# Reading Summary Week 6

**Aakash Agrawal**
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

## 1 Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

The paper highlights the challenges of training large models due to memory constraints. As model size increases, they can exceed the memory limits of processors. This issue is compounded by optimizer states, which demand additional memory per parameter, limiting the model size that can be effectively trained. The authors propose **Megatron-LM**, an efficient framework for **intra-layer model parallelism** that enables the training of transformer models with billions of parameters. This approach complements and works alongside pipeline-based model parallelism.

### 1.1 Parallelism Background and Challenges

#### 1.1.1 Data Parallelism

Data parallelism splits a training minibatch across multiple workers. As the minibatch size increases with the number of workers (**weak scaling**), training throughput scales nearly linearly. However, larger batches can complicate optimization, leading to reduced accuracy or longer convergence times, which may negate the benefits of faster training. Data parallelism can be enhanced with techniques like **activation checkpointing** for more efficient memory use. However, it is constrained by the requirement for the entire model to fit within a single worker, which becomes a challenge for large models like BERT and GPT-2, as neural networks near the memory limits of modern hardware accelerators.

#### 1.1.2 Model Parallelism

A solution to the limitations of data parallelism is model parallelism, where the model is split across multiple workers or devices. This reduces memory pressure and boosts parallelism, regardless of the micro-batch size. Model parallelism includes two main approaches: layer-wise **pipeline parallelism** and **distributed tensor parallelism**.

- In **pipeline model parallelism**, computations are divided into stages, with each device processing a portion of the model before passing its outputs to the next device. Some methods combine pipeline parallelism with a parameter server, but this can lead to inconsistencies, such as using outdated parameters, resulting in suboptimal performance.
- **Distributed tensor computation** is a more general approach that splits a tensor operation across multiple devices, helping to speed up computation or handle larger models. **Megatron-LM** utilizes this type of parallelism to improve efficiency and scale.

### 1.2 Model Parallel Transformers

A transformer consists of two main components: an **MLP** block and a **self-attention** block. The structure of transformer networks allows for a straightforward model parallel implementation by incorporating a few synchronization primitives.

Here's how parallelism works in both blocks:

- The **MLP block** involves a GEMM operation followed by a GeLU non-linearity. To implement parallelism, the weight matrix is split along its **columns** (column-partition), allowing the GeLU non-linearity to be applied independently to each partitioned GEMM output. This eliminates a synchronization point. Next, the second GEMM can be split along its rows, taking the output of the GeLU layer directly without requiring additional communication. This approach splits both GEMMs in the MLP block across GPUs and requires only a single **all-reduce** operation in the forward pass *(g operator)* and a single all-reduce in the backward pass *(f operator)*.

$$[Y_1, Y_2] = [GeLU(XA_1), GeLU(XA_2)]$$

- For the **self-attention block**, parallelism is exploited by splitting the matrix operations (GEMMs) for key (K), query (Q), and value (V) in a column-parallel way. This means each GPU handles the matrix multiplication for a specific attention head locally without the need for immediate communication. By doing this, the parameters and workload for each attention head are distributed across GPUs efficiently. In the next step, the GEMM from the output linear layer is split along its rows and directly uses the output from the parallelized attention layer, also without communication between GPUs.

This approach combines two GEMMs in both the MLP and self-attention layers, removing the need for synchronization in between, which helps the system scale better. As a result, the entire forward and backward pass of a transformer layer only requires two all-reduce operations each, significantly improving efficiency.

