rank decomposition matrices while sharing the same pre-trained model, making it particularly suitable for serving multiple downstream tasks.

The effectiveness of LoRA has been demonstrated across various model scales, from BERT and RoBERTa to GPT-3, showing consistent performance comparable to full fine-tuning while being significantly more parameter-efficient [24].

Chapter 3

Methodology

3.1 Preliminary Techniques

3.1.1 Tensor Parallelism in Megatron-LM

First introduced in the Megatron-LM paper [52], Tensor Parallelism (TP) proposes a way to shard Multihead Attention (MHA) and Multilayer Perceptron (MLP) weights to minimize communication between devices. For instance, the first linear layer in the MLP block partitions its weight matrix along columns, while the second linear projection layer is partitioned along rows (Figure 2.2a). Formally, for an input X and weight matrix A of the first linear layer, the calculation in the first linear layer is represented as:

$$[Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)], \quad A = [A_1, A_2] \tag{3.1}$$

where column-wise sharding of A allows independent General Matrix Multiplication (GEMM) and subsequent GeLU computations, eliminating the need for synchronization. Subsequently, the second linear layer, whose weight matrix is presented as B, takes this activation and performs a row-wise computation:

$$Z = Y_1B_1 + Y_2B_2, \quad B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \tag{3.2}$$

resulting in a single all-reduce synchronization after the computation. Similarly, in the self-attention block, the query (Q), key (K), and value (V) matrices are partitioned column-wise. The computation of each attention head or group of attention heads (in case the number of attention heads is a multiple of the TP sharding factor) occurs locally within a single GPU without intermediate synchronization. The subsequent output linear projection is then partitioned row-wise, taking outputs directly from the parallel attention heads, thereby further reducing communication (Figure 2.2b).

Consequently, the only synchronization required within each transformer block is an all-reduce operation at the end of each MHA or MLP block during the forward pass and at the beginning during the backward pass, simplifying communication to two all-reduce operations per block per training iteration.

3.1.2 LoRA Adapters

Each pair of LoRA [24] decomposition matrices, or *adapters*, consists of two projection matrices, namely A and B. Matrix $A \in \mathbb{R}^{d \times r}$ projects the input from the hidden dimension d to a lower intrinsic dimension r, and matrix $B \in \mathbb{R}^{r \times d}$ projects it back to the original

dimension. Let $\mathbf{x} \in \mathbb{R}^d$ denote an input hidden state vector, $\mathbf{W} \in \mathbb{R}^{d \times d}$ a pretrained weight matrix, and $\mathbf{y} \in \mathbb{R}^d$ the output logit. Then, each modified linear transformation is formulated as:

$$\mathbf{y} = \mathbf{xW} + \mathbf{xAB}. \tag{3.3}$$

Here, $\mathbf{W}$ remains frozen during training, while $\mathbf{A}$ and $\mathbf{B}$ are trainable and adapt to new data, functioning as modular plugins for fine-tuning on downstream tasks. The paradigm of LoRA is illustrated in Figure 3.1.

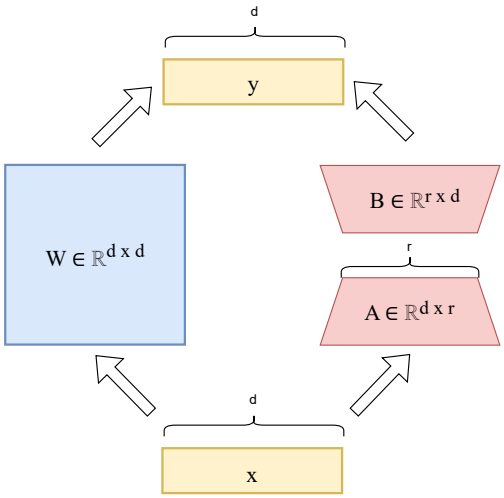


FIGURE 3.1: LoRA reparameterization.

3.1.3 Combining FSDP with Tensor Parallelism

Figure 3.2 illustrates the parameter distribution when combining FSDP with TP.

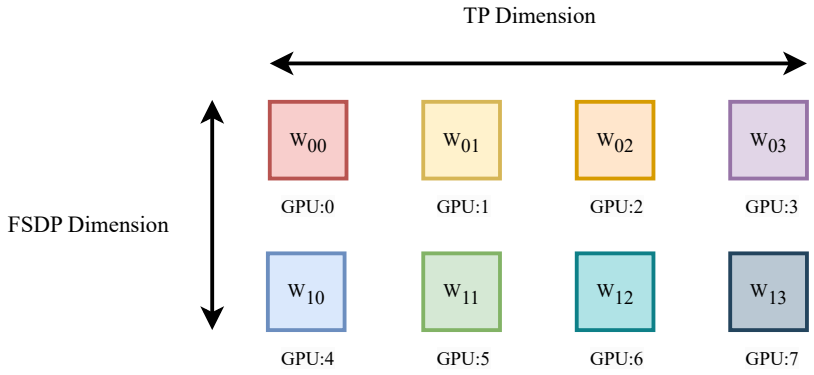


FIGURE 3.2: The sharding paradigm for weight W when combining FSDP and TP. Each weight is first sharded across the TP dimension and subsequently across the FSDP dimension. Examples of sharded weight shapes are given in Appendix A.2.

Here, each W_{ij} is a 2D-sharded weight that is sharded along the TP dimension and FSDP dimension. To perform the forward or backward pass, each GPU must contain the TP-sharded weight (i.e., a 1D-sharded weight) of the corresponding layer in use — for example, the Q-projection linear layer's weight in the 30th decoder layer of the LLaMA 3.1 70B model. This means that each device has to gather the weight from other devices along the FSDP dimension.

In the example shown in Figure 3.2, device `GPU:0` will gather the TP-sharded weight from `GPU:1`, and vice versa, to reconstruct the full 1D-sharded weight, preparing for forward/backward computations.

For convenience, we refer to each FSDP group as a DP group. During the forward and backward passes, each device fetches the TP-sharded weights only along the FSDP dimension, resulting in TP-sharded weights being available on each GPU. Together, this paradigm enables distributed computation of multiple micro-batches across DP groups while simultaneously parallelizing the computation of intermediate logits of each micro-batch across devices within each DP group (i.e., having the same DP rank but different TP ranks).

3.2 Proposed Tensor Parallelism Paradigm for LoRA Modules

Figure 3.3 illustrates our proposed tensor parallelism (TP) sharding paradigm for LoRA adapters alongside the original weight matrix $W \in \mathrm{R}^{d\times d}$. Depending on the sharding strategy of W—either column-wise or row-wise—the corresponding LoRA adapters adopt a consistent sharding strategy. This design ensures that the output logits produced by the LoRA adapters have the same shape as those produced by the pretrained matrix W, so the subsequent addition operation can be applied.

For a column-wise sharding strategy, W is split along its output (column) dimension, for example, into two shards: $W_1 \in \mathrm{R}^{d\times \frac{d}{2}}$ and $W_2 \in \mathrm{R}^{d\times \frac{d}{2}}$, such that $W = [W_1, W_2]$. To match this, the matrix $B \in \mathrm{R}^{r\times d}$ is likewise sharded column-wise as $B = [B_1, B_2]$, with $B_1, B_2 \in \mathrm{R}^{r\times \frac{d}{2}}$. Matrix $A \in \mathrm{R}^{d\times r}$ is **replicated** across devices to ensure a valid matrix multiplication operation with the sharded matrix B.

Let $X \in \mathrm{R}^{b\times s\times d}$ be a batch of input hidden states with b and s are batch size and sequence length, respectively. Then the output from each TP shard is computed as:

$$\begin{aligned} Y = [Y_1, Y_2] &= [\mathrm{GeLU}(XW_1 + XAB_1),\ \mathrm{GeLU}(XW_2 + XAB_2)]\,, \\ W = [W_1, W_2], &\quad B = [B_1, B_2]. \end{aligned} \tag{3.4}$$

Here, $Y_1, Y_2 \in \mathrm{R}^{b\times s\times \frac{d}{2}}$ are the outputs computed independently on each TP shard. The final output Y is the concatenation of Y_1 and Y_2 across the feature (column) dimension. This design enables parallelism over the output dimension d, balancing computation and memory while preserving the semantics of LoRA-enhanced linear transformations.

The computation for the row-wise strategy follows inherently the same logic, but in a row-wise manner, with the sharding applied to adapter A. The calculation is formulated in equation 3.5.

$$\begin{aligned} \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} &= \begin{bmatrix} \mathrm{GeLU}(X_1W_1 + X_1A_1B) \\ \mathrm{GeLU}(X_2W_2 + X_2A_2B) \end{bmatrix}, \\ W = \begin{bmatrix} W_1 \\ W_2 \end{bmatrix}, &\quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \end{aligned} \tag{3.5}$$

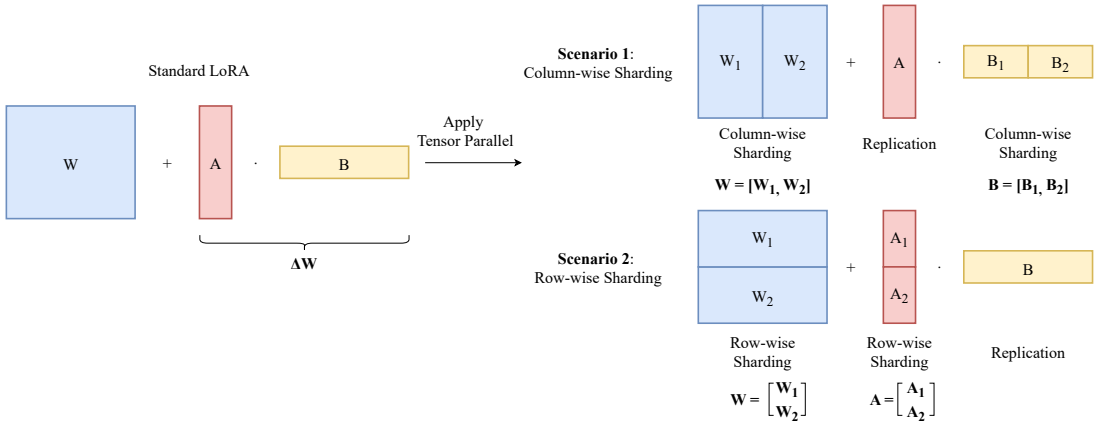


FIGURE 3.3: TP sharding paradigm for LoRA adapters.

