DOCUMENT

Tensor Parallelism for Multi-GPU Inference

Source: temp_input_1720a8a0afb94ca5bd40610a356bf466.md

Contents

Estimated reading time: 19 min

Introduction

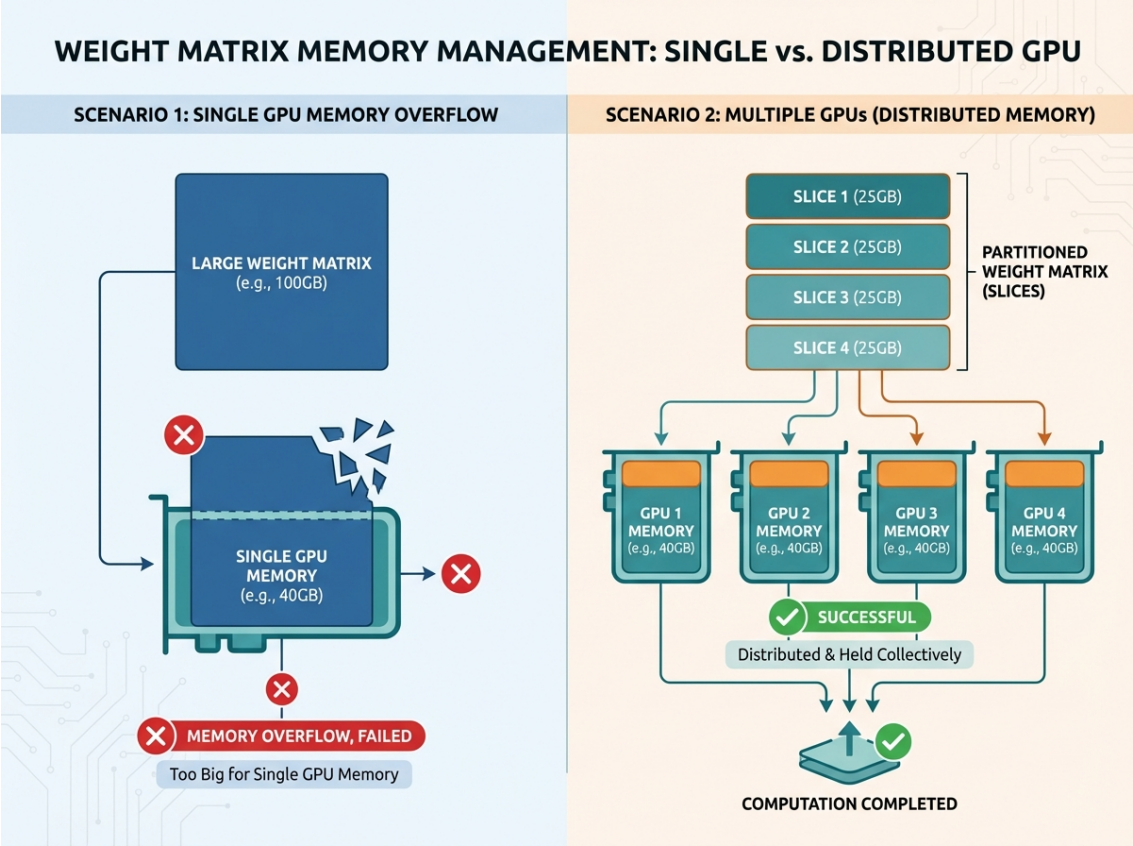

Figure 1: 1. Introduction

This diagram illustrates why Tensor Parallelism is crucial for running large models. A single GPU often lacks the memory to hold an entire model's weight matrix, resulting in an overflow error. By partitioning this large matrix into smaller slices and distributing them across multiple GPUs, Tensor Parallelism allows the devices to collectively hold and process the model successfully.

Running large language models, such as a 70-billion parameter model, on a single GPU is often impossible due to memory constraints. Even a high-end H100 GPU with 80GB of VRAM cannot accommodate a model like Llama-2-70B in full precision. This is the challenge where Tensor Parallelism (TP) provides a solution. By splitting a model's weight matrices across multiple GPUs, Tensor Parallelism enables the

execution of models that would otherwise be too large for a single device. This guide provides a hands-on approach to understanding and implementing Tensor Parallelism, covering the necessary theory before diving into practical code examples for running TP inference on platforms like RunPod and Lambda Cloud.

1. Introduction

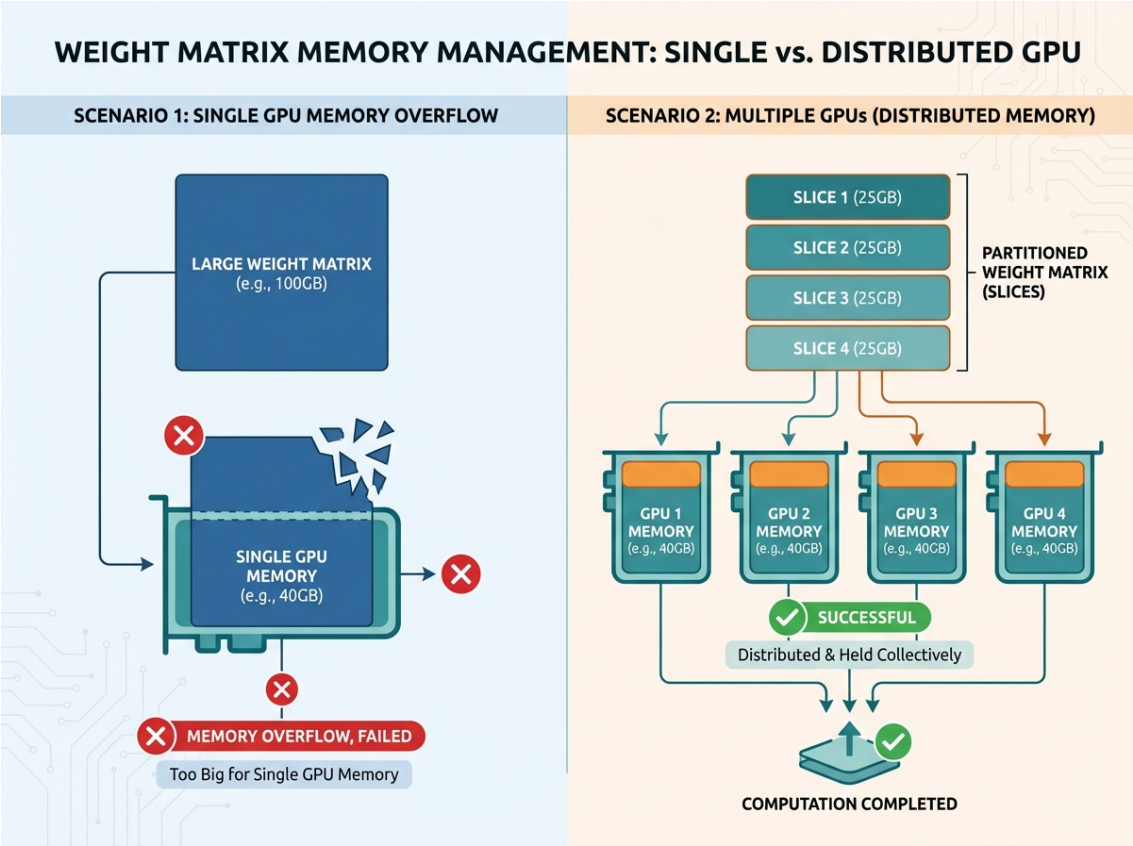

Figure 2: 1. Introduction

This diagram illustrates why Tensor Parallelism is crucial for running large models. A single GPU often lacks the memory to hold an entire model's weight matrix, resulting in an overflow error. By partitioning this large matrix into smaller slices and distributing them across multiple GPUs, Tensor Parallelism allows the devices to collectively hold and process the model successfully.

