11868 LLM Systems
Distributed GPU Training

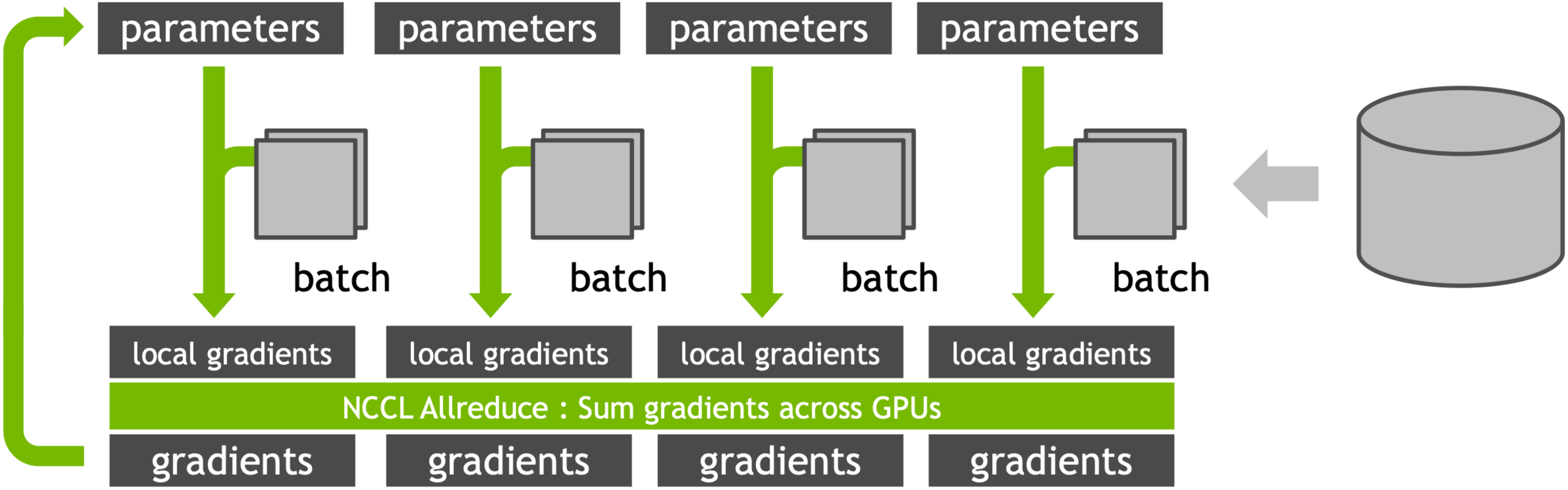
Lei Li

Carnegie Mellon University
Language Technologies Institute

Today's Topic

- Multi-GPU communication
- Distributed Data Parallel Training

Distributed Training with Multiple GPUs



need to communicate gradients across GPUs!

Multi-GPU Communication

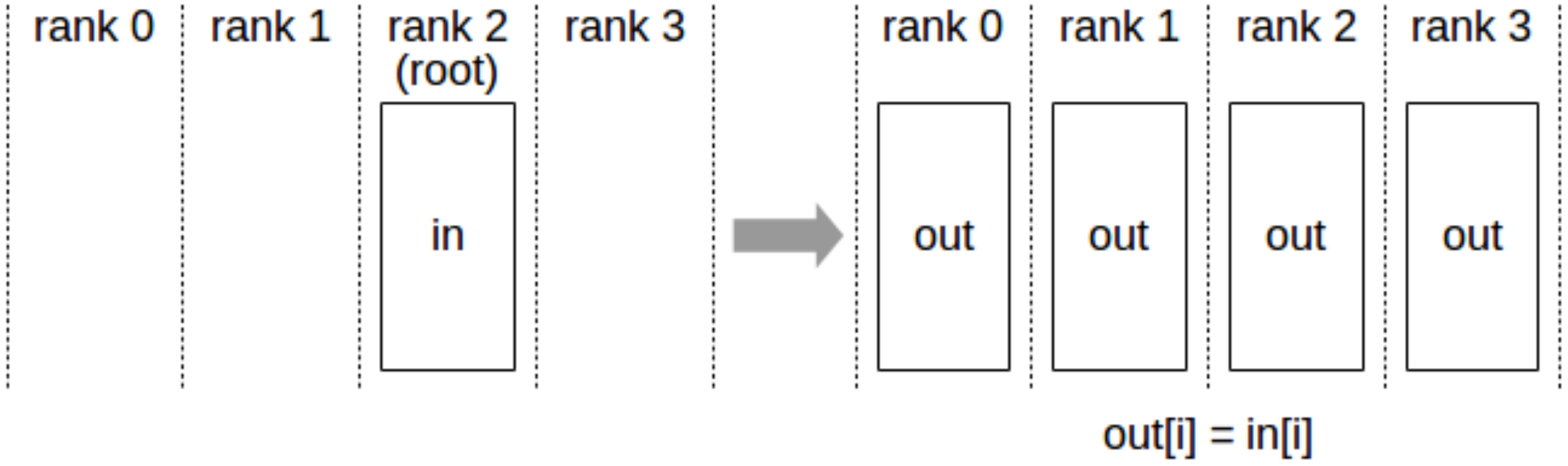
- NCCL (Nvidia Collective Communication Library)
  - provides inter-GPU communication APIs
  - both collective and point-to-point send/receive primitives
  - supports various of interconnect technologies
    - PCIe
    - NVLink
    - InfiniBand
    - IP sockets
  - Operations are tied to a CUDA stream.

NCCL Primitives

- Broadcast
- Reduce
- ReduceScatter
- AllGather
- AllReduce

Broadcast

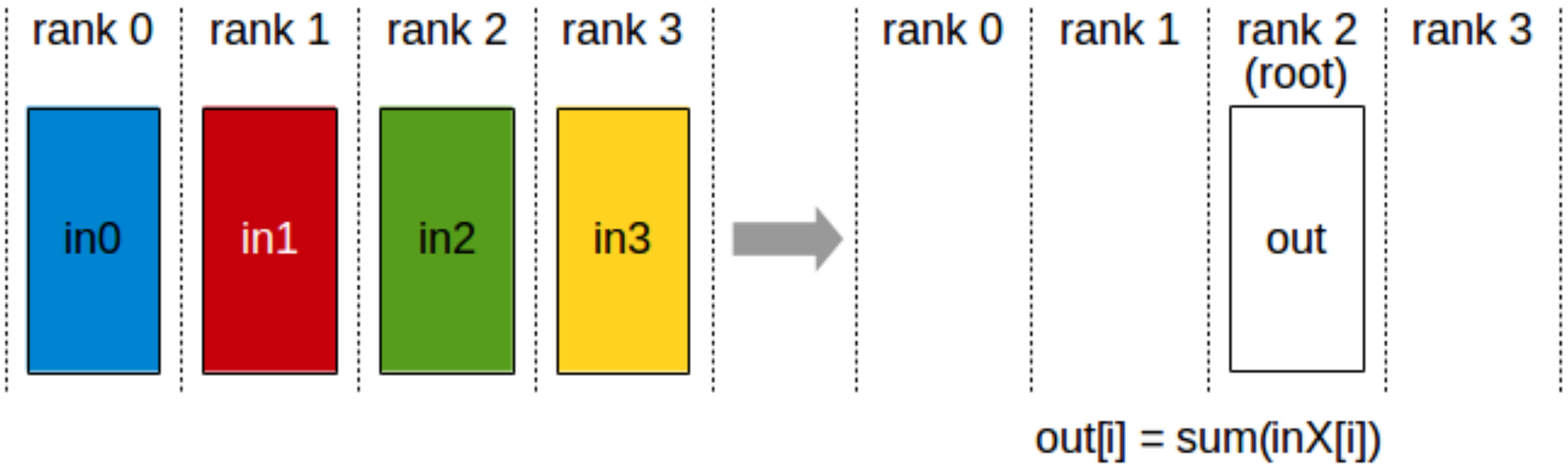
- The Broadcast operation copies an N-element buffer on the root rank to all ranks (devices).




```
ncclResult_t ncclBroadcast(const void* sendbuff, void* recvbuff,
size_t count, ncclDataType_t datatype,
int root, ncclComm_t comm, cudaStream_t stream)
```

