CS336 Language Modeling from Scratch

Lecture 8: Distributed Training Implementation and Collective Operations

Stanford University, Spring 2025

Abstract

This lecture provides hands-on implementation of distributed training strategies for large language models, building on the theoretical foundations from Lecture 7. Key topics include collective communication primitives (AllReduce, AllGather, ReduceScatter), NCCL and PyTorch.distributed implementation details, performance benchmarking of communication operations, and complete code implementations of data parallelism, tensor parallelism, and pipeline parallelism. The lecture bridges theory and practice by showing concrete distributed training implementations on multi-GPU systems.

Enhanced Summary by:
GitHub: HtmMhmd — LinkedIn: Hatem Mohamed

Contents

1 Introduction and Course Context

1.1 Lecture Overview

This is the second systems lecture focusing on multi-GPU parallelism, transitioning from single-GPU optimization (Lecture 6) to distributed training across multiple machines. The primary goal is to concretize the parallelism concepts from Lecture 7 through practical code implementations.

Key Point: Hardware Memory Hierarchy

Understanding the GPU cluster memory hierarchy is crucial for optimization: L1 cache (fastest, smallest) → HBM → NVLink → NVSwitch → Ethernet (slowest, largest). The goal is to minimize data transfer across slower links.

Learning Objectives:

1. Implement collective communication operations using PyTorch.distributed
2. Benchmark communication performance across GPU interconnects
3. Build complete distributed training systems from scratch
4. Understand practical trade-offs in parallelization strategies

1.2 Multi-GPU System Architecture

Definition: GPU Cluster Hierarchy

Modern GPU clusters consist of nodes (typically 8 GPUs each) connected via high-speed interconnects, with computation happening on Streaming Multiprocessors (SMs) within each GPU.

Communication Hierarchy:

- **Intra-SM**: L1 cache (extremely fast, very small)
- **Intra-GPU**: High Bandwidth Memory (HBM)
- **Intra-Node**: NVLink connections between GPUs
- **Inter-Node**: NVSwitch or InfiniBand connections

Warning: Bottleneck Management

Data transfer is typically the bottleneck in distributed training. Structuring computation to avoid transfer bottlenecks while maintaining high arithmetic intensity is the primary optimization challenge.

2 Collective Communication Primitives

2.1 Terminology and Concepts

Definition: Distributed Training Terminology

- **World Size**: Total number of devices/processes (e.g., 4)
- **Rank**: Device identifier (0, 1, 2, 3 for 4 devices)
- **Collective Operation**: Communication involving all devices simultaneously

2.2 Core Collective Operations

These operations are foundational primitives from parallel programming literature (since the 1980s) that provide better abstractions than point-to-point communication.

2.2.1 Broadcast Operation

Algorithm: Broadcast Implementation

Purpose: Distribute one tensor from a single rank to all ranks
Operation:

1. Source rank has tensor t_0
2. All ranks receive identical copy of t_0

Use Cases: Model initialization, hyperparameter distribution

2.2.2 Scatter and Gather Operations

Algorithm: Scatter Operation

Purpose: Distribute different parts of a tensor to different ranks
Operation:

1. Source has tensor with multiple values $[t_0, t_1, t_2, t_3]$
2. Each rank receives different portion: rank 0 gets t_0, rank 1 gets t_1, etc.

Algorithm: Gather Operation

Purpose: Collect tensors from all ranks to a single destination
Operation:

1. Each rank has different tensor value
2. Destination rank collects all values into single tensor

2.2.3 Reduce and AllReduce Operations

Algorithm: Reduce Operation

Purpose: Apply commutative operation (sum, max, min) across all ranks
Operation:

1. Each rank contributes a tensor
2. Operation applied elementwise across all tensors
3. Result stored on designated rank

Definition: AllReduce Equivalence

AllReduce is mathematically equivalent to Reduce followed by Broadcast:

$$\text{AllReduce}(x) \equiv \text{Broadcast}(\text{Reduce}(x)) \tag{1}$$

This equivalence is also implemented as ReduceScatter + AllGather with identical communication cost.

