Contents

Nebius H100 GPU Cluster - Complete Technical Guide

Document Information

Field	Value
Author	Supreeth Mysore
Date	December 17, 2025
Version	1.0
Scope	8-GPU and 16-GPU H100 Cluster Deployment

Table of Contents

---

1. Introduction

1.1 What is This Project?

This project demonstrates how to: 1. Validate and benchmark NVIDIA H100 GPU clusters on Nebius AI Cloud 2. Fine-tune Large Language Models (LLMs) using distributed training 3. Optimize performance for multi-GPU and multi-node configurations

1.2 Who is This For?

- ML Engineers setting up GPU clusters
- DevOps engineers validating hardware
- Researchers running distributed training
- Anyone learning about multi-GPU deep learning

1.3 Prerequisites

To understand this guide, you should be familiar with: - Basic Linux command line - Python programming - Deep learning concepts (training, loss, batches) - Basic understanding of GPUs

---

2. Project Overview

2.1 Two Configurations

We have two repository configurations:

Configuration	Repository	GPUs	Nodes
Single Node	nebius-h100-8gpu-benchmark	8	1
Multi-Node	nebius-h100-16gpu-benchmark	16	2

2.2 Two Exercises

Each repository contains two exercises:

Exercise 1 - LLM Fine-Tuning: - Fine-tune a 7 billion parameter language model (Qwen2-7B) - Use LoRA (Low-Rank Adaptation) for efficient training - Train the model to perform function calling tasks

Exercise 2 - GPU Benchmarks: - Validate GPU hardware health - Measure compute performance (matrix multiplication) - Measure memory bandwidth - Test inter-GPU communication (NCCL AllReduce)

2.3 Why These Exercises?

Exercise 1 demonstrates real-world AI workload - training an LLM is one of the most demanding GPU tasks. It tests: - GPU compute capability - Memory capacity and bandwidth - Multi-GPU communication - Software stack (PyTorch, CUDA, NCCL)

Exercise 2 provides quantitative benchmarks to: - Validate hardware meets specifications - Identify performance bottlenecks - Compare against expected values - Ensure cluster is production-ready

---

3. Hardware Architecture

3.1 Single Node Configuration (8 GPUs)

```
                    COMPUTE NODE


                     CPU SUBSYSTEM
Intel Xeon Platinum 8468 (2 sockets, 128 cores total)
1.5 TB DDR5 RAM


                        PCIe Gen5


                     GPU SUBSYSTEM

 GPU0    GPU1    GPU2    GPU3

       NVLink 4.0 (900 GB/s)

 GPU4    GPU5    GPU6    GPU7

 Each GPU: NVIDIA H100 80GB HBM3
```

3.2 Multi-Node Configuration (16 GPUs)

```
   NODE 0 (Master)                       NODE 1 (Worker)
 IP: 10.2.0.129                        IP: 10.2.0.0


 8x NVIDIA H100 80GB                   8x NVIDIA H100 80GB

GPU0  GPU1  GPU2  GPU3                GPU0  GPU1  GPU2  GPU3
GPU4  GPU5  GPU6  GPU7                GPU4  GPU5  GPU6  GPU7

Connected via NVLink                  Connected via NVLink



8x InfiniBand 400Gb/s                 8x InfiniBand 400Gb/s
```

```
                     InfiniBand
                       Network Fabric
                  (400 Gb/s per port)
```

3.3 Key Hardware Specifications

Component	Specification	Purpose
GPU	NVIDIA H100 80GB HBM3	AI compute accelerator
GPU Memory	80 GB HBM3 per GPU	Store model weights and activations
GPU Compute	989 TFLOPS (BF16)	Matrix operations for training
NVLink	900 GB/s bidirectional	Fast GPU-to-GPU communication within node
InfiniBand	400 Gb/s per port	Fast node-to-node communication
CPU	Intel Xeon Platinum 8468	Data preprocessing, orchestration
RAM	1.5 TB DDR5	Dataset loading, CPU operations

3.4 Understanding NVLink

NVLink is NVIDIA's high-speed GPU interconnect:

```
Without NVLink (PCIe only):
GPU0  PCIe > CPU  PCIe > GPU1
            ~64 GB/s

With NVLink:
GPU0   NVLink   GPU1
        ~900 GB/s

Speed improvement: ~14x faster!
```

Why it matters: During distributed training, GPUs need to exchange gradients. Faster interconnect = faster training.