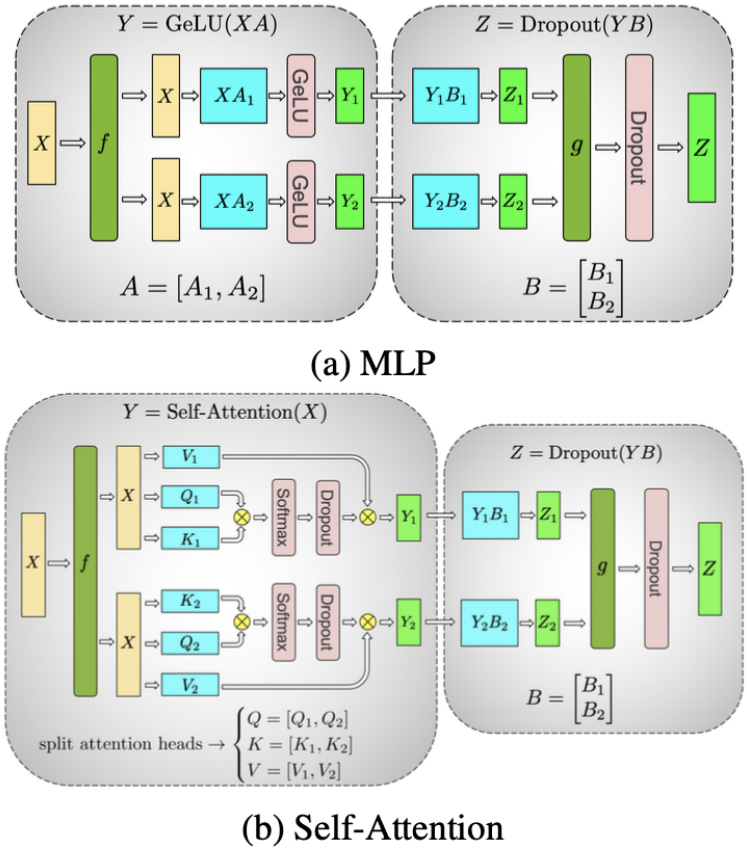


Figure 1: Blocks of Transformer with Model Parallelism.

### 1.3 Experiments and Results

**Short note on Setup**: To analyze the impact of model size **scaling** on accuracy, the authors train GPT-2 model and a BERT bidirectional transformer model, evaluating their performance on several downstream tasks. To create a diverse training set with long-term dependencies, several large language modeling datasets are combined: Wikipedia, CC-Stories, RealNews, and OpenWebText. The authors perform pre-processing to remove duplicated data points. The experiments use up to **32 DGX-2H servers** (a total of **512 Tesla V100** SXM3 32GB GPUs). The authors employ techniques like mixed precision training and activation checkpointing after every transformer layer to better manage the memory footprint.

- **Scaling analysis for model parallelism**: The authors establish a baseline by training a GPT-2 model with 1.2 billion parameters on a single V100, achieving 30% of the theoretical peak FLOPS. Scaling the model to 8.3 billion parameters across 8 GPUs with 8-way model parallelism achieves **77%** of linear **scaling**.
- **Scaling analysis for model + data parallelism**: Scaling the model to 8.3 billion parameters running on 512 GPUs achieves **74% scaling** relative to linear scaling of the strong single GPU baseline configuration.
- **Language Modeling Benchmark using GPT-2**: The authors also establish a language modeling benchmark using GPT-2, demonstrating that LLMs can further advance the SOTA results as we increase the size of the models. The **Perplexity** score on the Wikitext103 test set drops from 19.3 for the **355M** model to 10.8 for an **8.3B** parameter model. Further, larger models converge noticeably faster and converge to lower validation perplexities than their smaller counterparts.
- **Bi-directional Transformer Results Using BERT**: Prior to this work, it was observed that increasing model size beyond BERT-large with 336M parameters results in unexpected model degradation. In this paper, the authors show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased accuracies as the model size grows. Of the 336M, 1.3B, and 3.9B BERT models, the 3.9B Megatron model achieves a 90.9% accuracy on the RACE test dataset, beating the previous SOTA.

## 2 Everything about Distributed Training and Efficient Finetuning

The article delves into practical hacks and strategies for distributed training, spotlighting tools like DeepSpeed and Fully Sharded Data Parallelism (FSDP). It also covers efficient fine-tuning methodologies, particularly emphasizing multi-GPU and multi-node environments.

### 2.1 Distributed Training Basics

When we discuss training or fine-tuning LLMs, we typically handle massive datasets and models with billions of parameters. Speeding up the training of LLMs means maximizing throughput—processing as many samples per second as possible. Training these models demands substantial GPU vRAM because they involve large model weights, optimizer states, and intermediate activations. Therefore, we need clever distributed training strategies where each GPU worker manages only a part of the training state and data. Some of the main parallelism strategies:

- **Data Parallelism (DP)**: In this approach, each GPU worker processes only a segment of the dataset batch, computing gradients based on that segment. These gradients are then averaged across all workers, and the model weights are updated accordingly. A fundamental form of DP entails each GPU having its own set (replica) of model weights, optimizer states, and gradients for the specific data it's handling.
- **Model Parallelism (MP)**: In this approach, various layers of the model are distributed across different workers. This technique is particularly useful when the model is too large to fit within the memory of a single GPU, allowing each worker to handle a portion of the model's parameters and computations, thereby enabling the training of significantly larger models.

- **Pipeline Parallelism (PP)**: Here, groups of operations are performed on one worker before the outputs are passed to the next worker in the pipeline, where a different group of operations is performed.
- **Tensor Parallelism (TP)**: Here, each GPU processes only a slice of a tensor by horizontally slicing the model across GPU workers and computing the activations for the part of the weights each worker has; batch data is replicated across all the workers. A notable example is Megatron-LM, which we mentioned earlier.

**Baseline PyTorch DDP**: Simple data parallelism (Baseline), as seen in PyTorch's DistributedData-Parallel (DDP), involves each GPU worker having its own copy of the model weights, optimizer state, and gradients. After the backward pass, gradients are averaged across all workers in an **all-reduce** step before updating the model weights. If we have $\Psi$ parameters, we incur a communication cost of $2\Psi$ (sending out the data from the processor + receiving data from all other processors) with plain DP.