2.2.4 Advanced Operations

Algorithm: AllGather Operation

Purpose: Gather operation with all ranks as destinations
Operation:

1. Each rank has different tensor
2. All ranks receive concatenated result of all tensors

Algorithm: ReduceScatter Operation

Purpose: Reduce operation with scattered output
Operation:

1. Apply reduction across ranks
2. Scatter different parts of result to different ranks

Key Point: Operation Relationships

Understanding primitive relationships enables optimization:

- Scatter is inverse of Gather
- "All" prefix means operation applies to all destinations
- "Reduce" means applying associative/commutative operations

3 Hardware Implementation and NCCL

3.1 Traditional vs Modern GPU Networking

3.1.1 Traditional Architecture Limitations

Warning: Traditional GPU Communication Bottlenecks

Classical setup: GPU → PCI-E bus → CPU → Ethernet → Remote CPU → PCI-E → Remote GPU
This introduces significant overhead through:

- Kernel space transitions
- Buffer copying operations
- Transport protocol overhead
- General-purpose bus sharing

3.1.2 Modern High-Performance Interconnects

Performance: NVIDIA NVLink/NVSwitch Architecture

Modern GPU clusters bypass CPU bottlenecks:

- **NVLink**: Direct GPU-to-GPU connections within nodes
- **NVSwitch**: Direct GPU-to-GPU connections across nodes
- **Performance**: H100 provides 18 NVLink Gen4 connections = 900 GB/s total bandwidth

Speed Comparison:

- SM → HBM: 4000 GB/s (4× faster than NVLink)
- NVLink: 900 GB/s
- PCI-E: Much slower than NVLink
- Ethernet: Significantly slower than NVLink

3.2 NCCL: NVIDIA Collective Communications Library

Definition: NCCL Architecture

NCCL translates high-level collective operations into optimized low-level GPU kernels and communication patterns, handling topology optimization and efficient routing automatically.

Algorithm: NCCL Operation Workflow

1. **Initialization**: Discover and map hardware topology
2. **Optimization**: Determine optimal communication paths
3. **Execution**: Launch CUDA kernels for send/receive operations
4. **Coordination**: Manage synchronization across devices

Key NCCL Features:

- Automatic topology discovery and optimization
- High-performance kernel implementations
- Support for multiple communication patterns
- Integration with CUDA programming model

4 PyTorch Distributed Implementation

4.1 PyTorch.distributed Architecture

Definition: PyTorch Distributed Abstraction

PyTorch.distributed provides Python-level interfaces for collective operations with multiple backend support, enabling portable distributed training across different hardware configurations.

Supported Backends:

- **NCCL**: GPU-optimized backend for NVIDIA hardware
- **Gloo**: CPU backend for debugging and non-GPU environments
- **MPI**: Message Passing Interface for HPC environments

4.2 Practical Implementation Examples

4.2.1 Process Group Initialization

Code: Distributed Training Setup

```
import torch
import torch.distributed as dist

def run_distributed_function(func, world_size=4):
    """Wrapper for running function across multiple processes"""
    # Initialize process group
    dist.init_process_group(
        backend='nccl',  # Use NCCL for GPU
        world_size=world_size,
        rank=get_rank()  # 0, 1, 2, 3 for 4 processes
    )

    # Synchronization point
    dist.barrier()  # Wait for all processes

    # Execute distributed function
    func()

    # Cleanup
    dist.destroy_process_group()
```

4.2.2 AllReduce Implementation

Code: AllReduce Example

```
def allreduce_example():
    rank = dist.get_rank()

    # Create rank-specific tensor: [0,1,2,3] + rank
    tensor = torch.arange(4) + rank
    print(f"Rank␣{rank}␣before␣AllReduce:␣{tensor}")

    # Perform AllReduce with sum operation
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM, async_op=False)

    print(f"Rank␣{rank}␣after␣AllReduce:␣{tensor}")
    # Result: [6, 10, 14, 18] = sum across all ranks
```

4.2.3 ReduceScatter and AllGather

Code: ReduceScatter + AllGather

```
def reduce_scatter_allgather_example():
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # ReduceScatter
    input_tensor = torch.arange(world_size) + rank
    output_scalar = torch.zeros(1)

    dist.reduce_scatter(output_scalar, input_tensor, op=dist.
        ReduceOp.SUM)
    print(f"Rank␣{rank}␣ReduceScatter␣output:␣{output_scalar}")

    # AllGather
    gathered_tensors = [torch.zeros(1) for _ in range(world_size)]
    dist.all_gather(gathered_tensors, output_scalar)

    result = torch.cat(gathered_tensors)
    print(f"Rank␣{rank}␣AllGather␣result:␣{result}")
    # Demonstrates AllReduce = ReduceScatter + AllGather
```