1.1 The Challenge of Running Large Models

Modern Large Language Models (LLMs) have grown to immense sizes, creating a significant memory bottleneck. For instance, a Llama-3-70B model requires approximately 140 GB of memory in FP16 precision and 280 GB in FP32. This far exceeds the capacity of even a single A100 GPU with 80GB of VRAM, and this calculation does not even account for the additional memory overhead from the KV cache, activations, and batch size. To run models of this scale or larger, it is necessary to distribute the model's components across multiple GPUs.

1.2 Introducing Tensor Parallelism (TP)

Tensor Parallelism (TP) is a model parallelism technique designed to address the memory limitations of single-GPU systems. The fundamental principle of TP is to partition the individual weight matrices of a model and distribute these slices across multiple GPUs. This allows a collective of GPUs to hold and process a model that would be too large for any single one. By parallelizing the computations within each layer, TP enables inference for massive models while aiming to minimize latency.

2. Understanding the Need for Parallelism

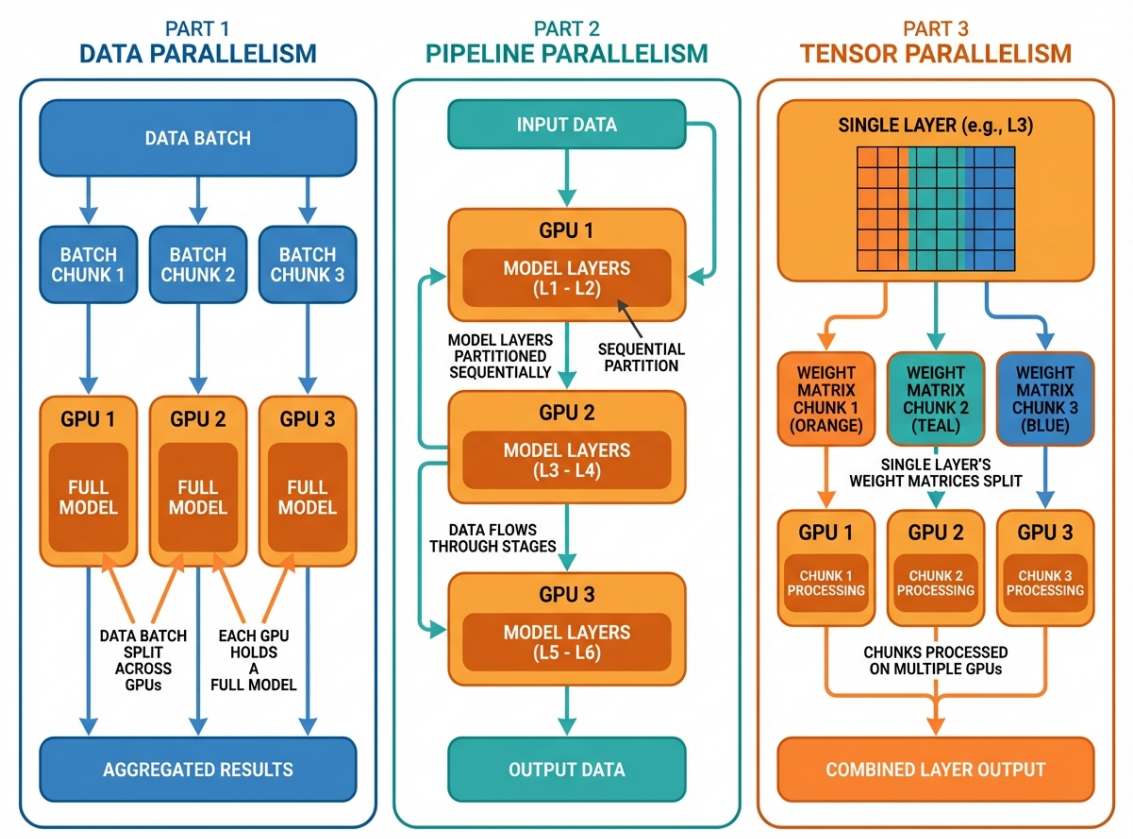

Figure 3: 2. Understanding the Need for Parallelism

This diagram visualizes three core strategies for running large models across multiple GPUs to overcome single-device memory limitations: Data, Pipeline, and Tensor Parallelism. Each approach differs in how it divides the workload, with Data Parallelism splitting the data batch, Pipeline Parallelism splitting sequential model layers, and Tensor Parallelism splitting the weight matrices within a single layer.

2.1 The Memory Wall Problem for LLMs

The growth in LLM parameter counts directly translates to increased memory requirements. The following table illustrates the memory footprint for various Llama-3 model sizes in both FP16 and FP32 precision:

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 shown, even a 70B model in FP16 exceeds the capacity of a standard 80GB GPU. For larger models, distributing the model across multiple GPUs is not just an optimization but a necessity.

2.2 Overview of Parallelism Strategies

Several strategies exist for distributing workloads across multiple GPUs, each with different trade-offs regarding memory usage and communication overhead:

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 between
Tensor Parallelism	Weight matrices	Slice of every layer	All-reduce within

Data Parallelism replicates the entire model on each GPU and splits the data batch, which is primarily useful for training. **Pipeline Parallelism** partitions the model layer-by-layer, with each GPU holding a subset of layers. **Tensor Parallelism** splits the weight matrices within each layer, meaning every GPU holds a slice of every layer.

2.3 When to Use Tensor Parallelism

Tensor Parallelism is the preferred strategy under specific conditions. It is most suitable when a model is too large to fit into the memory of a single GPU. It is also highly effective when the GPUs are connected with high-speed interconnects like **NVLink** or **InfiniBand**, as this minimizes the performance impact of the frequent communication required between GPUs. Finally, TP is often chosen for inference workloads where minimizing latency is a primary goal.

3. How Tensor Parallelism Works

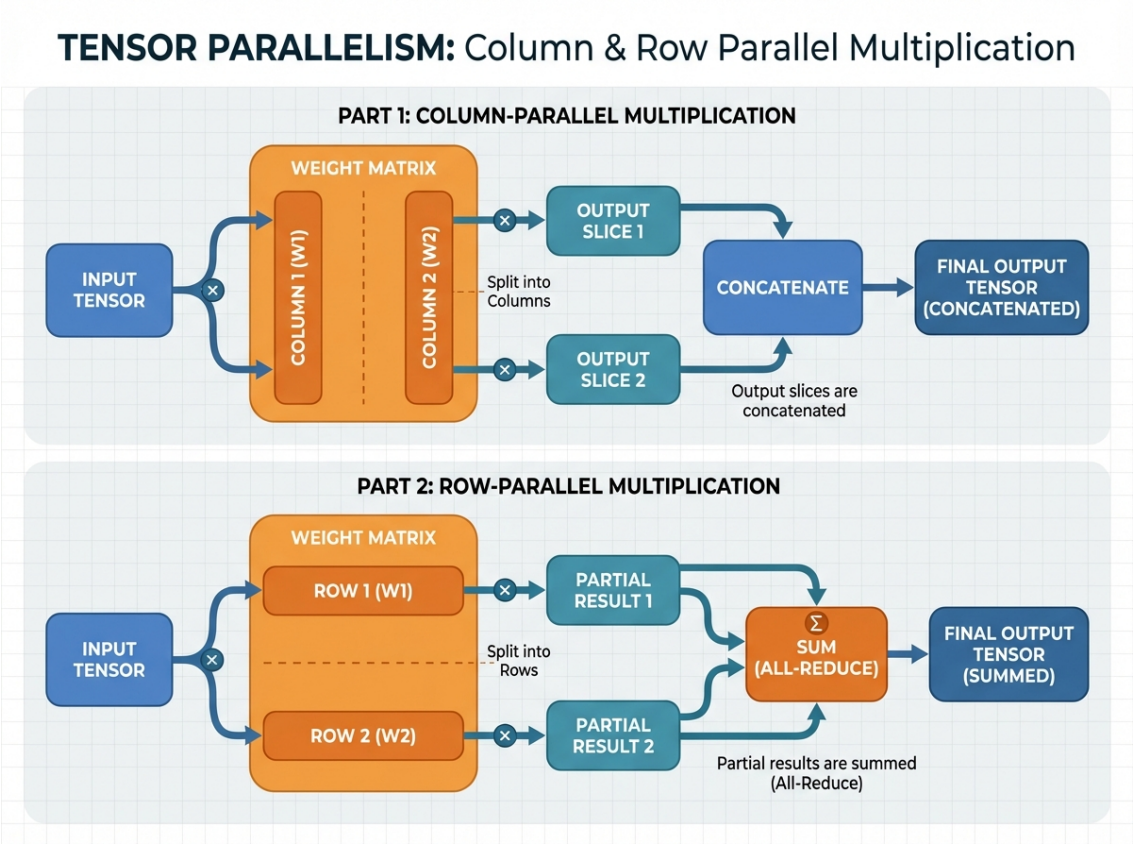

Figure 4: 3. How Tensor Parallelism Works

This diagram visualizes how tensor parallelism splits a model's weight matrix for parallel computation. In column-parallel multiplication, the matrix is split by columns, and the independent output slices are concatenated. Conversely, in row-parallel multiplication, the matrix is split by rows, and the partial results are summed to produce the final output tensor.

