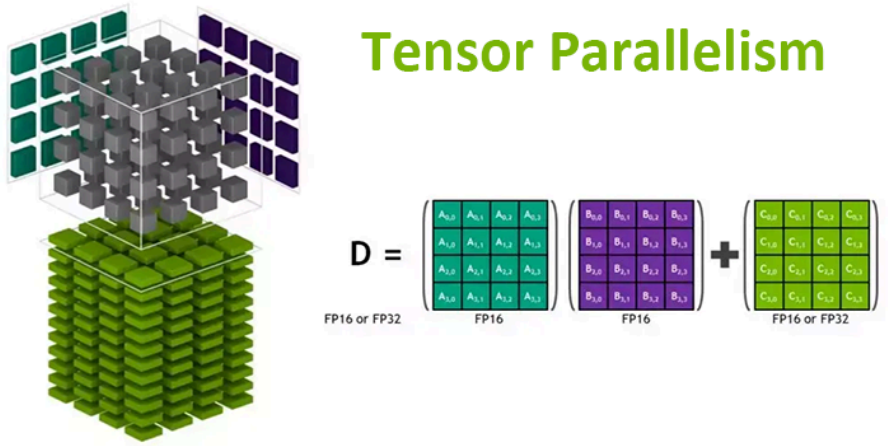
Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

Tensor Parallelism in Transformers: A Hands-On Guide for Multi-GPU Inference

Posted Dec 5, 2025 • Updated Dec 9, 2025



By Gian Paolo Santopaolo

15 min read

Contents ›

Running a 70B parameter model on a single GPU? Not happening. Even the beefiest H100 with 80GB of VRAM can't hold Llama-2-70B in full precision. This is where **Tensor Parallelism (TP)** comes in — it splits the model's weight matrices across multiple GPUs so you can run models that would otherwise be impossible.

This guide is hands-on. We'll cover the theory just enough to understand what's happening, then dive straight into code. By the end, you'll have working scripts for running tensor-parallel inference on **RunPod** and **Lambda Cloud**.

Why Tensor Parallelism? The Memory Wall Problem

Modern LLMs are massive. Here's a quick reality check:

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

A single A100 (80GB) can barely fit Llama-3-70B in FP16 — and that's before accounting for KV cache, activations, and batch size overhead. For anything larger, you need to split the model across GPUs.

The Parallelism Zoo

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

There are several ways to distribute work across GPUs:

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

When to use Tensor Parallelism:

- Model doesn't fit on a single GPU
- You have fast interconnects (NVLink, InfiniBand)
- You want to minimize latency for inference

How Tensor Parallelism Works

The core insight is simple: **matrix multiplications can be parallelized by splitting the matrices**.

Column-Parallel Matrix Multiplication

Suppose you need to compute $Y = XW$ where X is your input and W is a weight matrix. If you split W into column blocks:

$$W = [W_1 \mid W_2 \mid \ldots \mid W_n]$$

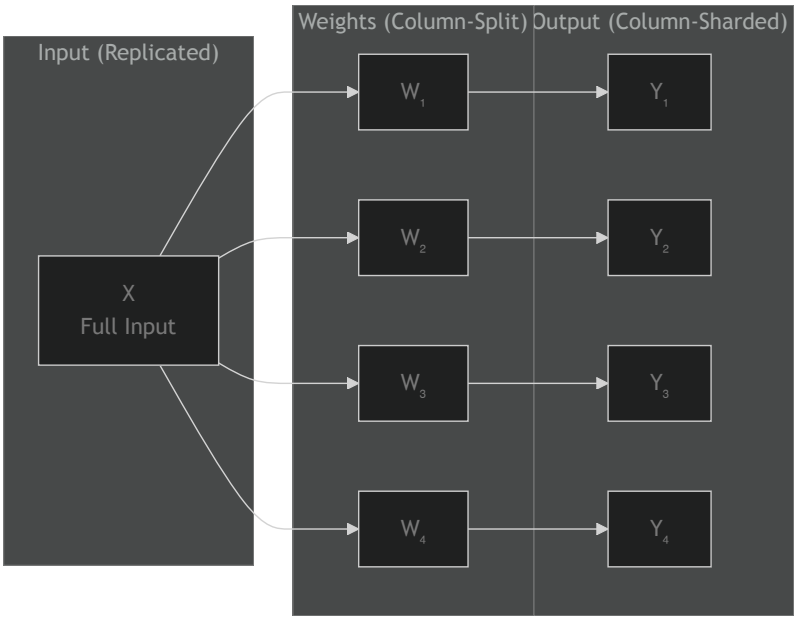
Then each GPU computes its slice independently:

$$Y_i = X \cdot W_i$$

The outputs are naturally sharded by columns — no communication needed yet.

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact



Row-Parallel Matrix Multiplication

Now suppose you have column-sharded input $X = [X_1 \mid X_2 \mid \ldots \mid X_n]$ and you split W into matching row blocks:

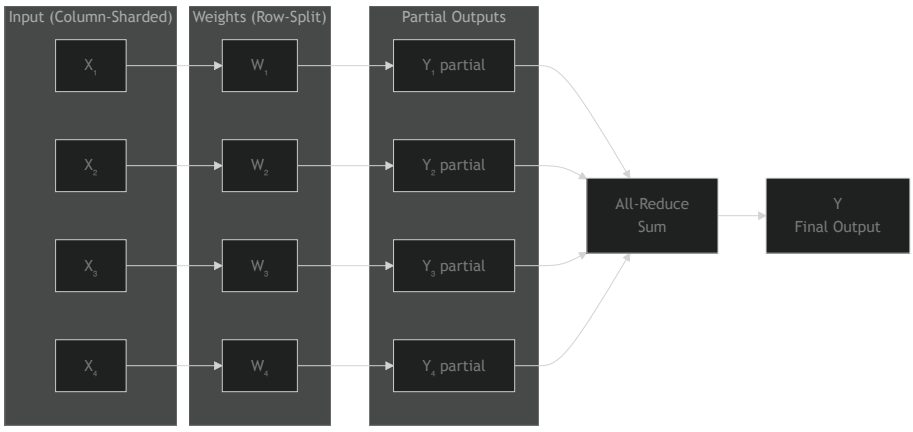
$$W = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix}$$

Each GPU computes a partial result:

$$Y_i = X_i \cdot W_i$$

Then you **sum across GPUs** (all-reduce) to get the final output:

$$Y = \sum_i Y_i$$



This is the key operation that requires GPU-to-GPU communication.

TP in Transformer Layers

Now let's see how these primitives apply to actual Transformer components.

Attention Layer

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

The attention mechanism has three projection matrices: W_Q, W_K, W_V (queries, keys, values) and an output projection W_O.

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

Each GPU gets a subset of attention heads. If you have 32 heads and 4 GPUs, each GPU handles 8 heads.

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

No communication needed — each GPU works independently.

Step 2: Local Attention Computation

Since attention heads are independent, each GPU computes attention for its heads locally:

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

Still no communication.

Step 3: Output Projection (Row-Parallel)

The output projection W_O is split by rows. Each GPU multiplies its attention output by its slice of W_O, then we all-reduce:

```
Text
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)
```

One all-reduce per attention layer.

Feed-Forward Network (FFN)

The FFN typically has two linear layers with an activation in between:

$$\text{FFN}(x) = \text{GELU}(xW_1)W_2$$

First Linear (Column-Parallel):

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

Second Linear (Row-Parallel):

```
Text
```

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


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

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

One all-reduce per FFN layer.

The Full Picture

Total communication per layer: 2 all-reduce operations.

Constraints

TP comes with a few practical constraints:

1. **TP size ≤ number of attention heads** — you can't split a single head across GPUs
2. **Heads must be divisible by TP size** — each GPU needs an equal share
3. **FFN hidden dimension must be divisible by TP size**

For Llama-3-70B with 64 heads, valid TP sizes are: 1, 2, 4, 8, 16, 32, 64.