Reduce

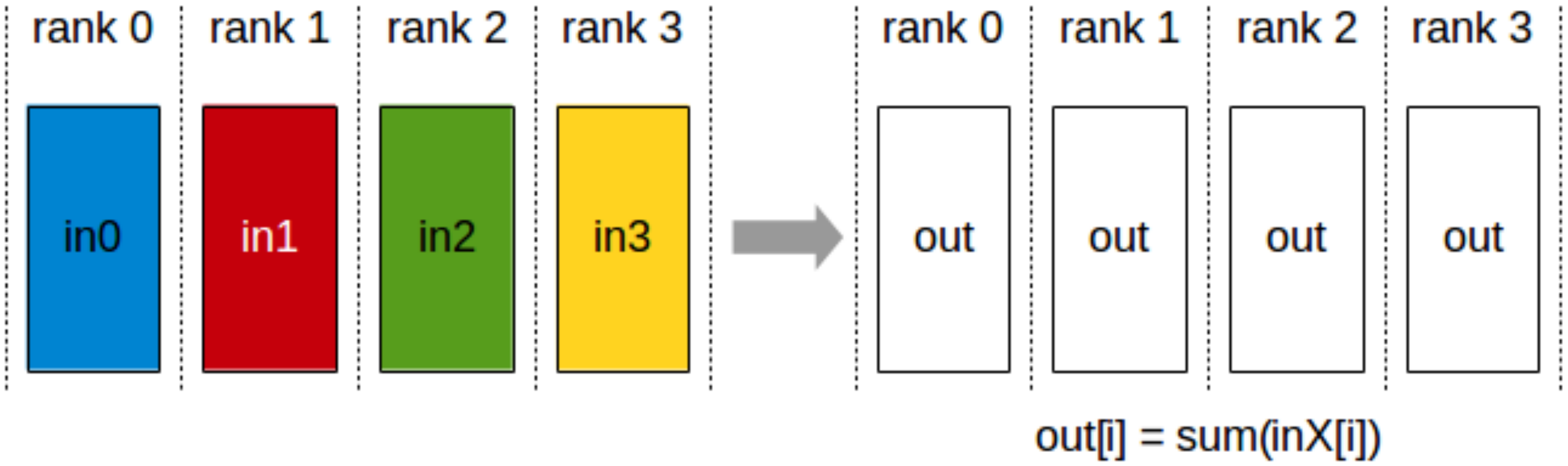
- Compute reduction (max, min, sum) across devices and write on one rank




```
ncclResult_t ncclReduce(const void* sendbuff, void* recvbuff,
size_t count, ncclDataType_t datatype, ncclRedOp_t op,
int root, ncclComm_t comm, cudaStream_t stream)
```

AllReduce (=Reduce & Broadcast)

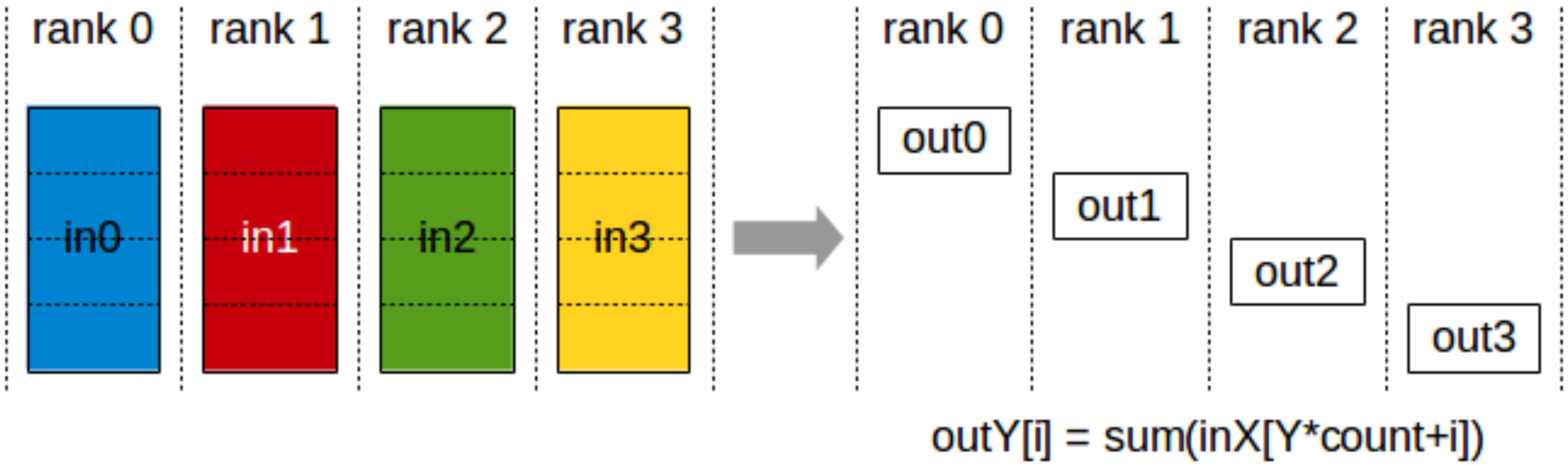
- Compute reduction (sum, min, max) across devices and writing the result in the receive buffers of every rank.




```
ncclResult_t ncclAllReduce(const void* sendbuff,
void* recvbuff, size_t count, ncclDataType_t datatype,
ncclRedOp_t op, ncclComm_t comm, cudaStream_t stream)
```

ReduceScatter

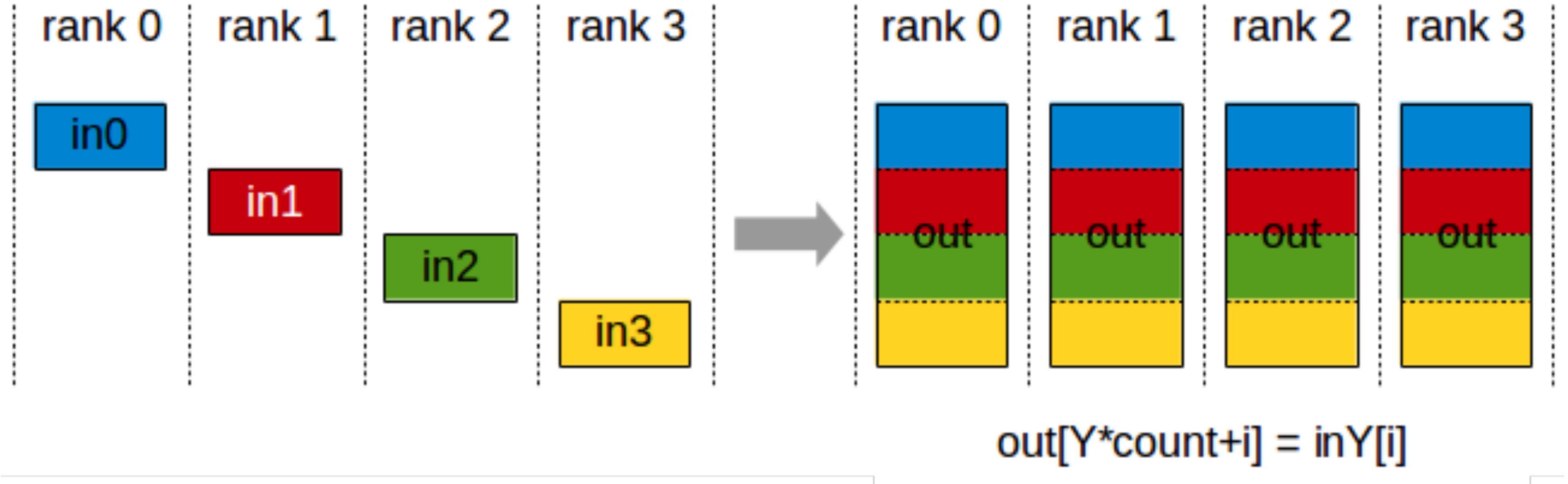
- Compute reduction (sum, min, max) and writing parts of results scattered in ranks




```
ncclResult_t ncclReduceScatter(const void* sendbuff,
void* recvbuff, size_t recvcount, ncclDataType_t datatype,
ncclRedOp_t op, ncclComm_t comm, cudaStream_t stream)
```

AllGather

- gathers N values from k ranks into an output of size k*N, and distributes that result to all ranks (devices).




```
ncclResult_t ncclAllGather(const void* sendbuff,
void* recvbuff, size_t sendcount, ncclDataType_t datatype,
ncclComm_t comm, cudaStream_t stream)
```

