DOCUMENT

Tensor Parallelism for Multi-GPU Transformer Inference

Source: temp_input_17d68ee69aa24e07b8e372ff8a3aa9da.md

Contents

Estimated reading time: 16 min

Introduction

Running large language models, such as a 70-billion parameter model, on a single GPU is often infeasible due to memory limitations. Even a high-end H100 GPU with 80GB of VRAM cannot accommodate a model like Llama-2-70B in full precision. Tensor Parallelism (TP) addresses this challenge by splitting a model's weight matrices across multiple GPUs, enabling the execution of models that would otherwise be too large. This guide provides a hands-on approach to understanding and implementing tensor-parallel inference, covering the underlying theory before diving into practical code examples for running inference on cloud platforms like RunPod and Lambda Cloud.

1. Introduction

This guide focuses on the practical application of Tensor Parallelism for large model inference. We will cover the necessary theoretical concepts to understand its mechanics and then proceed with implementation details. By the conclusion, you will have functional scripts for executing tensor-parallel inference on multi-GPU environments.

2. The Need for Tensor Parallelism

The primary driver for using parallelism strategies is the immense size of modern large language models, which creates a significant memory bottleneck.

2.1 The Memory Wall Problem with Large Models

Modern LLMs require substantial GPU memory, often exceeding the capacity of a single device. The memory footprint grows directly with the number of parameters and the precision used (e.g., FP16 or FP32).

Model	Parameters	FP16 Memory	FP32 Memory
Llama-3-8B	8B	~16 GB	~32 GB
Llama-3-70B	70B	~140 GB	~280 GB
Llama-3-405B	405B	~810 GB	~1.6 TB

As the table illustrates, a single A100 GPU with 80GB of VRAM can barely hold the Llama-3-70B model in FP16, and this does not account for the additional memory required for the KV cache, activations, and batch size overhead. To run larger models, it is essential to distribute the model across multiple GPUs.

2.2 Comparing Parallelism Strategies

Several strategies exist for distributing computational workloads across multiple GPUs, each with distinct characteristics regarding what is split, memory usage, and communication patterns.

Strategy	What's Split	Memory per GPU	Communication Overhead
Data Parallelism	Data batches	Full model on each GPU	Gradient sync after each step
Pipeline Parallelism	Layers	Subset of layers	Activations between stages
Tensor Parallelism	Weight matrices	Slice of every layer	All-reduce within each layer

Tensor Parallelism is the preferred strategy when the model is too large to fit on a single GPU, when fast interconnects like NVLink or InfiniBand are available, and when the primary goal is to minimize inference latency.

3. How Tensor Parallelism Works

The fundamental concept behind Tensor Parallelism is that matrix multiplications, which are core operations in Transformers, can be parallelized by splitting the matrices themselves across different processing units.

3.1 Column-Parallel Matrix Multiplication

Consider the computation `Y = XW`, where `X` is the input and `W` is a weight matrix. In a column-parallel approach, the weight matrix `W` is split into column blocks. Each GPU computes its portion of the output by multiplying the full input `X` with its assigned column slice of `W`. The resulting outputs `Y_i` are also sharded by columns. This step can be performed independently on each GPU without any inter-GPU communication.

3.2 Row-Parallel Matrix Multiplication

Now, if the input `X` is already sharded by columns, as in `X = [X_1 | X_2 | ... | X_n]`, and the weight matrix `W` is split into corresponding row blocks, each GPU can compute a partial result. To obtain the final output `Y`, the partial results from all GPUs must be summed together. This is achieved through an **all-reduce** communication operation, which is a key step requiring GPU-to-GPU data transfer.

4. Applying TP in Transformer Layers

Tensor Parallelism can be systematically applied to the main components of a Transformer layer: the attention mechanism and the feed-forward network.

4.1 Tensor Parallelism in the Attention Layer

The attention mechanism involves three projection matrices for queries (`W_Q`), keys (`W_K`), and values (`W_V`), along with an output projection matrix (`W_O`). TP is applied in three steps.