Tensor Parallelism with HuggingFace Transformers

The good news: HuggingFace Transformers now has built-in TP support. For supported models, it's a one-liner.

The 3-Line Solution

Python

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

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

tokenizer = AutoTokenizer.from_pretrained("meta-llama/Meta-Llama-3-8B-Ins

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

outputs = model.generate(**inputs, max_new_tokens=100)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Launching with torchrun

You can't just run `python tp_inference.py`. You need to launch it with `torchrun` to spawn multiple processes:

Shell

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


```
# Run on 4 GPUs
torchrun --nproc-per-node 4 tp_inference.py
```

Each process gets assigned to one GPU, and PyTorch's distributed runtime handles the communication.

Supported Models

As of late 2025, HuggingFace supports TP for:

- Llama (all versions)
- Mistral
- Mixtral
- Qwen
- Gemma
- And more…

Check the model's config for `_tp_plan` to see if it's supported:

```python
from transformers import AutoConfig
config = AutoConfig.from_pretrained("meta-llama/Meta-Llama-3-8B-Instruct")
print(config._tp_plan)  # Shows the default TP plan
```

Partitioning Strategies

Under the hood, HuggingFace uses these 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

You can define a custom `tp_plan` if needed:

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

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

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

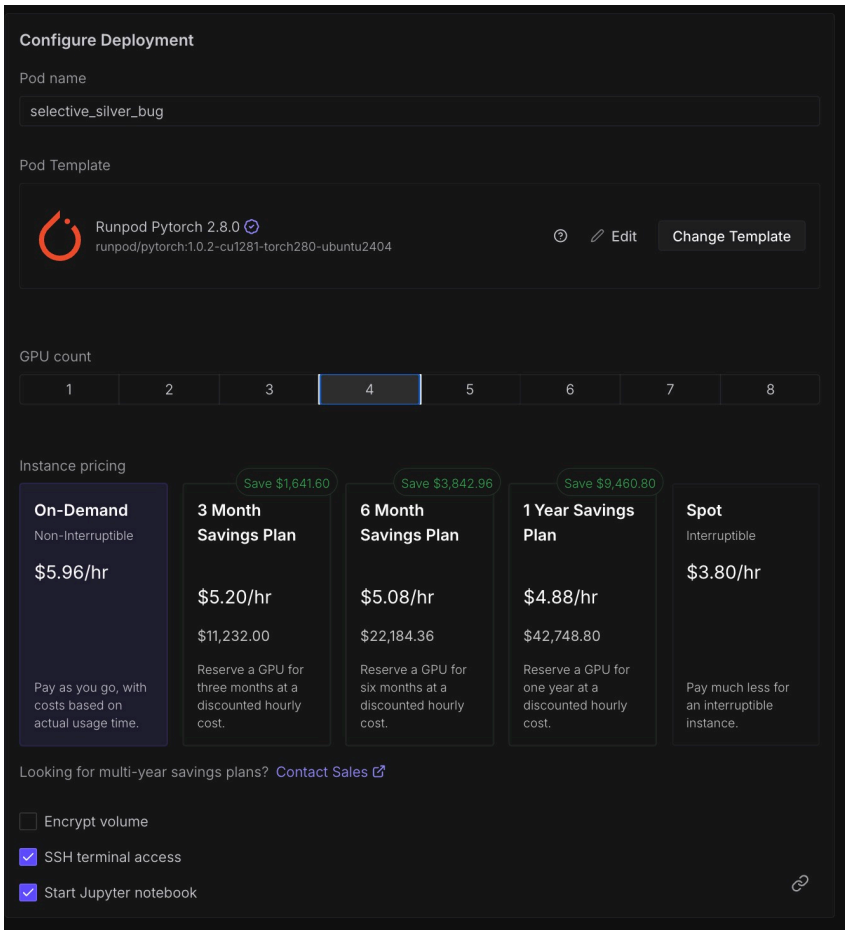
Hands-On: Running TP on RunPod

RunPod offers on-demand GPU pods with multi-GPU configurations. Let's run Llama-3-70B with tensor parallelism.