3.5 Understanding InfiniBand

InfiniBand connects nodes in a cluster:

```
Node 0                    Node 1
```

```
  InfiniBand Fabric
   (400 Gb/s = 50 GB/s per port)
   (8 ports = 400 GB/s aggregate)
```

Why it matters: Multi-node training requires gradient synchronization across nodes. InfiniBand provides low-latency, high-bandwidth connectivity.

---

4. Repository Structure

4.1 Directory Layout

```
nebius-h100-16gpu-benchmark/

  exercise1/                      # LLM Fine-Tuning
     configs/
        training_config.yaml           # Default training settings
        training_config_optimized.yaml # Optimized settings
        training_config_optimized_noDS.yaml  # Without DeepSpeed
        ds_config_zero3.json           # DeepSpeed ZeRO-3 config
        ds_config_zero3_optimized.json # Optimized DeepSpeed
     scripts/
         train_function_calling.py      # Main training script
         optimize_env.sh                # Environment variables

  exercise2/                      # GPU Benchmarks
     scripts/
        benchmark.py                   # Benchmark script
     configs/
        benchmark_config.yaml          # Benchmark thresholds
     results/
         benchmark_16gpu.json           # Saved results

  multinode.sh                    # Multi-node launcher (16 GPU)
  hardware_info.sh                # Hardware discovery script
  demo_banner.sh                  # Colorful demo banner
  demo_status.py                  # Rich GPU dashboard
  tmux_demo.sh                    # Multi-pane terminal layout
  run_demo.sh                     # Interactive demo sequence
  monitor_nccl.sh                 # NCCL/NVLink monitoring

  DEMO.md                         # Demo walkthrough guide
  OPTIMIZATION.md                 # Performance tuning guide
  READING.md                      # This document
  README.md                       # Project overview
```

4.2 8-GPU vs 16-GPU Differences

File	8-GPU Repo	16-GPU Repo
Launcher	`singlenode.sh`	`multinode.sh`
Nodes	1	2
GPU count	8	16
InfiniBand	Not used	Required
Coordination	Local only	Master-Worker

---

5. File-by-File Description

5.1 Launcher Scripts

`multinode.sh` (16-GPU) / `singlenode.sh` (8-GPU) **Purpose:** One-command launcher for training and benchmarks.

What it does: 1. Sets up conda environment 2. Configures environment variables (CUDA, NCCL) 3. Starts training or benchmarks on all GPUs 4. For multi-node: coordinates master and worker nodes

Usage:

```
# 8-GPU
source singlenode.sh exercise1          # Training
source singlenode.sh exercise2          # Benchmarks

# 16-GPU
source multinode.sh exercise1           # Training
source multinode.sh exercise2           # Benchmarks
```

Key sections explained:

```
# Environment variables for optimization
export CUDA_DEVICE_MAX_CONNECTIONS=1  # Optimize CUDA streams
export OMP_NUM_THREADS=8              # CPU thread count
export NCCL_IB_DISABLE=0              # Enable InfiniBand
export NCCL_NET_GDR_LEVEL=5           # GPU Direct RDMA level

# Launch distributed training
torchrun --nproc_per_node=8 \         # 8 processes (1 per GPU)
         --nnodes=2 \                  # 2 nodes total
         --node_rank=$RANK \           # This node's rank (0 or 1)
         --master_addr=10.2.0.129 \    # Master node IP
         --master_port=29500 \         # Communication port
         scripts/train_function_calling.py
```

`hardware_info.sh` **Purpose:** Discover and display hardware configuration.