Notably, in both of these scenarios, only one of the two adapters in each pair is sharded, while the other is replicated across devices.

3.3 Proposed Efficient Model Loading Method for FSDP-TP Sharded Models

In practice, initializing a model sharded with both TP (Tensor Parallelism) and FSDP (Fully Sharded Data Parallelism) is challenging due to redundant weight loading, which leads to excessive memory usage and slow initialization. Specifically, the standard loading approach results in each GPU redundantly loading the entire model, severely limiting scalability and efficiency. This issue arises because each GPU cannot independently identify and load only the specific subset of weights it requires, ultimately resulting in GPU memory overload—or even CPU memory overload if CPU offloading is employed—and causing bottlenecks in data transfer between CPU and GPU. A detailed discussion of this issue, including examples and quantification of memory overhead, is provided in Appendix A.3. To address this, we propose a straightforward and efficient loading method illustrated in Figure 3.4.

Initially, the pretrained model weights—either from a singleton model or a LoRA-infused model—are loaded onto the CPU.

Next, the correct portion of each weight matrix is identified by inspecting the shape of the local tensor corresponding to that TP rank. This step highlights the distribution of the FSDP-TP model. TP first shards each weight tensor along a specific dimension, i.e. colum-wise or row-wise. Subsequently, FSDP further shards this TP-sharded tensor across devices in the same DP group, i.e. devices that have the same TP rank but different DP rank (the number of devices in one DP group equals the DP dimension size, which is 4 in the example in Figure 3.4). Examples of the 2D-sharded weight shapes for both row-wise and column-wise manner are presented at A.2

However, prior to the stable release of FSDP2 (which remains under development), FSDP treats parameters as immutable and uninterpretable *FlatParameter* objects. Consequently, each device must explicitly retrieve the original weights from *FlatParameter* to modify the underlying tensor data (in this case, overwrite it with trained weights). It is important to note that the weight retrieved at each device remains a TP-sharded weight corresponding to the TP rank of the device (i.e., the size of the retrieved weight equals

the size of the full weight divided by the TP dimension size). This operation requires a temporary allocation of CPU memory equal to the size of the retrieved weights. However, as every device must perform this operation, it leads to potential CPU Out-of-memory (OOM) errors.

Specifically, since each device has to retrieve its corresponding TP-sharded weight, the total amount of CPU memory allocated equals to N times the full model size, where N is the FSDP dimension size, easily leading to CPU OOM error. A detailed description of this operation and calculation of the total amount of CPU memory allocated is present at A.4. Most hardware configurations lack sufficient CPU memory for this scenario. For instance, in our development environment, the maximum shared CPU memory on a single node with 8 GPUs is 480 GB, which is insufficient to store four LLaMA 3.1 70B models—requiring approximately 560 GB in total—assuming the model parameters are in bfloat16 format.

To address this issue, we propose a method wherein CPU weights are loaded exclusively to DP rank-0 devices (e.g., GPU:0 and GPU:1 in the example in Figure 3.4) and subsequently broadcasted to other devices within the same DP group (i.e., vertically as shown in Figure 3.4). This strategy provides two main benefits: (1) limiting maximum CPU memory consumption to the size of a single full model (e.g., 140GB for the LLaMA 3.1 70B model in bfloat16), regardless of the DP dimension size, and (2) significantly speeding up the loading process through GPU-based broadcast operations, which offer substantially higher bandwidth compared to CPU-GPU communication.

Here is an example of how the process of loading a weight W works. The weight W is first sharded by the factor of TP size, e.g., into W_0 and W_1, and loaded accordingly to GPU:0 and GPU:1, whose DP ranks are 0. GPU:0, whose TP rank is 0, proceeds to slice W_0 by a factor of DP size and distribute those slices (using broadcast operations) exclusively to other devices within the same DP group, i.e., GPU:2, GPU:3, and GPU:4. The same behavior is performed in DP Group 1, i.e., GPU:1, GPU:3, GPU:5, and GPU:6. As a result, the amount of data transferred through CPU-GPU communication is exactly the model size, compared to N times the model size in the standard loading mechanism. The rest of the communication is inter-GPU, which takes almost no time due to their extremely fast bandwidth, e.g., 200 GB/s unidirectional on our AMD MI250X.

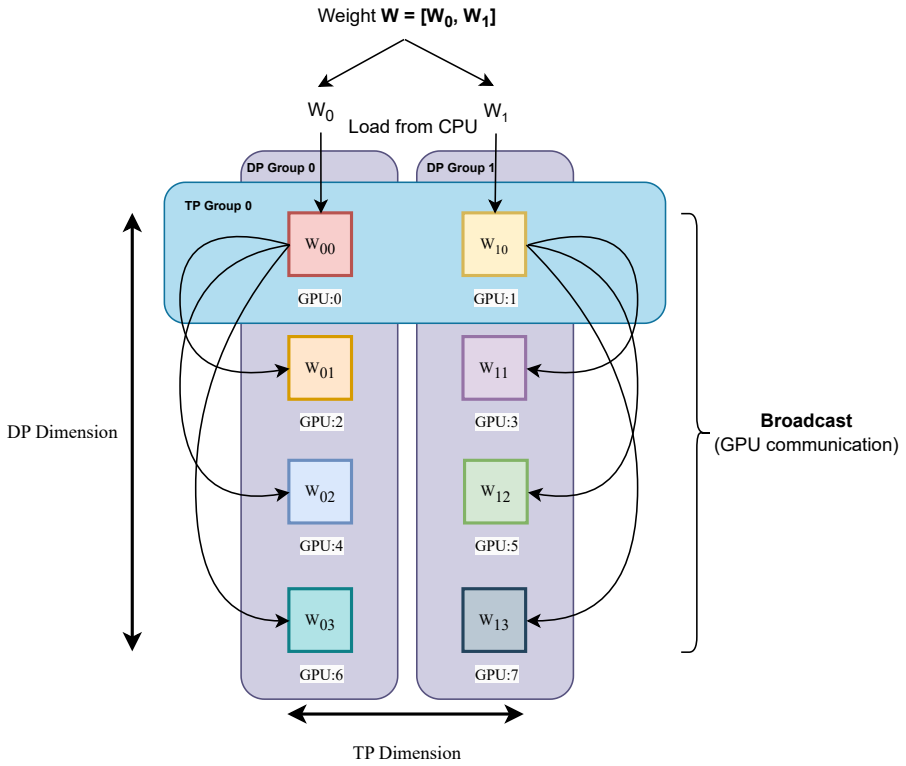


FIGURE 3.4: Efficient mechanism for loading model weights onto devices. In this example, the TP dimension size is 2, and the DP dimension size is 4.

Chapter 4

Experimental Results

4.1 Experiment Environment

All of our experiments are run on CSC's LUMI supercomputer using up to 4 GPU nodes. Each node features a single 64-core AMD EPYC 7A53 "Trento" CPU and four AMD Instinct MI250X GPUs. Each MI250X GPU is a multi-chip module containing two Graphics Compute Dies (GCDs), effectively presenting eight logical GPUs per node, with each logical GPU having 64 GB of memory. The maximum CPU memory that can be requested by users is 480 GB per node.

The model used in our experiments is LLaMA 3.1 70B in bfloat16 format, approximately 140 GB in size. Each experiment run has exclusive access to the requested resources, without any interference or overhead from other jobs.

The batch size in each run is determined by the minimum batch size of each configuration. For example, running FSDP-only on 8 GPUs requires the minimum batch size of 8, with each GPU processing one sample. Running TP-only on 8 GPUs enable batch size of 1 as it can distribute the computation of a single sample; therefore, the chosen batch size for this configuration will be 1. For hybrid approach configurations, the minimum batch size equals the FSDP dimension size. For the loss function, we choose cross entropy loss on the full input sequence. The metrics are measured by aggregating performance over four training epochs, which take approximately 1–2 hours depending on the number of nodes and the dataset size.

The maximum trainable sequence length was obtained through trial and error with a step of 100 tokens until reaching OOM error.

4.2 Single Node Performance

4.2.1 LoRA Rank Scaling

Our experiments reveal significant differences in maximum trainable sequence length (MTSL) across different parallelism strategies (Figure 4.1). For annotation, we denote each setting in the format *FSDP-TP (x, y)*, where x is the dimension size of FSDP (or DP for short), and y is that of TP. For example, FSDP-TP (8, 1) refers to an FSDP-only setup, as the TP dimension size is 1—meaning no TP sharding is applied. Similarly, FSDP-TP (4, 2) means an FSDP factor of 4 and a TP factor of 2. The exact MTSL at different LoRA ranks for each configuration is given in Table B.4.