3.1 Core Concept: Splitting Matrix Multiplications

The core idea behind Tensor Parallelism is that the matrix multiplications at the heart of neural networks can be parallelized. By strategically splitting the weight matrices involved in these operations, the computational workload can be distributed across multiple processing units. TP employs two primary methods for this: column-parallel and row-parallel matrix multiplication.

3.2 Column-Parallel Matrix Multiplication

In a column-parallel approach, a weight matrix `W` is split into vertical blocks, or columns. Consider the computation `Y = XW`. If `W` is split into n column blocks `[W1, W2, ..., Wn]`, then each of the n GPUs can compute a corresponding slice of the output `Yi = XWi` independently. The final output `Y` is the concatenation of these slices, `Y = [Y1, Y2, ..., Yn]`. A key advantage of this method is that the outputs are naturally sharded by columns, and no communication between GPUs is required at this stage.

3.3 Row-Parallel Matrix Multiplication

Row-parallel multiplication is used when the input `X` is already sharded by columns, such as `X = [X1, X2, ..., Xn]`. The weight matrix `W` is then split into horizontal blocks, or rows, to match the input sharding. Each GPU computes a partial result by multiplying its input slice with its corresponding weight slice. To obtain the final output, these partial results must be summed together across all GPUs. This summation is performed using an **all-reduce** operation, which constitutes the primary communication step in Tensor Parallelism.

4. Applying TP to Transformer Layers

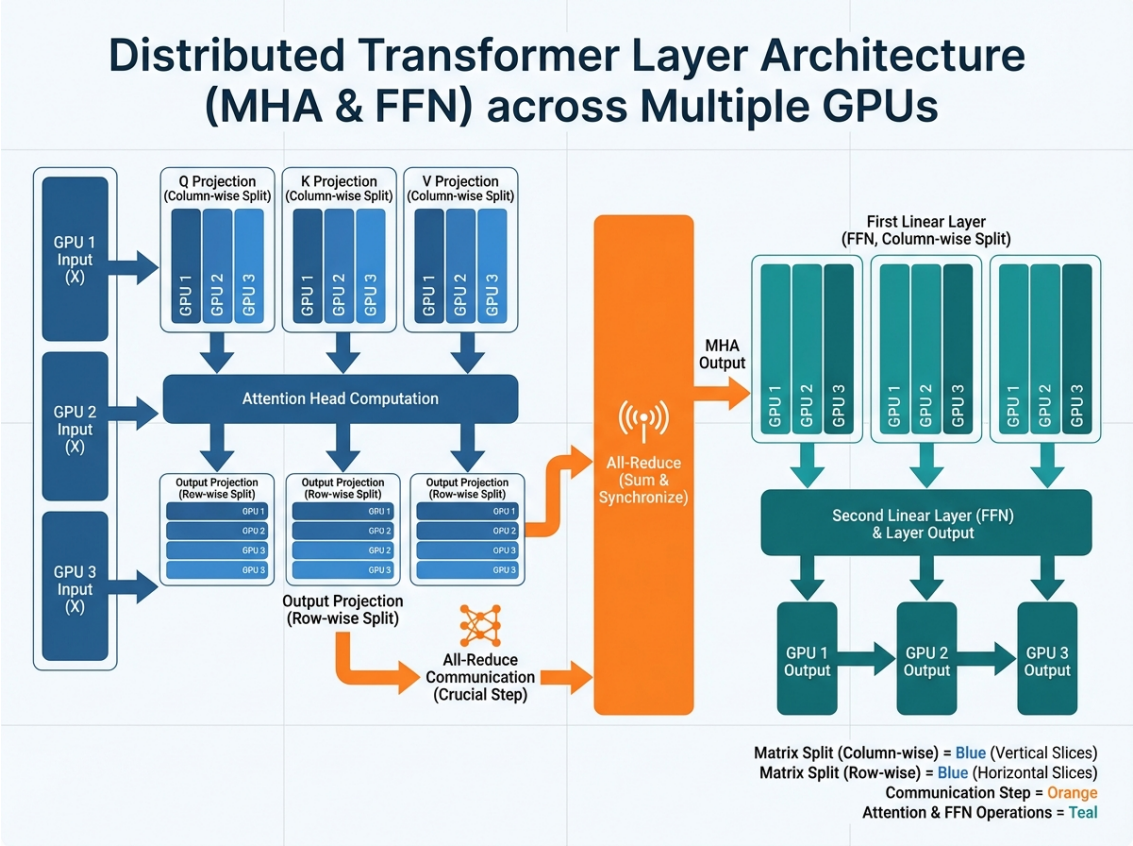

Figure 5: 4. Applying TP to Transformer Layers

This diagram illustrates how Tensor Parallelism distributes a single Transformer layer's architecture across multiple GPUs. The weight matrices for both the multi-head attention (MHA) and feed-forward network (FFN) are partitioned using column-wise and row-wise splits. Crucial all-reduce communication steps are required to sum and synchronize the partial results from each GPU, allowing the layer to function as a cohesive unit.

The principles of column-parallel and row-parallel multiplication are applied directly to the components of a Transformer layer, specifically the attention mechanism and the feed-forward network.

4.1 Splitting the Attention Layer

The self-attention mechanism in a Transformer involves three projection matrices for queries, keys, and values (`WQ`, `WK`, `WV`) and one output projection matrix (`WO`).

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

The `WQ`, `WK`, and `WV` matrices are split column-wise. This is equivalent to distributing the attention heads across the GPUs. For example, with 32 attention heads and 4 GPUs, each GPU would handle 8 heads. Each GPU computes its portion of the Q, K, and V projections independently, with no communication required.

```
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]
```

Step 2: Local Attention Computation

Since attention heads are independent, each GPU can compute the attention scores and resulting values for its assigned heads locally. This step also requires no inter-GPU communication.

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

Step 3: Output Projection (Row-Parallel)

The output projection matrix `WO` is split row-wise. Each GPU multiplies its local attention output by its slice of `WO` to produce a partial output. An **all-reduce** operation is then performed to sum these partial outputs from all GPUs, yielding the final attention output. This step introduces one `all-reduce` communication per attention layer.

```
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)
```

4.2 Splitting the Feed-Forward Network (FFN)

A standard Transformer FFN consists of two linear layers. TP is applied here in a similar two-step process.

First Linear Layer (Column-Parallel)

The first linear layer's weight matrix (`W1`) is split column-wise. Each GPU processes the input with its slice of the weights, and an activation function (like GELU) is applied. This step is performed independently on each GPU.

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

Second Linear Layer (Row-Parallel)

The weight matrix of the second linear layer (`W2`) is split row-wise. Each GPU computes a partial output using its local hidden state and its slice of `W2`. Another **all-reduce** operation is required to sum the partial outputs and produce the final FFN output.

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

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

In total, a standard Transformer layer requires two **all-reduce** operations: one for the attention output projection and one for the FFN output projection.

