HPC for Advanced ML/DL

Cynara Justine

College of Engineering Trivandrm

Workshop • January 2026

Workshop Schedule

10:00-11:30 Session 1: HPC Fundamentals & DDP Introduction

11:45-13:00 Session 2: Hands-On DDP Implementation

14:00-15:00 Session 3: Advanced Optimization

15:15-16:00 Session 4: SLURM & Deployment

Workshop Overview

1. Introduction to High-Performance Computing (HPC) for AI
2. Parallelization Techniques in Machine Learning
3. Deep Learning on HPC Systems
4. Big Data and HPC for ML/DL Projects
5. Optimization and Resource Management in HPC for AI

Workshop Overview: Demo

Demo 1: Single GPU baseline training on CIFAR-10

Demo 2: Multi-GPU training with DataParallel (DDP)

Demo 3: Mixed precision training with AMP

Demo 4: Performance profiling and optimization

Why HPC for ML/DL?

Speed: Multi-GPU training reduces training time from days to hours

Scale: Train larger models beyond single GPU memory

Efficiency: Mixed precision + optimization = faster convergence

Research: Iterate faster on hyperparameters and models

GPU Architecture Fundamentals

GPU Memory Hierarchy: L1/L2 cache → on-chip memory → HBM

Tensor Cores (A100): FP16 matrix ops (125 TFLOPS)

Memory Bandwidth: A100 has 2TB/s. Maximize utilization

Multi-GPU Communication: NVLink (600GB/s), PCIe (64GB/s)

Utilization Metrics: SM occupancy, memory coalescing, synchronization

CUDA & PyTorch Programming Model

CUDA Runtime: Manages GPU memory, kernels, streams, synchronization

PyTorch Abstractions: Tensors on device, ops are implicit GPU kernels

cuDNN/cuBLAS: Optimized libs for neural net ops

Streams: Enable overlapping GPU compute with data transfers

Synchronization Points: .to(device), .item(), backward() block until GPU work completes

Distributed Training Concepts

- **Data Parallelism:** Same model on multiple GPUs, different batches, sync gradients
- **Model Parallelism:** Split model across GPUs
- **Parameter Sharding:** Distribute parameters + optimizer state (ZeRO, FSDP)
- **Synchronization Overhead:** AllReduce during backward; communication is expensive
- **Scaling Laws:** Linear speedup requires careful tuning; real-world: 6-7x on 8 GPUs

Mixed Precision Training Deep Dive

FP32 vs FP16: Half precision = 2x memory, 2-3x speed

Gradient Underflow: FP16 range: 6×10^{-8} to 6×10^{4}. Small gradients disappear

Loss Scaling Mechanism: Scale loss before backward, unscale after

Autocast/GradScaler: PyTorch automates FP16 selection and scaling

Accuracy Trade-offs: Typically <0.1% loss in final accuracy

Data Loading & I/O Optimization

CPU-GPU Bottleneck: 20-30% of training time is data loading

num_workers: Multiprocessing to prefetch batches. Typical: 4-8 workers

pin_memory=True: Pin data in host memory for faster PCIe transfer

Prefetching & Double Buffering: GPU trains on batch N while CPU loads batch N+1

Batch Size Sweet Spot: Larger batch = better GPU utilization

🎬 LIVE DEMO 1: Single GPU Baseline

Running: demo_1_single_gpu.ipynb

Trains SimpleCNN on CIFAR-10 using single A100 GPU.

Expected: ~2,474,506 parameters

Expected throughput: ~1000-1200 samples/sec

DataParallel vs. DistributedDataParallel

DataParallel (DP): Single process, replicate model to all GPUs

DistributedDataParallel (DDP): Multi-process, one rank per GPU, sync via NCCL

Why DDP? Less overhead, better scaling, works with SLURM/torchrun

DataParallel in Notebooks: Works without special setup

Gradient Synchronization: AllReduce after backward

🎬 LIVE DEMO 2: Multi-GPU with DataParallel

Running: demo_2_ddp_training.ipynb

Trains same model across all 8 A100 GPUs.