AllReduce = ReduceScatter & AllGather

Data Pointers in CUDA

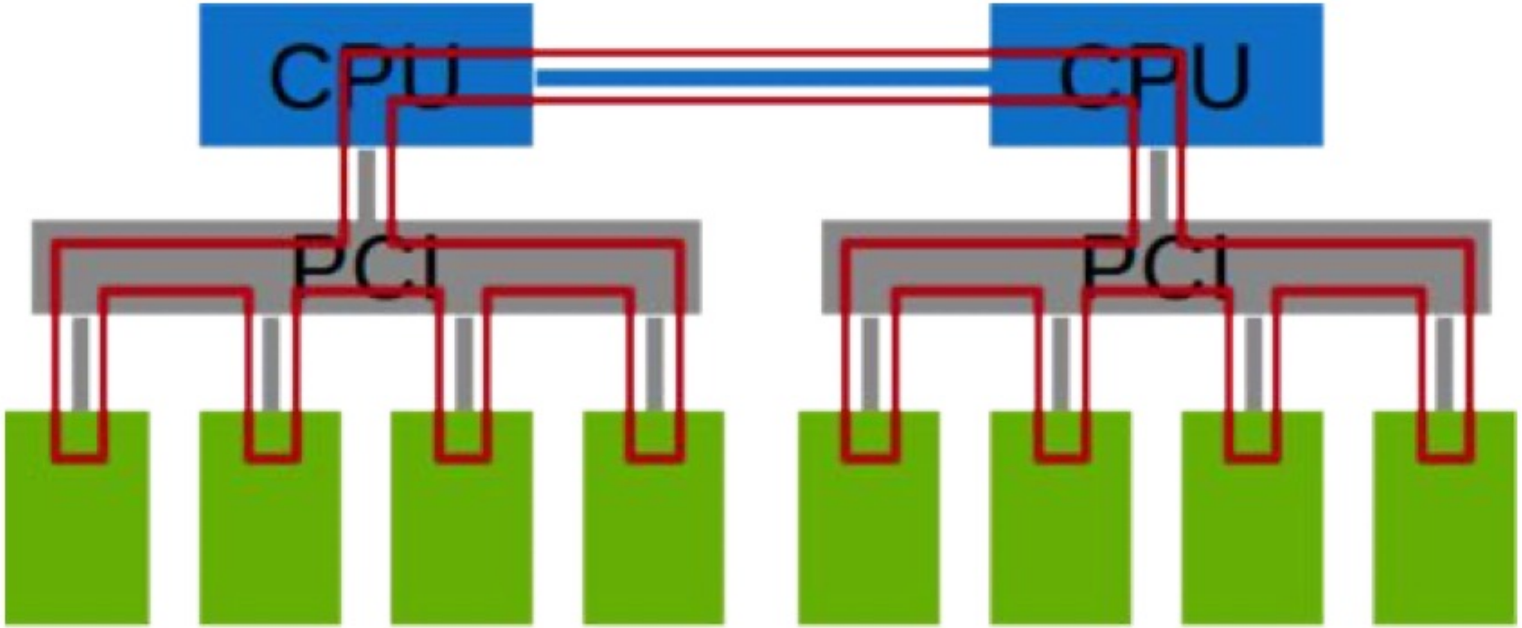
- device memory local to the CUDA device
- host memory registered using cudaHostRegister or cudaGetDevicePointer
- managed and unified memory.

Point-to-Point Communication

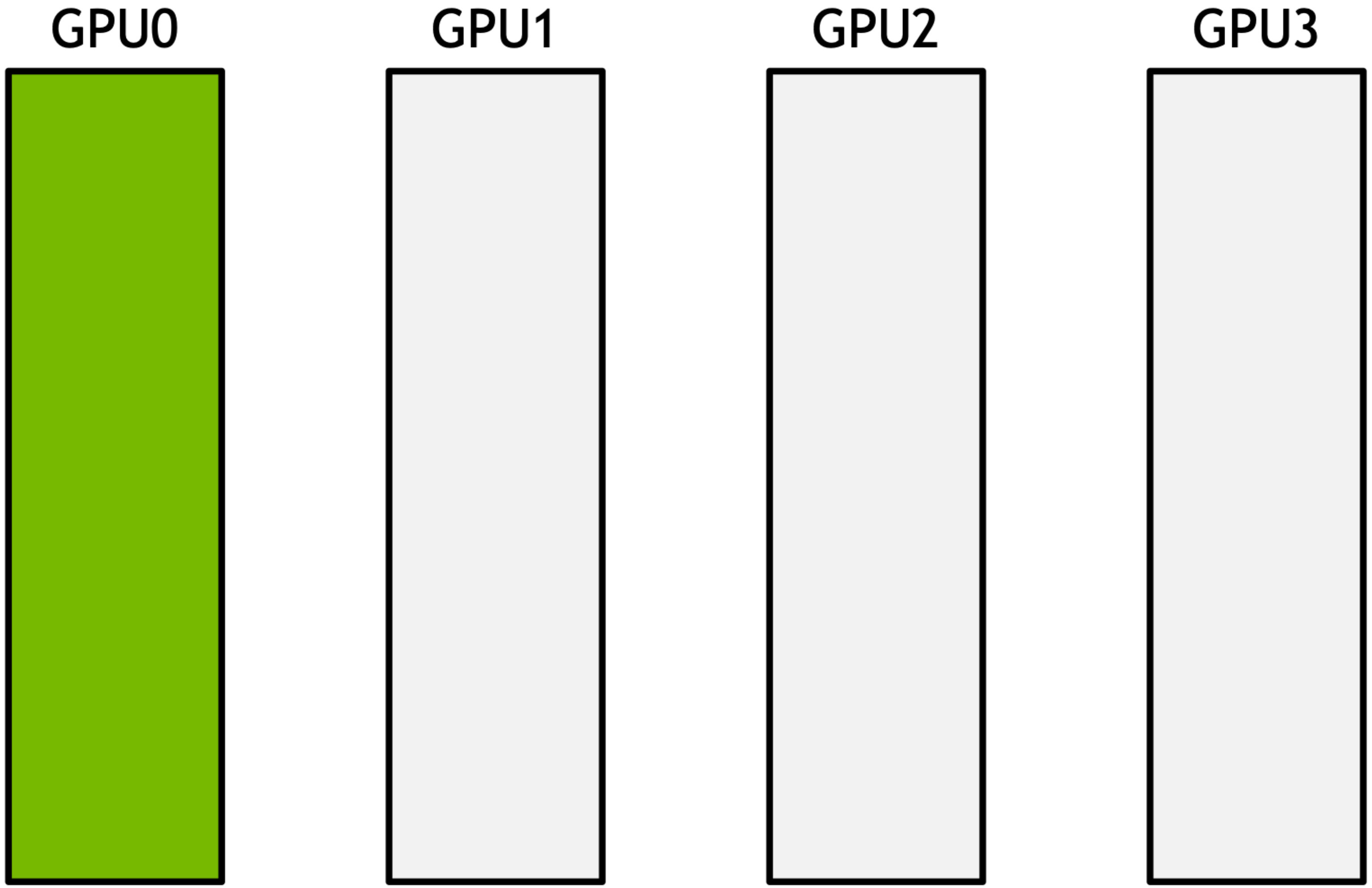
```
ncclGroupStart();

ncclSend(sendbuff, sendcount, sendtype, peer, comm, stream);

ncclRecv(recvbuff, recvcount, recvtype, peer, comm, stream);

ncclGroupEnd();
```

How Reduce is Implemented?

- NCCL uses rings to move data across all GPUs and perform reductions.