4.3 Communication Overhead: All-Reduce Operations

The performance of Tensor Parallelism is heavily dependent on the efficiency of the **all-reduce** operations. Each Transformer layer requires two such synchronizations. The speed of these operations is dictated by the interconnect bandwidth between the GPUs. High-speed interconnects like **NVLink** (providing 600-900 GB/s) are crucial for minimizing this communication overhead and achieving good scaling.

4.4 Practical Constraints of TP

Implementing Tensor Parallelism comes with several practical constraints related to the model's architecture. First, the TP size (number of GPUs) cannot exceed the number of attention heads, as a single head cannot be split. Second, the number of attention heads must be evenly divisible by the TP size to ensure each GPU receives an equal share of the workload. Similarly, the hidden dimension of the FFN must also be divisible by the TP size. For a model like Llama-3-70B with 64 attention heads, valid TP sizes are powers of two up to 64 (e.g., 2, 4, 8, 16, 32, 64).

5. Implementing TP with HuggingFace Transformers

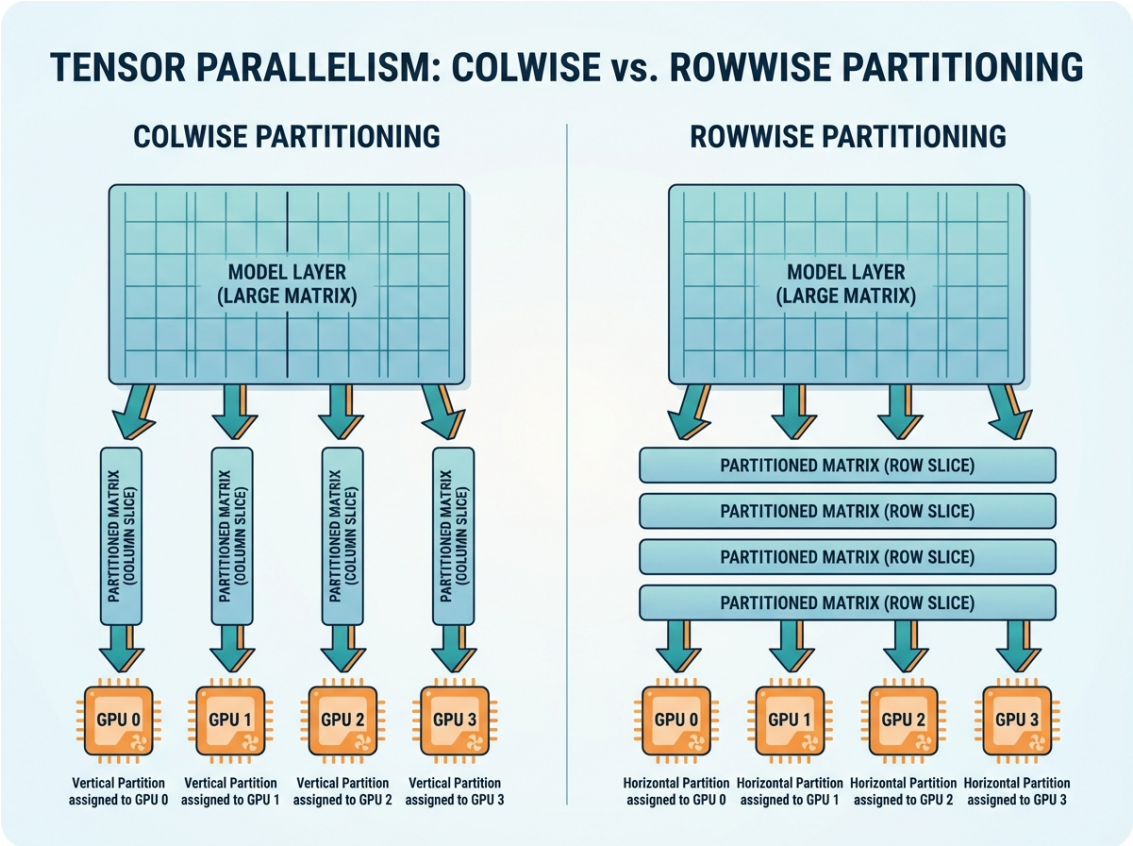

Figure 6: 5. Implementing TP with HuggingFace Transformers

Tensor parallelism addresses memory limitations by splitting a model's large weight matrices across multiple GPUs. This diagram illustrates two fundamental approaches for partitioning a model layer. In colwise partitioning, the matrix is split vertically into column slices, while in rowwise partitioning, it is split horizontally into row slices, with each slice assigned to a different GPU.

The HuggingFace `transformers` library has integrated support for Tensor Parallelism, making it straightforward to implement for supported models.

5.1 The Simple `tp_plan` Solution

For models that support it, enabling Tensor Parallelism can be as simple as adding a single argument when loading the model. By setting `tp_plan="auto"`, the library

will automatically handle the partitioning of 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 Scripts with `torchrun`

A script using Tensor Parallelism cannot be run with a simple `python` command. Instead, it must be launched using `torchrun`, which is part of PyTorch's distributed runtime. This utility spawns multiple processes, one for each GPU, and manages the necessary communication between them.

To run the script on 4 GPUs, you would use the following command:

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

5.3 Verifying Model Support for TP

As of late 2025, HuggingFace provides built-in TP support for popular models including Llama, Mistral, Mixtral, Qwen, and Gemma. To verify if a specific model supports TP, you can inspect its configuration for a `_tp_plan` attribute, which details the default partitioning strategy.

```
from transformers import AutoConfig

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

5.4 Understanding Partitioning Strategies

HuggingFace uses several internal strategies to partition model parameters. The most common are `colwise` for column-parallel splitting (used for Q, K, V projections and the first FFN layer) and `rowwise` for row-parallel splitting (used for output projections). Other strategies include `sequence_parallel` for components like LayerNorm and Dropout, and `replicate` for parameters that need to be fully copied to each GPU.

5.5 Defining a Custom `tp_plan`

While `tp_plan="auto"` is sufficient for most cases, you can also define a custom partitioning plan by providing a dictionary that maps specific model layers to a partitioning strategy. This allows for fine-grained control over how the model is distributed.