Step 1: Split Q, K, V Projections (Column-Parallel)

The `W_Q`, `W_K`, and `W_V` weight matrices are split by columns. This is equivalent to distributing the attention heads across the available GPUs. For instance, with 32 attention heads and 4 GPUs, each GPU would handle 8 heads. Each GPU computes its slice independently.

```
1 ■ GPU i computes (column slices of weight matrices):
2 ■ Q_i = X × W_Q[all_rows, columns_for_heads_i]
3 ■ K_i = X × W_K[all_rows, columns_for_heads_i]
4 ■ V_i = X × W_V[all_rows, columns_for_heads_i]
```

This stage requires no communication.

Step 2: Local Attention Computation

Since attention heads operate independently, each GPU can compute the attention mechanism for its assigned heads locally without communicating with other GPUs.

```
GPU i computes attention locally:
attn_i = softmax(Q_i × K_i^T / √d_k) × V_i
```

Step 3: Output Projection (Row-Parallel)

The output projection matrix `W_O` is split by rows. Each GPU multiplies its local attention output by its slice of `W_O` to get a partial result. An all-reduce operation is then performed to sum these partial results into the final output.

```
GPU i computes partial output (row slice of W_O):
partial_i = attn_i × W_O[rows_for_gpu_i, all_cols]

All GPUs synchronize:
output = AllReduce(partial_0 + partial_1 + ... + partial_n)
```

This process results in one all-reduce operation per attention layer.

4.2 Tensor Parallelism in the Feed-Forward Network

The feed-forward network (FFN) in a Transformer typically consists of two linear layers. TP is applied to these layers as follows:

First Linear Layer (Column-Parallel)

The first linear layer's weight matrix, `W1`, is split by columns. Each GPU computes a partial hidden state independently.

```
GPU i computes:
hidden_i = GELU(x × W1[all_rows, cols_for_gpu_i])
```

Second Linear Layer (Row-Parallel)

The second linear layer's weight matrix, `W2`, is split by rows. Each GPU computes a partial output using its hidden state slice. These partial outputs are then summed across all GPUs using an all-reduce operation.

```
GPU i computes:
partial_i = hidden_i × W2[rows_for_gpu_i, all_cols]

All GPUs synchronize:
output = AllReduce(sum of all partial_i)
```

This adds a second all-reduce operation, bringing the total to two all-reduce operations per complete Transformer layer.

4.3 Communication and Practical Constraints

In total, a standard Transformer layer requires **two all-reduce operations** when using Tensor Parallelism. However, there are practical constraints to consider:

1. The TP size (number of GPUs) cannot exceed the number of attention heads.

2. The number of attention heads must be divisible by the TP size to ensure an equal workload on each GPU.

3. The FFN hidden dimension must also be divisible by the TP size.

For example, the Llama-3-70B model has 64 attention heads. Therefore, valid TP sizes are 1, 2, 4, 8, 16, 32, and 64.

5. Implementing TP with HuggingFace Transformers

The HuggingFace `transformers` library offers built-in support for Tensor Parallelism, simplifying its implementation for many models to just a single parameter.

5.1 A Simple Code Implementation

For supported models, enabling Tensor Parallelism can be done by setting the `tp_plan` argument to `"auto"` when loading the model. This instructs the library to automatically partition the model's weights across the available GPUs.

```
# tp_inference.py
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained(
    &amp;quot;meta-llama/Meta-Llama-3-8B-Instruct&amp;quot;,
    torch_dtype=torch.bfloat16,
    tp_plan=&amp;quot;auto&amp;quot;  # &amp;lt;-- This enables tensor parallelism
)

tokenizer = AutoTokenizer.from_pretrained(&amp;quot;meta-llama/Meta-Llama-3-8B-Instruct&amp;quot;)

prompt = &amp;quot;Explain tensor parallelism in one paragraph:&amp;quot;
inputs = tokenizer(prompt, return_tensors=&amp;quot;pt&amp;quot;).to(model.device)

outputs = model.generate(**inputs, max_new_tokens=100)

print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

5.2 Launching with torchrun

A script using Tensor Parallelism cannot be run directly with `python`. Instead, it must be launched using `torchrun`, which spawns multiple processes and manages the distributed environment. Each process is assigned to a separate GPU.

To run the script on 4 GPUs:

