A Deep Dive into Parallelism with Llama 4 Scout

June 2025

1 The Goal

This write-up provides an in-depth look at how data parallelism and tensor parallelism (row and column parallelism), are applied to a defined model using LLaMA 4 Scout in PyTorch. For simplicity, we focus on a single Transformer block from LLaMA 4 Scout language part. We explore the underlying implementation techniques, including the use of forward hook functions and the DTensor module.

The outline of this review is as follows: we begin by examining a single Transformer block from LLaMA 4, analyzing it from both the network architecture and parameter perspectives. Next, we explain how tensor parallelism is applied, including examples of different types such as row and column parallelism. We then describe how data parallelism is integrated. Finally, we delve into the underlying implementation techniques, focusing on forward hook functions and the DTensor module.

2 Intro to Llama 4 Scout

2.1 Network Structure

LLaMA 4 Scout comprises two primary components: a language module and a vision module. The architecture of the language decoder is detailed in Table **??**, using the PyTorch model format. Visually, it closely resembles the structure shown in Figure 1 for LLaMA 3.2. The main difference lies in the MLP layer—LLaMA 4 incorporates a Mixture-of-Experts (MoE) layer, whereas LLaMA 3.2 does not. For illustration purposes, only a single transformer block is shown; the full model consists of 32 identical transformer blocks.

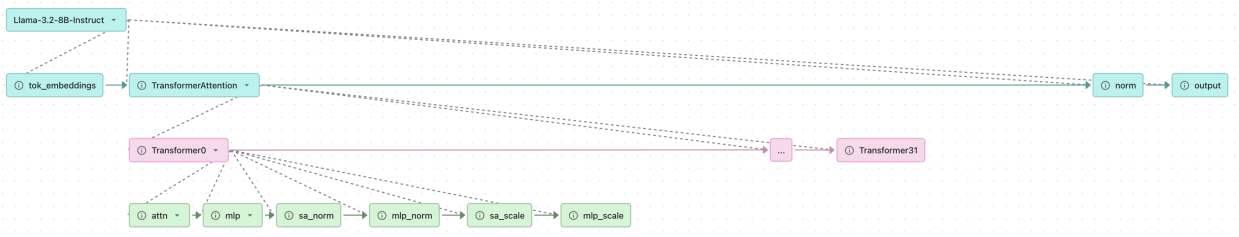

Figure 1: LLama 4 scout language decoder network structure

```
TransformerDecoder(
  (tok_embeddings): Embedding(202048, 5120)
  (layers): ModuleList(
    (0): TransformerSelfAttentionLayer(
      (attn): MultiHeadAttention(
        (q_proj): Linear(in_features=5120, out_features=5120, bias=False)
        (k_proj): Linear(in_features=5120, out_features=1024, bias=False)
        (v_proj): Linear(in_features=5120, out_features=1024, bias=False)
        (output_proj): Linear(in_features=5120, out_features=5120, bias=False)
        (pos_embeddings): Llama4ScaledRoPE()
      )
      (mlp): MoE(
        (experts): GroupedExperts()
        (router): TokenChoiceTopKRouter(
          (gate): Linear(in_features=5120, out_features=16, bias=False)
        )
        (shared_expert): FeedForward(
          (w1): Linear(in_features=5120, out_features=8192, bias=False)
          (w2): Linear(in_features=8192, out_features=5120, bias=False)
          (w3): Linear(in_features=5120, out_features=8192, bias=False)
          (activation): SiLU()
        )
      )
      (sa_norm): RMSNorm()
      (mlp_norm): RMSNorm()
      (sa_scale): Identity()
      (mlp_scale): Identity()
    )
  )
  (norm): RMSNorm()
  (output): Linear(in_features=5120, out_features=202048, bias=False)
)
```

Table 1: Structure of the `TransformerDecoder` module

2.2 Parameters

The table below lists the parameters and their shapes in LLaMA-4, similar to the network structure part, considering only a single layer of the decoder. The vision component is omitted in this overview.