```
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: TP on RunPod

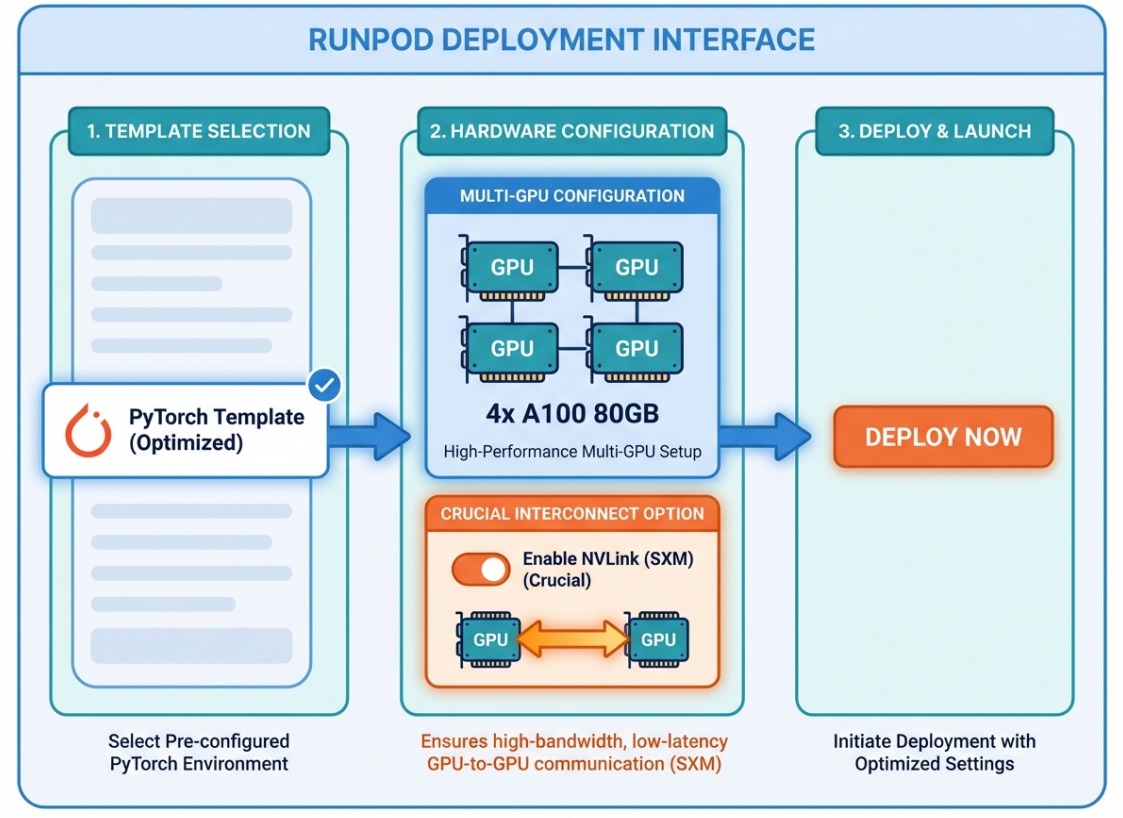

Figure 7: 6. Hands-On Guide: TP on RunPod

To deploy a multi-GPU environment for Tensor Parallelism on RunPod, start by selecting an optimized PyTorch template. Next, configure your hardware with a

multi-GPU setup, such as four A100 80GB GPUs. Crucially, enable the NVLink (SXM) interconnect to ensure the high-bandwidth, low-latency communication between GPUs required for efficient model parallelism.

RunPod provides on-demand GPU instances suitable for running multi-GPU workloads like Tensor Parallelism. The following steps outline how to run Llama-3-70B with TP on this platform.

6.1 Deploying a Multi-GPU Pod with NVLink

First, deploy a multi-GPU pod on RunPod. Select a PyTorch template (e.g., `runpod/pytorch:2.1.0-py3.10-cuda11.8.0`) and choose a configuration with multiple GPUs, such as 4x A100 80GB for a 70B model. It is critical to select instances with **NVLink** interconnects, typically designated as `SXM` variants (e.g., A100-SXM or H100-SXM). NVLink offers 600-900 GB/s of inter-GPU bandwidth, which is essential for efficient `all-reduce` operations. Standard PCIe interconnects are limited to around 64 GB/s and will create a severe communication bottleneck, negating the benefits of TP.

6.2 Environment Setup and Verification

Once connected to the pod via SSH, update the necessary packages and verify the GPU setup.

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

# Verify GPU setup
nvidia-smi

# Check NCCL (the communication backend)
python -c &quot;import torch; print(f&#x27;CUDA available: {torch.cuda.is_available()}\nGPU c
```

The expected output should confirm that CUDA is available and show the correct number of GPUs.

6.3 Creating the Inference Script

Create a Python script to load the model with TP and generate text. The script should use `os.environ.get("RANK", 0)` to ensure that only the main process (rank 0) prints the output.

```
# 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()
```

6.4 Launching with `torchrun`

Before running the script, set your HuggingFace token as an environment variable if you are using a gated model. Then, launch the script using `torchrun`, specifying the number of 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
```

6.5 Using vLLM with TP on RunPod

For production environments, vLLM often delivers higher performance. To use vLLM with Tensor Parallelism on RunPod, you can install it and launch its OpenAI-compatible API server, specifying the `tensor-parallel-size`.

```
# 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 vLLM workers can automatically handle TP configuration through the handler settings.

7. Hands-On Guide: TP on Lambda Cloud

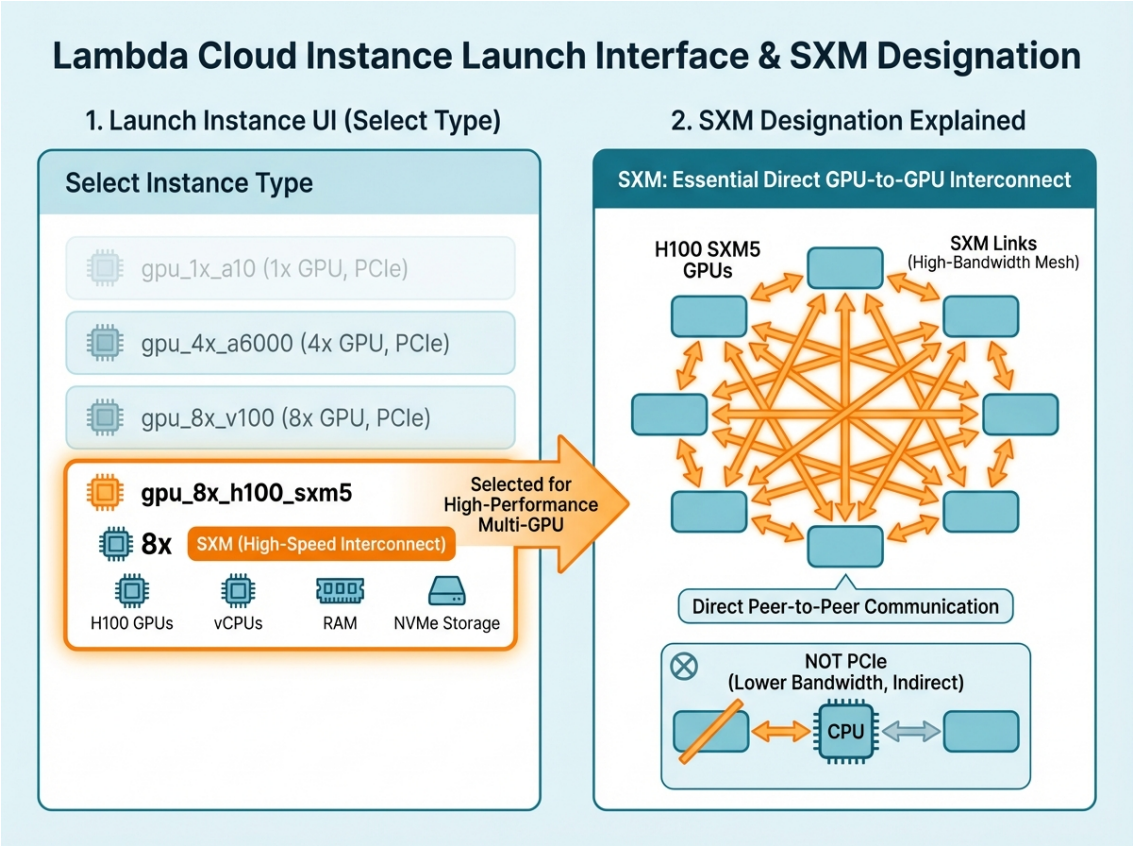

Figure 8: 7. Hands-On Guide: TP on Lambda Cloud

When running Tensor Parallelism on Lambda Cloud, selecting an instance with an SXM interconnect is crucial for high performance. This technology provides a direct, high-bandwidth communication mesh between GPUs, enabling the rapid peer-to-peer data exchange required for multi-GPU workloads. This avoids the performance bottleneck of standard PCIe connections, which route communication indirectly through the CPU.

Lambda Cloud also offers multi-GPU instances with high-speed interconnects, making it another excellent choice for Tensor Parallelism.

7.1 Launching a Multi-GPU SXM Instance

When launching an instance on Lambda Cloud, choose a type with multiple GPUs, such as `gpu_8x_h100_sxm5` or `gpu_4x_a100_80gb_sxm4`. As with RunPod, it is crucial to select **SXM** variants (`sxm4`, `sxm5`). The `SXM` designation confirms that the GPUs are connected via high-bandwidth **NVLink**, which is essential for TP performance. PCIe-based instances will introduce a significant communication bottleneck that undermines the efficiency of Tensor Parallelism.

7.2 Creating a Benchmarking Inference Script

The setup process after SSHing into the instance is similar to RunPod. For benchmarking, you can create a script that measures model loading time and inference throughput (tokens per second). The script should be designed to run with `torchrun` and use environment variables like `RANK` and `WORLD_SIZE`.

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

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

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

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

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

    prompt = &quot;Write a short poem about distributed computing:&quot;
    inputs = tokenizer(prompt, return_tensors=&quot;pt&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

```

```
    if rank == 0:
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        tokens_generated = outputs.shape[1] - inputs[&quot;input_ids&quot;].shape[1]
        tokens_per_sec = tokens_generated / gen_time
        print(f&quot;\nPrompt: {prompt}&quot;)
        print(f&quot;Response: {response}&quot;)
        print(f&quot;\n--- Performance ---&quot;)
        print(f&quot;Tokens generated: {tokens_generated}&quot;)
        print(f&quot;Time: {gen_time:.2f}s&quot;)
        print(f&quot;Throughput: {tokens_per_sec:.1f} tokens/sec&quot;)

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

7.3 Launching on a Single Node

To run the benchmarking script on a single Lambda Cloud instance with multiple GPUs, use `torchrun` and specify the number of processes per node.