```
torchrun --nproc-per-node 4 tp_inference.py
```

5.3 Supported Models and Partitioning Strategies

As of late 2025, HuggingFace provides native TP support for a growing list of models, including Llama, Mistral, Mixtral, Qwen, and Gemma. You can check if a model is supported by inspecting its configuration for a `_tp_plan` attribute.

```
1 ■ from transformers import AutoConfig
2 ■
3 ■ config = AutoConfig.from_pretrained(&amp;quot;meta-llama/Meta-Llama-3-8B-Instruct&amp;quot;)
4 ■ print(config._tp_plan)  # Shows the default TP plan
```

Internally, HuggingFace uses several partitioning strategies:

Strategy	Description
`colwise`	Column-parallel (for Q, K, V projections)
`rowwise`	Row-parallel (for output projections)
`sequence_parallel`	For LayerNorm, Dropout
`replicate`	Keep full copy on each GPU

For advanced use cases, you can define a custom `tp_plan` dictionary to specify how each layer should be partitioned.

```
tp_plan = {
    &quot;model.layers.*.self_attn.q_proj&quot;: &quot;colwise&quot;,
    &quot;model.layers.*.self_attn.k_proj&quot;: &quot;colwise&quot;,
    &quot;model.layers.*.self_attn.v_proj&quot;: &quot;colwise&quot;,
    &quot;model.layers.*.self_attn.o_proj&quot;: &quot;rowwise&quot;,
    &quot;model.layers.*.mlp.gate_proj&quot;: &quot;colwise&quot;,
    &quot;model.layers.*.mlp.up_proj&quot;: &quot;colwise&quot;,
    &quot;model.layers.*.mlp.down_proj&quot;: &quot;rowwise&quot;,
}

model = AutoModelForCausalLM.from_pretrained(
    &quot;meta-llama/Meta-Llama-3-8B-Instruct&quot;,
    torch_dtype=torch.bfloat16,
    tp_plan=tp_plan
)
```

6. Hands-On Guide: Tensor Parallelism on RunPod

RunPod provides on-demand multi-GPU instances suitable for running large models with Tensor Parallelism. This section outlines the steps to run Llama-3-70B on a RunPod pod.

6.1 Setting Up a Multi-GPU Pod

First, deploy a new pod on RunPod:

1. Navigate to **RunPod** → **Pods** → **Deploy**.

2. Select a suitable PyTorch template, such as `runpod/pytorch:2.1.0-py3.10-cuda11.8.0`.

3. Choose a multi-GPU configuration, like 4x A100 80GB for Llama-3-70B or 8x H100 for larger models.

Critical: It is essential to select instances with **NVLink** interconnects (e.g., SXM variants like A100-SXM or H100-SXM) rather than PCIe. NVLink offers 600-900 GB/s

of inter-GPU bandwidth, whereas PCIe is limited to approximately 64 GB/s. The high communication cost of `all-reduce` operations makes NVLink crucial for TP performance; without it, the communication overhead becomes a severe bottleneck and negates performance gains.

6.2 Environment Setup and Inference Script

Once connected to the pod via SSH, set up the environment by installing the necessary libraries.