Figure 2: Overview tensor parallel and data parallel applying to model

Parameter Name	**Size**
decoder.tok_embeddings.weight	[202048, 5120]
decoder.layers.0.attn.q_proj.weight	[5120, 5120]
decoder.layers.0.attn.k_proj.weight	[1024, 5120]
decoder.layers.0.attn.v_proj.weight	[1024, 5120]
decoder.layers.0.attn.output_proj.weight	[5120, 5120]
decoder.layers.0.mlp.experts.gate_proj	[16, 5120, 8192]
decoder.layers.0.mlp.experts.down_proj	[16, 8192, 5120]
decoder.layers.0.mlp.experts.up_proj	[16, 5120, 8192]
decoder.layers.0.mlp.router.gate.weight	[16, 5120]
decoder.layers.0.mlp.shared_expert.w1.weight	[8192, 5120]
decoder.layers.0.mlp.shared_expert.w2.weight	[5120, 8192]
decoder.layers.0.mlp.shared_expert.w3.weight	[8192, 5120]
decoder.layers.0.sa_norm.scale	[5120]
decoder.layers.0.mlp_norm.scale	[5120]
decoder.norm.scale	[5120]
decoder.output.weight	[202048, 5120]

Table 2: Model Parameters and Their Shapes

3 Parallelism

Initially, the model is defined as shown above. Next, a tensor parallelism plan (for example, defined in torchtune.models.llama4.decoder_only_tp_plan) is created and applied to each module in the network. Following that, data parallelism—specifically, Fully Sharded Data Parallel (FSDP)—is applied to each module to further distribute training across devices. The entire process is presented in the Fig 2.

3.1 Tensor Parallel

After tensor parallel sharding, the network parameters are updated as shown in the table 3 below. If a parameter remains of type Parameter, it means that the corresponding layer is not compatible with tensor parallelism. If a parameter is shown as a DTensor, it indicates that the parameter has been sharded and converted from a regular tensor to a distributed tensor. The table also specifies the devices across which the parameter is sharded, as well as the number of devices involved.

For example, the parameter decoder.layers.0.attn.q_proj.weight has been split into two parts, distributed across CUDA 0 and CUDA 1. The mesh_dim_names field is set to tp, denoting that the sharding follows the tensor parallel dimension.

Parameter Name	Shape	Type
decoder.tok_embeddings.weight	[202048, 5120]	Parameter
decoder.layers.0.attn.q_proj.weight	[5120, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.attn.k_proj.weight	[1024, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.attn.v_proj.weight	[1024, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.attn.output_proj.weight	[5120, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp.experts.gate_proj	[16, 5120, 8192]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp.experts.down_proj	[16, 8192, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp.experts.up_proj	[16, 5120, 8192]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp.router.gate.weight	[16, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp.shared_expert.w1.weight	[8192, 5120]	Parameter
decoder.layers.0.mlp.shared_expert.w2.weight	[5120, 8192]	Parameter
decoder.layers.0.mlp.shared_expert.w3.weight	[8192, 5120]	Parameter
decoder.layers.0.sa_norm.scale	[5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.layers.0.mlp_norm.scale	[5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.norm.scale	[5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))
decoder.output.weight	[202048, 5120]	DTensor (cuda [0, 1], mesh_dim_names: (tp))

Table 3: Model Parameters and their Distribution Type

3.1.1 Tensor Parallel Type

There are several types of tensor parallelism: row parallelism, column parallelism, sequence parallelism, and customized tensor parallelism. In Table 4, we list the tensor parallelism type associated with each layer. This association is not arbitrary; it is determined by the inherent characteristics of each parallelism type, as illustrated in the following example.