As shown in the figure, FSDP-only setting FSDP-only (i.e., DP size = 8) exhibits the most limited performance, with a maximum trainable length of approximately 11,500 at

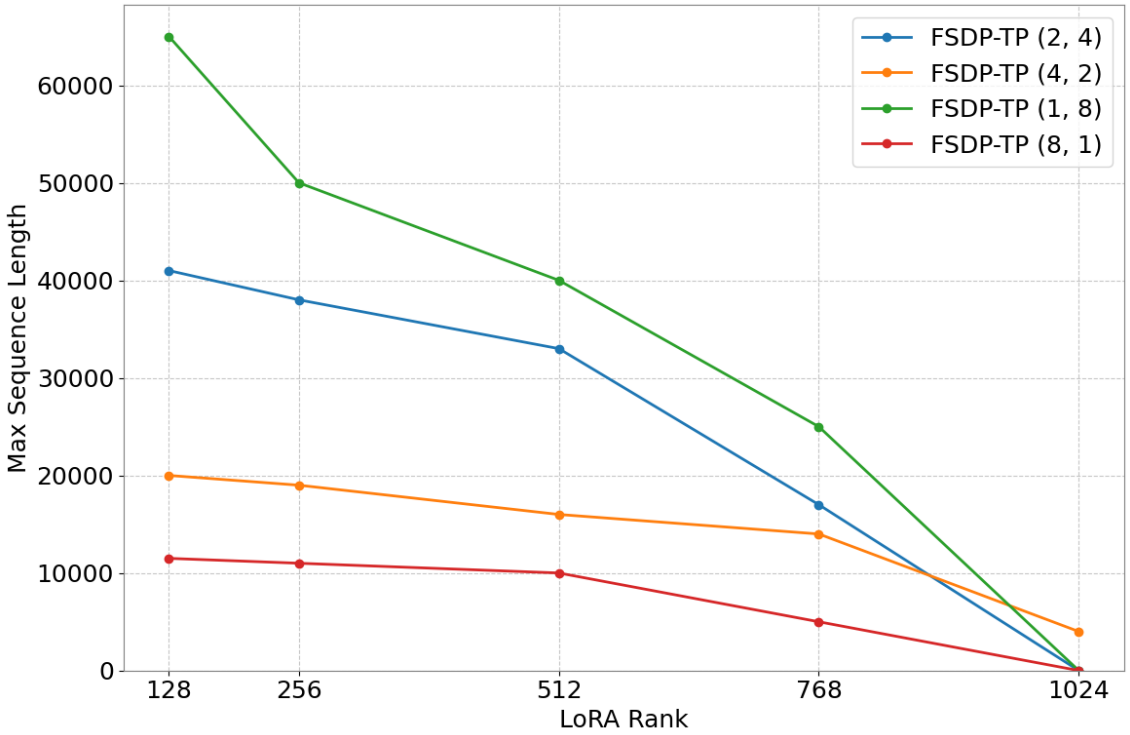


FIGURE 4.1: Comparison of Maximum Trainable Sequence Length Across Configurations

LoRA rank = 128. In contrast, TP-only (i.e., TP size = 8) consistently demonstrates the longest trainable sequence length across almost all LoRA rank values.

Hybrid approaches combining FSDP and TP show intermediate performance between the two pure strategies. Configurations with larger FSDP dimension sizes (i.e., DP size = 4) behave more similarly to FSDP-only, while those with larger TP dimension sizes (i.e., TP size = 4) exhibit characteristics closer to TP-only. At high LoRA ranks, where the percentage of trainable parameters approaches 25% of the original model size, three out of four configurations encounter out-of-memory (OOM) errors due to GPU memory constraints.

The detailed explanation for the difference in MTSL between parallelism configurations is further discussed in section 5.1. However, in exchange for the extended trainable sequence length, TP introduces a significantly lower throughput compared to FSDP, which is shown in the next section.

4.2.2 Throughput (Processed Tokens/Second)

Table B.1 presents the maximum total number of trainable tokens per node for different parallelism configurations. This number is calculated as the batch size multiplied by the sequence length. Note that it differs from the MTSL mentioned above, as MTSL measures the maximum trainable sequence length at the minimal batch size of each configuration. In the following experiments, we employed weak scaling of sequence length and batch size, i.e., maintaining roughly this maximum total token count per node for every configuration. Specifically, we fixed sequence length and varied batch size.

Figure 4.2 demonstrates the superior performance of FSDP over TP in throughput, measured by the number of tokens processed per second at a fixed sequence length and

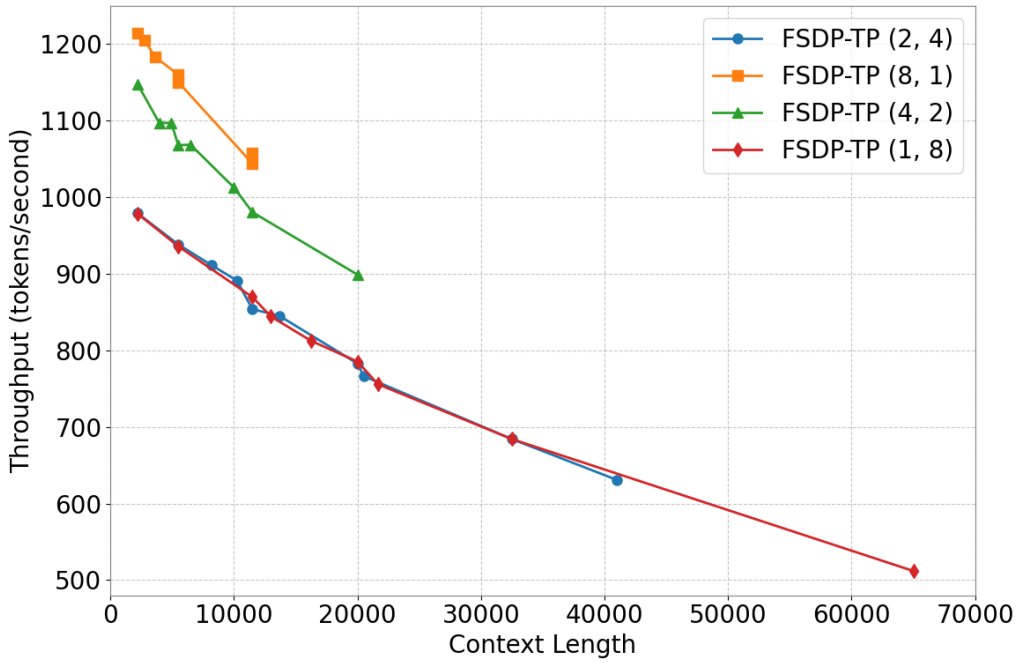


FIGURE 4.2: Efficient mechanism for loading model weights onto devices. In this example, the TP dimension size is 2, and the DP dimension size is 4.

maximum trainable batch size. Another minor observation is the throughput drop over time, with FSDP exhibiting more significant decreases compared to TP. These two observations are further discussed in section 5.1. The detailed experimental values corresponding to figure 4.2 is given in table B.5

4.2.3 Processing Time per Sample (PTS)

Another metric of interest is Processing Time per Sample (PTS), calculated as Processing time / batch size. As illustrated in Figure 4.3, we observe a consistent increase in PTS across all FSDP-TP configurations when scaling up sequence length and scaling down batch size by the same factor. This upward trend in PTS directly correlates with the previously observed drop in throughput, confirming that longer sequences require more processing time per sample even when maintaining constant total token count. The detailed values corresponding to this figure are presented in Table B.6.

4.3 Multi-node Scaling

In our multi-node scaling experiments, we employed weak scaling by fixing sequence lengths to 10,000 and 20,000 while maximizing batch size. This approach is referred to as weak scaling, ensuring that the total tokens processed on each node is unchanged and matched the maximum trainable token count identified for each parallelism configuration in the single-node experiments. Additionally, we used only FSDP for multi-node scaling; for example, FSDP-TP (4,2) on a single node becomes FSDP-TP (8,2) and FSDP-TP (12,2) in 2-node and 3-node training, respectively. For simplicity of annotation, we always refer to the single-node configuration; for example, we use FSDP-TP (4,2) instead of FSDP-TP (4×#nodes, 2) in multi-node training. We do not scale the TP dimension, as TP introduces more communication overhead compared to FSDP. This makes it unsuitable for

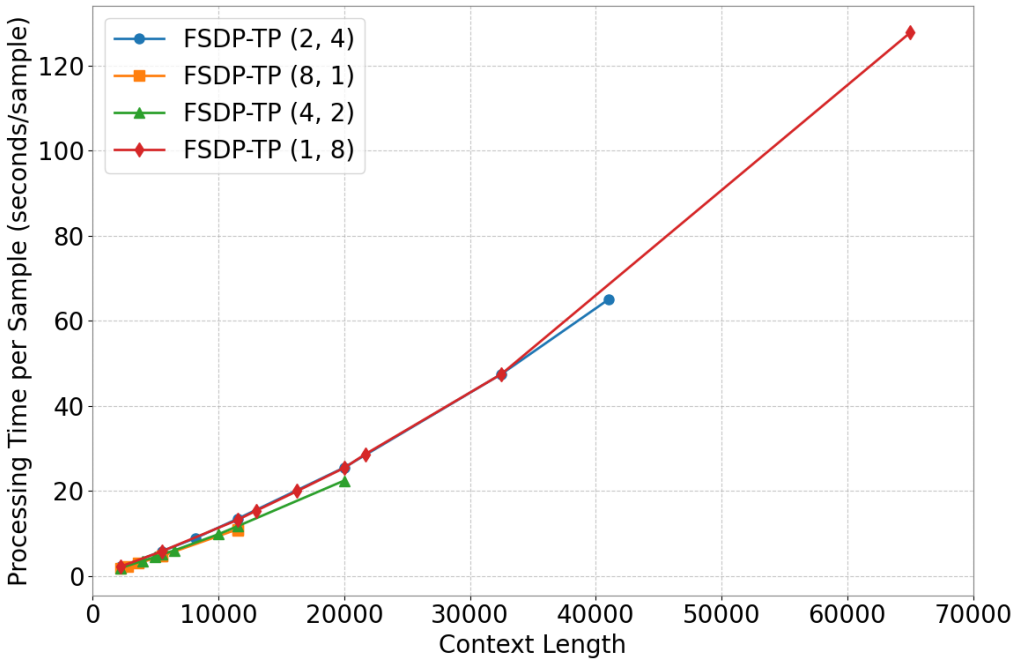


FIGURE 4.3: Processing Time per Sample (PTS) vs Context Length Across Different Parallelism Settings

scaling beyond a single node, especially over low-bandwidth communication channels such as inter-node InfiniBand, as noted in the Megatron-LM paper [52].