```
# 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
```

7.4 Multi-Node Setup for Large-Scale TP

For models that require more than 8 GPUs, `torchrun` supports multi-node execution. You must provide additional arguments to specify the total number of nodes, the rank of the current node, and the address and port of the master node.

On the master node (Node 0):

```
torchrun \
    --nproc-per-node 8 \
    --nnodes 2 \
    --node-rank 0 \
    --master-addr &lt;master-ip&gt; \
    --master-port 29500 \
    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 setup enables 16-way Tensor Parallelism. However, cross-node TP is highly sensitive to the network interconnect. It requires high-bandwidth connections like **InfiniBand** to be effective; otherwise, communication overhead will severely degrade performance.

8. Performance Benchmarks and Observations

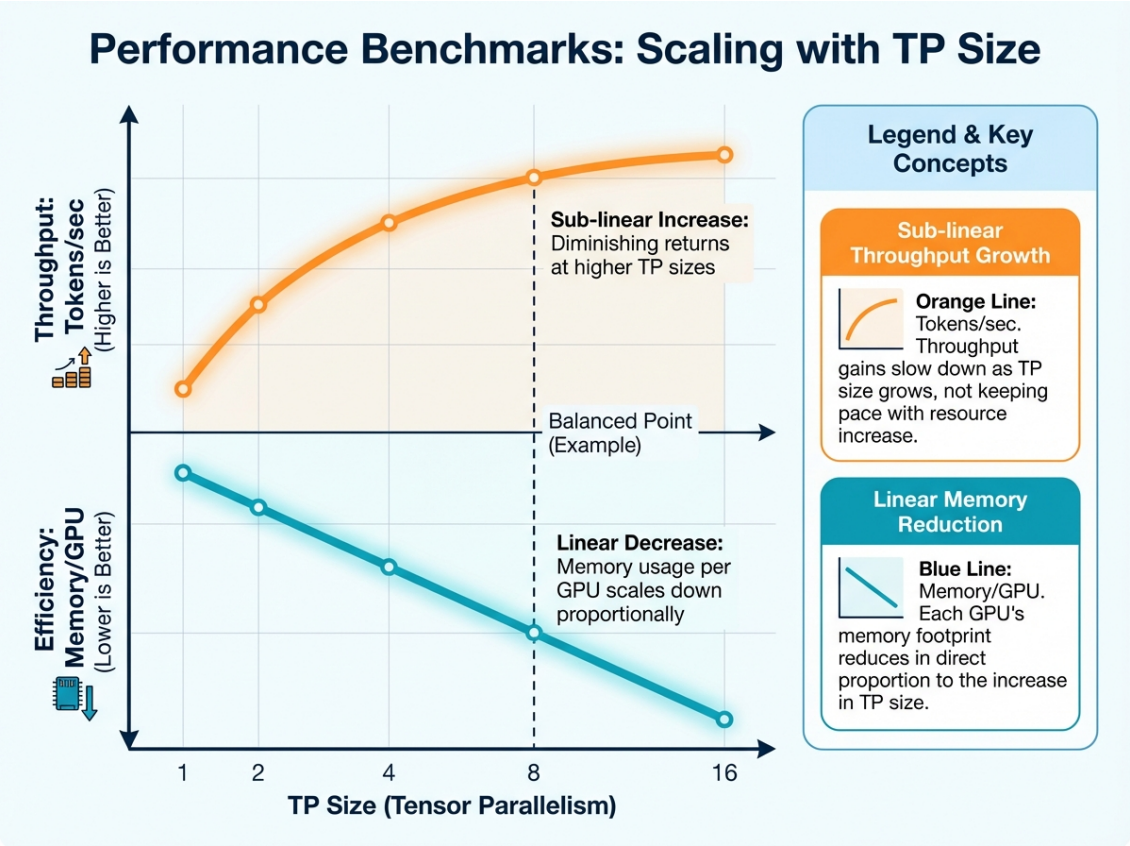

Figure 9: 8. Performance Benchmarks and Observations

This performance benchmark graph illustrates the trade-offs of scaling Tensor Parallelism (TP). As the TP size increases, memory efficiency improves with a linear decrease in memory usage per GPU. However, throughput gains, measured in tokens per second, are sub-linear, showing diminishing returns as more resources are added. This highlights the importance of finding a balanced point to optimize both memory efficiency and processing speed.

Benchmarking reveals how Tensor Parallelism scales with an increasing number of GPUs.

8.1 Llama-3-70B Inference Throughput

The following table shows typical inference performance for Llama-3-70B on H100 80GB GPUs with varying TP sizes. 'OOM' indicates an Out Of Memory error.

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 Observations on Scalability

From the benchmarks, two key patterns emerge. First, memory usage per GPU scales linearly: using four times the number of GPUs reduces the memory footprint on each GPU by approximately four times. Second, throughput scales sub-linearly. While adding more GPUs increases the tokens per second, the gains diminish as the TP size grows due to the increasing communication overhead from the `all-reduce` operations. The sweet spot for performance is often between 4 and 8 GPUs; beyond this point, communication costs can begin to dominate computation time.