What it does: - Shows CPU model and core count - Shows GPU model, count, and memory - Displays GPU topology (NVLink connections) - Shows InfiniBand status - Displays memory and storage info

Usage:

```
./hardware_info.sh summary    # Quick overview
./hardware_info.sh gpu        # GPU details
./hardware_info.sh ib         # InfiniBand status
./hardware_info.sh all        # Everything
```

Sample output:

```
==============================================
CLUSTER SUMMARY
==============================================
Node 0 (Master): computeinstance-xxx - 10.2.0.129
Node 1 (Worker): computeinstance-yyy - 10.2.0.0

Per Node:
  - CPU: Intel Xeon Platinum 8468 (128 cores)
  - RAM: 1.5 TB
  - GPUs: 8x NVIDIA H100 80GB HBM3
  - InfiniBand: 8x Mellanox ConnectX (400 Gb/s each)
```

5.2 Training Scripts

`exercise1/scripts/train_function_calling.py` **Purpose:** Main training script for LLM fine-tuning.

What it does: 1. Loads pre-trained Qwen2-7B model 2. Applies LoRA adapters for efficient fine-tuning 3. Loads function calling dataset 4. Runs distributed training across GPUs 5. Saves checkpoints and logs

Key components:

```python
# 1. Model Loading
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype=torch.bfloat16,      # Use BF16 for efficiency
    device_map="auto"
)

# 2. LoRA Configuration
lora_config = LoraConfig(
    r=64,                             # Rank (size of adaptation)
    lora_alpha=128,                   # Scaling factor
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
```

```python
    lora_dropout=0.05
)

# 3. Training Arguments
training_args = TrainingArguments(
    per_device_train_batch_size=2,     # Samples per GPU
    gradient_accumulation_steps=4,      # Effective batch = 2*4*8 = 64
    learning_rate=2e-5,
    bf16=True,                          # Use BF16 precision
    max_steps=100,
    output_dir="./checkpoints"
)

# 4. Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    data_collator=collator
)
trainer.train()
```

`exercise1/configs/training_config.yaml` **Purpose:** Configuration file for training parameters.

Structure explained:

```yaml
model:
  name: "Qwen/Qwen2-7B-Instruct"      # Base model to fine-tune
  torch_dtype: "bfloat16"             # Precision (16-bit brain float)

lora:
  r: 64                                # LoRA rank
  alpha: 128                           # LoRA scaling (alpha/r = 2)
  dropout: 0.05                        # Regularization
  target_modules:                      # Which layers to adapt
    - q_proj                           # Query projection
    - k_proj                           # Key projection
    - v_proj                           # Value projection
    - o_proj                           # Output projection

training:
  per_device_train_batch_size: 2      # Batch size per GPU
  gradient_accumulation_steps: 4       # Accumulate before update
  learning_rate: 2.0e-5               # How fast to learn
  max_steps: 100                       # Training iterations
  warmup_steps: 10                     # Gradual LR increase
```

```yaml
  bf16: true                        # Use BF16 training
  gradient_checkpointing: true      # Save memory

data:
  dataset_name: "glaiveai/glaive-function-calling-v2"
  max_seq_length: 4096             # Maximum token length

distributed:
  deepspeed: null                   # DeepSpeed config (optional)
```

5.3 Benchmark Scripts

`exercise2/scripts/benchmark.py` **Purpose:** Measure GPU performance metrics.

Benchmarks performed:

1. **Health Check**
    - Verify all GPUs are detected
    - Check memory availability
    - Validate driver and CUDA versions
2. **MatMul Benchmark (Single GPU)**
    - Performs large matrix multiplication
    - Measures TFLOPS (trillion floating-point operations per second)
    - Tests raw compute capability
3. **Memory Bandwidth (Single GPU)**
    - Copies data to/from GPU memory
    - Measures TB/s throughput
    - Tests HBM3 memory performance
4. **NCCL AllReduce (Multi-GPU)**
    - Synchronizes data across all GPUs
    - Measures collective communication bandwidth
    - Tests NVLink/InfiniBand performance
5. **Training Throughput (Multi-GPU)**
    - Simulates training with synthetic data
    - Measures tokens per second
    - Tests end-to-end training performance