```
1 ■ # Update and install dependencies
2 ■ pip install --upgrade transformers accelerate torch
3 ■
4 ■ # Verify GPU setup
5 ■ nvidia-smi
6 ■
7 ■ # Check NCCL (the communication backend)
8 ■ python -c &quot;import torch; print(f&#x27;CUDA available: {torch.cuda.is_available()}\nGPU c
```

The expected output should confirm CUDA availability and list the correct number of GPUs. Next, create the inference script:

```
# runpod_tp_inference.py
import os
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

def main():
    model_id = &quot;meta-llama/Meta-Llama-3-70B-Instruct&quot;

    # Load model with tensor parallelism
    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        torch_dtype=torch.bfloat16,
        tp_plan=&quot;auto&quot;
    )

    tokenizer = AutoTokenizer.from_pretrained(model_id)
    tokenizer.pad_token = tokenizer.eos_token

    # Only rank 0 should print
    rank = int(os.environ.get(&quot;RANK&quot;, 0))

    prompts = [
        &quot;What is tensor parallelism?&quot;,
        &quot;Explain the difference between data and model parallelism.&quot;,
    ]

    for prompt in prompts:
        inputs = tokenizer(prompt, return_tensors=&quot;pt&quot;).to(model.device)
        with torch.no_grad():
            outputs = model.generate(
                **inputs,
                max_new_tokens=200,
                do_sample=True,
                temperature=0.7,
                top_p=0.9,
            )

        if rank == 0:
            response = tokenizer.decode(outputs[0], skip_special_tokens=True)
            print(f&quot;\n{&#x27;=&#x27;*50}&quot;)
            print(f&quot;Prompt: {prompt}&quot;)
            print(f&quot;Response: {response}&quot;)
            print(f&quot;{&#x27;=&#x27;*50}\n&quot;)

if __name__ == &quot;__main__&quot;:
    main()
```

6.3 Launching the Inference Job

To run the script, first set your HuggingFace token as an environment variable, then use `torchrun` to launch the job on all available GPUs.

```
# Set your HuggingFace token for gated models
export HF_TOKEN=&amp;quot;your_token_here&amp;quot;

# Launch on 4 GPUs
torchrun --nproc-per-node 4 runpod_tp_inference.py
```

6.4 Using vLLM with Tensor Parallelism

For production-level inference, vLLM is often a more performant option. To use vLLM with tensor parallelism, you can run its OpenAI-compatible API server:

```
# Install vLLM
pip install vllm

# Run with tensor parallelism
python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Meta-Llama-3-70B-Instruct \
    --tensor-parallel-size 4 \
    --dtype bfloat16 \
    --port 8000
```

Alternatively, RunPod's serverless platform can automatically handle vLLM and TP configuration through a handler.

```
# In your RunPod serverless handler
handler_config = {
    &amp;quot;model_name&amp;quot;: &amp;quot;meta-llama/Meta-Llama-3-70B-Instruct&amp;quot;,
    &amp;quot;tensor_parallel_size&amp;quot;: 4,
    &amp;quot;dtype&amp;quot;: &amp;quot;bfloat16&amp;quot;,
}
```

7. Hands-On Guide: Tensor Parallelism on Lambda Cloud

Lambda Cloud provides high-performance GPU instances, including configurations with up to 8x H100 GPUs, making it another excellent choice for large model inference.

7.1 Launching a Multi-GPU Instance

To start, launch a multi-GPU instance from the Lambda Cloud dashboard:

1. Navigate to **Lambda Cloud** → **Instances** → **Launch**.

2. Select an appropriate instance type, such as `gpu_8x_h100_sxm5` or `gpu_4x_a100_80gb_sxm4`.

Critical: As with RunPod, always choose **SXM** variants (e.g., `sxm4`, `sxm5`) over PCIe. The SXM designation confirms that the GPUs are interconnected with high-bandwidth NVLink (600-900 GB/s), which is essential for efficient tensor parallelism. PCIe-based instances are limited to the CPU's PCIe lane bandwidth (~64 GB/s), creating a bottleneck that severely degrades performance.

7.2 Creating and Launching the Inference Script

After SSHing into the instance and completing the environment setup, create the following script, which includes simple benchmarking to measure performance.