4.3.1 Throughput and PTS at Sequence Length = 10,000

Figure 4.4 illustrates throughput scaling when adding more nodes. Table 4.1 presents the exact throughput of different parallelism configurations at varying numbers of nodes.

TABLE 4.1: Throughput (tokens/s) vs. Number of Nodes for Different Configurations at Sequence Length of 10,000.

# Nodes	FSDP-TP (8, 1)	FSDP-TP (4, 2)	FSDP-TP (2, 4)	FSDP-TP (1, 8)
1	1057.00	1016.52	877.19	886.26
2	889.00	1415.93	1301.87	914.63
3	1350.00	2108.96	1900.24	1374.05
4	1758.00	2775.37	2496.10	1803.16

Surprisingly, although FSDP-only demonstrates the highest throughput in single-node training, FSDP-TP configurations show clearly superior scaling in multi-node scenarios.

All three configurations exhibit nearly linear scaling, with the exception of a minor performance drop at node count = 2 for FSDP-only (or FSDP-TP (8, 1) as shown in the figure). While we cannot fully explain this performance degradation in dual-node PyTorch FSDP training and attribute it to practical implementation details, the overall results demonstrate strong scaling capabilities for both FSDP and FSDP-TP. At this sequence length range, FSDP-TP (4,2) consistently outperforms FSDP-TP (2,4), suggesting it represents the optimal configuration for this case. Further discussion and explanation is presented in section 5.1

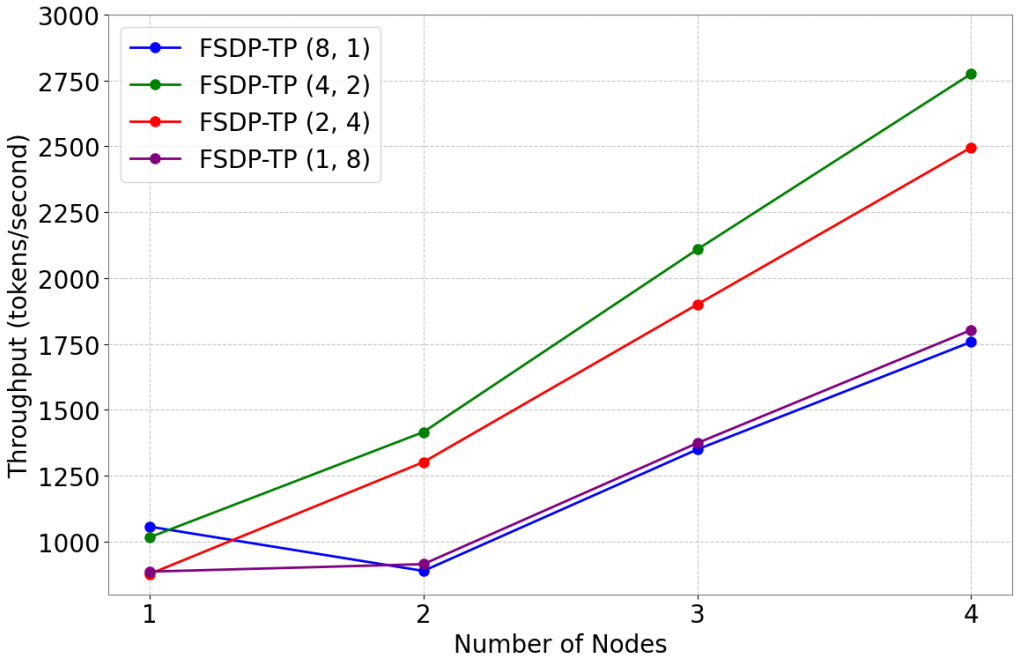


FIGURE 4.4: Throughput of different parallelism configurations when scaling to multi-node scaling at sequence length = 10,000.

Table 4.2 quantifies the throughput improvements of hybrid configurations compared to FSDP-only.

Copy

TABLE 4.2: Percentage higher throughput of FSDP-TP (8,1) compared to hybrid FSDP-TP approaches across device scales.

# Nodes	FSDP-TP (4,2) to FSDP-TP (8,1) (% higher)	FSDP-TP (2,4) to FSDP-TP (8,1) (% higher)	FSDP-TP (1,8) to FSDP-TP (8,1) (% higher)
1	-3.83	-17.01	-16.15
2	59.27	46.44	2.88
3	56.22	40.76	1.78
4	57.87	41.99	2.57

PTS As a direct consequence of improved throughput scaling, FSDP-TP (4,2) and FSDP (2,4) demonstrate significant PTS improvements compared to FSDP-only, while FSDP-TP (1,8) shows only marginal improvement due to the high TP dimension, which results in increased communication overhead—even with high-bandwidth intra-node communication. Figure 4.5 illustrates the multi-node scaling of PTS of different parallelism configurations. Table 4.3 shows the exact PTS of each configuration and table 4.4 shows the detailed reduction in PTS (in percentage) of 3 hybrid approaches configurations compared to FSDP-only (i.e. FSDP-TP (8,1)).

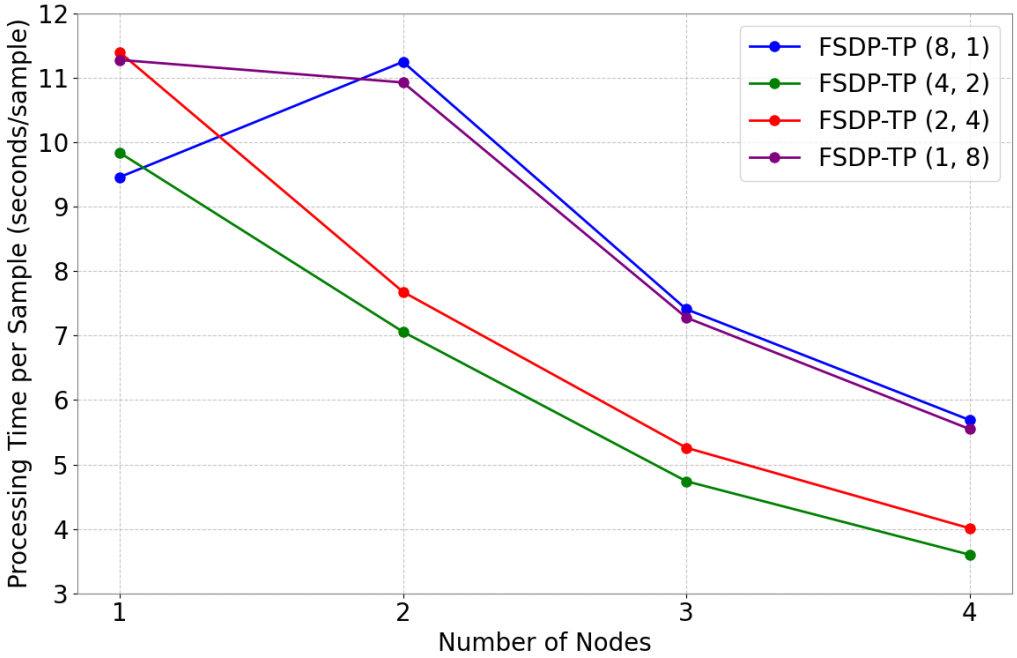


FIGURE 4.5: PTS of different parallelism configurations when scaling to multi-node scaling at sequence length = 10,000.

TABLE 4.3: Processing Time per Sample (PTS) (secs/sample) vs. Number of Nodes for Different Configurations at Sequence Length of 10,000.

# Nodes	FSDP-TP (8, 1)	FSDP-TP (4, 2)	FSDP-TP (2, 4)	FSDP-TP (1, 8)
1	9.46	9.84	11.40	11.28
2	11.25	7.06	7.68	10.93
3	7.41	4.74	5.26	7.28
4	5.69	3.60	4.01	5.55

4.3.2 Throughput and PTS at Sequence Length = 20,000

At this sequence length, FSDP-only training no longer works due to an out-of-memory (OOM) error; therefore, we can only compare the three hybrid approaches. The three approaches continue to scale linearly. Notably, for sequence lengths less than or equal to 20,000 (i.e., the MTSL of FSDP-TP (4,2), according to table B.4), FSDP-TP (4,2) represents the optimal configuration. However, for longer sequences, FSDP-TP (2,4) and FSDP-TP (1,8) become the preferred options.

Throughput Improvement: Table 4.6 and figure 4.6 show the improvement in throughput of FSDP-TP (4,2) and FSDP-TP (2,4) over FSDP-TP (1,8). The table also demonstrates the consistent outperformance of FSDP-TP (4,2) when scaling to multiple nodes at a sequence length of 20,000. Detailed experimental records corresponding to Figure 4.6 are given in Table 4.5.

Processing Time per Sample (PTS) Reduction: Consequently, we also observe an outperformance of FSDP-TP (4,2) over FSDP-TP (2,4) and FSDP-TP (1,8) in terms of

TABLE 4.4: Percentage reduction in pretraining time per step (PTS) of hybrid approaches compared to FSDP-only across device scales.