9. Limitations of Tensor Parallelism

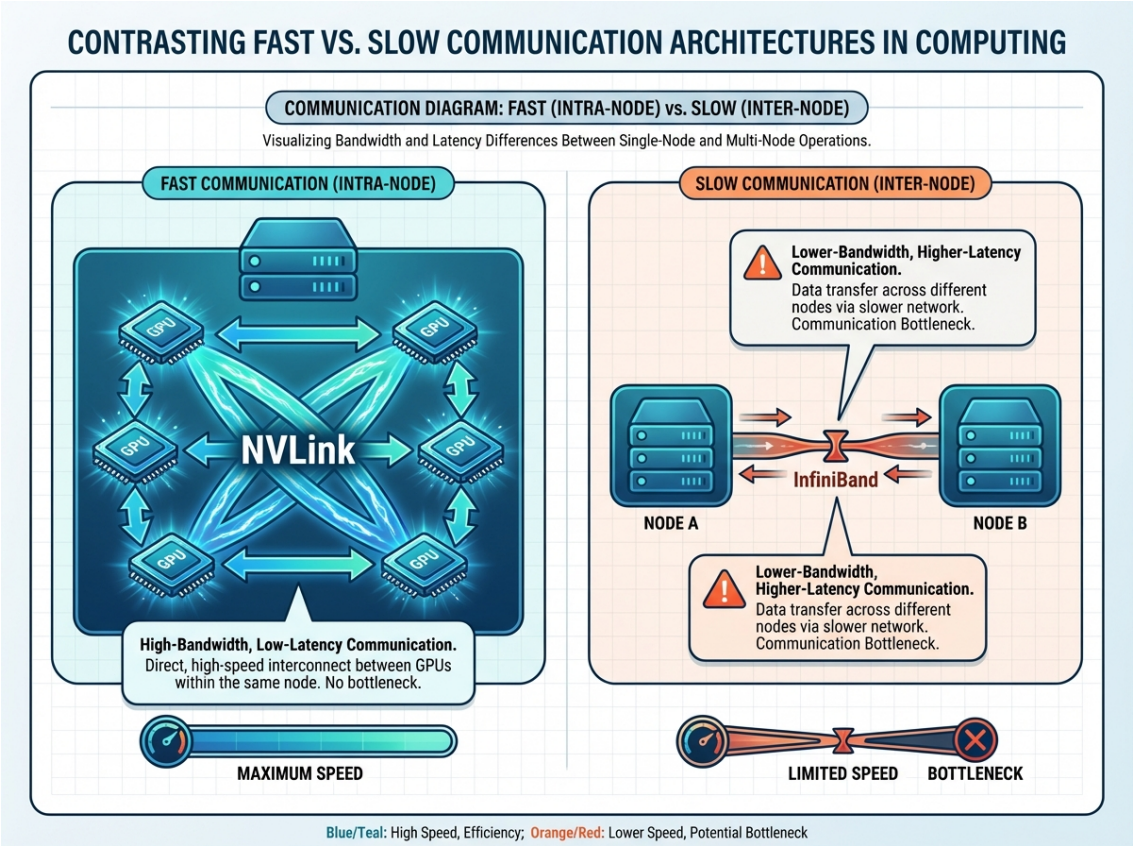

Figure 10: 9. Limitations of Tensor Parallelism

While tensor parallelism splits a model across GPUs, its performance hinges on the communication speed between them. Fast, intra-node communication using high-speed interconnects like NVLink enables maximum throughput within a single server. However, a key limitation arises with slower inter-node communication, where networks like InfiniBand create a high-latency bottleneck that can throttle performance across different machines.

While powerful, Tensor Parallelism is not a universal solution and has several important limitations.

9.1 Scalability Capped by Attention Heads

The maximum TP size is fundamentally limited by the number of attention heads in the model. You cannot split a single attention head across multiple GPUs. In practice, to maintain efficiency, the TP size should be considerably smaller than the total number of heads.

9.2 Communication Overhead Across Nodes

TP relies on frequent `all-reduce` operations (two per Transformer layer). Within a single node connected by **NVLink** (900 GB/s), this communication is very fast. However, when TP is extended across nodes, the interconnect is typically **InfiniBand** (~400 GB/s) or, in worse cases, Ethernet (~100 Gbps). This lower bandwidth makes cross-node communication a significant bottleneck. A general rule of thumb is to confine TP to a single node and use Pipeline Parallelism for scaling across multiple nodes.

9.3 Ineffectiveness Against Activation Memory Growth

Tensor Parallelism effectively reduces the memory required to store the model's weights, but it does not reduce the memory needed for activations. When processing very long sequences, the KV cache and other activations can consume a large amount of memory, which may still lead to out-of-memory errors even when using TP. In such cases, other techniques like gradient checkpointing may be necessary.

10. Combining Tensor and Pipeline Parallelism

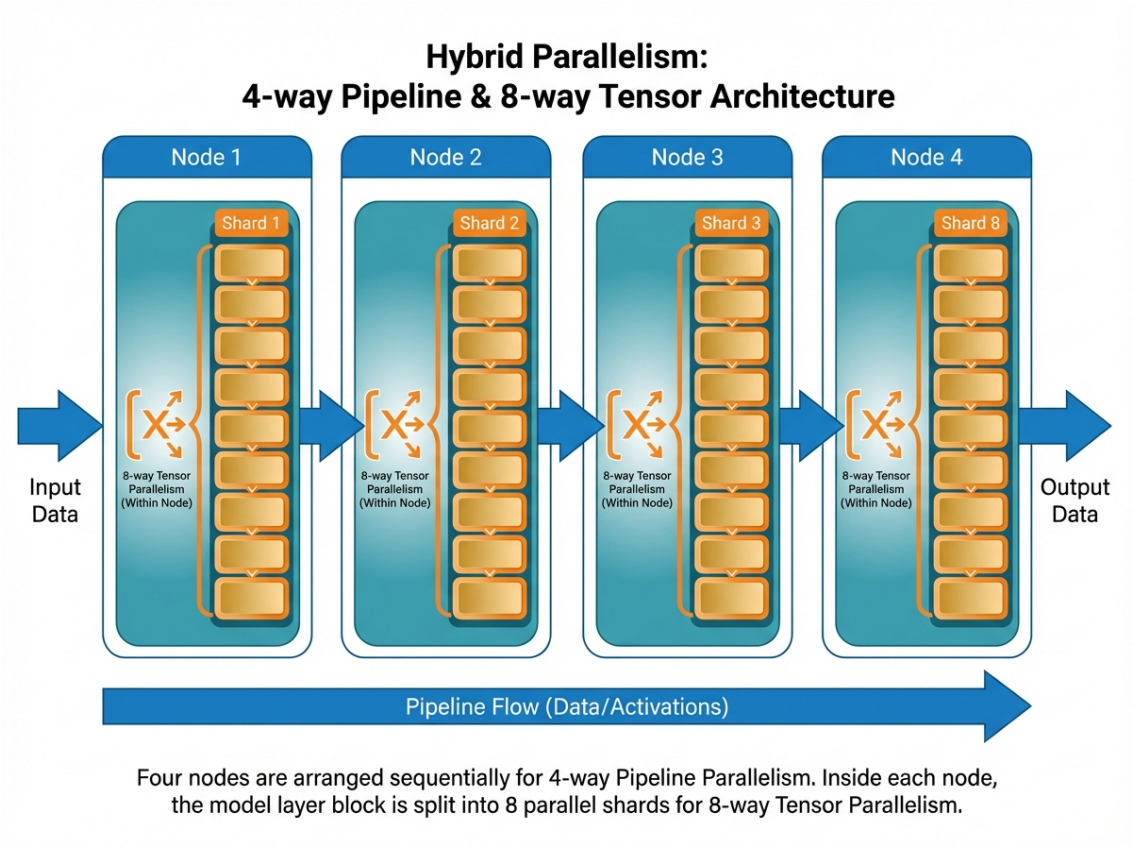

Figure 11: 10. Combining Tensor and Pipeline Parallelism

This diagram illustrates a hybrid architecture combining two powerful techniques for running large models. The model is distributed across four sequential nodes using 4-way pipeline parallelism, where data flows from one stage to the next. Within each node, 8-way tensor parallelism is applied, splitting the model's layers into parallel shards to further distribute the computational load.

10.1 Strategy for Truly Massive Models

For extremely large models with hundreds of billions of parameters, a hybrid approach combining Tensor Parallelism (TP) and Pipeline Parallelism (PP) is often the most effective strategy. In this setup, TP is used to split the layers within each node, while PP is used to distribute sequential blocks of layers across different nodes.