Module	Tensor Parallel Style
decoder	PrepareModuleInput
decoder.norm	SequenceParallel
decoder.output	ColwiseParallel
decoder.layers.0.sa_norm	SequenceParallel
decoder.layers.0.attn	PrepareModuleInput
decoder.layers.0.attn.q_proj	ColwiseParallel
decoder.layers.0.attn.k_proj	ColwiseParallel
decoder.layers.0.attn.v_proj	ColwiseParallel
decoder.layers.0.attn.output_proj	RowwiseParallel
decoder.layers.0.mlp_norm	SequenceParallel
decoder.layers.0.mlp	torchtune defined PrepareModuleInputOutput
decoder.layers.0.mlp.router.gate	torchtune defined NoParallel
decoder.layers.0.mlp.experts	torchtune defined ExpertTensorParallel

Table 4: Model Parameters and their Distribution Type

3.2 How to Apply Tensor Parallelism to Each Layer/Module

To apply tensor parallelism to an existing module, a wrapper is typically created around the original module. In addition, the inputs, parameters, and outputs must be modified to ensure compatibility with the parallelism strategy. This overall process is illustrated in Fig. 3.

When are these transformations of the inputs parameters, and outputs applied? Typically, when a module is invoked, its forward function is automatically executed. To integrate tensor parallelism, the prepare input and partition functions are registered as forward pre-hooks, which are executed before the module's forward

function. Conversely, the prepare output function is registered as a forward hook, which is executed after the forward function completes.

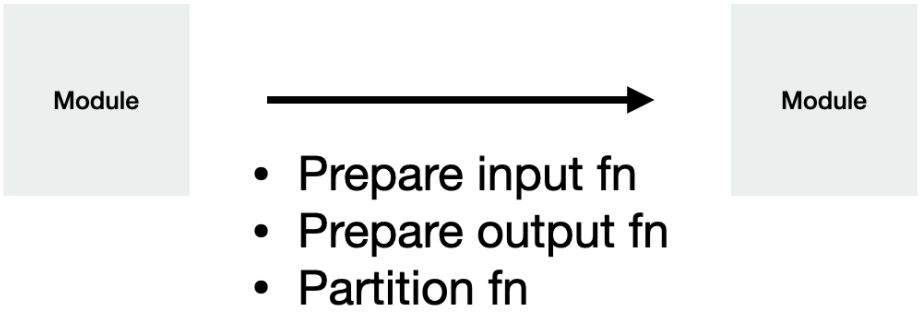

Figure 3: Turn Module to Tensor Parallelized Module with the three defined functions

As shown above, the wrapper functions require three components: a function to prepare the input, a function to prepare the output, and a partitioning function.

3.2.1 Examples of Tensor Parallel

As shown above, there are various types of tensor parallelism. In this section, we examine column parallelism, row parallelism, and components related to PrepareModuleInput.

Column Parallel Example as shown in the Fig 4: Suppose that the input has a shape of $10 * 512$, and the weight has a shape of $2048 * 512$, the linear layer operation is $input * weight^T$. In column parallelism, we can divide $weights^T$ into four equal parts, and each shape is $512 * 512$, and distribute them across four different GPUs. On each GPU, the linear operation is performed independently, producing a shape output $10 * 512$. Concatenating the output of all four GPUs results in the final shape output $10 * 2048$.

The fundamental idea behind column parallelism is that in matrix multiplication $A * B$, the i^{th} column of the output depends only on the matrix A and the i^{th} column of the matrix B.