### 2.2 ZeRO-powered Data-Parallelism

Zero Redundancy Optimizer has 3 different methods, also called stages:

1. **ZeRO Stage-1 /** $P_{os}$: Here, only the optimizer state is partitioned or sharded across GPU workers, while the model weights and gradients are replicated on each worker. Following the backward pass, there's a standard **all-reduce** operation to compute the average gradient across all workers. This approach offers a **4x** memory reduction (in this specific example), maintaining the same communication volume as the baseline without extra inter-GPU communication. Here, as well, if we have $\Psi$ parameters, we incur a communication cost of $2\Psi$.
2. **ZeRO Stage-2/** $P_{os+g}$: Here, both the optimizer state and the gradients are partitioned or sharded across the workers. This means that two GPU workers are handling different micro-batches of data and are responsible for the gradients of different subsets of model parameters. The key advantage is that each worker updates only its segment of the optimizer state, needing just the corresponding gradients for that update. At the gradient level, a **reduce-scatter** is performed. This approach offers a **8x** memory reduction with the same communication volume as the Baseline (PyTorch DDP). With Both ZeRO Stage 1 and 2, we need the entire model to fit on 1 GPU.
3. **ZeRO Stage-3/** $P_{os+g+p}$: Stage-3 extends the sharding principle to encompass not just the optimizer state and gradients but also the model parameters. Each layer of the model is horizontally divided, with each worker holding only a portion of the weight tensors. During both the forward and backward passes (where each GPU worker processes different micro-batches of data), these workers communicate the necessary parts of each layer's weights as needed (on-demand parameter communication), allowing them to compute activations and gradients. This approach offers a **64x** memory reduction, with 1.5x communication volume as compared to the Baseline, since for every training step, we have an extra all-gather operation for model parameters in the forward pass.

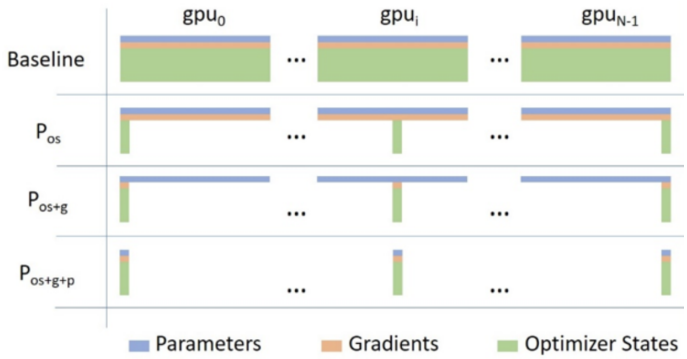


Figure 2: High-level overview of different stages/methods in ZeRO on the basis of partitioning and replication schemes.

**ZeRO-R** advances ZeRO-DP by targeting activation memory consumption and managing memory fragmentation, reducing the activation memory footprint through partitioning. Similar to the idea of CPU Swapping, ZeRO includes optimizations like **ZeRO-Offload/Infinity**, which offload some of the computations to the CPU/NVMe disk.

### 2.3 Fully-Sharded Data Parallel

Fully-Sharded Data Parallel (FSDP) is a data parallelism technique designed to enhance memory efficiency while minimizing communication overhead, thereby boosting throughput. FSDP employs two sharding strategies: Full Sharding and Hybrid Sharding.

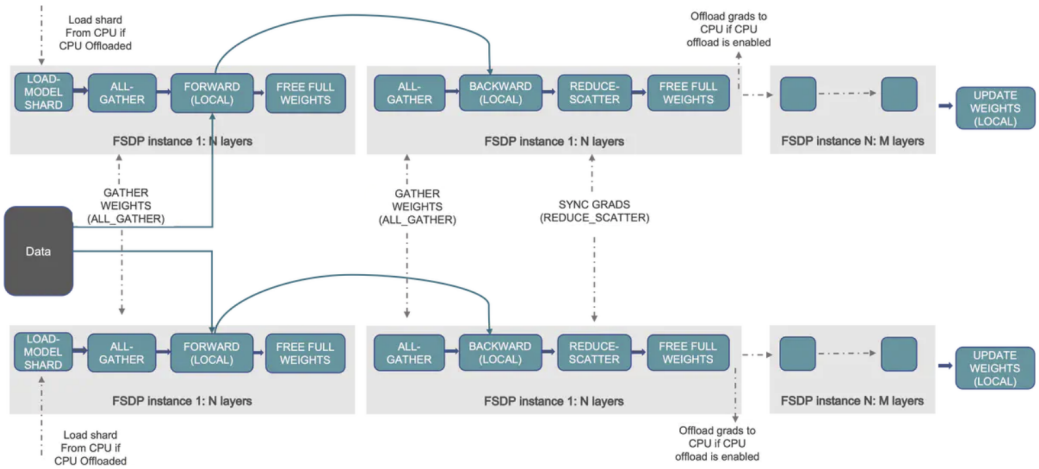


Figure 3: Fully-sharded FSDP. A low-level visualization of different operations involved with 2 devices.