```
# lambda_tp_inference.py
import os
import time
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

def main():
    model_id = &amp;quot;meta-llama/Meta-Llama-3-70B-Instruct&amp;quot;
    rank = int(os.environ.get(&amp;quot;RANK&amp;quot;, 0))
    world_size = int(os.environ.get(&amp;quot;WORLD_SIZE&amp;quot;, 1))

    if rank == 0:
        print(f&amp;quot;Loading {model_id} with TP across {world_size} GPUs...&amp;quot;)

    start_time = time.time()
    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        torch_dtype=torch.bfloat16,
        tp_plan=&amp;quot;auto&amp;quot;
    )
    if rank == 0:
        load_time = time.time() - start_time
        print(f&amp;quot;Model loaded in {load_time:.2f}s&amp;quot;)

    tokenizer = AutoTokenizer.from_pretrained(model_id)
    tokenizer.pad_token = tokenizer.eos_token

    # Benchmark inference
    prompt = &amp;quot;Write a short poem about distributed computing:&amp;quot;
    inputs = tokenizer(prompt, return_tensors=&amp;quot;pt&amp;quot;).to(model.device)

    # Warmup
    with torch.no_grad():
        _ = model.generate(**inputs, max_new_tokens=10)

    # Timed generation
    torch.cuda.synchronize()
    start = time.time()
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=100,
            do_sample=False,  # Greedy for reproducibility
        )
    torch.cuda.synchronize()
    gen_time = time.time() - start
```

```
47 ■
48 ■     if rank == 0:
49 ■         response = tokenizer.decode(outputs[0], skip_special_tokens=True)
50 ■         tokens_generated = outputs.shape[1] - inputs[&amp;quot;input_ids&amp;quot;].shape[1]
51 ■         tokens_per_sec = tokens_generated / gen_time
52 ■
53 ■         print(f&amp;quot;\nPrompt: {prompt}&amp;quot;)
54 ■         print(f&amp;quot;Response: {response}&amp;quot;)
55 ■         print(f&amp;quot;\n--- Performance ---&amp;quot;)
56 ■         print(f&amp;quot;Tokens generated: {tokens_generated}&amp;quot;)
57 ■         print(f&amp;quot;Time: {gen_time:.2f}s&amp;quot;)
58 ■         print(f&amp;quot;Throughput: {tokens_per_sec:.1f} tokens/sec&amp;quot;)
59 ■
60 ■ if __name__ == &amp;quot;__main__&amp;quot;:
61 ■     main()
```

Launch the script using `torchrun`, specifying the number of GPUs on the node.

```
1 ■ # For a single node with 4 GPUs
2 ■ torchrun --nproc-per-node 4 lambda_tp_inference.py
3 ■
4 ■ # For 8 GPUs
5 ■ torchrun --nproc-per-node 8 lambda_tp_inference.py
```

7.3 Multi-Node Setup for Large-Scale Inference

For models that require more than 8 GPUs, you can extend Tensor Parallelism across multiple nodes using `torchrun`'s multi-node capabilities. This requires a high-bandwidth interconnect like InfiniBand between nodes to be effective.

On the master node (Node 0):

```
1 ■ torchrun \
2 ■     --nproc-per-node 8 \
3 ■     --nnodes 2 \
4 ■     --node-rank 0 \
5 ■     --master-addr &amp;lt;master-ip&amp;gt; \
6 ■     --master-port 29500 \
7 ■     lambda_tp_inference.py
```

On the worker node (Node 1):