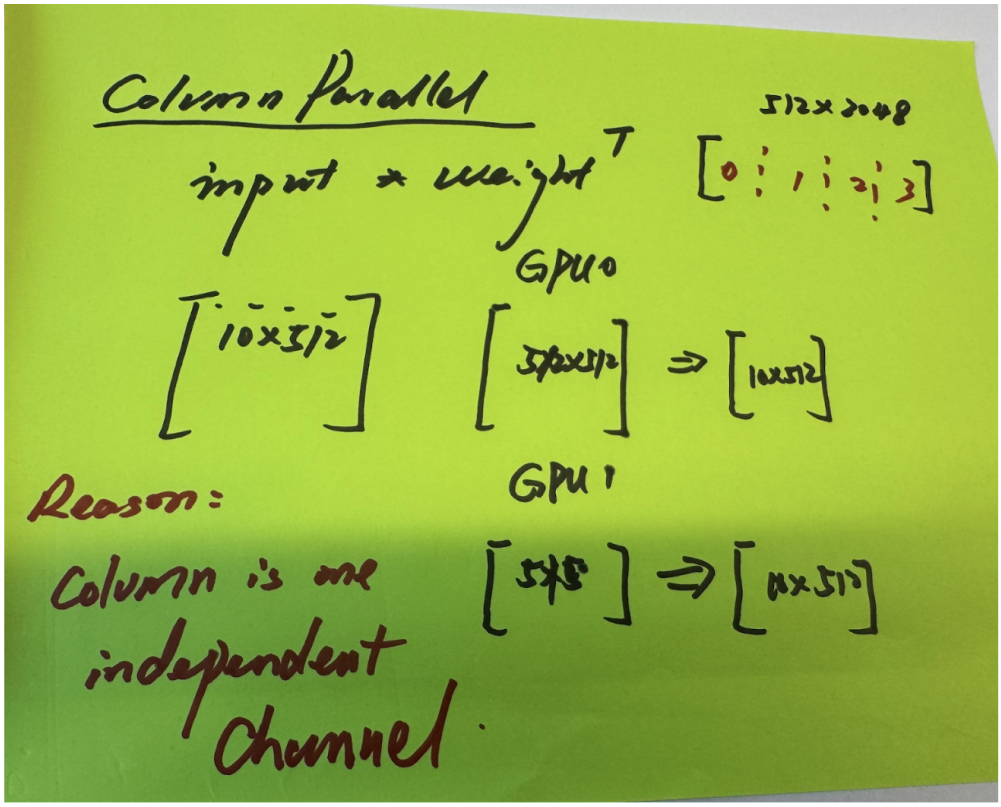

Figure 4: Column Parallel

Row Parallel Example as shown in the Fig 5: after applying column parallelism, the output is distributed across the GPUs, with each GPU holding a local tensor of shape $10 * 512$. Instead of concatenating the output from different GPUs, which would incur communication overhead, we keep the outputs local to each GPU. For the subsequent linear operation matrix multiplication $output * W$, where W has a shape of $2048 * 2048$, we partition W row-wise into four chunks, each with shape $512 * 2048$. Each GPU performs the matrix multiplication locally, shape perspective $(10 * 512) * (512 * 2028)$, resulting in a local output of shape $10 * 2048$. The final result is obtained by summing the outputs from all GPUs. From an operational standpoint, this step corresponds to an all-reduce operation, and the aggregated result maintains the shape $10 * 2048$.