Step 1: Spin Up a Multi-GPU Pod

1. Go to RunPod → Pods → Deploy
2. Select a template with PyTorch (e.g., `runpod/pytorch:2.1.0-py3.10-cuda11.8.0`)
3. Choose a multi-GPU configuration:
   - **4× A100 80GB** for Llama-3-70B
   - **8× H100** for larger models or faster inference

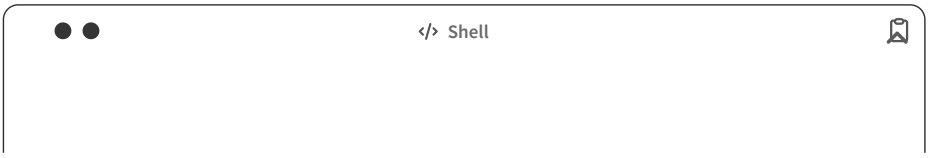
> **Critical:** Select instances with **NVLink** interconnect (e.g., SXM variants like A100-SXM or H100-SXM), not PCIe. NVLink provides 600-900 GB/s bandwidth between GPUs, while PCIe is limited to ~64 GB/s. Without NVLink, the all-reduce operations in tensor parallelism become a severe bottleneck, negating most of the performance gains.



Selecting a 4×A100 pod on RunPod — look for SXM variants with NVLink

Step 2: Environment Setup

SSH into your pod and set up the environment:



Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


```
# Update and install dependencies
pip install --upgrade transformers accelerate torch

# Verify GPU setup
nvidia-smi

# Check NCCL (the communication backend)
python -c "import torch; print(f'CUDA available: {torch.cuda.is_available(
```

Expected output:

```
CUDA available: True
GPU count: 4
```

Step 3: Create the Inference Script

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


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

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

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

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

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

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

    for prompt in prompts:
        inputs = tokenizer(prompt, return_tensors="pt").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=T
            print(f"\n{'='*50}")
            print(f"Prompt: {prompt}")
            print(f"Response: {response}")
            print(f"{'='*50}\n")

if __name__ == "__main__":
    main()
```

Step 4: Launch with torchrun

```shell
# Set your HuggingFace token for gated models
export HF_TOKEN="your_token_here"

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

Using vLLM with Tensor Parallelism on RunPod

For production inference, vLLM is often faster. RunPod has native vLLM support:

```shell
```

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


```
# 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
```

Or use RunPod's serverless vLLM workers which handle TP automatically:

Python

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

Hands-On: Running TP on Lambda Cloud

Lambda Cloud offers GPU instances with up to 8× H100s. The setup is similar but with some Lambda-specific details.

