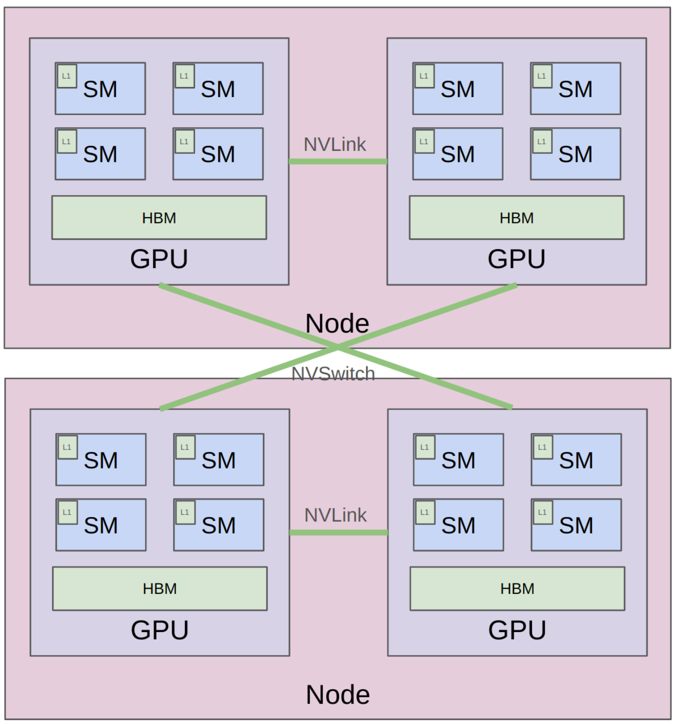
lecture_08.py

```python
import torch
import time
import os
from typing import List, Callable
import torch.nn.functional as F
import torch.distributed as dist
import torch.distributed.fsdp
from execute_util import text, image, link, system_text
from torch_util import get_device
from lecture_util import article_link
from lecture_08_utils import spawn, int_divide, summarize_tensor, get_init_params, render_duration

def main():
```

Last week: parallelism within a single GPU
This week: parallelism across multiple GPUs



In both cases, **compute** (arithmetic logic units) is far from inputs/outputs (**data**).
Unifying theme: orchestrate computation to avoid data transfer bottlenecks

Last week: reduce memory accesses via fusion/tiling
This week: reduce communication across GPUs/nodes via replication/sharding

Generalized hierarchy (from small/fast to big/slow):

- Single node, single GPU: L1 cache / shared memory
- Single node, single GPU: HBM
- Single node, multi-GPU: NVLink
- Multi-node, multi-GPU: NVSwitch

This lecture: concretize the concepts from last lecture in code

[stdout for this lecture]

Part 1: building blocks of distributed communication/computation

```
collective_operations()    # Conceptual programming interface
torch_distributed()        # How this is implemented in NCCL/PyTorch
benchmarking()             # Measure actual NCCL bandwidth
```

Part 2: distributed training

Walk through bare-bones implementations of each strategy on deep MLPs.
Recall that MLPs are the compute bottleneck in Transformers, so this is representative.

```
data_parallelism()         # Cut up along the batch dimension
tensor_parallelism()       # Cut up along the width dimension
pipeline_parallelism()     # Cut up along the depth dimension
```

What's missing?

- More general models (with attention, etc.)
- More communication/computation overlap
- This require more complex code with more bookkeeping
- Jax/TPUs: just define the model, the sharding strategy, and the Jax compiler handles the rest [levanter]
- But we're doing PyTorch so you can see how one builds up from the primitives

Summary

- Many ways to parallelize: data (batch), tensor/expert (width), pipeline (depth), sequence (length)
- Can **re-compute** or store in **memory** or store in another GPUs memory and **communicate**
- Hardware is getting faster, but will always want bigger models, so will have this hierarchical structure

```
def collective_operations():
```