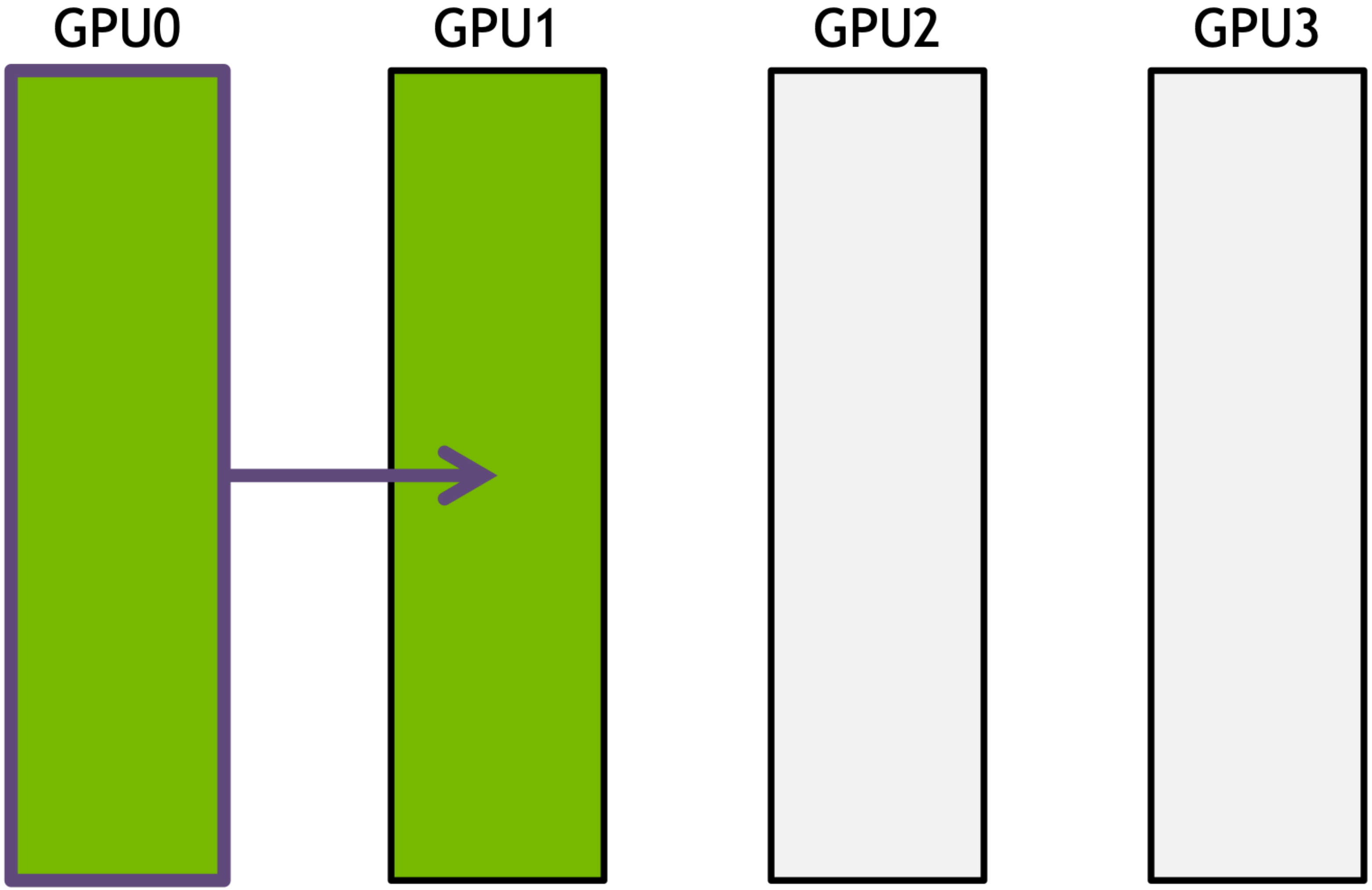


Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

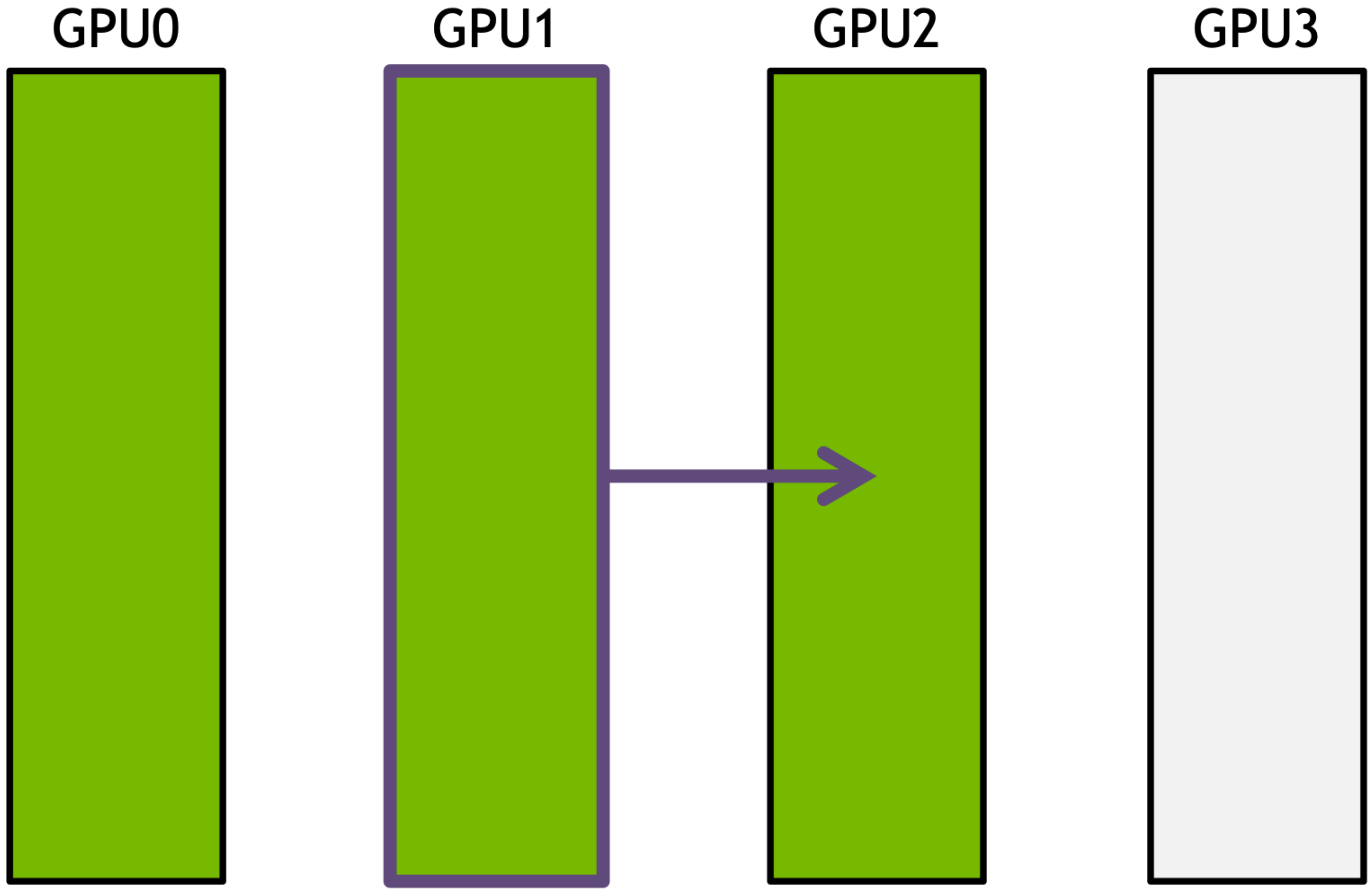
Broadcast with unidirectional ring



Step 1: $t = N/B$

N=bytes to transfer
B=bandwidth

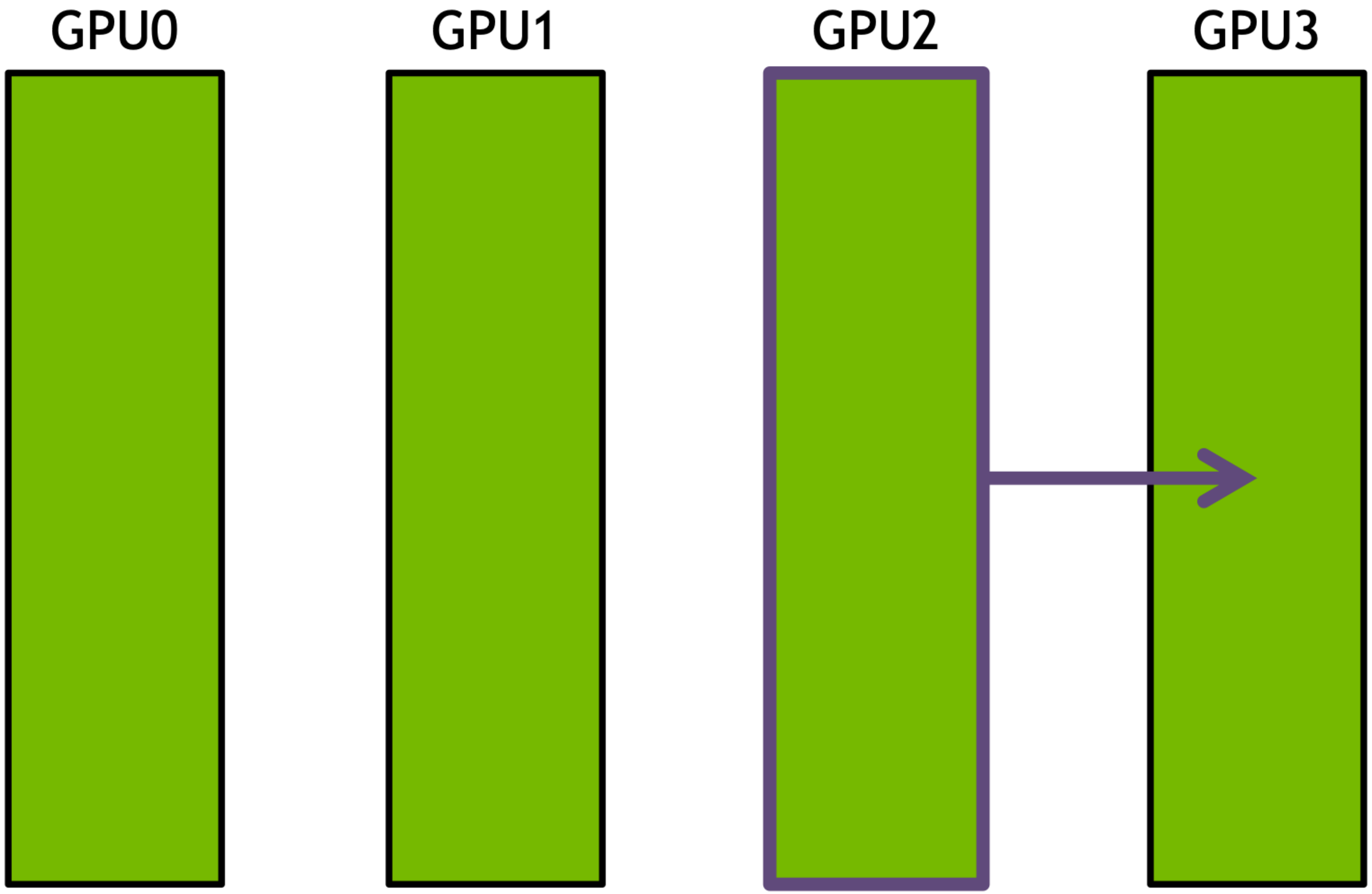
Broadcast with unidirectional ring



Step 1: t = N/B
Step 2: t = N/B

N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring



Step 1: $t = N/B$

Step 2: $t = N/B$

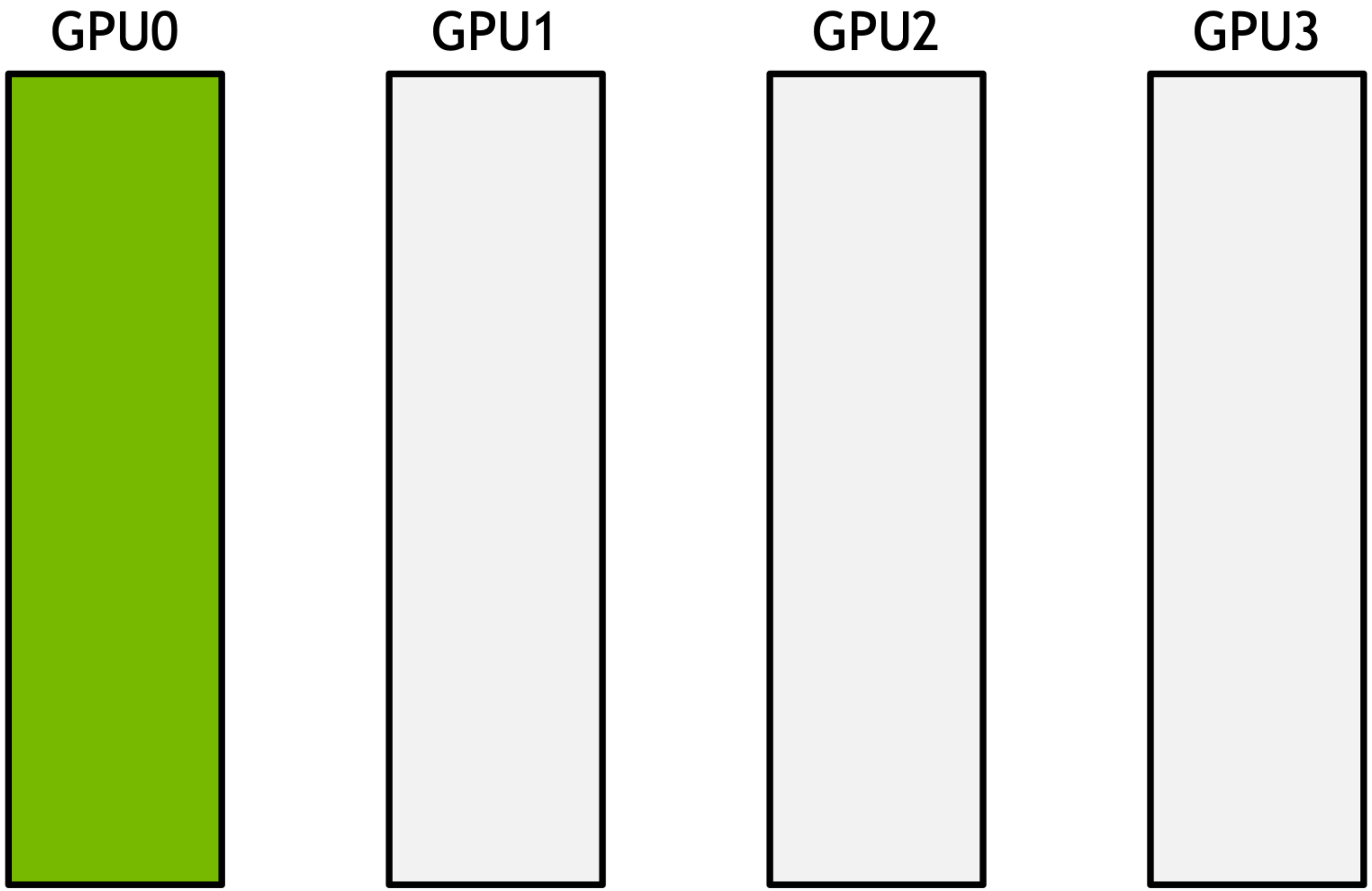
Step 3: $t = N/B$

total time=$(K-1)\ N/B$

N=bytes to transfer

B=bandwidth

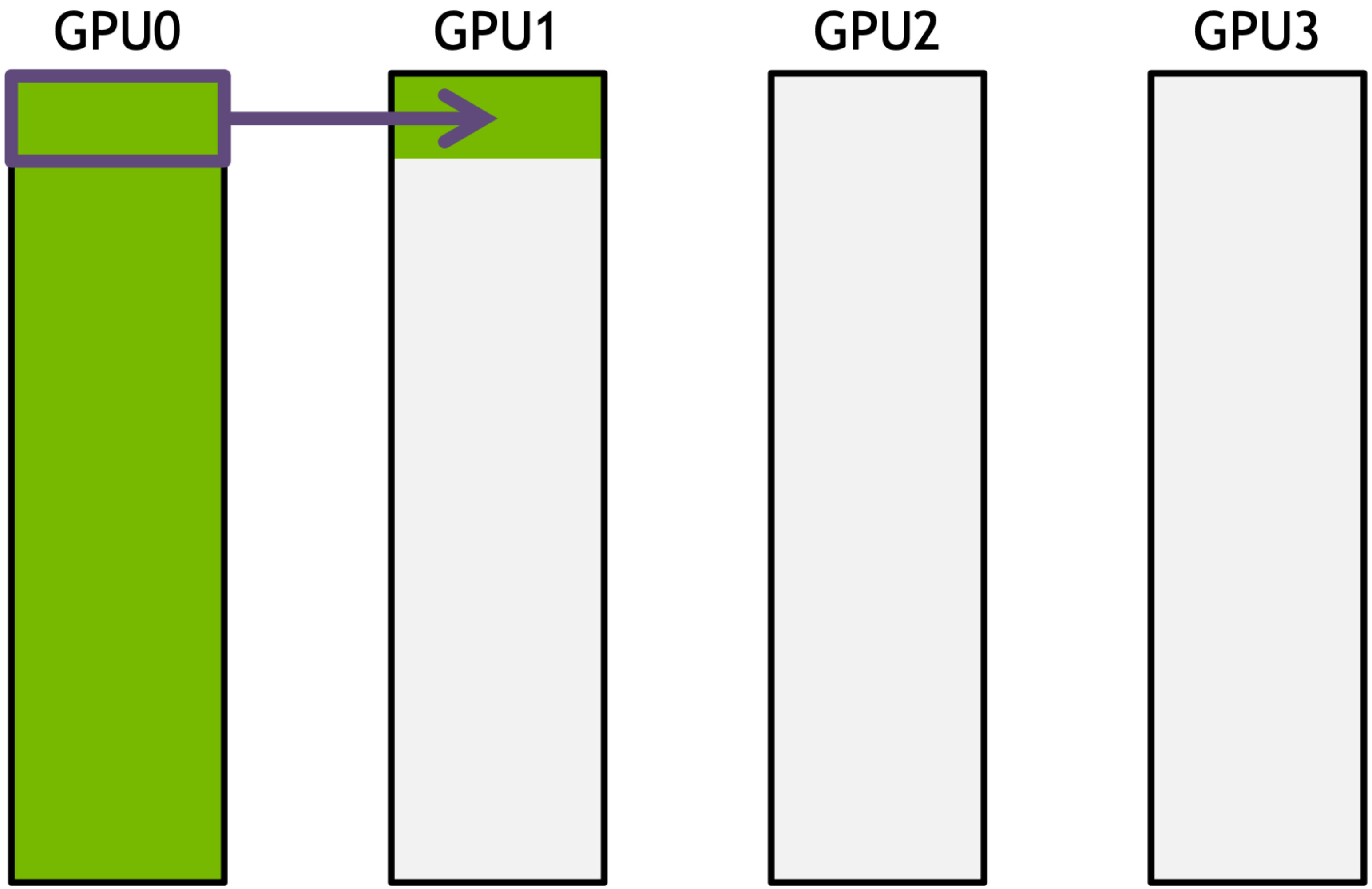
Broadcast with unidirectional ring



N=bytes to transfer
B=bandwidth

Broadcast with unidirectional ring