1. **Full Sharding**: Similar to ZeRO-3, full Sharding in FSDP shards model parameters, optimizer states, and gradients across all workers/devices, significantly reducing memory usage per device while maintaining high computational efficiency.
2. **Hybrid Sharding**: Hybrid Sharding in FSDP combines full sharding with some level of replication, allowing for a trade-off between memory usage and communication overhead based on the specific needs of the training scenario as it involves an **extra all-gather** across nodes (inter-node) to get an averaged gradient value for the total mini-batch of data being processed in each training step.

One of the primary benefits of using DeepSpeed's ZeRO or FSDP is that they provide memory savings and throughput enhancements typically associated with data and tensor parallelism, all while operating within a data-parallel framework.

### 2.4 Efficient Finetuning

Here are some strategies for the efficient fine-tuning of large models (including LLMs):

1. **Mixed Precision**: Weights, activations, and gradients are stored in half-precision (FP16) to save memory and speed up computations, while the "master copy" of the weights is kept in full precision (FP32). This improves performance by reducing memory usage without losing model accuracy.
2. **Parameter-Efficient Fine-Tuning (PEFT)**: PEFT reduces memory use during fine-tuning by freezing most model weights and updating only a small subset. Popular methods like LoRA and $(IA)^3$ inject small, trainable vectors into key model layers. These vectors are merged into the base weights at inference, avoiding extra computations. However, fine-tuning may be less effective/accurate compared to full-tuning.

3. **Flash Attention**: Flash Attention is a fast, memory-efficient attention algorithm that maximizes GPU performance. FlashAttention 2 can achieve up to 220+ TFLOPS on an A100 80GB (which has a maximum of 315 TFLOPS), delivering speed and efficiency without approximations.
4. **Gradient/Activation Checkpointing**: This technique saves memory by storing only some intermediate activations and recomputing others during the backward pass. The tradeoff is added recomputation time to save memory.
5. **Quantization**: Reduces model size and speeds up inference by lowering numerical precision, say from FP32 to INT8 or lower. Benefits include lower memory usage, faster inference, and efficient deployment on resource-constrained devices like edge hardware. However, aggressive quantization may degrade model accuracy.
    - **Quantization-Aware Training** (QAT): The model is trained while simulating quantization effects, allowing it to adapt and minimize accuracy loss.
    - **Post-Training Quantization** (PTQ): A pre-trained model is quantized after training, often with calibration data, to reduce precision without retraining.
6. **Gradient Accumulation**: Enables training with large batch sizes on limited memory by splitting a batch into smaller "micro-batches." Gradients are computed and accumulated over multiple iterations before updating the model. This smooths updates, reduces noise, and improves efficiency, especially in multi-GPU training. Reducing all-reduce operations lowers inter-GPU communication overhead and speeds up training.

**Is increasing batch size always beneficial?** No. The goal should be to train the best possible model as fast as possible for easy experimentation using available hardware.

### 2.5 Practical Guidelines

Here are my key practical takeaways from the blog when experimenting and fine-tuning models with **10–100B+** parameters on **1M+** size datasets:

- **Use BF16/FP16 by default**: BF16 is preferred for its simplicity and minimal overflow issues without the need for extra configuration.
- **Apply LoRA**: Add trainable parameters to all linear layers to enable efficient fine-tuning with minimal memory.
- **Enable Flash Attention**: Use it if your GPU supports it for faster and more memory-efficient attention computation.
- **Use Gradient/Activation Checkpointing**: Reduces memory usage at the cost of slight throughput reduction. If Flash Attention is enabled, checkpointing might not be necessary.
- **For multiple GPUs**: Start with BF16 + LoRA + Gradient Checkpointing + DeepSpeed ZeRO 3 for efficient memory management and training performance.
- **Use Quantization**: For limited GPU memory, apply quantization to reduce model size and improve inference speed.
- **With large GPU clusters (8+ V100s or A100s)**: DeepSpeed ZeRO-3 is optimal. ZeRO-2 can be used but may hit CPU RAM limits due to model replication across workers.
- **Gradient Accumulation**: Use it if the batch size is still limited after applying the optimizations above. This can speed up training, especially for large models and multi-GPU/multi-node setups.
- **Activate CPU/Disk Offloading**: If GPU memory is extremely limited, offloading can help free up space and maintain training progress.