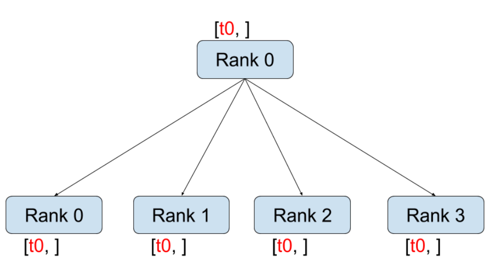
Collective operations are the conceptual primitives used for distributed programming [article]

- Collective means that you specify communication pattern across many (e.g., 256) nodes.
- These are classic in the parallel programming literature from the 1980s.
- Better/faster abstraction than managing point-to-point communication yourself.

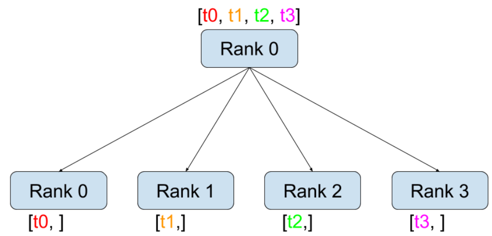
Terminology:

- **World size**: number of devices (e.g., 4)
- **Rank**: a device (e.g., 0, 1, 2, 3)

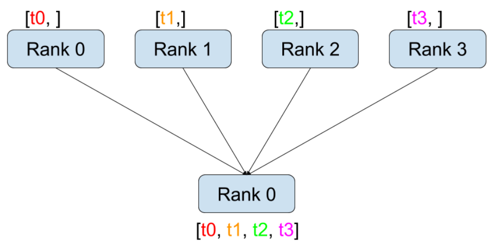
Broadcast



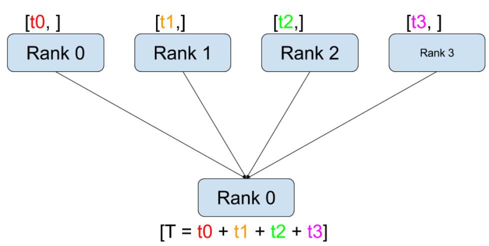
Scatter



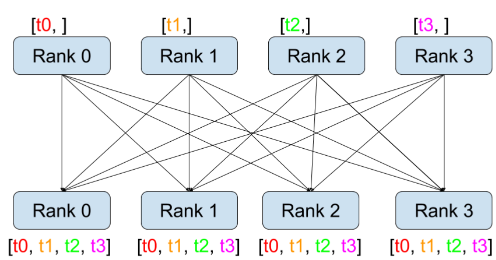
Gather



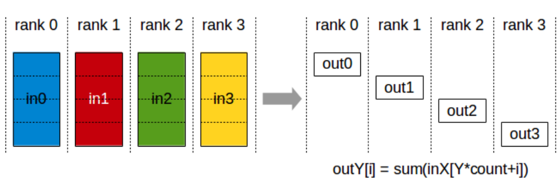
Reduce



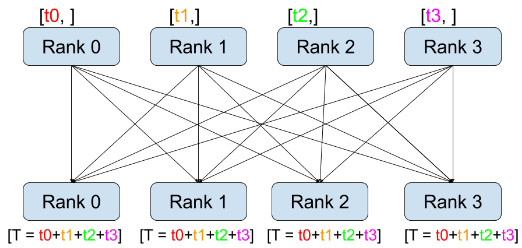
All-gather



Reduce-scatter



All-reduce = reduce-scatter + all-gather



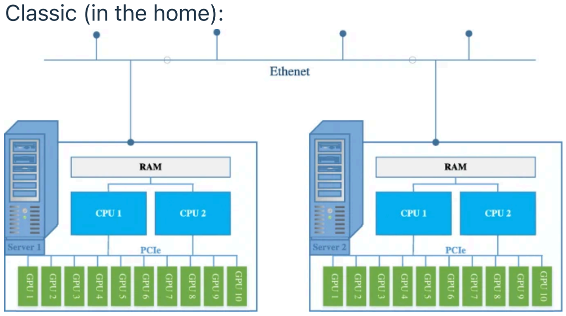
Way to remember the terminology:

- Reduce: performs some associative/commutative operation (sum, min, max)
- Broadcast/scatter is inverse of gather
- All: means destination is all devices

```
def torch_distributed():
```

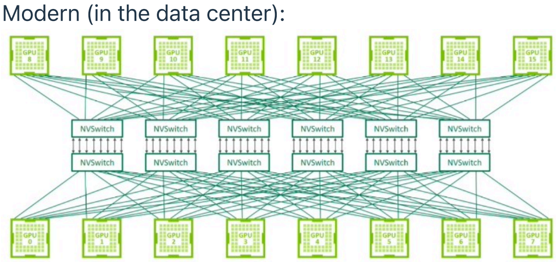
Hardware

Classic (in the home):



- GPUs on same node communicate via a PCI(e) bus (v7.0, 16 lanes => 242 GB/s) [article]
- GPUs on different nodes communicate via Ethernet (~200 MB/s)

Modern (in the data center):



- Within a node: NVLink connects GPUs directly, bypass CPU
- Across nodes: NVSwitch connects GPUs directly, bypass Ethernet

Each H100 has 18 NVLink 4.0 links, for a total of 900GB/s [article]
In comparison, memory bandwidth for HBM is 3.9 TB/s [article]

Let's check what our hardware setup is. [article]

```
if torch.cuda.is_available():
    os.system("nvidia-smi topo -m")
```

Note GPUs are connected via NV18, also connected to NICs (for PCIe)

NVIDIA Collective Communication Library (NCCL)

NCCL translates collective operations into low-level packets that are sent between GPUs. [talk]

- Detects topology of hardware (e.g., number of nodes, switches, NVLink/PCIe)

- Optimizes the path between GPUs
- Launches CUDA kernels to send/receive data

PyTorch distributed library (`torch.distributed`)

[Documentation]

- Provides clean interface for collective operations (e.g., `all_gather_into_tensor`)
- Supports multiple backends for different hardware: gloo (CPU), nccl (GPU)
- Also supports higher-level algorithms (e.g., `FullyShardedDataParallel`) [not used in this course]

Let's walk through some examples.

```
    spawn(collective_operations_main, world_size=4)


def collective_operations_main(rank: int, world_size: int):
    """This function is running asynchronously for each process (rank = 0, ..., world_size - 1)."""
    setup(rank, world_size)

    # All-reduce
    dist.barrier()  # Waits for all processes to get to this point (in this case, for print statements)

    tensor = torch.tensor([0., 1, 2, 3], device=get_device(rank)) + rank  # Both input and output

    print(f"Rank {rank} [before all-reduce]: {tensor}", flush=True)
    dist.all_reduce(tensor=tensor, op=dist.ReduceOp.SUM, async_op=False)  # Modifies tensor in place
    print(f"Rank {rank} [after all-reduce]: {tensor}", flush=True)

    # Reduce-scatter
    dist.barrier()

    input = torch.arange(world_size, dtype=torch.float32, device=get_device(rank)) + rank  # Input
    output = torch.empty(1, device=get_device(rank))  # Allocate output

    print(f"Rank {rank} [before reduce-scatter]: input = {input}, output = {output}", flush=True)
    dist.reduce_scatter_tensor(output=output, input=input, op=dist.ReduceOp.SUM, async_op=False)
    print(f"Rank {rank} [after reduce-scatter]: input = {input}, output = {output}", flush=True)

    # All-gather
    dist.barrier()

    input = output  # Input is the output of reduce-scatter
    output = torch.empty(world_size, device=get_device(rank))  # Allocate output

    print(f"Rank {rank} [before all-gather]: input = {input}, output = {output}", flush=True)
    dist.all_gather_into_tensor(output_tensor=output, input_tensor=input, async_op=False)
    print(f"Rank {rank} [after all-gather]: input = {input}, output = {output}", flush=True)
```

Indeed, all-reduce = reduce-scatter + all-gather!