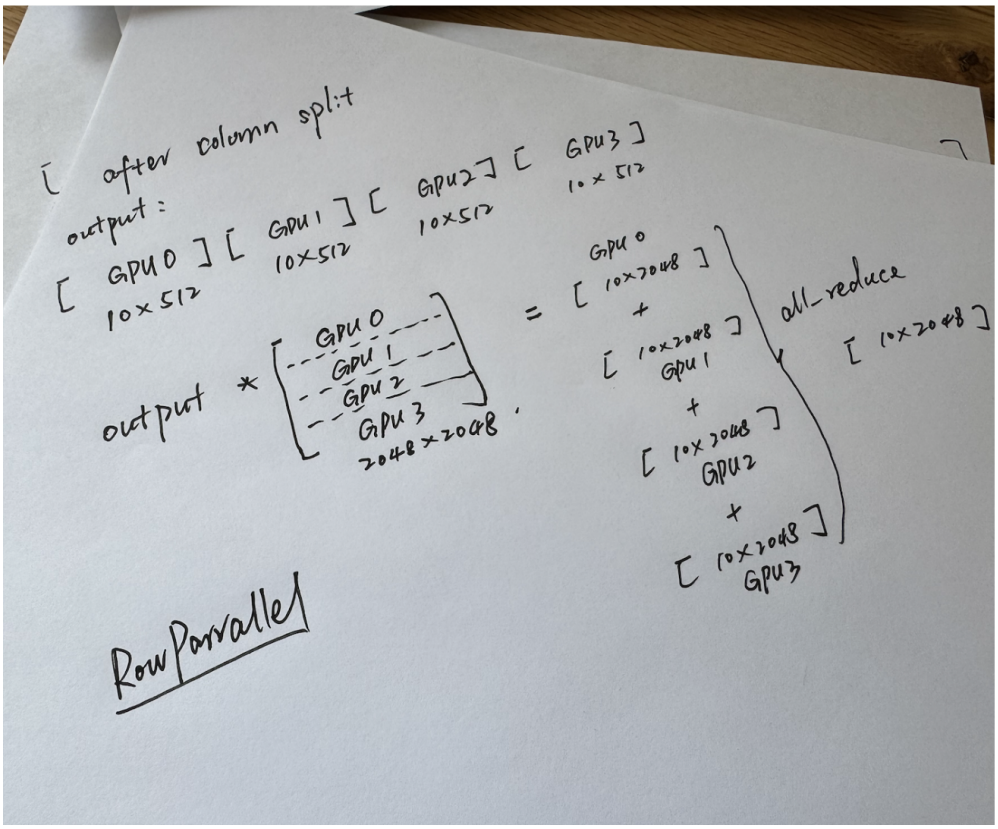

Figure 5: Row Parallel

Another example PrepareModuleInput: At the very beginning, the input tokens have a shape of $1 * 84$ where the batch size is 1 and the sequence length is 84. Before entering the PrepareModuleInput class and executing prepare_input_fn, the input embeddings are represented as a torch.Tensor of shape $1 * 84 * 5120$ where 5120 is the embedding dimension. Within the PrepareModuleInput class, this tensor is first replicated across all devices (e.g., cuda:0 and cuda:2) and then redistributed using shard(1). As a result, each device holds a slice of the tensor with shape $1 * 42 * 5120$ stored as a DTensor. Logically, however, the input embedding remains a single tensor of shape $1 * 84 * 5120$.

3.3 Data Parallel: FSDP

After applying tensor parallel, we further apply fully shard data parallel (FSDP), which shards a model's parameters across multiple GPUs to improve efficiency and scalability. Table 5 illustrates how parameter are distributed after applying tensor parallel and FSDP. Let's walk through a few example for clarity.

- The parameter decoder.tok_embeddding.weight is only sharded using FSDP. As indicated in the deviceMesh field, its mesh_dim_names is set to dp_shard, meaning the parameter is sharded across two devices: cuda:0 and cuda:1.
- The parameter decoder.layers.0.cwm.attn.q_proj.weight uses both tensor parallelism and FSDP. This is reflected in the mesh_dim_names being (dp_shard, tp), indicating the parameter is sharded along both the data parallel and tensor parallel dimensions.

Parameter Name	Shape	DeviceMesh
decoder.tok_embeddings.weight	[202048, 5120]	('cuda', [0, 2], mesh_dim_names=('dp_shard',))
decoder.layers.0.cwm.attn.q_proj.weight	[5120, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.attn.k_proj.weight	[1024, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.attn.v_proj.weight	[1024, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.attn.output_proj.weight	[5120, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp.experts.gate_proj	[16, 5120, 8192]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp.experts.down_proj	[16, 8192, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp.experts.up_proj	[16, 5120, 8192]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp.router.gate.weight	[16, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp.shared_expert.w1.weight	[8192, 5120]	('cuda', [0, 2], mesh_dim_names=('dp_shard',))
decoder.layers.0.cwm.mlp.shared_expert.w2.weight	[5120, 8192]	('cuda', [0, 2], mesh_dim_names=('dp_shard',))
decoder.layers.0.cwm.mlp.shared_expert.w3.weight	[8192, 5120]	('cuda', [0, 2], mesh_dim_names=('dp_shard',))
decoder.layers.0.cwm.sa_norm.scale	[5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.layers.0.cwm.mlp_norm.scale	[5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.norm.scale	[5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))
decoder.output.weight	[202048, 5120]	('cuda', [[0, 1], [2, 3]], mesh_dim_names=('dp_shard', 'tp'))