Key code sections:

```python
# MatMul Benchmark
def benchmark_matmul(size=16384, dtype=torch.bfloat16):
    A = torch.randn(size, size, dtype=dtype, device='cuda')
    B = torch.randn(size, size, dtype=dtype, device='cuda')

    # Warmup
    for _ in range(10):
        C = torch.matmul(A, B)
    torch.cuda.synchronize()
```

```python
    # Benchmark
    start = time.time()
    for _ in range(100):
        C = torch.matmul(A, B)
    torch.cuda.synchronize()
    elapsed = time.time() - start

    # Calculate TFLOPS
    flops = 2 * size**3 * 100  # 2*N^3 per matmul, 100 iterations
    tflops = flops / elapsed / 1e12
    return tflops
```

5.4 Demo Scripts

`demo_banner.sh` **Purpose:** Display a colorful ASCII banner for presentations.

What it does: - Clears screen - Shows ASCII art "NEBIUS" logo - Displays cluster configuration summary - Lists demo contents

Sample output:

```
...

Cluster Configuration:

  Nodes:           2 (Master + Worker)
  GPUs:            16x NVIDIA H100 80GB HBM3
  GPU Memory:      1.28 TB Total
```

`demo_status.py` **Purpose:** Real-time GPU monitoring dashboard using Rich library.

What it does: - Shows live GPU utilization percentage - Displays memory usage per GPU - Shows temperature and power draw - Updates every second - Color-coded status indicators

Usage:

```
python demo_status.py
```

Output format:

```
       NVIDIA H100 GPU Status
 GPU    Util       Memory      Temp    Power

  0     95%     65G / 80G     62C     420W
  1     93%     64G / 80G     61C     415W
 ...    ...        ...        ...     ...
```

`tmux_demo.sh` **Purpose:** Create a multi-pane terminal layout for demos.

What it does: - Creates a tmux session with 4 panes - Pane 0: GPU status (gpustat) - Pane 1: GPU metrics (nvidia-smi) - Pane 2: Training logs - Pane 3: Command input

Layout:

```
  Pane 0            Pane 2
  GPU Status        Training Logs
  (gpustat)

  Pane 1            Pane 3
  GPU Metrics       Commands
  (nvidia-smi)
```

Usage:

```
./tmux_demo.sh start      # Create session
tmux attach -t gpu_demo   # Attach to session
./tmux_demo.sh kill       # Kill session
```

`run_demo.sh` **Purpose:** Automated interactive demo sequence.

What it does: 1. Shows welcome banner 2. Displays hardware configuration 3. Runs health check 4. Runs benchmarks 5. Runs training demo 6. Shows results summary

Modes:

```
./run_demo.sh full        # Complete demo (15-20 min)
./run_demo.sh quick       # Quick overview (5 min)
./run_demo.sh benchmark   # Benchmarks only
./run_demo.sh training    # Training only
```

`monitor_nccl.sh` **Purpose:** Monitor NCCL, NVLink, and InfiniBand traffic.

What it does: - Shows NCCL environment variables - Displays NVLink status and bandwidth - Shows InfiniBand port states - Live monitoring mode

Usage:

```
./monitor_nccl.sh all    # All info
./monitor_nccl.sh nvlink # NVLink only
./monitor_nccl.sh ib     # InfiniBand only
./monitor_nccl.sh live   # Continuous monitoring
```

---

6. Exercise 1: LLM Fine-Tuning

6.1 What is Fine-Tuning?

Fine-tuning takes a pre-trained model and adapts it for a specific task:

```
Pre-trained Model          Fine-tuned Model
(General knowledge)   -->  (Specific skill)

Qwen2-7B-Instruct     -->  Function Calling Expert
(Chat assistant)           (API/tool usage)
```

6.2 What is LoRA?

LoRA (Low-Rank Adaptation) is an efficient fine-tuning method:

Traditional Fine-Tuning: - Update ALL 7 billion parameters - Requires massive memory - Slow training

LoRA Fine-Tuning: - Freeze original weights - Add small trainable "adapters" - Only update 161 million parameters (2.1%) - Much faster and memory-efficient

```
Original Weight Matrix W (4096 x 4096 = 16M params)



  W_original (frozen, not updated)
         +
  ΔW = A × B (trainable)
  A: 4096 x 64 = 262K params
  B: 64 x 4096 = 262K params
  Total: 524K params (vs 16M)
```

6.3 The Dataset

We use the Glaive Function Calling dataset:

Sample data:

```json
{
  "instruction": "You have access to these functions: get_weather(city)",
  "input": "What's the weather in Tokyo?",
  "output": "<function_call>get_weather(city='Tokyo')</function_call>"
}
```

The model learns to: 1. Understand when to call a function 2. Extract the correct parameters 3. Format the function call properly

6.4 Training Flow

```
                      TRAINING PIPELINE

  1. Load Data

       Dataset   > Tokenize  > Create Batches


  2. Forward Pass (on each GPU)

        Input     >    Model     >    Output


  3. Calculate Loss

                            Loss = CrossEntropy
                            (predicted vs actual)


  4. Backward Pass

                            Calculate Gradients
                            (how to update weights)


  5. Gradient Synchronization

      GPU0  GPU1  GPU2  GPU3  ...




            NCCL AllReduce      <-- Average gradients
           (via NVLink/IB)          across all GPUs
```

```
  6. Update Weights


           Optimizer Step
           (AdamW update)


  7. Repeat for next batch
```

6.5 Distributed Data Parallel (DDP)

How training is distributed across GPUs:

```
Global Batch Size = 64 samples


                    DATA DISTRIBUTION

  Total: 64 samples

            GPU 0: 8 samples (batch_size=2 × accum=4)
            GPU 1: 8 samples
            GPU 2: 8 samples
            GPU 3: 8 samples
            GPU 4: 8 samples
            GPU 5: 8 samples
            GPU 6: 8 samples
            GPU 7: 8 samples

  Each GPU processes different data, computes gradients,
  then all GPUs synchronize via AllReduce
```

---

7. Exercise 2: GPU Benchmarks

7.1 Health Check

What it tests: - All GPUs are visible to PyTorch - Each GPU has expected memory (80 GB) - No hardware errors

Expected output:

```
GPU Health Check:
  GPU 0: NVIDIA H100 80GB HBM3 - 79.19 GB available - PASS
  GPU 1: NVIDIA H100 80GB HBM3 - 79.19 GB available - PASS
```

```
  ...
  Total: 8/8 GPUs healthy
```

7.2 Matrix Multiplication (TFLOPS)

What it tests: - Raw compute performance - Tensor core utilization

How it works:

```
Matrix A (16384 x 16384) × Matrix B (16384 x 16384) = Matrix C

FLOPs per MatMul = 2 × N³ = 2 × 16384³ = 8.8 trillion operations

Time: ~12ms per operation

TFLOPS = 8.8T / 0.012s = 733 TFLOPS
```

Expected results: | Precision | Expected | H100 Peak | |———-|———-|———-| | BF16 | >700 TFLOPS | 989 TFLOPS | | FP32 | >60 TFLOPS | 67 TFLOPS |

7.3 Memory Bandwidth

What it tests: - HBM3 memory throughput - Memory subsystem health

How it works:

```
1. Allocate large tensor (32 GB)
2. Copy tensor within GPU memory
3. Measure time and calculate bandwidth

Bandwidth = Data Copied / Time
         = 32 GB / 0.01s
         = 3.2 TB/s
```

Expected results: | Metric | Expected | H100 Peak | |——-|———-|———-| | Bandwidth | >2.5 TB/s | 3.35 TB/s |

7.4 NCCL AllReduce

What it tests: - Inter-GPU communication - NVLink bandwidth (intra-node) - InfiniBand bandwidth (inter-node)