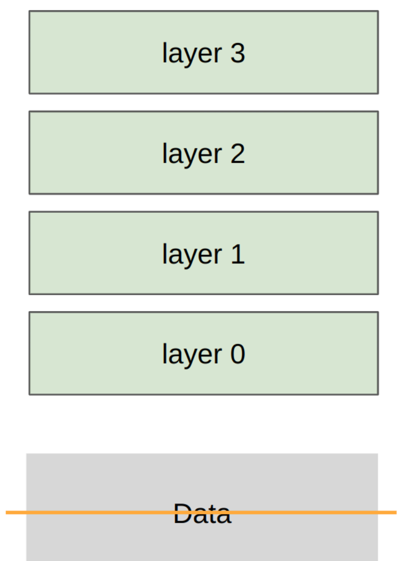
```
    cleanup()


def benchmarking():
```

Let's see how fast communication happens (restrict to one node).

```
    # All-reduce
```

```
    spawn(all_reduce, world_size=4, num_elements=100 * 1024**2)

    # Reduce-scatter
    spawn(reduce_scatter, world_size=4, num_elements=100 * 1024**2)

    # References
    How to reason about operations
    Sample code


def all_reduce(rank: int, world_size: int, num_elements: int):
    setup(rank, world_size)

    # Create tensor
    tensor = torch.randn(num_elements, device=get_device(rank))

    # Warmup
    dist.all_reduce(tensor=tensor, op=dist.ReduceOp.SUM, async_op=False)
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA kernels to finish
        dist.barrier()            # Wait for all the processes to get here

    # Perform all-reduce
    start_time = time.time()
    dist.all_reduce(tensor=tensor, op=dist.ReduceOp.SUM, async_op=False)
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA kernels to finish
        dist.barrier()            # Wait for all the processes to get here
    end_time = time.time()

    duration = end_time - start_time
    print(f"[all_reduce] Rank {rank}: all_reduce(world_size={world_size}, num_elements={num_elements}) took
{render_duration(duration)}", flush=True)

    # Measure the effective bandwidth
    dist.barrier()
    size_bytes = tensor.element_size() * tensor.numel()
    sent_bytes = size_bytes * 2 * (world_size - 1)  # 2x because send input and receive output
    total_duration = world_size * duration
    bandwidth = sent_bytes / total_duration
    print(f"[all_reduce] Rank {rank}: all_reduce measured bandwidth = {round(bandwidth / 1024**3)} GB/s", flush=True)

    cleanup()


def reduce_scatter(rank: int, world_size: int, num_elements: int):
    setup(rank, world_size)

    # Create input and outputs
    input = torch.randn(world_size, num_elements, device=get_device(rank))  # Each rank has a matrix
    output = torch.empty(num_elements, device=get_device(rank))

    # Warmup
    dist.reduce_scatter_tensor(output=output, input=input, op=dist.ReduceOp.SUM, async_op=False)
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA kerels to finish
        dist.barrier()            # Wait for all the processes to get here

```

```
    # Perform reduce-scatter
    start_time = time.time()
    dist.reduce_scatter_tensor(output=output, input=input, op=dist.ReduceOp.SUM, async_op=False)
    if torch.cuda.is_available():
        torch.cuda.synchronize()  # Wait for CUDA kerels to finish
        dist.barrier()            # Wait for all the processes to get here
    end_time = time.time()

    duration = end_time - start_time
    print(f"[reduce_scatter] Rank {rank}: reduce_scatter(world_size={world_size}, num_elements={num_elements}) took {render_duration(duration)}", flush=True)

    # Measure the effective bandwidth
    dist.barrier()
    data_bytes = output.element_size() * output.numel()  # How much data in the output
    sent_bytes = data_bytes * (world_size - 1)  # How much needs to be sent (no 2x here)
    total_duration = world_size * duration  # Total time for transmission
    bandwidth = sent_bytes / total_duration
    print(f"[reduce_scatter] Rank {rank}: reduce_scatter measured bandwidth = {round(bandwidth / 1024**3)} GB/s", flush=True)

    cleanup()


def data_parallelism():
```