Table 5: Parameter Details after Tensor Parallel and FSDP. Note: cwm for checkpoint_wrapped_module

Figure 6 presents a module-level comparison before and after applying tensor parallelism and FSDP. As shown, the original EarlyFusionModel is transformed into FSDPEarlyFusionModel, instructing PyTorch to invoke the forward function associated with the FSDP-wrapped module during execution.

```
EarlyFusionModel(
(decoder): TransformerDecoder(
  (tok_embeddings): Embedding(202048, 5120)
  (layers): ModuleList(
    (0): TransformerSelfAttentionLayer(
      (attn): MultiHeadAttention(
        (q_proj): Linear(in_features=5120, out_features=5120, bias=False)
        (k_proj): Linear(in_features=5120, out_features=1024, bias=False)
        (v_proj): Linear(in_features=5120, out_features=1024, bias=False)
        (output_proj): Linear(in_features=5120, out_features=5120, bias=False)
        (pos_embeddings): Llama4ScaledRoPE()
      )
      (mlp): MoE(
        (experts): GroupedExperts()
        (router): TokenChoiceTopKRouter(
          (gate): Linear(in_features=5120, out_features=16, bias=False)
        )
        (shared_expert): FeedForward(
          (w1): Linear(in_features=5120, out_features=8192, bias=False)
          (w2): Linear(in_features=8192, out_features=5120, bias=False)
          (w3): Linear(in_features=5120, out_features=8192, bias=False)
          (activation): SiLU()
        )
      )
      (sa_norm): RMSNorm()
      (mlp_norm): RMSNorm()
      (sa_scale): Identity()
      (mlp_scale): Identity()
    )
  )
  (norm): RMSNorm()
  (output): Linear(in_features=5120, out_features=202048, bias=False)
)
```

(a) Llama 4 with 1 tranformer layer

```
FSDPEarlyFusionModel(
(decoder): TransformerDecoder(
  (tok_embeddings): Embedding(202048, 5120)
  (layers): ModuleList(
    (0): FSDPCheckpointWrapper(
      (_checkpoint_wrapped_module): TransformerSelfAttentionLayer(
        (attn): MultiHeadAttention(
          (q_proj): Linear(in_features=5120, out_features=5120, bias=False)
          (k_proj): Linear(in_features=5120, out_features=1024, bias=False)
          (v_proj): Linear(in_features=5120, out_features=1024, bias=False)
          (output_proj): Linear(in_features=5120, out_features=5120, bias=False)
          (pos_embeddings): Llama4ScaledRoPE()
        )
        (mlp): MoE(
          (experts): GroupedExperts()
          (router): TokenChoiceTopKRouter(
            (gate): Linear(in_features=5120, out_features=16, bias=False)
          )
          (shared_expert): FeedForward(
            (w1): Linear(in_features=5120, out_features=8192, bias=False)
            (w2): Linear(in_features=8192, out_features=5120, bias=False)
            (w3): Linear(in_features=5120, out_features=8192, bias=False)
            (activation): SiLU()
          )
        )
        (sa_norm): RMSNorm()
        (mlp_norm): RMSNorm()
        (sa_scale): Identity()
        (mlp_scale): Identity()
      )
    )
  )
  (norm): RMSNorm()
  (output): Linear(in_features=5120, out_features=202048, bias=False)
)
```

(b) Llama 4 with 1 tranformer layer after FSDP

Figure 6: Module Comparison before and after TP and FSDP

4 Forward Hook Functions

As previously mentioned, when applying parallelism to individual module, we register forward hooks—specifically, forward pre-hooks and forward hooks—to modify behavior during execution. In general:

- Forward pre-hooks are used for input preprocessing, normalization, and routing.
- Forward hooks are used for output logging, masking, and postprocessing.

n the context of parallelism, these hooks play a crucial role: they are responsible for sharding, replicating, and redistributing inputs and outputs to support parallel execution.