How it works:

```
AllReduce Operation:

Before:
  GPU0: [1, 2, 3, 4]
  GPU1: [5, 6, 7, 8]
  GPU2: [9, 10, 11, 12]
  GPU3: [13, 14, 15, 16]
```

```
After (sum reduction):
  GPU0: [28, 32, 36, 40]
  GPU1: [28, 32, 36, 40]
  GPU2: [28, 32, 36, 40]
  GPU3: [28, 32, 36, 40]

All GPUs end up with the same summed values.
```

Expected results: | Configuration | Expected | Notes | |———–|——–|—–| | 8 GPUs (NVLink) | >400 GB/s | All via NVLink | | 16 GPUs (IB) | >400 GB/s | Cross-node via IB |

---

8. Optimizations Explained

8.1 Overview of Optimizations

Optimization	Default	Optimized	Impact
Batch size	2	4	+30-50% throughput
DataLoader workers	4	8	+5-10% throughput
Prefetch factor	2	4	Reduced data stalls
NCCL IB	Default	Tuned	+10-20% communication

8.2 Batch Size Optimization

What is batch size? Number of samples processed before updating weights.

Why increase it?

```
Smaller batch (2):
  - Forward pass: 10ms
  - Backward pass: 15ms
  - Communication: 20ms
  - GPU idle time: HIGH (waiting for communication)

Larger batch (4):
  - Forward pass: 18ms
  - Backward pass: 28ms
  - Communication: 20ms (same)
  - GPU idle time: LOW (more compute per sync)
```

Trade-off: - Larger batch = more memory usage - H100 has 80 GB - we can fit batch size 4

8.3 DataLoader Optimization

What is DataLoader? Component that loads and preprocesses training data.

Default settings:

```yaml
dataloader_num_workers: 4    # 4 CPU workers loading data
prefetch_factor: 2           # Pre-load 2 batches ahead
```

Optimized settings:

```yaml
dataloader_num_workers: 8    # 8 CPU workers (more parallelism)
prefetch_factor: 4           # Pre-load 4 batches (reduce stalls)
```

Why it helps:

```
Without prefetching:
GPU: [Compute]----[Wait]----[Compute]----[Wait]
CPU:          [Load]                [Load]

With prefetching:
GPU: [Compute]----[Compute]----[Compute]
CPU: [Load][Load][Load][Load][Load]

Data is ready before GPU needs it!
```

8.4 NCCL Optimization

What is NCCL? NVIDIA Collective Communications Library - handles GPU-to-GPU data transfer.

Key environment variables:

```bash
# Enable InfiniBand (don't disable it)
export NCCL_IB_DISABLE=0

# Use correct IB interface
export NCCL_IB_GID_INDEX=3

# Enable GPU Direct RDMA (GPU talks directly to network)
export NCCL_NET_GDR_LEVEL=5

# Use Ring algorithm for AllReduce
export NCCL_ALGO=Ring
```

GPU Direct RDMA explained:

```
Without GPU Direct:
GPU Memory -> CPU Memory -> Network -> CPU Memory -> GPU Memory
              (2 extra copies!)

With GPU Direct RDMA:
GPU Memory -> Network -> GPU Memory
              (Direct transfer!)
```

8.5 Gradient Checkpointing

What it does: Trades compute for memory by recomputing activations during backward pass.

Without checkpointing:

```
Forward: Store all activations in memory
  Layer 1 -> [Activation 1] -> Layer 2 -> [Activation 2] -> ...
  Memory: ALL activations stored (HIGH memory usage)

Backward: Use stored activations
  Memory: Still high
```

With checkpointing:

```
Forward: Only store checkpoints (every few layers)
  Layer 1 -> [Checkpoint] -> Layer 2 -> Layer 3 -> [Checkpoint] -> ...
  Memory: Only checkpoints stored (LOW memory usage)

Backward: Recompute activations from checkpoints
  Memory: Low (but more compute)
```

Impact: - Memory usage: -40% - Compute overhead: +20% - Net benefit: Can use larger batch sizes