Sharding strategy: each rank gets a slice of the data

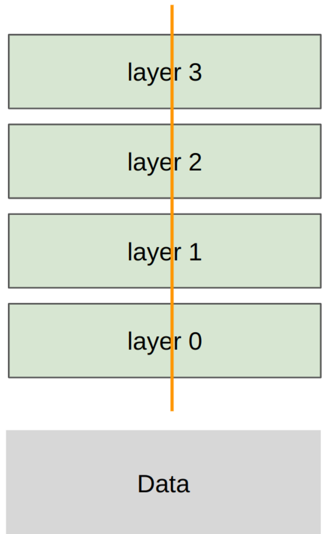
```
    data = generate_sample_data()
    spawn(data_parallelism_main, world_size=4, data=data, num_layers=4, num_steps=1)
```

Notes:

- Losses are different across ranks (computed on local data)
- Gradients are all-reduced to be the same across ranks
- Therefore, parameters remain the same across ranks

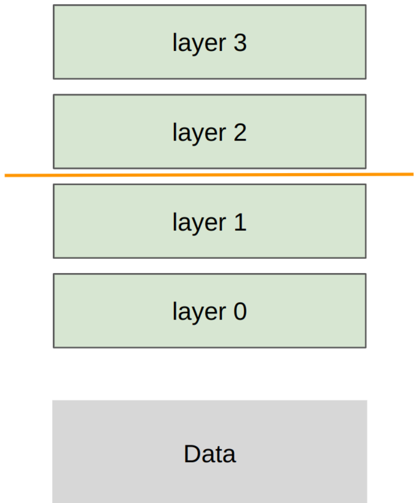
```
def generate_sample_data():
    batch_size = 128
    num_dim = 1024
    data = torch.randn(batch_size, num_dim)
```

```
    return data


def data_parallelism_main(rank: int, world_size: int, data: torch.Tensor, num_layers: int, num_steps: int):
    setup(rank, world_size)

    # Get the slice of data for this rank (in practice, each rank should load only its own data)
    batch_size = data.size(0)  # @inspect batch_size
    num_dim = data.size(1)  # @inspect num_dim
    local_batch_size = int_divide(batch_size, world_size)  # @inspect local_batch_size
    start_index = rank * local_batch_size  # @inspect start_index
    end_index = start_index + local_batch_size  # @inspect end_index
    data = data[start_index:end_index].to(get_device(rank))

    # Create MLP parameters params[0], ..., params[num_layers - 1] (each rank has all parameters)
    params = [get_init_params(num_dim, num_dim, rank) for i in range(num_layers)]
    optimizer = torch.optim.AdamW(params, lr=1e-3)  # Each rank has own optimizer state

    for step in range(num_steps):
        # Forward pass
        x = data
        for param in params:
            x = x @ param
            x = F.gelu(x)
        loss = x.square().mean()  # Loss function is average squared magnitude

        # Backward pass
        loss.backward()

        # Sync gradients across workers (only difference between standard training and DDP)
        for param in params:
            dist.all_reduce(tensor=param.grad, op=dist.ReduceOp.AVG, async_op=False)

        # Update parameters
        optimizer.step()

        print(f"[data_parallelism] Rank {rank}: step = {step}, loss = {loss.item()}, params = {[summarize_tensor(params[i]) for i in range(num_layers)]}", flush=True)

    cleanup()


def tensor_parallelism():
```