break data into S messages



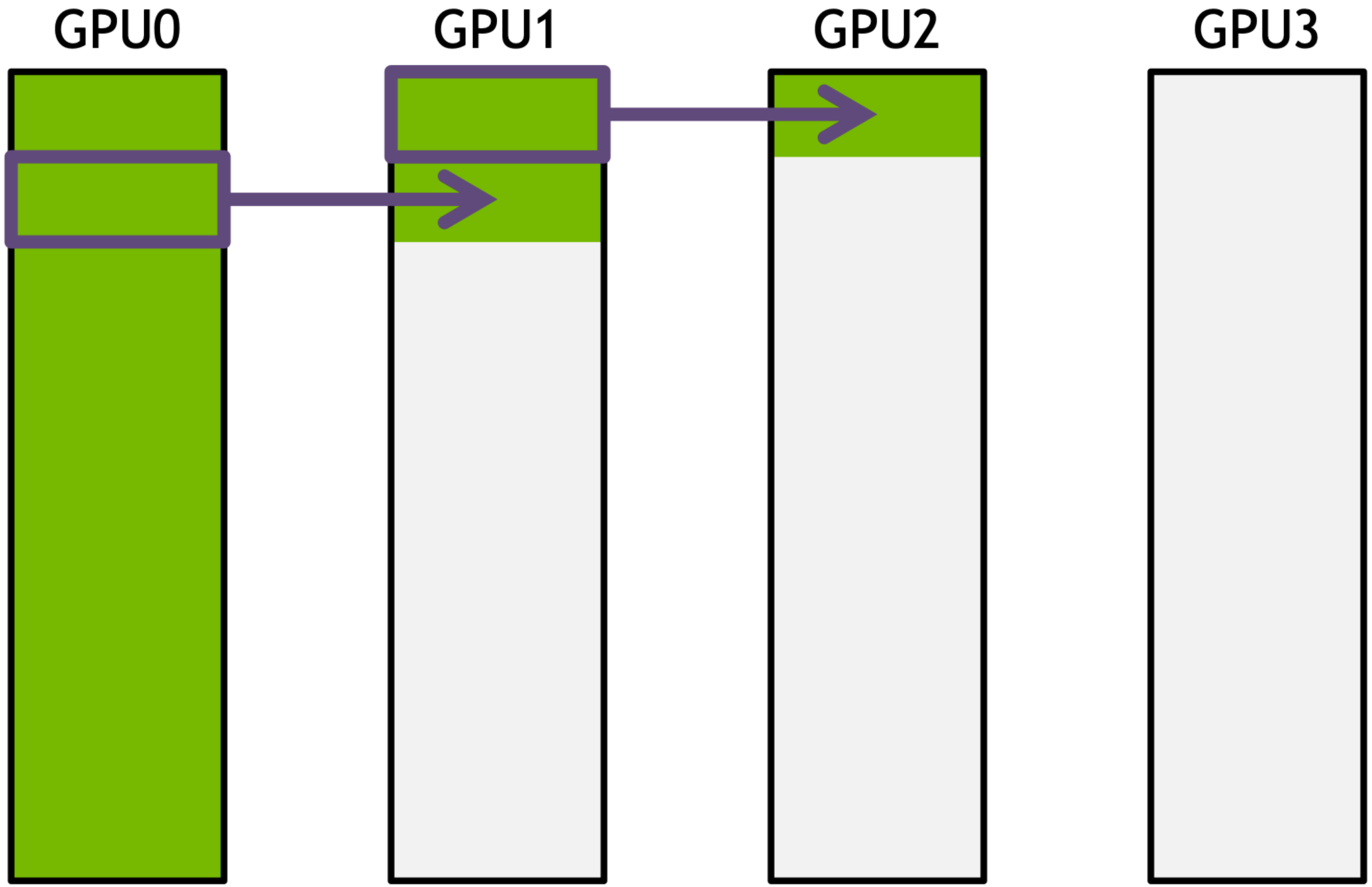
Step 1: $t = N/SB$

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: t = N/SB

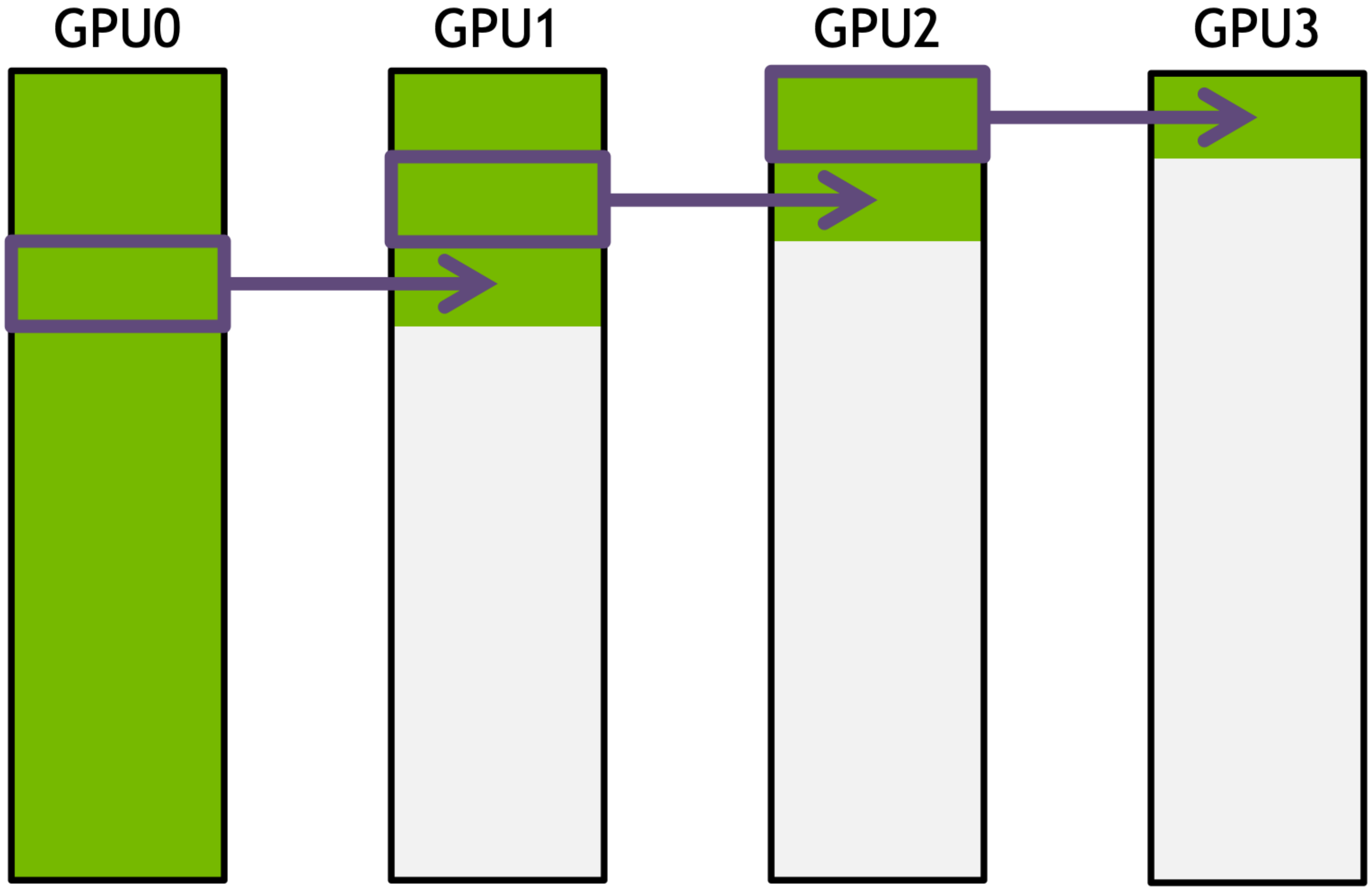
Step 2: t = N/SB

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: $t = N/SB$

Step 2: $t = N/SB$

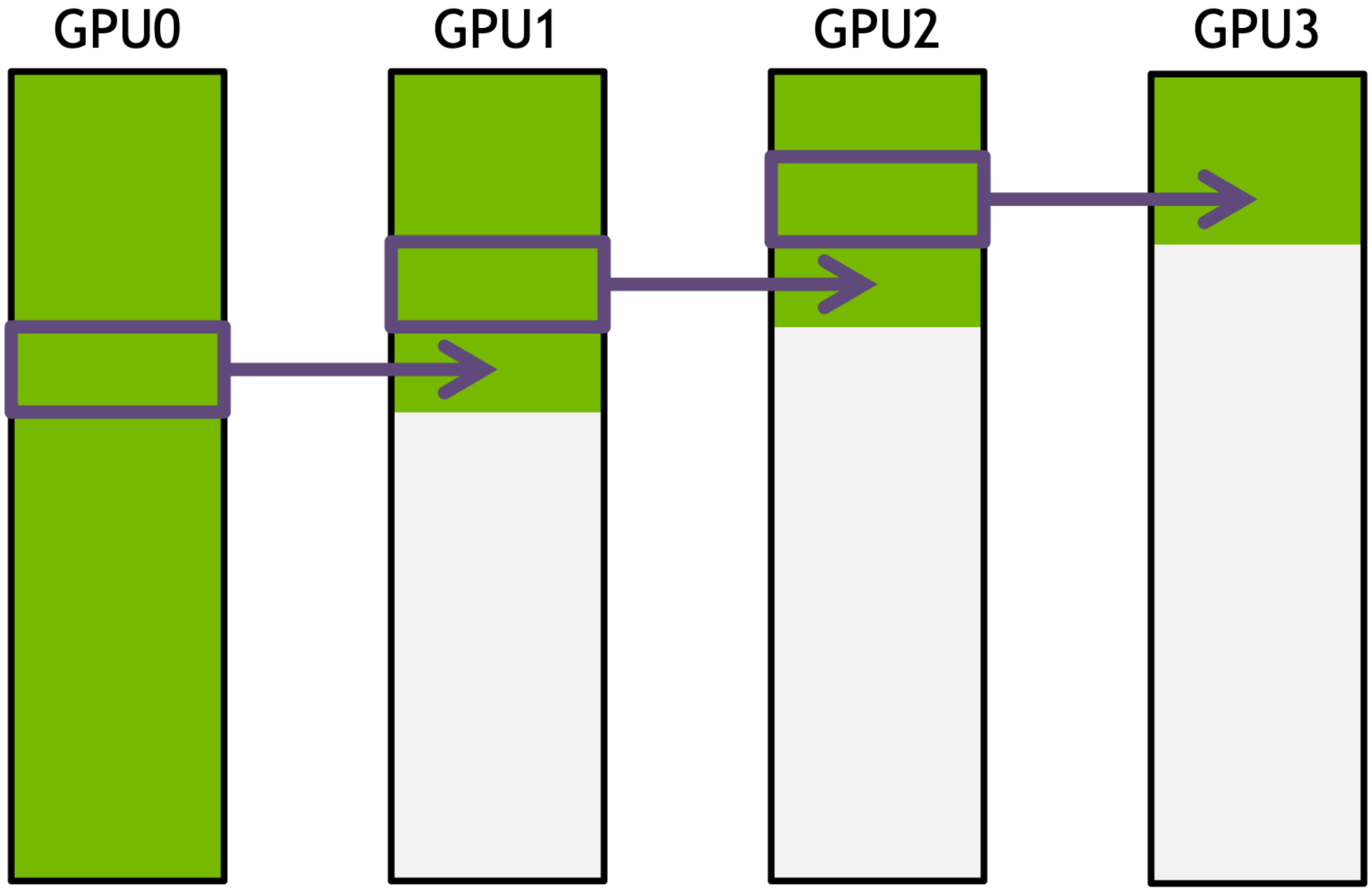
Step 3: $t = N/SB$

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: t = N/SB

Step 2: t = N/SB

Step 3: t = N/SB

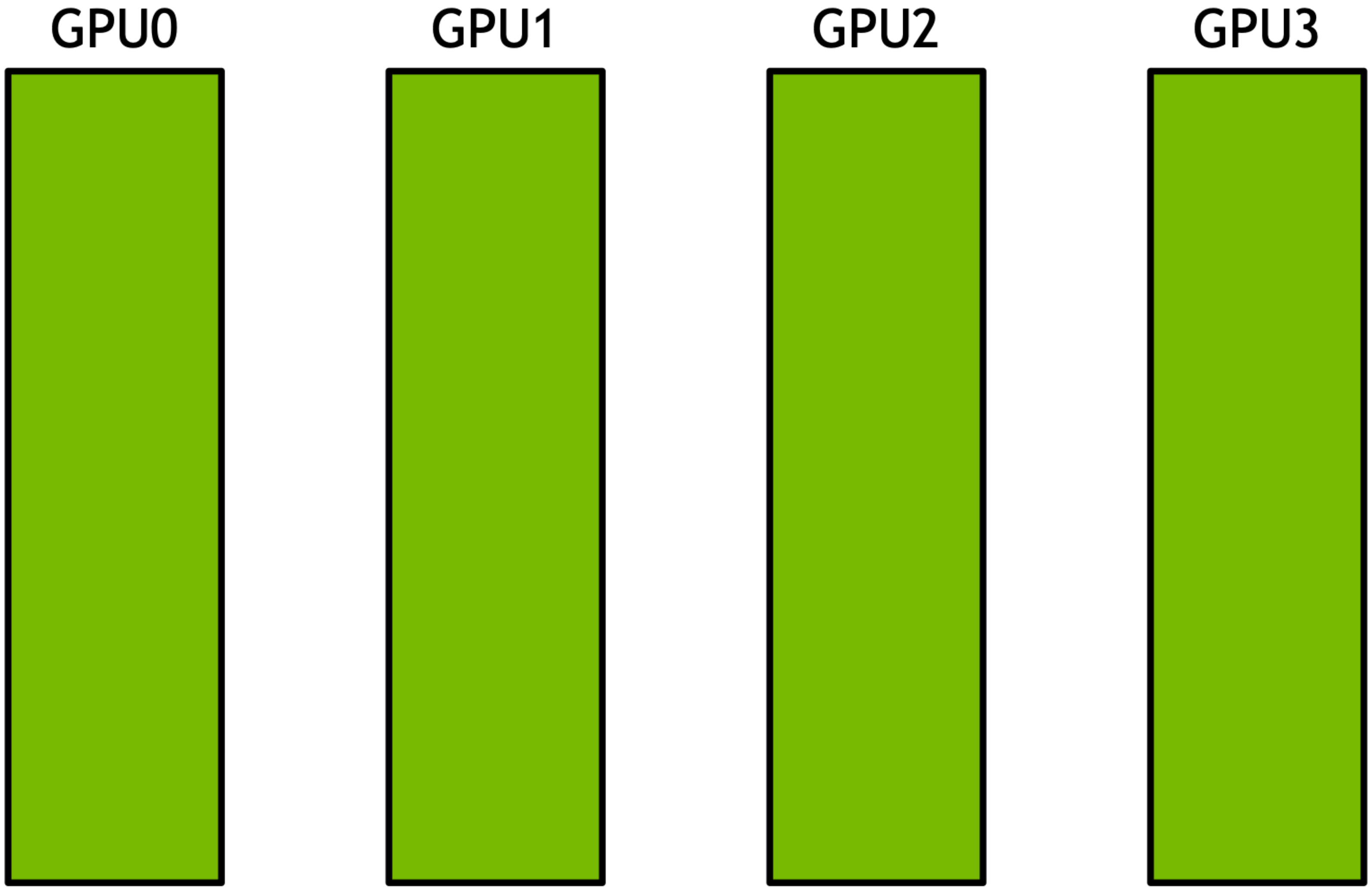
Step 4: t = N/SB

N=bytes to transfer

B=bandwidth

Broadcast with unidirectional ring

break data into S messages



Step 1: $t = N/SB$

Step 2: $t = N/SB$

Step 3: $t = N/SB$