Table 6 lists the modules along with their registered forward pre-hooks, providing insight into how parallelization is applied at each stage.

Module Name	**Hook Source**
	fsdp
decoder	style
decoder.layers.0	fsdp
decoder.layers.0._checkpoint_wrapped_module.attn	style
decoder.layers.0._checkpoint_wrapped_module.attn.q_proj	api
decoder.layers.0._checkpoint_wrapped_module.attn.k_proj	api
decoder.layers.0._checkpoint_wrapped_module.attn.v_proj	api
decoder.layers.0._checkpoint_wrapped_module.attn.output_proj	api
decoder.layers.0._checkpoint_wrapped_module.mlp	style
decoder.layers.0._checkpoint_wrapped_module.mlp.experts	api
decoder.layers.0._checkpoint_wrapped_module.mlp.router.gate	api
decoder.layers.0._checkpoint_wrapped_module.sa_norm	api
decoder.layers.0._checkpoint_wrapped_module.mlp_norm	api
decoder.norm	api
decoder.output	api

Table 6: Forward Hook function Source

In the table, FSDP refers to Fully Sharded Data Parallel; Style refers to the parallelism strategy, typically defined in PrepareModuleInput; API refers to the distribute_module function provided by PyTorch. FSDP is applied during the data parallel (DP) sharding stage, while both style and PrepareModuleInput are defined during the tensor parallel (TP) planning stage.

Table 7 lists post forward hook for each module for information purpose.

Module Name	**Function Name**	**Source**
	_post_forward	FSDP
decoder.layers.0	_post_forward	FSDP
decoder.layers.0._checkpoint_wrapped_module.attn.q_proj	lambda	API
decoder.layers.0._checkpoint_wrapped_module.attn.k_proj	lambda	API
decoder.layers.0._checkpoint_wrapped_module.attn.v_proj	lambda	API
decoder.layers.0._checkpoint_wrapped_module.attn.output_proj	lambda	API
decoder.layers.0._checkpoint_wrapped_module.mlp	lambda	Style
decoder.layers.0._checkpoint_wrapped_module.mlp.experts	lambda	API
decoder.layers.0._checkpoint_wrapped_module.mlp.router.gate	lambda	API
decoder.layers.0._checkpoint_wrapped_module.sa_norm	lambda	API
decoder.layers.0._checkpoint_wrapped_module.mlp_norm	lambda	API
decoder.norm	lambda	API
decoder.output	lambda	API

Table 7: Post Forward Hooks in Modules

5 Dtensor

DTensor is a fundamental building block for distributed and parallel training in PyTorch. It serves as the bridge between standard tensors and distributed/parallel computation environments. Important operations related to Dtensors

- Replicate: replicates the tensor across all participating devices. Each device holds a complete copy of the tensor. This is commonly used in data parallelism
- Shard(0): splits (or shards) the tensor along dimension 0 across devices.
- Shard(1): splits (or shards) the tensor along dimension 1 across devices.

The operations described above are extensively used in the prepare_input, prepare_output, and partition functions to manage and distribute tensors appropriately across devices.

6 Summary

In this write-up, we illustrate the end-to-end process of LLaMA 4 in the PyTorch environment—from model definition to the application of tensor parallelism and data parallelism—highlighting how parameters and modules are transformed throughout. We also explore how PyTorch implements these parallelism strategies at a high level, including key components such as DTensor and forward hook functions