5 Communication Performance Benchmarking

5.1 Benchmarking Methodology

Algorithm: Communication Benchmarking Protocol

1. **Warmup**: Execute operation once to load kernels
2. **Synchronization**: Barrier to ensure clean timing
3. **Timing**: Start clock, execute operation, synchronize, stop clock
4. **Analysis**: Calculate bandwidth and compare to theoretical peaks

5.2 AllReduce Performance Analysis

Code: AllReduce Benchmarking

```
def benchmark_allreduce():
    num_elements = 100_000_000  # 100M elements
    world_size = 4

    # Create test tensor
    tensor = torch.randn(num_elements, device='cuda')

    # Warmup
    dist.all_reduce(tensor.clone())
    dist.barrier()

    # Benchmark
    start_time = time.time()
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM)
    torch.cuda.synchronize()
    duration = time.time() - start_time

    # Calculate bandwidth
    bytes_per_element = 4  # FP32
    size_bytes = num_elements * bytes_per_element
    total_bytes = 2 * world_size * size_bytes  # Factor of 2 for
        send+receive
    bandwidth_gbps = (total_bytes / duration) / (1024**3)

    print(f"AllReduce bandwidth: {bandwidth_gbps:.1f} GB/s")
```

Performance: Typical Benchmark Results

AllReduce Performance: 277 GB/s (may vary based on tensor size, hardware configuration)
ReduceScatter Performance: 70 GB/s (typically lower than AllReduce due to different optimization patterns)
Performance Factors:

- Tensor size and memory layout
- Number of participating devices
- Hardware interconnect topology
- NCCL optimization algorithms (e.g., SHARP acceleration)

6 Distributed Training Implementation

6.1 Data Parallelism (DDP)

Data parallelism splits the batch across devices while replicating the model on each device.

Algorithm: Data Parallelism Strategy

Model Distribution: Full model replicated on each rank **Data Distribution:** Batch split across ranks **Communication:** AllReduce gradients after backward pass

Code: Data Parallelism Implementation

```
def data_parallel_training():
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # Split batch across ranks
    batch_size = 128
    local_batch_size = batch_size // world_size
    start_idx = rank * local_batch_size
    end_idx = start_idx + local_batch_size

    # Get rank-specific data slice
    local_data = full_data[start_idx:end_idx]

    # Initialize model (same on all ranks)
    model = create_mlp(num_layers=4, hidden_dim=1024)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

    for step in range(num_steps):
        # Forward pass on local data
        output = model(local_data)
        loss = compute_loss(output, local_targets)

        # Backward pass
        loss.backward()

        # DDP: AllReduce gradients across ranks
        for param in model.parameters():
            dist.all_reduce(param.grad, op=dist.ReduceOp.AVG)

        # Update parameters (now synchronized)
        optimizer.step()
        optimizer.zero_grad()
```

Key Point: DDP Synchronization

AllReduce serves as implicit synchronization point. All ranks must reach this point before proceeding, ensuring lockstep training across devices.

6.2 Tensor Parallelism

Tensor parallelism splits the model's hidden dimensions across devices while keeping the full batch on each device.

Algorithm: Tensor Parallelism Strategy

Model Distribution: Split weight matrices along hidden dimension **Data Distribution:** Full batch replicated on each rank **Communication:** AllGather activations between layers

Code: Tensor Parallelism Implementation

```
def tensor_parallel_training():
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # Split hidden dimension across ranks
    hidden_dim = 1024
    local_hidden_dim = hidden_dim // world_size  # 256 per rank

    # Initialize model shards
    model_shard = create_mlp_shard(
        hidden_dim=local_hidden_dim,
        num_layers=4
    )

    # Forward pass with communication
    x = input_data  # Full batch on each rank

    for layer in model_shard.layers:
        # Compute local activations
        local_activations = layer(x)  # [batch_size,
            local_hidden_dim]

        # AllGather to reconstruct full activations
        gathered_activations = [
            torch.zeros_like(local_activations)
            for _ in range(world_size)
        ]
        dist.all_gather(gathered_activations, local_activations)

        # Concatenate to get full activations
        x = torch.cat(gathered_activations, dim=1)  # [batch_size,
            hidden_dim]

    return x
```

Warning: Tensor Parallelism Communication Overhead

Tensor parallelism requires AllGather operations between every layer, resulting in high communication volume. This approach works best with high-bandwidth intra-node connections (NVLink) but degrades rapidly across slower inter-node links.