# Nodes	FSDP-TP (4,2) Reduction (%)	FSDP-TP (2,4) Reduction (%)	FSDP-TP (1,8) Reduction (%)
1	-4.02	-20.51	-19.24
2	37.24	31.73	2.84
3	36.03	29.01	1.75
4	36.73	29.53	2.46

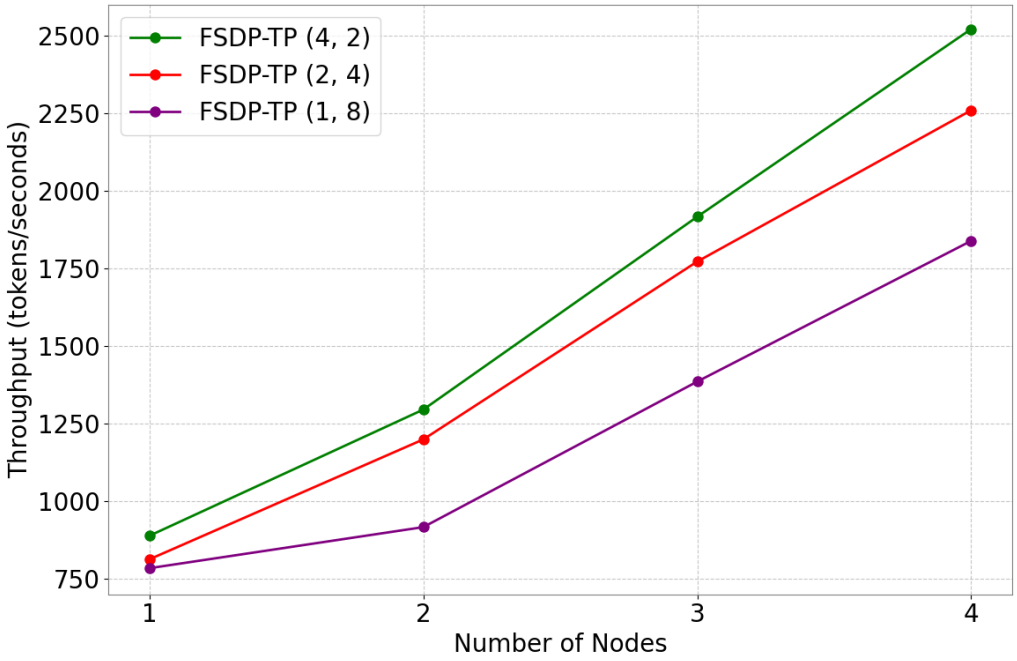


FIGURE 4.6: Throughput of different parallelism configurations when scaling to multi-node scaling at sequence length = 20,000.

PTS. Table 4.8 and Figure 4.7 show the detailed reduction in PTS of FSDP-TP (4,2) compared to FSDP-TP (2,4). Detailed experimental records corresponding to Figure 4.7 are given in Table 4.7.

4.4 Efficient Model Loading Performance

To benchmark the efficiency of our proposed efficient model loading method, we choose the baseline as the CPU-GPU communication-based method, which requires each GPU to load the full weight and only keep its shard of that weight while discarding the rest. Note that for both the baseline and our method, the weight matrices are loaded consecutively, one-by-one, with no multiple weight matrices loaded at the same time.

Since it is impossible to benchmark the baseline method with the Llama 3.1 70B model as it encounters CPU OOM errors, we reduce the original Llama 3.1 70B model, which has 70 billion parameters, to approximately 50 billion parameters by reducing the number of decoder layers. As a result, for FSDP-TP (4,2), the baseline method requires approximately 400 GB of CPU memory (i.e., 4 times the model size) and takes 9 minutes to complete model loading. Our proposed method requires only 100 GB of CPU memory (i.e., exactly 1

TABLE 4.5: Throughput (tokens/s) vs. Number of Nodes for Different Configurations at Sequence Length of 20,000.

# Nodes	FSDP-TP (4, 2)	FSDP-TP (2, 4)	FSDP-TP (1, 8)
1	888.89	813.01	784.31
2	1295.55	1199.40	916.73
3	1916.93	1772.53	1385.68
4	2521.67	2259.89	1839.08

TABLE 4.6: Percentage improvement in throughput of FSDP-TP (4,2) and FSDP-TP (2,4) over FSDP-TP (1,8) across different device counts.

# Nodes	FSDP-TP (4,2) over (1,8) (% improvement)	FSDP-TP (2,4) over (1,8) (% improvement)
1	11.76	3.53
2	29.23	23.54
3	27.74	21.85
4	27.08	18.62

copy of the model size) and takes 4 times less time (i.e., 2 minutes 20 seconds) to complete loading due to reduced data transfer through CPU-GPU communication while moving the rest of the weight transfer to GPU-GPU communication, which takes almost no time thanks to the extremely high bandwidth.

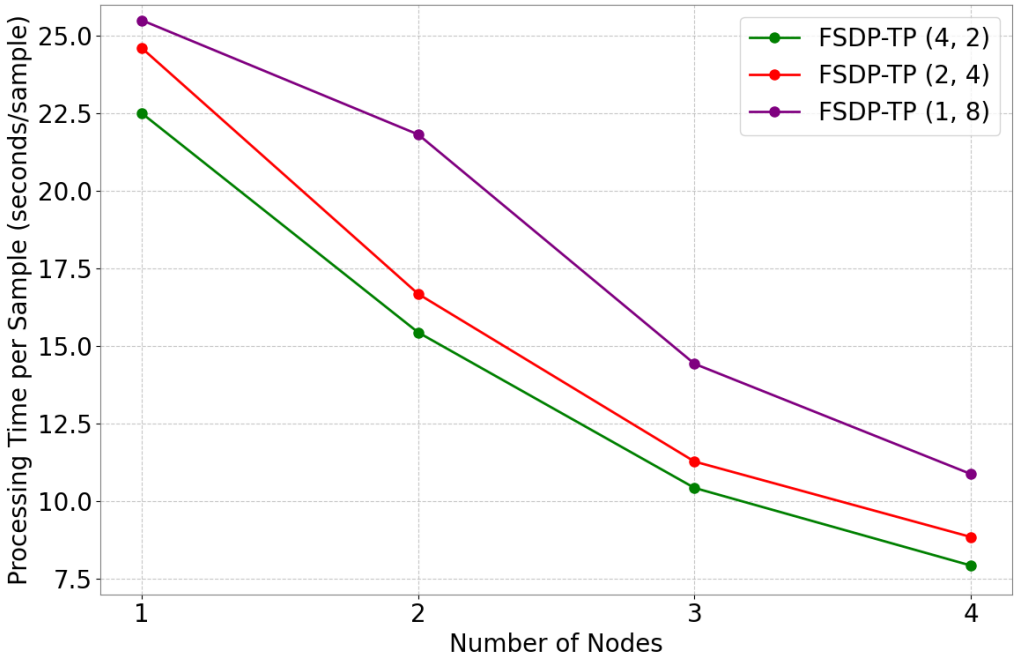


FIGURE 4.7: PTS of different parallelism configurations when scaling to multi-node scaling at sequence length = 20,000.

TABLE 4.7: Processing Time per Sample (PTS) (secs/sample) vs. Number of Nodes for Different Configurations at Sequence Length of 20,000.

# Nodes	FSDP-TP (4, 2)	FSDP-TP (2, 4)	FSDP-TP (1, 8)
1	22.50	24.60	25.50
2	15.44	16.68	21.82
3	10.43	11.28	14.43
4	7.93	8.85	10.88

TABLE 4.8: Percentage reduction in pretraining time per step (PTS) of FSDP-TP (4,2) and FSDP-TP (2,4) compared to FSDP-TP (1,8) across different node counts.

# Nodes	FSDP-TP (4,2) PTS Reduction (%)	FSDP-TP (2,4) PTS Reduction (%)
1	11.76	3.53
2	29.23	23.54
3	27.74	21.85
4	27.08	18.62

Chapter 5

Discussion

In this chapter, we dive deeper into the interpretation and explanation for the observations in the experiment results. This chapter consists of two parts: Single-node Performance and Multi-node Scaling. In the first part, we discuss the differences in performance across various parallelism configurations in the single-node setting, focusing on the key predefined metrics: MTSL, throughput, and PTS. This provides insight into the key advantages and disadvantages of the two parallelism techniques and their combinations. The second part expands the knowledge and insights into multi-node settings where inter-node communication introduces new artifacts into the performance characteristics of the two parallelism techniques. By analyzing the key design differences, supported by the experimental results, we gain insights into multi-node scaling and how the combination of two parallelism techniques offers greater performance benefits from both approaches.

5.1 Single node Performance

TP enables longer MTSL. The difference in MTSL can be explained by examining the fundamental characteristics of each parallelism strategy. FSDP, at its core, is a Data Parallel (DP) method that requires each rank to process a complete input. This can trigger OOM errors since the peak memory consumption during computation of a single multi-head attention (MHA) or multi-layer perceptron (MLP) block remains unchanged. Expressions 2.1 and 2.2 show the memory usage of these 2 building blocks.

FSDP distributes layers evenly across devices but does not distribute the computation of individual layers, resulting in fixed memory consumption within each MHA/MLP computation. The memory savings come solely from the distribution of parameters, optimizer states, and gradients, not activations.

On the other hand, TP shards individual weights, effectively distributing computation and reducing memory consumption for intermediate logits. According to the theoretical model, if memory pressure primarily stems from MHA and MLP computations, we would expect TP to support longer sequences by a factor proportional to the TP factor. Our experiments confirm this, with TP-only demonstrating approximately 5 times longer trainable sequences compared to FSDP-only at LoRA ranks of 128, 256, 512, and 768 on a full 8-GPU single node. This highlights TP's advantage over FSDP in extending trainable sequence length.

TP demonstrates lower throughput and higher PTS. According to TP literature [52], TP requires more communication compared to FSDP due to two synchronization points at the beginning and end of each MHA/MLP block. The communication tensor

has a shape of (B, L, H) and cannot be overlapped with computation. In contrast, FSDP can perform all-gather operations for the next parameter group (i.e., FSDP unit or FSDP group) during the forward or backward computation of a previous group. This reduces idle time and consequently increases throughput and decreases PTS.

Throughput drop. Since the attention mechanism has a computational complexity of $O(L^2 \times \text{hidden\_dim})$, throughput decreases in both TP and FSDP when increasing L (sequence length) and decreasing B (batch size) by the same factor. However, TP demonstrates less severe relative drops compared to FSDP. This is mathematically proven in B.6.

In summary, although TP enables longer context training, it supports a smaller maximum total number of trainable tokens per step and exhibits notably lower throughput and higher PTS compared to FSDP-only and FSDP-TP (4,2) hybrid approach, which have higher FSDP dimension sizes than TP sizes, as shown in table B.2 and table B.3.