```
torchrun \
    --nproc-per-node 8 \
    --nnodes 2 \
    --node-rank 1 \
    --master-addr &lt;master-ip&gt; \
    --master-port 29500 \
    lambda_tp_inference.py
```

This configuration creates a 16-GPU setup with tensor parallelism. However, be aware that cross-node communication overhead can significantly impact performance if the interconnect is not sufficiently fast.

8. Performance Benchmarks and Observations

Performance with Tensor Parallelism scales with the number of GPUs, but not perfectly due to communication overhead.

8.1 Llama-3-70B Inference Throughput

The following table shows approximate inference throughput for Llama-3-70B on different H100 GPU configurations.

Configuration	TP Size	Tokens/sec	Memory/GPU
1× H100 80GB	1	OOM	-
2× H100 80GB	2	~45	~38 GB
4× H100 80GB	4	~85	~20 GB
8× H100 80GB	8	~140	~12 GB

8.2 Key Performance Scaling Observations

Based on these benchmarks, several key patterns emerge:

1. **Memory scales linearly**: Using four times the number of GPUs reduces the memory requirement per GPU by approximately four times.

2. **Throughput scales sub-linearly**: Performance gains diminish as the TP size increases because the communication overhead from `all-reduce` operations becomes more significant.

3. **There is a sweet spot**: For many models, the optimal balance between performance and cost is often found with a TP size of 4 to 8 GPUs. Beyond this, communication can start to dominate computation time.

9. Limitations and Advanced Strategies

While powerful, Tensor Parallelism has inherent limitations and is often combined with other strategies for very large-scale systems.

9.1 What Tensor Parallelism Does Not Solve

Tensor Parallelism has three main limitations:

1. **Scalability is Capped by Attention Heads**: The TP size cannot be larger than the model's number of attention heads. For a model with 64 heads, you cannot use more than 64 GPUs for TP.

2. **Communication Overhead Across Nodes**: TP requires frequent communication. This is fast within a node using NVLink (900 GB/s), but becomes a bottleneck across nodes connected by InfiniBand (~400 GB/s) or Ethernet (~100 Gbps).

3. **Activation Memory**: TP primarily reduces the memory required for model weights, but it does not reduce the memory needed for activations. For very long sequences, activation memory can still become a bottleneck.

9.2 Combining with Pipeline Parallelism

For extremely large models (400B+ parameters), a hybrid approach combining Tensor Parallelism (TP) and Pipeline Parallelism (PP) is common. The general rule of thumb is to use TP within a single node to leverage fast NVLink interconnects and use PP across different nodes.

For example, to run a model on 32 GPUs, you could use an 8-way TP within each of four nodes, and a 4-way PP across the nodes:

```
1 ■ Node 0: Layers 0-19  (TP=8 within node)
2 ■ Node 1: Layers 20-39 (TP=8 within node)
3 ■ Node 2: Layers 40-59 (TP=8 within node)
4 ■ Node 3: Layers 60-79 (TP=8 within node)
```

This hybrid strategy effectively scales to a large number of GPUs by minimizing slow cross-node communication.

10. Key Takeaways

Tensor parallelism is an essential technique for running large language models that do not fit on a single GPU. The core principles involve splitting the model's weight matrices across multiple GPUs—using a column-wise split for projection layers and a row-wise split for output layers—and using all-reduce operations to aggregate partial results. A standard Transformer layer requires two such communication steps. For optimal performance, it is best to confine Tensor Parallelism within a single, NVLink-connected node. The HuggingFace `transformers` library simplifies

implementation with its `tp_plan="auto"` feature. For production environments, frameworks like vLLM provide highly optimized TP implementations, while for training, FSDP (Fully Sharded Data Parallel) offers an alternative that combines elements of both tensor and data parallelism.

Key Takeaways

Tensor Parallelism is the go-to technique for running models that don’t fit on a single GPU. The key ideas are splitting weight matrices across GPUs (column-wise for projections, row-wise for outputs) and using all-reduce to aggregate partial results, with two such operations per transformer layer. For best performance, TP should be kept within a single node with fast interconnects like NVLink. For ease of use, HuggingFace's `tp_plan="auto"` is a straightforward solution, while specialized frameworks like vLLM are recommended for production inference.