Step 4: $t = N/SB$

…

total time$=(K-2+S)N/SB$

$\sim=N/B$

N=bytes to transfer

B=bandwidth

Example

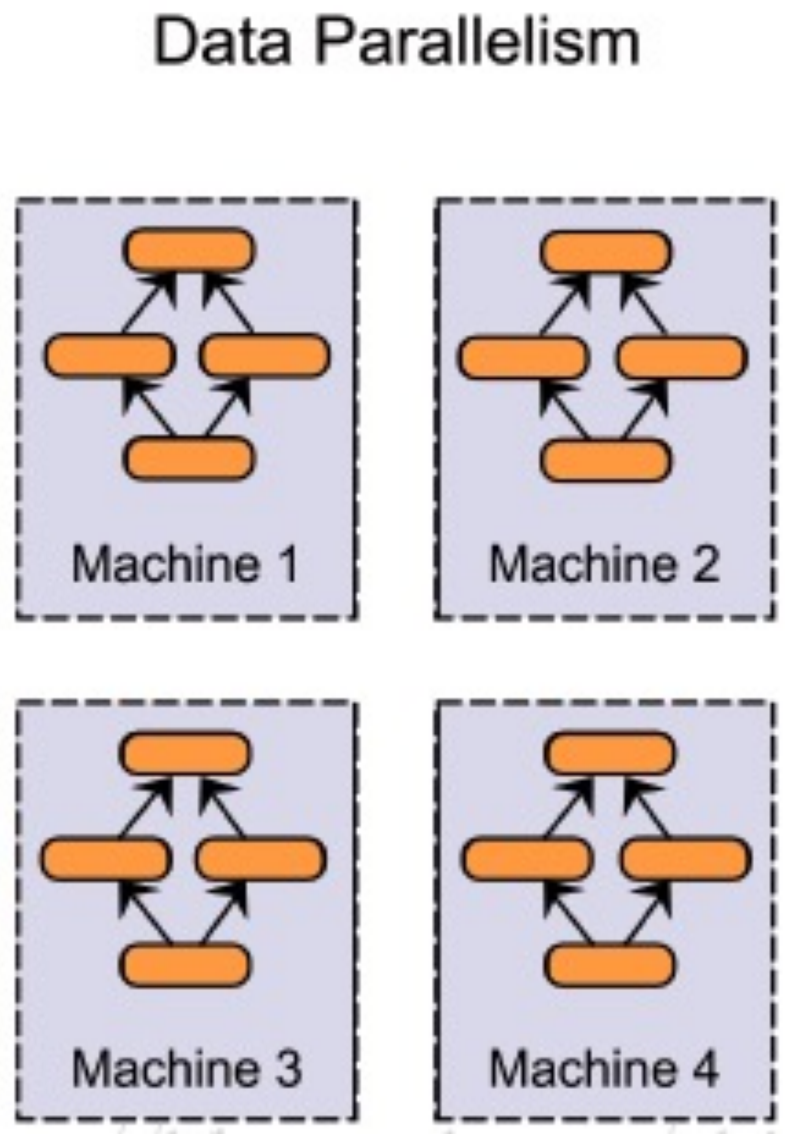
```
//initializing NCCL, group API is required around ncclCommInitRank as it is
//called across multiple GPUs in each thread/process
NCCLCHECK(ncclGroupStart());
for (int i=0; i<nDev; i++) {
  CUDACHECK(cudaSetDevice(localRank*nDev + i));
  NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + i));
}
NCCLCHECK(ncclGroupEnd());
//calling NCCL communication API. Group API is required when using
//multiple devices per thread/process
NCCLCHECK(ncclGroupStart());
for (int i=0; i<nDev; i++)
  NCCLCHECK(ncclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i], size,