6.3 Pipeline Parallelism

Pipeline parallelism splits the model by layers across devices and processes microbatches to minimize pipeline bubbles.

Algorithm: Pipeline Parallelism Strategy

Model Distribution: Consecutive layers assigned to different ranks **Data Distribution:** Full batch split into microbatches **Communication:** Point-to-point activation passing between adjacent ranks

Code: Pipeline Parallelism Implementation

```
def pipeline_parallel_training():
    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # Assign layers to ranks
    total_layers = 4
    layers_per_rank = total_layers // world_size  # 2 layers per
        rank

    start_layer = rank * layers_per_rank
    end_layer = start_layer + layers_per_rank
    local_layers = model.layers[start_layer:end_layer]

    # Split batch into microbatches
    batch_size = 128
    microbatch_size = 32
    num_microbatches = batch_size // microbatch_size

    for microbatch_idx in range(num_microbatches):
        if rank == 0:
            # First rank: use input data
            x = input_microbatches[microbatch_idx]
        else:
            # Receive from previous rank
            x = torch.zeros(microbatch_size, hidden_dim, device='
                cuda')
            dist.recv(x, src=rank-1)

        # Forward pass through local layers
        for layer in local_layers:
            x = layer(x)

        if rank < world_size - 1:
            # Send to next rank
            dist.send(x, dst=rank+1)
        else:
            # Last rank: compute loss
            loss = compute_loss(x, targets_microbatch)
```

Performance: Pipeline Parallelism Trade-offs

Advantages:

- Low communication bandwidth requirements
- Point-to-point communication pattern
- Scales well across slow inter-node links

Disadvantages:

- Pipeline bubbles reduce GPU utilization
- Complex scheduling for forward/backward overlap
- Microbatch size tuning required

7 Advanced Implementation Considerations

7.1 Communication-Computation Overlap

Algorithm: Asynchronous Communication Strategy

1. Use `async_op=True` in collective operations
2. Launch communication and continue computation
3. Synchronize only when results are needed
4. Overlap gradient computation with parameter synchronization

Code: Asynchronous Communication Example

```
def async_communication_example():
    # Start asynchronous AllReduce
    handle = dist.all_reduce(tensor, async_op=True)

    # Continue with other computation
    other_computation()

    # Wait for communication to complete when needed
    handle.wait()

    # Use synchronized tensor
    continue_with_tensor(tensor)
```

7.2 Memory and Synchronization Management

Warning: Synchronization Points

Collective operations serve as implicit synchronization barriers. Missing AllReduce calls will cause deadlocks, while extra synchronization can hurt performance.

Algorithm: Best Practices for Distributed Training

1. **Initialization**: Use identical random seeds across ranks
2. **Error Handling**: Implement timeout mechanisms for communication
3. **Memory Management**: Be careful with tensor device placement
4. **Debugging**: Use CPU backend (Gloo) for development/debugging

7.3 Framework Comparison: PyTorch vs JAX

Performance: JAX/TPU Approach

JAX provides higher-level declarative sharding specifications:

- Define sharding strategy declaratively
- Compiler automatically generates communication code
- Better for rapid experimentation
- Less control over low-level optimizations

Code: JAX Sharding Example

```
# JAX declarative approach (conceptual)
model = create_model()

# FSDP in ~10 lines
sharded_model = shard(model, axis_name='batch')

# Tensor parallelism
sharded_model = shard(model,
                      head_axis='model_parallel',
                      embed_axis='model_parallel')
```

8 Practical Trade-offs and Optimization

8.1 Parallelism Strategy Selection

Algorithm: Strategy Selection Framework

1. **Memory Constraints**: Does model fit on single device?
2. **Communication Bandwidth**: High-speed intra-node vs slower inter-node
3. **Implementation Complexity**: Development and maintenance costs
4. **Debugging Requirements**: Ease of troubleshooting distributed issues