8.6 BF16 (Brain Float 16) Precision

What is BF16? A 16-bit floating point format optimized for deep learning.

```
FP32 (32 bits):
  Sign: 1 bit
  Exponent: 8 bits
  Mantissa: 23 bits

BF16 (16 bits):
  Sign: 1 bit
  Exponent: 8 bits  (same range as FP32!)
  Mantissa: 7 bits  (less precision)
```

Why use BF16? - Same range as FP32 (important for training stability) - Half the memory - 2x faster compute on Tensor Cores - H100 optimized for BF16 operations

---

9. Demo Flow

9.1 Recommended Demo Sequence

```
                      DEMO TIMELINE
```

```
  0:00
         1. INTRODUCTION
            - Show demo_banner.sh
            - Explain cluster configuration
  2:00

  2:00
         2. HARDWARE VALIDATION
            - Run hardware_info.sh summary
            - Show GPU topology
            - Explain NVLink connections
  4:00

  4:00
         3. GPU BENCHMARKS
            - Run health check
            - Show MatMul TFLOPS
            - Explain NCCL AllReduce
  7:00

  7:00
         4. LLM TRAINING DEMO
            - Start tmux_demo.sh
            - Run training
            - Watch GPU utilization
            - Observe loss decreasing
 12:00

 12:00
         5. RESULTS & SUMMARY
            - Show final loss
            - Show checkpoint files
            - Recap performance numbers
 15:00
```

9.2 Quick Commands Reference

```
# Show banner
./demo_banner.sh

# Hardware info
./hardware_info.sh summary
./hardware_info.sh gpu

# Start multi-pane view
```

```bash
./tmux_demo.sh start
tmux attach -t gpu_demo

# Run benchmarks
source multinode.sh exercise2

# Run training
source multinode.sh exercise1

# Monitor GPUs
python demo_status.py
gpustat -i 1 --color

# View TensorBoard
tensorboard --logdir /home/supreethlab/training/logs --port 6006
```

---

10. Performance Results

10.1 Benchmark Results Summary

Benchmark	8 GPUs	16 GPUs	Status
GPU Health	8/8 PASS	16/16 PASS	PASS
MatMul (BF16)	730.97 TFLOPS	N/A	PASS
Memory Bandwidth	3.02 TB/s	N/A	PASS
NCCL AllReduce	442.28 GB/s	435.91 GB/s	PASS
Training Throughput	331,842 tok/s	136,970 tok/s	PASS

10.2 Training Results

Metric	8 GPUs	16 GPUs
Model	Qwen2-7B-Instruct	Qwen2-7B-Instruct
Trainable Params	161M (2.1%)	161M (2.1%)
Training Steps	100	100
Initial Loss	~1.7	~1.7
Final Training Loss	0.3895	0.3918
Final Eval Loss	0.396	0.3958
Time per Step	~1.7s	~1.9s

10.3 Scaling Analysis

```
Per-GPU Throughput:
  8 GPUs:  331,842 / 8  = 41,480 tokens/s per GPU
  16 GPUs: 136,970 / 16 = 8,561 tokens/s per GPU
```

```
Scaling Efficiency:
  Expected (linear): 41,480 tokens/s per GPU
  Actual: 8,561 tokens/s per GPU
  Efficiency: 8,561 / 41,480 = 20.6%
```

Why the drop in multi-node? 1. InfiniBand latency higher than NVLink 2. Cross-node gradient sync overhead 3. Communication not fully overlapped with compute

How to improve: 1. Larger batch sizes (more compute per sync) 2. DeepSpeed ZeRO-3 (requires code changes) 3. Gradient compression 4. Better overlap of compute and communication

---

11. Troubleshooting Guide

11.1 Common Errors

CUDA Out of Memory

```
RuntimeError: CUDA out of memory
```

Solutions: 1. Reduce batch size: `per_device_train_batch_size: 1` 2. Enable gradient checkpointing (already enabled) 3. Check for other GPU processes: `nvidia-smi` 4. Use smaller model or more aggressive LoRA

NCCL Timeout

```
torch.distributed.DistStoreError: Timed out after 901 seconds
```

Solutions: 1. Check network connectivity: `ping 10.2.0.0` 2. Verify SSH works: `ssh 10.2.0.0 hostname` 3. Start worker before master 4. Check firewall rules

InfiniBand Warnings

```
libibverbs: Warning: couldn't load driver 'libvmw_pvrdma-rdmav34.so'
```

Solution: Ignore - this is harmless. VMware driver not needed.