5.2 Multi-node Scaling

In our experimental setup, weights are always sharded across all devices, resulting in smaller local parameter sets on each GPU. In FSDP-only configurations, scaling to high factors may create scenarios where computation outpaces communication despite operation overlapping (facilitated by weight pre-fetching). This causes GPUs to idle while waiting for parameters.

In hybrid approaches, although weights are sharded by the same factor as in FSDP-only, a portion is sharded horizontally across devices within the same DP group. Consider a model with 8 decoder blocks, representing 8 FSDP units. With FSDP-only at a factor of 8, each device stores a single block. However, with FSDP-TP (4,2), each DP group contains 2 devices, with each group storing $8/4 = 2$ blocks. Within each TP group, parameters are sharded horizontally rather than at block boundaries, resulting in 2 sub-blocks per device. This paradigm excels in multi-node training as it reduces weight transfers between TP groups, which can locate on different nodes. Instead, part of the communication occurs between devices within a TP group, which benefit from much higher bandwidth thanks to intra-node communication.

In summary, although TP requires more communication than FSDP-only, when scaling to multiple nodes, inter-node bandwidth limitations severely impact FSDP communication while TP communication remains intra-node. This makes hybrid approaches the optimal configurations in multi-node training. Concretely, the optimal configuration is determined by the maximum sequence length of the training data. Each configuration has its own MTSL, and based on that insight and the actual sequence length of our training data, we can decide the optimal configuration. Table B.4 shows the MTSL of each configuration at different values of LoRA ranks. Based on this, the optimal parallelism configuration for a case will be the one with the shortest MTSL that is still longer than the longest sequence in our training data.

5.3 Limitations and Future Outlooks

This study has several important limitations that should be acknowledged. First, our experiments are constrained to the Llama 3 model family architecture, and while we believe the findings generalize to models with similar architectures, this has not been empirically validated. Additionally, from an implementation standpoint, our training runs were conducted using PyTorch 2.5 before a stable FSDP2 release was available. This means that

absolute performance metrics may differ when FSDP2 is eventually deployed, though the theoretical foundations and clear performance distinctions between FSDP and TP in metrics such as throughput and MTSL remain valid for guiding optimal configuration selection.

Hardware constraints also present limitations to our findings. Our experiments were conducted exclusively on AMD GPUs using ROCm rather than NVIDIA's CUDA ecosystem. Given PyTorch's limited support for AMD GPUs, complete experimental reproducibility cannot be guaranteed across different hardware environments. Nevertheless, the relative performance trends and overall patterns observed should remain consistent regardless of the specific accelerator architecture.

Despite these limitations, our comprehensive analysis of FSDP-TP hybrid approaches for LoRA-infused model fine-tuning on single-digit node configurations provides a solid foundation for future research directions. To conclude, this study has deeply analyzed the practical performance of different parallelism configurations of FSDP and TP for LoRA-infused model fine-tuning on a limited single-digit number of nodes. To extend to an even near-infinite context length, future studies on the combination of FSDP-TP hybrid approaches with methods such as sequence parallelism [34] or, more recently, ring attention with the Blockwise Transformer [36], would be a promising starting point to enable long-context fine-tuning with LoRA. On the other hand, to increase the trainable batch size while localizing FSDP weight exchanges–especially critical for very large models–pipeline parallelism techniques [25, 39, 40] are also a worthwhile extension to the current FSDP-TP schema. Correctly defined metrics are key to incorporating new training methods critically.

Bibliography

[1] Armen Aghajanyan, Akshat Shrivastava, Anchit Gupta, Naman Goyal, Luke Zettlemoyer, and S. Gupta. "Better Fine-Tuning by Reducing Representational Collapse". In: *ArXiv* abs/2008.03156 (2020). URL: https://api.semanticscholar.org/CorpusID:221083147.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 4895–4901. DOI: 10.18653/v1/2023.emnlp-main.298. URL: https://aclanthology.org/2023.emnlp-main.298.

[3] Jean-Baptiste Alayrac et al. "Flamingo: a Visual Language Model for Few-Shot Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 23716–23736. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf.

[4] Silas Alberti, Niclas Dern, Laura Thesing, and Gitta Kutyniok. "Sumformer: Universal Approximation for Efficient Transformers". In: *Proceedings of 2nd Annual Workshop on Topology, Algebra, and Geometry in Machine Learning (TAG-ML)*. Ed. by Timothy Doster, Tegan Emerson, Henry Kvinge, Nina Miolane, Mathilde Papillon, Bastian Rieck, and Sophia Sanborn. Vol. 221. Proceedings of Machine Learning Research. PMLR, 28 Jul 2023, pp. 72–86. URL: https://proceedings.mlr.press/v221/alberti23a.html.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473 (2014). URL: https://api.semanticscholar.org/CorpusID:11212020.

[6] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. "BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 1–9. DOI: 10.18653/v1/2022.acl-short.1. URL: https://aclanthology.org/2022.acl-short.1/.

[7] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–

1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[8] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. "Reading Wikipedia to Answer Open-Domain Questions". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Regina Barzilay and Min-Yen Kan. Vancouver, Canada: Association for Computational Linguistics, July 2017, pp. 1870–1879. DOI: 10.18653/v1/P17-1171. URL: https://aclanthology.org/P17-1171.

[9] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. "Generating Long Sequences with Sparse Transformers". In: *ArXiv* abs/1904.10509 (2019). URL: https://api.semanticscholar.org/CorpusID:129945531.

[10] Krzysztof Choromanski et al. "Rethinking Attention with Performers". In: *ArXiv* abs/2009.14794 (2020). URL: https://api.semanticscholar.org/CorpusID:222067132.

[11] Aakanksha Chowdhery et al. "PaLM: scaling language modeling with pathways". In: *J. Mach. Learn. Res.* 24.1 (Mar. 2024). ISSN: 1532-4435.

[12] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 2978–2988. DOI: 10.18653/v1/P19-1285. URL: https://aclanthology.org/P19-1285.

[13] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher R'e. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *ArXiv* abs/2205.14135 (2022). URL: https://api.semanticscholar.org/CorpusID:249151871.

[14] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. "LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale". In: *ArXiv* abs/2208.07339 (2022). URL: https://api.semanticscholar.org/CorpusID:251564521.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *North American Chapter of the Association for Computational Linguistics*. 2019. URL: https://api.semanticscholar.org/CorpusID:52967399.

[16] Danny Driess et al. "PaLM-E: an embodied multimodal language model". In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[17] William Fedus, Barret Zoph, and Noam Shazeer. "Switch transformers: scaling to trillion parameter models with simple and efficient sparsity". In: *J. Mach. Learn. Res.* 23.1 (Jan. 2022). ISSN: 1532-4435.

[18] William Fedus, Barret Zoph, and Noam M. Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *ArXiv* abs/2101.03961 (2021). URL: https://api.semanticscholar.org/CorpusID:231573431.

[19] Elias Frantar and Dan Alistarh. "SparseGPT: massive language models can be accurately pruned in one-shot". In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[20] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. "OPTQ: Accurate Quantization for Generative Pre-trained Transformers". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=tcbBPnfwxS.

[21] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. "MiniLLM: Knowledge Distillation of Large Language Models". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=5h0qf7IBZZ.

[22] Shwai He, Liang Ding, Daize Dong, Jeremy Zhang, and Dacheng Tao. "SparseAdapter: An Easy Approach for Improving the Parameter-Efficiency of Adapters". In: *Findings of the Association for Computational Linguistics: EMNLP 2022*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 2184–2190. DOI: 10.18653/v1/2022.findings-emnlp.160. URL: https://aclanthology.org/2022.findings-emnlp.160/.

[23] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. "Parameter-Efficient Transfer Learning for NLP". In: *arXiv:1902.00751* (2019).

[24] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9.

[25] Yanping Huang et al. "GPipe: efficient training of giant neural networks using pipeline parallelism". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[26] Albert Q. Jiang et al. "Mixtral of Experts". In: *ArXiv* abs/2401.04088 (2024). URL: https://api.semanticscholar.org/CorpusID:266844877.

[27] Yuxin Jiang, Chunkit Chan, Mingyang Chen, and Wei Wang. "Lion: Adversarial Distillation of Proprietary Large Language Models". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 3134–3154. DOI: 10.18653/v1/2023.emnlp-main.189. URL: https://aclanthology.org/2023.emnlp-main.189/.

[28] Dhiraj D. Kalamkar et al. "A Study of BFLOAT16 for Deep Learning Training". In: *ArXiv* abs/1905.12322 (2019). URL: https://api.semanticscholar.org/CorpusID:168170136.

[29] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. "Large language models are zero-shot reasoners". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[30] Tom Kwiatkowski et al. "Natural Questions: A Benchmark for Question Answering Research". In: *Transactions of the Association for Computational Linguistics* 7 (2019). Ed. by Lillian Lee, Mark Johnson, Brian Roark, and Ani Nenkova, pp. 452–466. DOI: 10.1162/tacl_a_00276. URL: https://aclanthology.org/Q19-1026.

[31] Lambda Labs. *Demystifying GPT-3*. https://lambdalabs.com/blog/demystifying-gpt-3#1. Accessed: December 13, 2024. 2020.

[32] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. "{GS}hard: Scaling Giant Models with Conditional Computation and Automatic Sharding". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=qrwe7XHTmYb.

[33] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault. Online: Association for Computational Linguistics, July 2020, pp. 7871–7880. DOI: 10.18653/v1/2020.acl-main.703. URL: https://aclanthology.org/2020.acl-main.703.

[34] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. "Sequence Parallelism: Long Sequence Training from System Perspective". In: *Workshop on Efficient Systems for Foundation Models @ ICML2023*. 2023. URL: https://openreview.net/forum?id=SvUmzK7dLZ.

[35] Chen Liang, Simiao Zuo, Qingru Zhang, Pengcheng He, Weizhu Chen, and Tuo Zhao. "Less is more: task-aware layer-wise distillation for language model compression". In: *Proceedings of the 40th International Conference on Machine Learning*. ICML'23. Honolulu, Hawaii, USA: JMLR.org, 2023.