Performance: Trade-off Analysis

Data Parallelism:

- ✓ Simple implementation and debugging
- ✓ Works well with slower networks
- × Limited by single-device memory for model size

Tensor Parallelism:

- ✓ Enables larger models than single device
- ✓ Good GPU utilization
- × High communication overhead
- × Requires high-bandwidth interconnects

Pipeline Parallelism:

- ✓ Low communication bandwidth requirements
- ✓ Scales across slow interconnects
- × Complex implementation with bubble optimization
- × GPU utilization challenges

8.2 Future Hardware Trends

Key Point: Hardware Evolution

Despite hardware improvements, the memory hierarchy will persist:

- Physical limits constrain single-device scaling
- Models will continue growing to hardware limits
- Hierarchical memory systems are fundamental to computer architecture
- Parallelization strategies remain relevant regardless of hardware advances

9 Conclusion and Implementation Guidance

9.1 Key Takeaways

This lecture demonstrated the practical implementation of distributed training strategies, showing how theoretical concepts from Lecture 7 translate into working code.

Key Point: Implementation Principles

1. **Start Simple**: Begin with data parallelism before moving to complex strategies
2. **Measure Performance**: Benchmark communication overhead before optimization
3. **Understand Trade-offs**: Balance implementation complexity against performance gains
4. **Use Abstractions**: Leverage PyTorch FSDP and other frameworks for production use

9.2 From Theory to Practice

Algorithm: Development Workflow

1. **Prototype**: Start with single-GPU implementation
2. **Scale Gradually**: Add data parallelism first
3. **Profile Bottlenecks**: Identify communication vs computation limits
4. **Optimize Strategically**: Apply advanced parallelism where needed
5. **Production Deploy**: Use established frameworks (FSDP, Megatron-LM)

Warning: Production Considerations

This lecture's implementations are educational. Production systems require additional complexity:

- Fault tolerance and recovery mechanisms
- Dynamic load balancing
- Memory optimization (activation checkpointing)
- Mixed precision training integration
- Complex scheduling for communication overlap

9.3 Assignment 2 Connection

The concepts and implementations from this lecture directly support Assignment 2 requirements:

- Understanding collective operations for gradient synchronization
- Implementing data parallelism for transformer training
- Benchmarking communication performance
- Debugging distributed training issues

The transition from single-GPU kernels (Lecture 6) to multi-GPU coordination (this lecture) provides the complete foundation for efficient large language model training at scale.

10 Chapter Mind Map

The following mind map provides a comprehensive visual overview of the distributed training implementation concepts covered in this lecture. The diagram illustrates the interconnected nature of collective operations, hardware optimizations, software frameworks, and parallelization strategies that form the foundation of modern large-scale language model training.