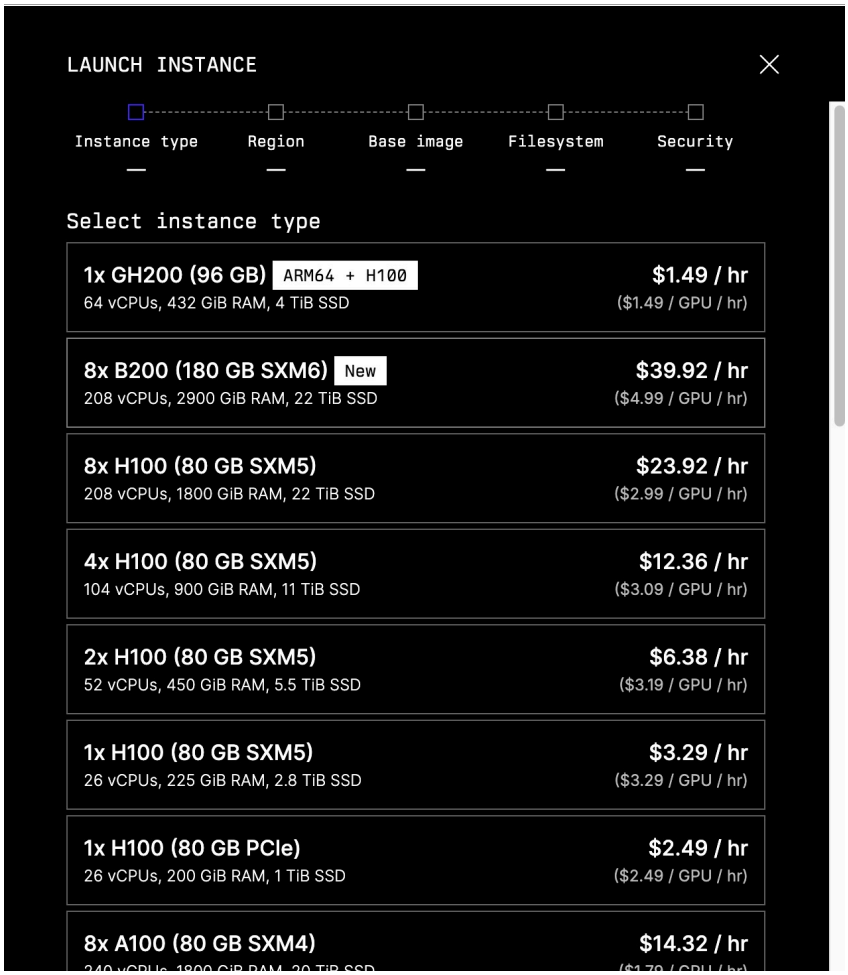
Step 1: Launch a Multi-GPU Instance

1. Go to Lambda Cloud → Instances → Launch
2. Select instance type:
   - **gpu_8x_h100_sxm5** (8× H100 80GB) — best for large models
   - **gpu_4x_a100_80gb_sxm4** (4× A100 80GB) — good for 70B models

> **Critical:** Always choose **SXM** variants (e.g., `sxm4`, `sxm5`) over PCIe. The "SXM" designation indicates GPUs connected via **NVLink** with 600-900 GB/s inter-GPU bandwidth. PCIe-based instances share bandwidth through the CPU's PCIe lanes (~64 GB/s), creating a communication bottleneck that cripples tensor parallelism performance.

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact



Selecting a multi-GPU instance on Lambda Cloud — SXM variants have NVLink

Step 2: SSH and Setup

Shell

```shell
# SSH into your instance
ssh ubuntu@<your-instance-ip>

# Lambda instances come with PyTorch pre-installed
# Just update transformers
pip install --upgrade transformers accelerate

# Verify setup
python -c "import torch; print(f'GPUs: {torch.cuda.device_count()}')"
```

Step 3: Create the Inference Script

Python

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact


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

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

    rank = int(os.environ.get("RANK", 0))
    world_size = int(os.environ.get("WORLD_SIZE", 1))

    if rank == 0:
        print(f"Loading {model_id} with TP across {world_size} GPUs...")
        start_time = time.time()

    model = AutoModelForCausalLM.from_pretrained(
        model_id,
        torch_dtype=torch.bfloat16,
        tp_plan="auto"
    )

    if rank == 0:
        load_time = time.time() - start_time
        print(f"Model loaded in {load_time:.2f}s")

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

    # Benchmark inference
    prompt = "Write a short poem about distributed computing:"
    inputs = tokenizer(prompt, return_tensors="pt").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

    if rank == 0:
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        tokens_generated = outputs.shape[1] - inputs["input_ids"].shape[1
        tokens_per_sec = tokens_generated / gen_time

        print(f"\nPrompt: {prompt}")
        print(f"Response: {response}")
        print(f"\n--- Performance ---")
        print(f"Tokens generated: {tokens_generated}")
        print(f"Time: {gen_time:.2f}s")
        print(f"Throughput: {tokens_per_sec:.1f} tokens/sec")

if __name__ == "__main__":
    main()
```

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

Step 4: Launch with torchrun

```shell
# For a single node with 4 GPUs
torchrun --nproc-per-node 4 lambda_tp_inference.py