[36] Hao Liu, Matei Zaharia, and Pieter Abbeel. "RingAttention with Blockwise Transformers for Near-Infinite Context". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=WsRHpHH4s0.

[37] Xinyin Ma, Gongfan Fang, and Xinchao Wang. "LLM-Pruner: On the Structural Pruning of Large Language Models". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=J8Ajf9WfXP.

[38] Paulius Micikevicius et al. "Mixed Precision Training". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[39] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. "Memory-Efficient Pipeline-Parallel DNN Training". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 7937–7947. URL: https://proceedings.mlr.press/v139/narayanan21a.html.

[40] Deepak Narayanan 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*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476209. URL: https://doi.org/10.1145/3458817.3476209.

[41] Alec Radford and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training". In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[42] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners". In: 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[43] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. "Compressive Transformers for Long-Range Sequence Modelling". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=SylKikSYDH.

[44] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *J. Mach. Learn. Res.* 21 (2019), 140:1–140:67. URL: https://api.semanticscholar.org/CorpusID:204838007.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "ZeRO: memory optimizations toward training trillion parameter models". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[46] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. "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*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476205. URL: https://doi.org/10.1145/3458817.3476205.

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Ed. by Jian Su, Kevin Duh, and Xavier Carreras. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264. URL: https://aclanthology.org/D16-1264.

[48] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. "Learning multiple visual domains with residual adapters". In: *ArXiv* abs/1705.08045 (2017). URL: https://api.semanticscholar.org/CorpusID:215826266.

[49] Stephen Roller et al. "Recipes for Building an Open-Domain Chatbot". In: *Conference of the European Chapter of the Association for Computational Linguistics*. 2020. URL: https://api.semanticscholar.org/CorpusID:216562425.

[50] A. See, Peter J. Liu, and Christopher D. Manning. "Get To The Point: Summarization with Pointer-Generator Networks". In: *ArXiv* abs/1704.04368 (2017). URL: https://api.semanticscholar.org/CorpusID:8314118.

[51] Noam M. Shazeer. "Fast Transformer Decoding: One Write-Head is All You Need". In: *ArXiv* abs/1911.02150 (2019). URL: https://api.semanticscholar.org/CorpusID:207880429.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: *ArXiv* abs/1909.08053 (2019). URL: https://api.semanticscholar.org/CorpusID:202660670.

[53] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. "A Simple and Effective Pruning Approach for Large Language Models". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=PxoFut3dWW.

[54] Chaofan Tao, Lu Hou, Wei Zhang, Lifeng Shang, Xin Jiang, Qun Liu, Ping Luo, and Ngai Wong. "Compression of Generative Pre-trained Language Models via Quantization". In: Jan. 2022, pp. 4821–4836. DOI: 10.18653/v1/2022.acl-long.331.

[55] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. "Efficient Transformers: A Survey". In: *ACM Computing Surveys (ACM CSUR)* (2022).

[56] Nllb team et al. "No Language Left Behind: Scaling Human-Centered Machine Translation". In: *ArXiv* abs/2207.04672 (2022). URL: https://api.semanticscholar.org/CorpusID:250425961.

[57] Romal Thoppilan et al. "LaMDA: Language Models for Dialog Applications". In: *ArXiv* abs/2201.08239 (2022). URL: https://api.semanticscholar.org/CorpusID:246063428.

[58] Jacob-Junqi Tian, David B. Emerson, Sevil Zanjani Miyandoab, Deval Pandya, Laleh Seyyed-Kalantari, and Faiza Khan Khattak. "Soft-prompt Tuning for Large Language Models to Evaluate Bias". In: *ArXiv* abs/2306.04735 (2023). URL: https://api.semanticscholar.org/CorpusID:259108572.

[59] Inar Timiryasov and Jean-Loup Tastet. "Baby Llama: knowledge distillation from an ensemble of teachers trained on a small dataset with no performance penalty". In: *Proceedings of the BabyLM Challenge at the 27th Conference on Computational Natural Language Learning*. Ed. by Alex Warstadt et al. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 279–289. DOI: 10.18653/v1/2023.conll-babylm.24. URL: https://aclanthology.org/2023.conll-babylm.24/.

[60] Hugo Touvron et al. "Llama 2: Open Foundation and Fine-Tuned Chat Models". In: *ArXiv* abs/2307.09288 (2023). URL: https://api.semanticscholar.org/CorpusID:259950998.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[62] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. "Linformer: Self-Attention with Linear Complexity". In: *ArXiv* abs/2006.04768 (2020). URL: https://api.semanticscholar.org/CorpusID:219530577.

[63] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H. Chi, and Denny Zhou. "Self-Consistency Improves Chain of Thought Reasoning in Language Models". In: *ArXiv* abs/2203.11171 (2022). URL: https://api.semanticscholar.org/CorpusID:247595263.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. "Chain-of-thought prompting elicits reasoning in large language models". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2024. ISBN: 9781713871088.

[65] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *ArXiv* abs/1609.08144 (2016). URL: https://api.semanticscholar.org/CorpusID:3603249.

[66] Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. "Sheared LLaMA: Accelerating Language Model Pre-training via Structured Pruning". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=09iOdaeOzp.

[67] Manzil Zaheer et al. "Big Bird: Transformers for Longer Sequences". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 17283–17297. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/c8512d142a2d849725f31a9a7a361ab9-Paper.pdf.

[68] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. "PEGASUS: pre-training with extracted gap-sentences for abstractive summarization". In: *Proceedings of the 37th International Conference on Machine Learning*. ICML'20. JMLR.org, 2020.

[69] Yanli Zhao et al. *PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel*. 2023. arXiv: 2304.11277 [cs.DC]. URL: https://arxiv.org/abs/2304.11277.

Appendix A

Supplementary Materials and Examples

A.1 Batch Script Configurations for Llama 3.1 70B Training

This appendix provides the batch script configuration used to train Llama 3.1 70B models on two nodes of CSC's Puhti supercomputer.

A.1.1 Slurm Batch Script Example

The following script requests two GPU nodes, each equipped with four NVIDIA V100 GPUs, and dynamically configures multi-node execution.

Listing A.1: Slurm batch script for Llama 3.1 70B training on Puhti

```
#!/bin/bash
#SBATCH --output="slurm-training.out"
#SBATCH --job-name="training-gpu"
#SBATCH --account=<your_project_account>
#SBATCH --partition=gpu
#SBATCH --ntasks-per-node=1
#SBATCH --nodes=2
#SBATCH --cpus-per-task=10
#SBATCH --gres=gpu:v100:4
#SBATCH --mem=373G
#SBATCH --time=8:00:00

# Load necessary modules
module use /appl/soft/ai/singularity/modulefiles/
module load pytorch/2.3

# Get the hostnames for the allocated nodes
nodes=$(scontrol show hostnames $SLURM_JOB_NODELIST)
node_array=($nodes)

# Total number of nodes allocated
num_nodes=${#node_array[@]}
```

```
# Calculate total workers
total_workers=$((num_nodes * 4))

# Run python scripts on each node dynamically
for (( i=0; i<num_nodes; i++ )); do
    if [[ $i -eq 0 ]]; then
        # First node runs without a host node parameter
        srun --nodes=1 --ntasks=1 -w ${node_array[i]} bash -c "
            cd /path/to/training
            source .venv/bin/activate
            python training/train_script.py -m -w $total_workers --ngpus 4
    -nid $i
        " &
    else
        # Subsequent nodes use the first node as the host
        srun --nodes=1 --ntasks=1 -w ${node_array[i]} bash -c "
            cd /path/to/training
            source .venv/bin/activate
            python training/train_script.py -m -w $total_workers --ngpus 4
    -nid $i --host_node ${node_array[0]}
        "&
    fi
done

# Wait for all background jobs to finish
wait

echo "All scripts have completed"
```

A.1.2 Explanation of Key Parameters

- `#SBATCH -nodes=2`: Requests two compute nodes.
- `#SBATCH -ntasks-per-node=1`: Allocates one task per node.
- `#SBATCH -gres=gpu:v100:4`: Assigns four NVIDIA V100 GPUs per node.
- `#SBATCH -cpus-per-task=10`: Allocates 10 CPU cores per task.
- `#SBATCH -mem=373G`: Requests 373GB of system memory per node.
- `#SBATCH -time=8:00:00`: Sets a time limit of 8 hours.
- `module load pytorch/2.3`: Loads PyTorch version 2.3.
- `scontrol show hostnames $SLURM_JOB_NODELIST`: Retrieves node hostnames for dynamic allocation.
- `python training/train_script.py -m -w $total_workers -ngpus 4 -nid $i`: Runs the training script with dynamic worker allocation across nodes.

This script dynamically configures the execution environment to ensure efficient distributed training across multiple nodes.

A.2 Demonstrative Example of 2D Column-wise Sharded Weight Shape

Suppose a weight matrix W has shape $[4096, 1024]$ and the dimension sizes of FSDP and TP are 4 and 2, respectively. After applying 2D Column-wise sharding, the resulting local weight shard W_{ij} will have shape:

$$[4096/\#\text{FSDP},\ 1024/\#\text{TP}] = [2048,\ 256], \text{for column-wise sharding}$$

$$[4096/(\#\text{FSDP} \times \#\text{TP}), 1024] = [512,\ 1024], \text{for row-wise sharding}$$

where #FSDP and #TP denote the sizes of FSDP and TP dimensions, respectively. Here, the indices i and j refer to the ranks along the FSDP (also referred to as data parallel or DP) and TP dimensions.

A.3 Bottlenecks in Naïve Parameter Loading under TP + FSDP

Practically, PyTorch does not support efficient loading of model shards for 2D parallelism when loading Huggingface (HF) models through the `transformers` library. Although efficient loading is supported for FSDP alone, it fails when TP is involved. In fact, it results in loading the full model onto each GPU because each device cannot automatically determine which part of the model weights it should load. Therefore, there is a need for customization and optimization for zero-redundant and fast model loading.