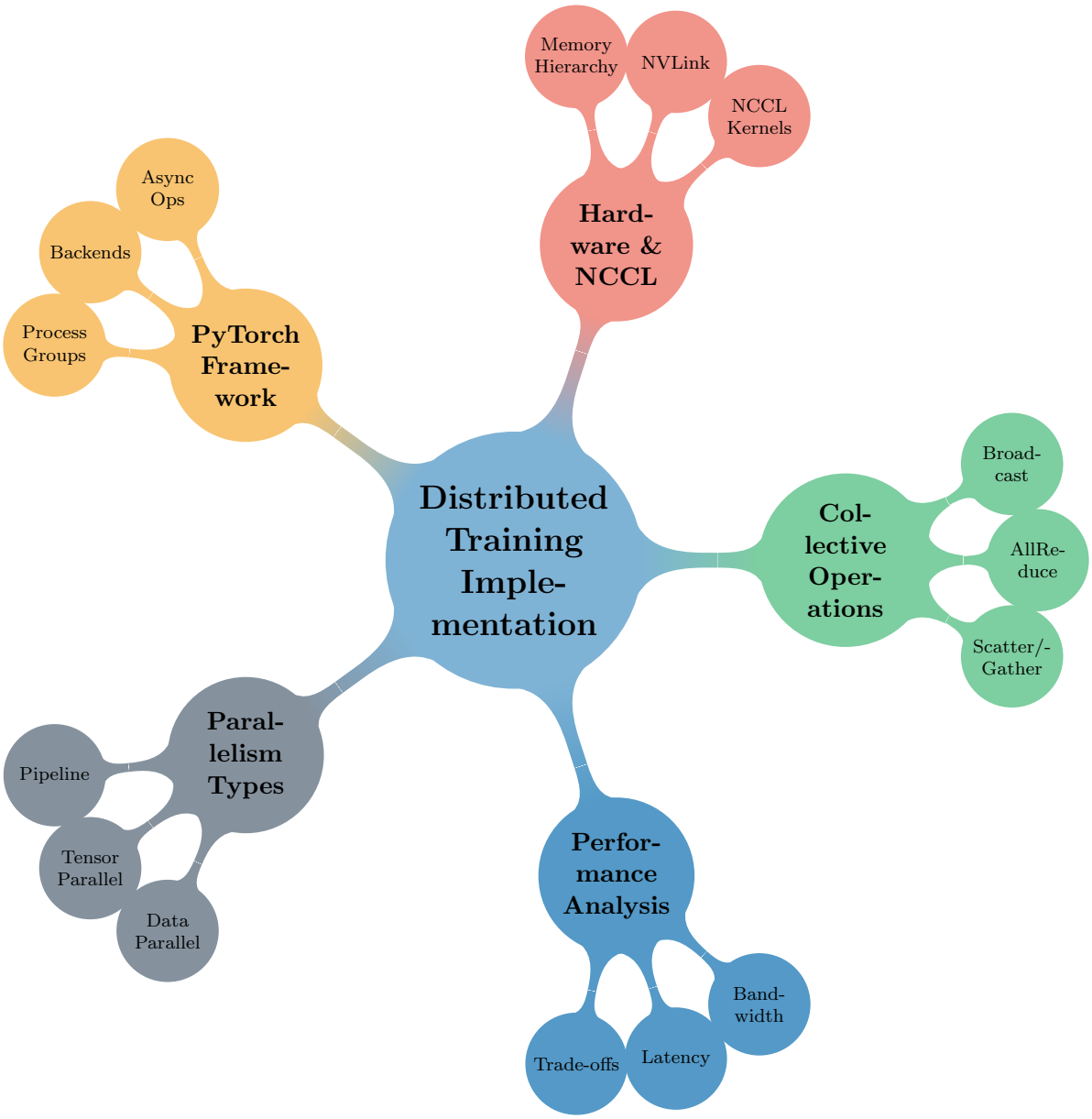

Figure 1: CS336 Lecture 8: Distributed Training Implementation Mind Map - This comprehensive visualization shows the five core areas of distributed training: collective communication primitives that enable multi-device coordination, hardware acceleration through NCCL and modern GPU interconnects, PyTorch's distributed computing framework, the three primary parallelization strategies (data, tensor, and pipeline), and performance analysis considerations for optimizing large-scale training systems.

Mind Map Description:

This comprehensive mind map illustrates the five fundamental pillars of distributed training implementation for large language models:

- **Collective Operations (Green)**: The mathematical primitives that enable coordinated computation across multiple devices, including broadcast for parameter distribution, AllReduce for gradient synchronization, and scatter/gather operations for data distribution.
- **Hardware & NCCL (Red)**: The physical and software infrastructure that makes high-performance distributed training possible, from understanding GPU memory hierarchies to leveraging NVIDIA's optimized communication libraries and NVLink interconnects.
- **PyTorch Framework (Orange)**: The software abstraction layer that translates high-level distributed training concepts into executable code, managing process groups, communication backends, and asynchronous operations.
- **Parallelism Types (Dark Blue)**: The three primary strategies for scaling training across multiple devices - data parallelism for scaling batch sizes, tensor parallelism for scaling model dimensions, and pipeline parallelism for scaling model depth.
- **Performance Analysis (Light Blue)**: The critical evaluation metrics and trade-offs that guide optimization decisions, balancing communication bandwidth, latency constraints, and implementation complexity to achieve optimal training efficiency.

The interconnected nature of these components reflects the reality that successful distributed training requires coordinated optimization across all layers of the system stack, from low-level hardware utilization to high-level algorithmic choices.