Sharding strategy: each rank gets part of each layer, transfer all data/activations

```
    data = generate_sample_data()
    spawn(tensor_parallelism_main, world_size=4, data=data, num_layers=4)


def tensor_parallelism_main(rank: int, world_size: int, data: torch.Tensor, num_layers: int):
    setup(rank, world_size)

    data = data.to(get_device(rank))
    batch_size = data.size(0)  # @inspect batch_size
    num_dim = data.size(1)  # @inspect num_dim
    local_num_dim = int_divide(num_dim, world_size)  # Shard `num_dim`  @inspect local_num_dim

    # Create model (each rank gets 1/world_size of the parameters)
    params = [get_init_params(num_dim, local_num_dim, rank) for i in range(num_layers)]

    # Forward pass
    x = data
    for i in range(num_layers):
        # Compute activations (batch_size x local_num_dim)
        x = x @ params[i]  # Note: this is only on a slice of the parameters
        x = F.gelu(x)

        # Allocate memory for activations (world_size x batch_size x local_num_dim)
        activations = [torch.empty(batch_size, local_num_dim, device=get_device(rank)) for _ in range(world_size)]

        # Send activations via all gather
        dist.all_gather(tensor_list=activations, tensor=x, async_op=False)

        # Concatenate them to get batch_size x num_dim
        x = torch.cat(activations, dim=1)

    print(f"[tensor_parallelism] Rank {rank}: forward pass produced activations {summarize_tensor(x)}", flush=True)

    # Backward pass: homework exercise

    cleanup()


def pipeline_parallelism():
```



Sharding strategy: each rank gets subset of layers, transfer all data/activations

```python
    data = generate_sample_data()
    spawn(pipeline_parallelism_main, world_size=2, data=data, num_layers=4, num_micro_batches=4)


def pipeline_parallelism_main(rank: int, world_size: int, data: torch.Tensor, num_layers: int, num_micro_batches: int):
    setup(rank, world_size)

    # Use all the data
    data = data.to(get_device(rank))
    batch_size = data.size(0)  # @inspect batch_size
    num_dim = data.size(1)  # @inspect num_dim

    # Split up layers
    local_num_layers = int_divide(num_layers, world_size)  # @inspect local_num_layers

    # Each rank gets a subset of layers
    local_params = [get_init_params(num_dim, num_dim, rank) for i in range(local_num_layers)]

    # Forward pass

    # Break up into micro batches to minimize the bubble
    micro_batch_size = int_divide(batch_size, num_micro_batches)  # @inspect micro_batch_size
    if rank == 0:
        # The data
        micro_batches = data.chunk(chunks=num_micro_batches, dim=0)
    else:
        # Allocate memory for activations
        micro_batches = [torch.empty(micro_batch_size, num_dim, device=get_device(rank)) for _ in range(num_micro_batches)]

    for x in micro_batches:
        # Get activations from previous rank
        if rank - 1 >= 0:
            dist.recv(tensor=x, src=rank - 1)

        # Compute layers assigned to this rank
        for param in local_params:
            x = x @ param
            x = F.gelu(x)

```

```
        # Send to the next rank
        if rank + 1 < world_size:
            print(f"[pipeline_parallelism] Rank {rank}: sending {summarize_tensor(x)} to rank {rank + 1}", flush=True)
            dist.send(tensor=x, dst=rank + 1)
```

Not handled: overlapping communication/computation to eliminate pipeline bubbles

```
    # Backward pass: homework exercise

    cleanup()

############################################################

def setup(rank: int, world_size: int):
    # Specify where master lives (rank 0), used to coordinate (actual data goes through NCCL)
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "15623"

    if torch.cuda.is_available():
        dist.init_process_group("nccl", rank=rank, world_size=world_size)
    else:
        dist.init_process_group("gloo", rank=rank, world_size=world_size)


def cleanup():
    torch.distributed.destroy_process_group()


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