11.2 Diagnostic Commands

```bash
# Check GPU status
nvidia-smi

# Check GPU processes
nvidia-smi --query-compute-apps=pid,process_name,used_memory --format=csv

# Check InfiniBand
ibstat | grep -E "State|Rate"
```

```bash
# Check connectivity
ping -c 3 10.2.0.0

# Check disk space
df -h /home/supreethlab/training/

# Kill stuck processes
pkill -f torchrun

# View logs
tail -f /home/supreethlab/training/logs/*.log
```

---

12. Glossary

Term	Definition
AllReduce	Collective operation that sums values across all GPUs and distributes the result to all
BF16	Brain Float 16 - a 16-bit floating point format optimized for deep learning
Batch Size	Number of samples processed before updating model weights
Checkpoint	Saved state of model during training for recovery or evaluation
DDP	Distributed Data Parallel - PyTorch's method for multi-GPU training
DeepSpeed	Microsoft library for efficient distributed training
Fine-tuning	Adapting a pre-trained model for a specific task
Gradient	Direction and magnitude of weight updates during training
Gradient Accumulation	Summing gradients over multiple batches before updating
Gradient Checkpointing	Trading compute for memory by recomputing activations
HBM3	High Bandwidth Memory 3 - fast GPU memory technology
InfiniBand	High-speed networking technology for cluster computing
LoRA	Low-Rank Adaptation - efficient fine-tuning method
Loss	Measure of model prediction error (lower is better)
NCCL	NVIDIA Collective Communications Library
NVLink	NVIDIA's high-speed GPU interconnect
RDMA	Remote Direct Memory Access - direct memory access over network
TFLOPS	Trillion floating-point operations per second
Tensor Cores	Specialized GPU cores for matrix operations

Term	Definition
torchrun	PyTorch distributed training launcher
Warmup	Gradual increase of learning rate at training start
ZeRO	Zero Redundancy Optimizer - DeepSpeed memory optimization

---

Appendix A: Quick Reference Card

```
                    QUICK REFERENCE CARD


  LAUNCHERS:
    source singlenode.sh exercise1    # 8-GPU training
    source singlenode.sh exercise2    # 8-GPU benchmarks
    source multinode.sh exercise1     # 16-GPU training
    source multinode.sh exercise2     # 16-GPU benchmarks

  MONITORING:
    ./demo_banner.sh                  # Show banner
    ./hardware_info.sh summary        # Cluster info
    python demo_status.py             # Rich GPU dashboard
    gpustat -i 1 --color              # Simple GPU status
    nvidia-smi                        # Detailed GPU info

  TMUX:
    ./tmux_demo.sh start              # Create demo session
    tmux attach -t gpu_demo           # Attach to session
    Ctrl+B, Arrow                     # Navigate panes
    Ctrl+B, z                         # Zoom pane
    Ctrl+B, d                         # Detach
    ./tmux_demo.sh kill               # Kill session

  TENSORBOARD:
    tensorboard --logdir ~/training/logs --port 6006 --bind_all

  TROUBLESHOOTING:
    pkill -f torchrun                 # Kill training
    nvidia-smi                        # Check GPU status
    ibstat | grep State               # Check InfiniBand
```

Document End

Repository URLs: - 8-GPU: https://github.com/drmysore/nebius-h100-8gpu-benchmark - 16-GPU: https://github.com/drmysore/nebius-h100-16gpu-benchmark

Author: Supreeth Mysore **Date:** December 17, 2025 **Version:** 1.0