# For 8 GPUs
torchrun --nproc-per-node 8 lambda_tp_inference.py
```

Multi-Node Setup on Lambda Cloud

If you need more than 8 GPUs, you can run across multiple nodes. Lambda instances support this via `torchrun`:

```shell
# On Node 0 (master)
torchrun \
    --nproc-per-node 8 \
    --nnodes 2 \
    --node-rank 0 \
    --master-addr <master-ip> \
    --master-port 29500 \
    lambda_tp_inference.py

# On Node 1 (worker)
torchrun \
    --nproc-per-node 8 \
    --nnodes 2 \
    --node-rank 1 \
    --master-addr <master-ip> \
    --master-port 29500 \
    lambda_tp_inference.py
```

This gives you 16 GPUs with tensor parallelism across nodes.

> **Warning:** Cross-node TP requires high-bandwidth interconnects (InfiniBand). Without it, communication overhead can kill performance.

Performance Benchmarks

Here's what you can expect with tensor parallelism on different configurations:

Llama-3-70B Inference Throughput

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

Key Observations

1. **Memory scales linearly** — 4× GPUs = ~4× less memory per GPU

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

2. **Throughput scales sub-linearly** — communication overhead increases with TP size
3. **Sweet spot is often 4-8 GPUs** — beyond that, communication dominates

---

What TP Doesn't Solve

Tensor parallelism is powerful, but it has limitations:

1. Scalability is Capped by Attention Heads

If your model has 64 attention heads, TP size can't exceed 64. In practice, you want TP size much smaller than head count to maintain efficiency.

2. Communication Overhead Across Nodes

TP requires frequent all-reduce operations (2 per layer). Within a node with NVLink (900 GB/s), this is fast. Across nodes with InfiniBand (~400 GB/s) or worse, Ethernet (~100 Gbps), it becomes a bottleneck.

Rule of thumb: Keep TP within a single node. Use Pipeline Parallelism (PP) across nodes.

3. Doesn't Help with Activation Memory

TP reduces weight memory but not activation memory. For very long sequences, you may still need gradient checkpointing or other techniques.

When to Combine with Pipeline Parallelism

For truly massive models (400B+), combine TP and PP:

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

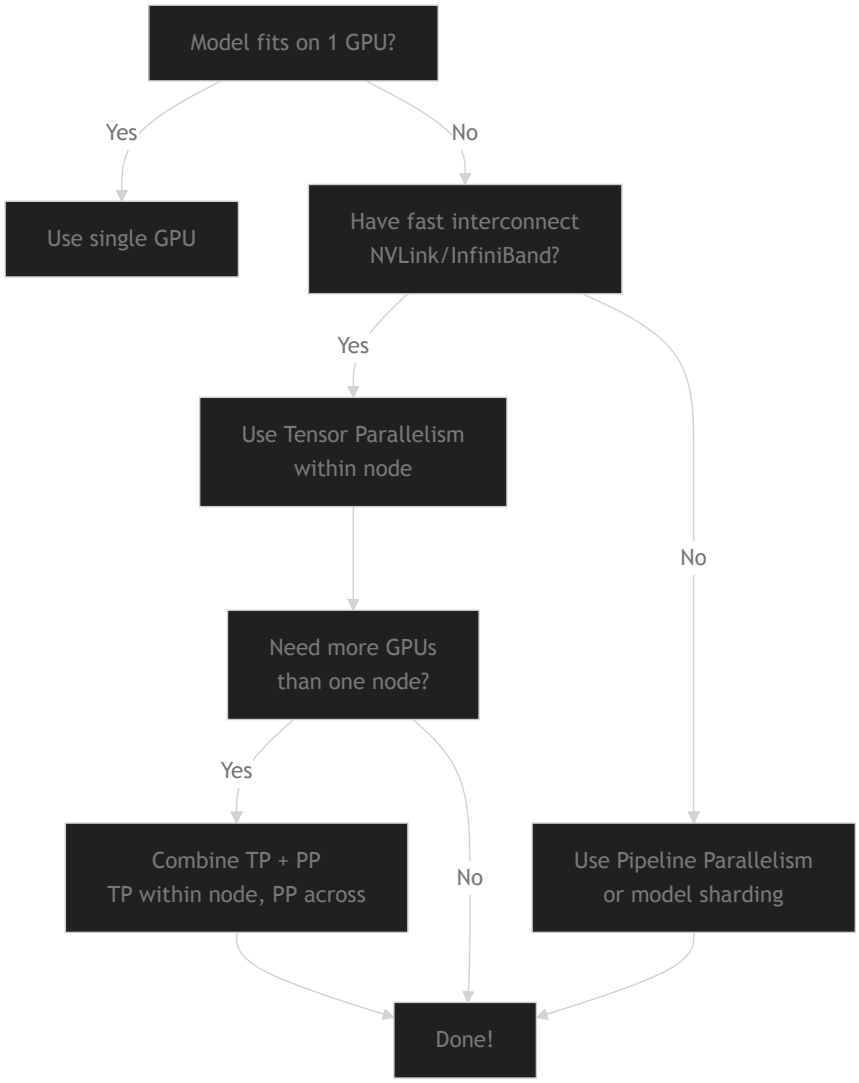
This gives you 32 GPUs total: 8-way TP × 4-way PP.

Practical Takeaways

Decision Tree: Which Parallelism Strategy?

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact



Quick Reference Commands

Shell