For example, to run a model on 32 GPUs, you could configure it as follows:

```
■ 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 configuration results in an 8-way TP and a 4-way PP setup, effectively utilizing all 32 GPUs.

10.2 When to Use Each Parallelism Strategy

The choice of parallelism strategy depends on the specific bottlenecks and hardware setup. The general best practice is to use **Tensor Parallelism within a single node** to take advantage of high-speed NVLink interconnects. For scaling beyond a single node, use **Pipeline Parallelism across nodes**. This approach minimizes the high-frequency communication of TP over the slower inter-node network, while restricting the less frequent PP communication (passing activations between stages) to that network.

11. Key Takeaways

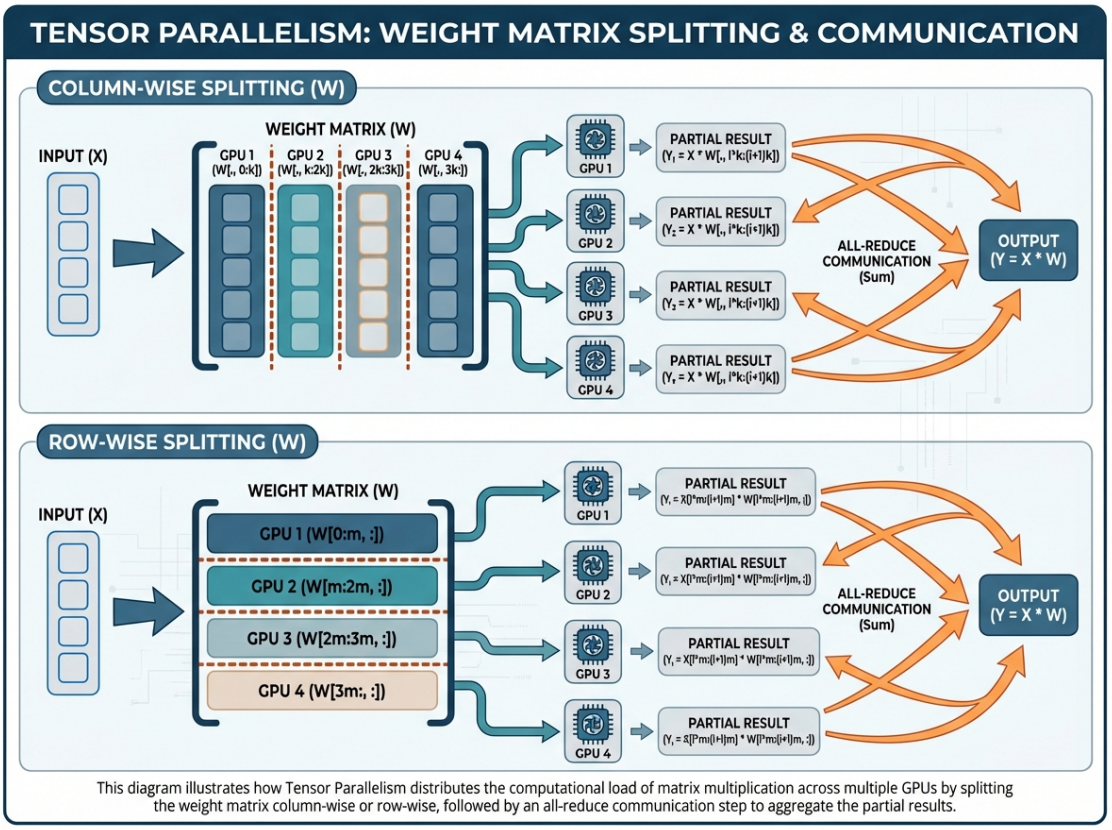

Figure 12: 11. Key Takeaways

This diagram illustrates how Tensor Parallelism splits a model's large weight matrix across multiple GPUs, a key technique for running models that exceed single-GPU memory. It showcases two methods: splitting the matrix by columns or by rows. In both cases, each GPU computes a partial result, which is then aggregated through an all-reduce communication step to produce the final output.

11.1 Core Principles of Tensor Parallelism

Tensor Parallelism is a fundamental technique for running large models that exceed single-GPU memory. Its core principles involve splitting the weight matrices of a model across multiple GPUs—typically using a column-wise split for initial projections and a row-wise split for output projections. This distribution requires communication between GPUs, which is handled by two `all-reduce` operations per Transformer

layer to aggregate the partial results.

11.2 Best Practices for Implementation

For optimal performance, Tensor Parallelism should be confined within a single server node where GPUs are connected by high-speed **NVLink**. This minimizes the latency of the required communication. The HuggingFace `transformers` library simplifies implementation significantly with the `tp_plan="auto"` argument, which automates the model partitioning process. Scripts utilizing TP must be launched with `torchrun` to manage the distributed processes.

11.3 Considering Alternatives like vLLM and FSDP

While the built-in HuggingFace support is convenient, for production inference workloads, frameworks like **vLLM** often provide more highly optimized TP implementations that can yield better throughput. For training large models, an alternative to consider is **FSDP** (Fully Sharded Data Parallel), which offers a different approach by combining aspects of both Tensor Parallelism and Data Parallelism to optimize memory usage and training efficiency.

Key Takeaways

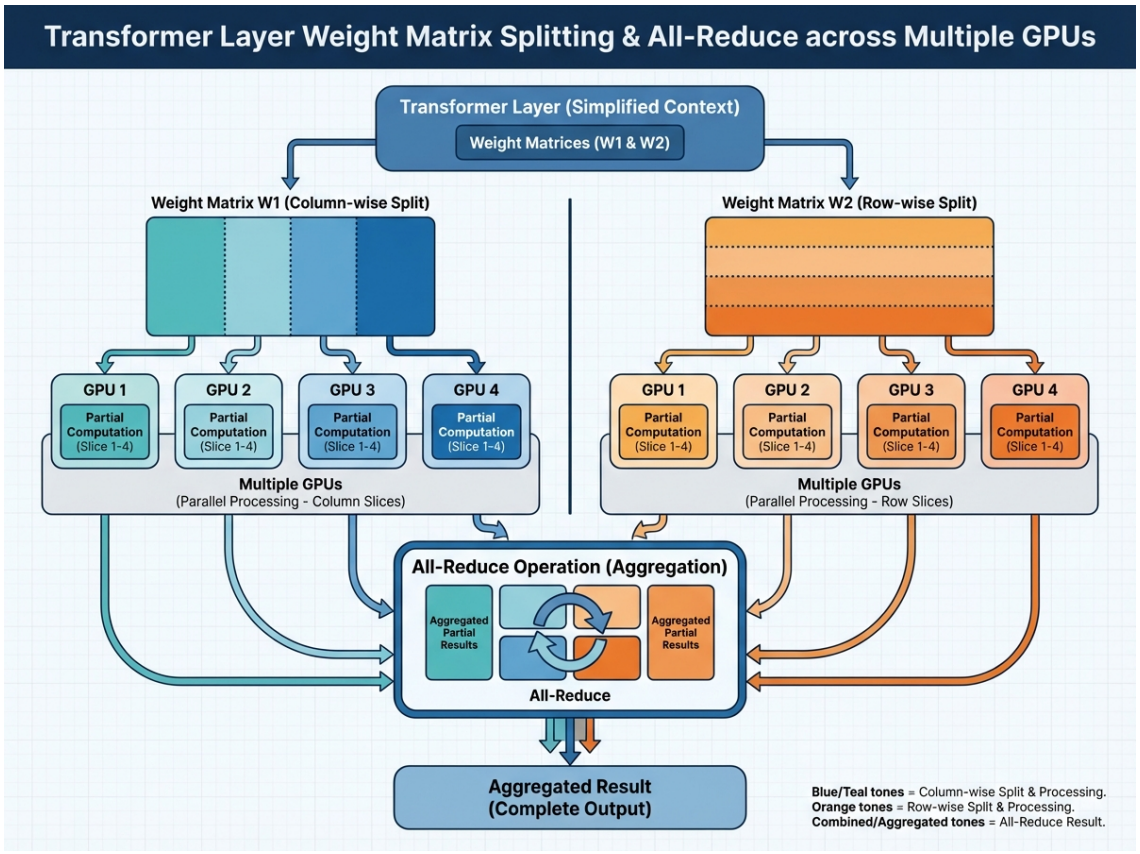

Figure 13: Key Takeaways

Tensor Parallelism enables running large models by splitting a transformer layer's weight matrices across multiple GPUs. This diagram illustrates how matrices are split either column-wise or row-wise, with each GPU processing its slice in parallel. The partial results are then efficiently combined through an "All-Reduce" operation to produce the final aggregated output.

Tensor Parallelism is the go-to technique for running models that do not fit on a single GPU. The key ideas are to split weight matrices across GPUs (column-wise for projections, row-wise for outputs) and use `all-reduce` operations to aggregate partial results, with two such operations per Transformer layer. For best performance, it is crucial to keep TP within a single node equipped with high-speed interconnects like NVLink. The HuggingFace `transformers` library offers a simple implementation path with `tp_plan="auto"`. For more specialized use cases, consider highly optimized inference servers like vLLM for production or FSDP for large-scale training.