Expected: Distributed batch (16 samples per GPU)

Expected speedup: ~6-7x (due to communication overhead)

Checkpoint Saving & Resumption

Why Checkpoints? Fault tolerance, model selection, resuming interrupted jobs

What to Save: Model weights, optimizer state, scheduler state, epoch counter

DDP Gotcha: Save from rank 0 only (otherwise corruption)

torch.save/load: Handles DataParallel's module prefix

Resume Training: Load checkpoint, set epoch, continue loop

Mixed Precision in Practice

Autocast Context: Auto-casts ops to FP16/TF32 where safe

GradScaler: Scales loss, unscales gradients—prevents underflow

Typical Pattern: with autocast(): forward; scaler.scale(loss).backward(); scaler.step()

Numerics: Batch norm/LayerNorm stay FP32; linear/conv do FP16

Compatibility: Works with single GPU, DataParallel, DDP

🎬 LIVE DEMO 3: Mixed Precision with AMP

Running: demo_3_mixed_precision.ipynb

FP32 vs AMP (FP16 + FP32) on ResNet-18.

Expected: AMP time ~50-60% of FP32

Expected speedup: ~1.5-2x with mixed precision

Profiling & Performance Analysis

torch.profiler: Profile GPU kernels, measure op-level time

nvidia-smi: Real-time GPU utilization, memory, power, temperature

Bottleneck Detection: If GPU util < 80%, you have a bottleneck

Metrics to Track: Throughput, GPU memory, GPU utilization %, temperature

Amdahl's Law: Parallelization helps proportionally to parallelizable fraction

🎬 LIVE DEMO 4: Profiling & Optimization

Running: demo_4_profiling_optimization.ipynb

Three benchmarks:

1. Data loading (vary num_workers: 0, 2, 4, 8)
2. Batch size vs. memory & throughput
3. FP32 vs. AMP throughput comparison

Production Deployment Strategy

torchrun: Multi-GPU launcher; handles process groups automatically

SLURM Integration: sbatch script with torchrun --nproc_per_node=N

Multi-Node Setup: Set MASTER_ADDR, MASTER_PORT, RANK, WORLD_SIZE

Monitoring: Use tensorboard, wandb, or mlflow for metrics

Fault Tolerance: Save checkpoints frequently; support resume-from-checkpoint

Combined Optimization Results

Demo 1: Single GPU

~1000 samples/sec

(Baseline)

Demo 2: 8 GPUs

~6500 samples/sec

(6.5× speedup)

Demo 3: +AMP

~1.7× speedup

(×8 GPUs = 11× total)

Combined: ~11× speedup = 11,000 samples/sec

Troubleshooting & Common Issues

OOM: Reduce batch size, enable gradient checkpointing, use AMP

DDP "process group not initialized": Use torchrun, not python

NaN loss during AMP: Reduce LR, increase GradScaler initial_scale

Slow data loading: Increase num_workers, enable pin_memory

Uneven GPU utilization: Check thermal throttling, batch assignment

Training diverges after scaling: Increase LR with GPU count, use warmup

Next Steps & Advanced Topics

PyTorch Lightning: High-level abstraction handling DDP, AMP, checkpointing

FSDP: Parameter sharding + data parallelism for huge models

Gradient Checkpointing: Trade compute for memory; fit larger models

Horovod: Multi-node distributed training framework

Custom CUDA Kernels: Domain-specific ops for 10-50× speedup

Summary & Key Takeaways

Baseline is crucial: Measure single-GPU performance first

Distributed training scales: 8 GPUs = 6-7× speedup with data parallelism

AMP is essential: 1.5-2× faster on modern GPUs, minimal accuracy loss

Profile before optimizing: Identify bottlenecks; target systematically

Combined techniques dominate: DDP + AMP + optimized I/O = 10-15× speedup

Questions & Discussion

Let's dive deeper into topics of interest

Feedback, suggestions, and questions welcome!