```shell
# HuggingFace Transformers with TP
torchrun --nproc-per-node 4 inference.py

# vLLM with TP
python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Meta-Llama-3-70B-Instruct \
    --tensor-parallel-size 4

# Check GPU topology (important for TP performance)
nvidia-smi topo -m

# Monitor GPU usage during inference
watch -n 0.5 nvidia-smi
```

Key Constraints Checklist

Before deploying with TP, verify:

- ◯ TP size ≤ number of attention heads
- ◯ Attention heads divisible by TP size
- ◯ FFN hidden dim divisible by TP size
- ◯ All GPUs have NVLink or fast interconnect
- ◯ Using `torchrun` or equivalent launcher

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

Wrapping Up

Tensor parallelism is the go-to technique for running models that don't fit on a single GPU. The key ideas:

1. **Split weight matrices** across GPUs (column-wise for projections, row-wise for outputs)
2. **All-reduce** to aggregate partial results (2× per transformer layer)
3. **Keep TP within a node** for best performance
4. **Use** `tp_plan="auto"` in HuggingFace for the easy path

For production inference, consider vLLM which has highly optimized TP implementations. For training, look into FSDP (Fully Sharded Data Parallel) which combines aspects of TP and data parallelism.

References

- HuggingFace Distributed Inference Documentation
- Megatron-LM: Training Multi-Billion Parameter Language Models
- RunPod Multi-GPU Training Guide
- Lambda Labs Multi-Node PyTorch Guide
- vLLM Documentation
- PyTorch Distributed Overview

Tensor Parallelism, Tensor, Transformers, Multi-GPU, PyTorch, RunPod, Lambda Cloud

Share:

Further Reading

Dec 2, 2025

What Is a Tensor? A Practical Guide for AI Engineers

When dealing with deep learning, we quickly encounter the word tensor. But what exactly i…

Dec 3, 2025

Why GPUs Love Tensors: Understanding Tensor Cores and…

Understanding why modern AI is so fast requires understanding the hardware that …

Dec 18, 2025

PyTorch Training Loop: A Production-Ready Template for …

You can train almost anything in PyTorch with seven lines of code. The problem? Those seve…

OLDER NEWER

Why GPUs Love Tensors: Understanding Tensor Cores and AI Acceleration

Choosing the Right Loss Function for your ML Problem

Gian Paolo Santopaolo

Hands-On ML Engineering Meets Strategic Vision—Bridging Code and Cloud to Drive Enterprise AI for Real-World Impact

HOME

POSTS

PROJECTS

ABOUT