A straightforward approach involves initially loading the entire model onto CPU memory and subsequently transferring only the required weight shards to each GPU. However, when this strategy is employed across N GPUs, it requires storing N times the full model size in CPU memory. In our setup on LUMI, this situation is exacerbated since all GPUs in a node share the same CPU memory. This approach introduces two critical bottlenecks: out-of-memory (OOM) errors in CPU memory, and significantly increased loading times due to limited CPU-GPU bandwidth.

A.4 FSDP-TP Weight Gathering: Elaboration, Example, and CPU Memory Allocation Analysis

Each device now holds a local shard that is equal to $1/(\#\text{FSDP} \times \#\text{TP})$ of the original weight size. After the retrieval operation, each device stores a weight of size $1/\#\text{TP}$ of the original weight in CPU memory. As a result, the total memory allocated across all devices is

$$\#\text{devices} \times \frac{1}{\#\text{TP}} = \#\text{FSDP} \times \#\text{TP} \times \frac{1}{\#\text{TP}} = \#\text{FSDP}$$

times the original weight size. Therefore, we end up loading N times more memory onto the CPU, where $N = \#\text{FSDP}$, i.e., the size of the FSDP dimension.

Appendix B

Supplementary Data and Experiments

B.1 Maximum number of trainable tokens per node of different parallelism configurations

TABLE B.1: Maximum trainable sequence length with different FSDP-TP configurations

Configuration	Maximum Tokens
FSDP-TP (8, 1)	88000
FSDP-TP (4, 2)	80000
FSDP-TP (2, 4)	82000
FSDP-TP (1, 8)	65000

B.2 Throughput Improvement and PTS reduction (in percentage) of different Parallelism configurations compared to TP-only configuration

TABLE B.2: Throughput comparison showing percentage higher performance of FSDP-only, FSDP-TP (4,2), and FSDP-TP (2,4) configurations relative to TP-only configuration across different context lengths.

Context Length	FSDP-only to TP-only (% higher)	FSDP-TP (4,2) to TP-only (% higher)	FSDP-TP (2,4) to TP-only (% higher)
2200	24.06	17.30	0.05
5496	23.95	14.13	0.27
11496	20.06	12.75	-1.83
20000	25.81	13.31	-0.33

TABLE B.3: Processing Time per Sample (PTS) comparison showing percentage lower PTS of FSDP-only, FSDP-TP (4,2), and FSDP-TP (2,4) configurations relative to TP-only configuration across different context lengths.

Context Length	FSDP-only to TP-only (% lower)	FSDP-TP (4,2) to TP-only (% lower)	FSDP-TP (2,4) to TP-only (% lower)
2200	19.56	14.67	0.00
5496	19.25	12.27	0.17
11496	16.70	11.27	-1.89
20000	54.32	11.74	-0.31

B.3 MTSL of different parallelism strategies at different LoRA ranks

TABLE B.4: Maximum trainable sequence length (MTSL) of different parallelism strategies across various LoRA ranks.

LoRA Rank	FSDP-only (FSDP-TP (8,1))	FSDP-TP (4,2)	FSDP-TP (2,4)	TP-only (FSDP-TP (1,8))
128	11500	20000	41000	65000
256	11000	19000	38000	50000
512	10000	16000	33000	40000
768	5000	14000	17000	25000
1024	0	4000	0	0

B.4 Throughput at different context length for different configurations

Table B.5 shows the specific throughput values at different context lengths for each parallelism configuration. Note that the empty entries are due to the absence of the corresponding experiments, as they could not ensure that the total number of processed tokens was approximately equal to the maximum total trainable tokens of the respective parallelism configurations.

B.5 PTS at different context length for different configurations

Table B.6 shows the specific PTS values at different context lengths for each parallelism configuration. Note that the empty entries are due to the absence of the corresponding experiments, as they could not ensure that the total number of processed tokens was approximately equal to the maximum total trainable tokens of the respective parallelism configurations.

TABLE B.5: Throughput (tokens/sec) vs. Context Length for Different Configurations

Context Length	**FSDP-TP (2, 4)**	**FSDP-TP (8, 1)**	**FSDP-TP (4, 2)**	**FSDP-TP (1, 8)**
2,200	978.99	1213.79	1147.83	978.53
3,950	-	-	1097.22	-
5,496	938.34	1160.00	1068.05	935.85
8,200	911.11	-	-	-
11,496	853.66	1044.00	980.47	869.59
13,000	-	-	-	844.16
20,000	782.78	-	889.88	785.34
32,496	684.13	-	-	684.13
41,000	630.77	-	-	-
65,000	-	-	-	509.01

TABLE B.6: Processing Time per Sample (PTS) (secs/sample) vs. Context Length for Different Configurations

Context Length	**FSDP-TP (2, 4)**	**FSDP-TP (8, 1)**	**FSDP-TP (4, 2)**	**FSDP-TP (1, 8)**
2,200	2.25	1.81	1.92	2.25
3,950	-	-	3.60	-
5,496	5.86	4.74	5.15	5.87
8,200	9.00	-	-	-
11,496	13.47	11.01	11.73	13.22
13,000	-	-	-	15.40
20,000	25.55	-	22.48	25.47
32,496	47.50	-	-	47.50
41,000	65.00	-	-	-
65,000	-	-	-	127.70

B.6 Mathematical Analysis of Throughput Degradation in Parallelism Strategies

This section provides a rigorous mathematical proof demonstrating why Tensor Parallelism (TP) exhibits less significant throughput degradation compared to Fully Sharded Data Parallelism (FSDP) when sequence length increases and batch size decreases proportionally.

B.6.1 Theoretical Framework

Let us define the following notation for our analysis:

- a_0: Computational cost (measured in time) for FSDP with batch size B_0 and sequence length L_0.
- b_0: Computational cost for TP with identical parameters (B_0, L_0).
- C: Fixed communication overhead inherent to TP implementation.
- a_1: Computational cost for FSDP with adjusted parameters $\big(B_0/n,\ L_0 \times n\big)$.
- b_1: Computational cost for TP with adjusted parameters $\big(B_0/n,\ L_0 \times n\big)$.

To maintain constant token throughput during our analysis, we establish that:

$$B_0 \times L_0 = (B_0/n) \times (L_0 \times n) \tag{B.1}$$

This constraint ensures that the total number of tokens processed remains invariant, while only the distribution between batch size and sequence length changes by a factor of n.

B.6.2 Computational Complexity Analysis

The computational complexity of attention mechanisms in transformer-based models is known to scale quadratically with sequence length:

- For FSDP: The computational cost is proportional to $L^2 \times \text{hidden\_dimension}$, with each rank processing the entire hidden dimension.
- For TP: The computational cost is proportional to $L^2 \times (\text{hidden\_dimension}/p)$, where p represents the number of tensor-parallel ranks.

When scaling sequence length by a factor of n, the computational cost increases by approximately n^2 for both parallelism strategies. We can thus establish:

$$\frac{a_1}{a_0} = n^2 \quad \text{and} \quad \frac{b_1}{b_0} = n^2 \tag{B.2}$$

B.6.3 Statement of the Theorem

We aim to prove that FSDP experiences a more significant relative throughput degradation compared to TP when sequence length increases and batch size decreases proportionally. Mathematically, we need to demonstrate:

$$\frac{a_1}{a_0} > \frac{b_1 + C}{b_0 + C} \tag{B.3}$$

Given the following conditions:

- $a_1/a_0 = n^2$
- $b_1/b_0 = n^2$
- $n > 1$ (sequence length increases)
- $C > 0$ (positive communication overhead)
- $b_0 > 0$ (positive initial computational cost)

B.6.4 Formal Proof

We proceed with a step-by-step proof:

1. Substituting $b_1 = n^2 b_0$ into the right side of our inequality:

$$\frac{b_1 + C}{b_0 + C} = \frac{n^2 b_0 + C}{b_0 + C} \tag{B.4}$$

2. Comparing n^2 with this fraction:

$$n^2 - \frac{n^2 b_0 + C}{b_0 + C} = \frac{n^2(b_0 + C) - (n^2 b_0 + C)}{b_0 + C} \tag{B.5}$$

3. Simplifying the numerator:

$$\begin{aligned} n^2(b_0 + C) - (n^2 b_0 + C) &= n^2 b_0 + n^2 C - n^2 b_0 - C && \text{(B.6)} \\ &= n^2 C - C && \text{(B.7)} \\ &= C(n^2 - 1) && \text{(B.8)} \end{aligned}$$

4. Since $n > 1$ implies $n^2 - 1 > 0$, and given $C > 0$, we can conclude:

$$C(n^2 - 1) > 0 \tag{B.9}$$

5. Therefore:

$$n^2 - \frac{n^2 b_0 + C}{b_0 + C} > 0 \Rightarrow n^2 > \frac{n^2 b_0 + C}{b_0 + C} \tag{B.10}$$

6. Finally, since $\frac{a_1}{a_0} = n^2$, we have:

$$\frac{a_1}{a_0} > \frac{b_1 + C}{b_0 + C} \tag{B.11}$$

B.6.5 Implications for Parallel Training Strategies

This mathematical analysis demonstrates that as sequence length increases (and batch size decreases proportionally), FSDP experiences a steeper relative throughput degradation compared to TP. This phenomenon is explained by the fixed communication overhead in TP implementations, which becomes relatively less significant as computational costs increase with longer sequences.

Specifically, while both parallelism strategies experience increased computational costs proportional to n^2 (due to the quadratic nature of attention), the relative impact on TP is

moderated by the fixed communication overhead. This theoretical finding aligns with our experimental results, where we observed that TP-based configurations maintained better relative throughput when processing longer sequences.

This mathematical insight provides crucial guidance for practitioners in selecting optimal parallelism strategies based on expected sequence lengths and computational resources. It suggests that as sequence lengths increase, the relative advantage of incorporating TP becomes more pronounced, particularly in scenarios where maximum sequence length is a priority over raw throughput.