ncclFloat, ncclSum, comms[i], s[i]));
NCCLCHECK(ncclGroupEnd());
//synchronizing on CUDA stream to complete NCCL communication
for (int i=0; i<nDev; i++)
  CUDACHECK(cudaStreamSynchronize(s[i]));
```

Today's Topic

- Multi-GPU communication
- Distributed Data Parallel Training

Distributed Data Parallel

- Basic Idea:
  - Create replicas of a model on multiple GPUs
  - Each model performs the forward pass and the backward pass independently
  - Synchronize gradients before the optimizer step



Design Goal of DDP

- Non-intrusive: Develops should be able to reuse the local training script with minimal modifications.

- Interceptive: The API needs to allow the implementation to intercept various signals and trigger appropriate algorithms promptly. The API must expose as many optimization opportunities as possible to the internal implementation.

Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, VLDB 2020.

Distributed Data Parallel

- You can use DDP with minimal code change in pytorch!

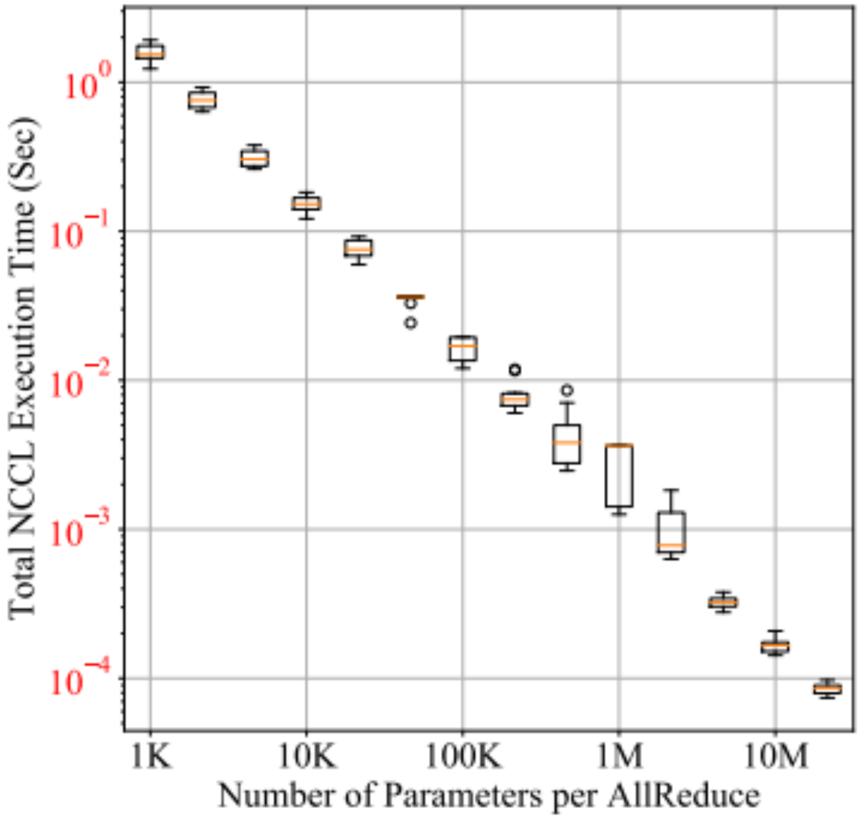
```
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
net = par.DistributedDataParallel(net)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```

How to Implement Distributed Data Parallel

- Naïve solution: synchronize gradients after the *entire* backward pass finishes
  - What can be improved?

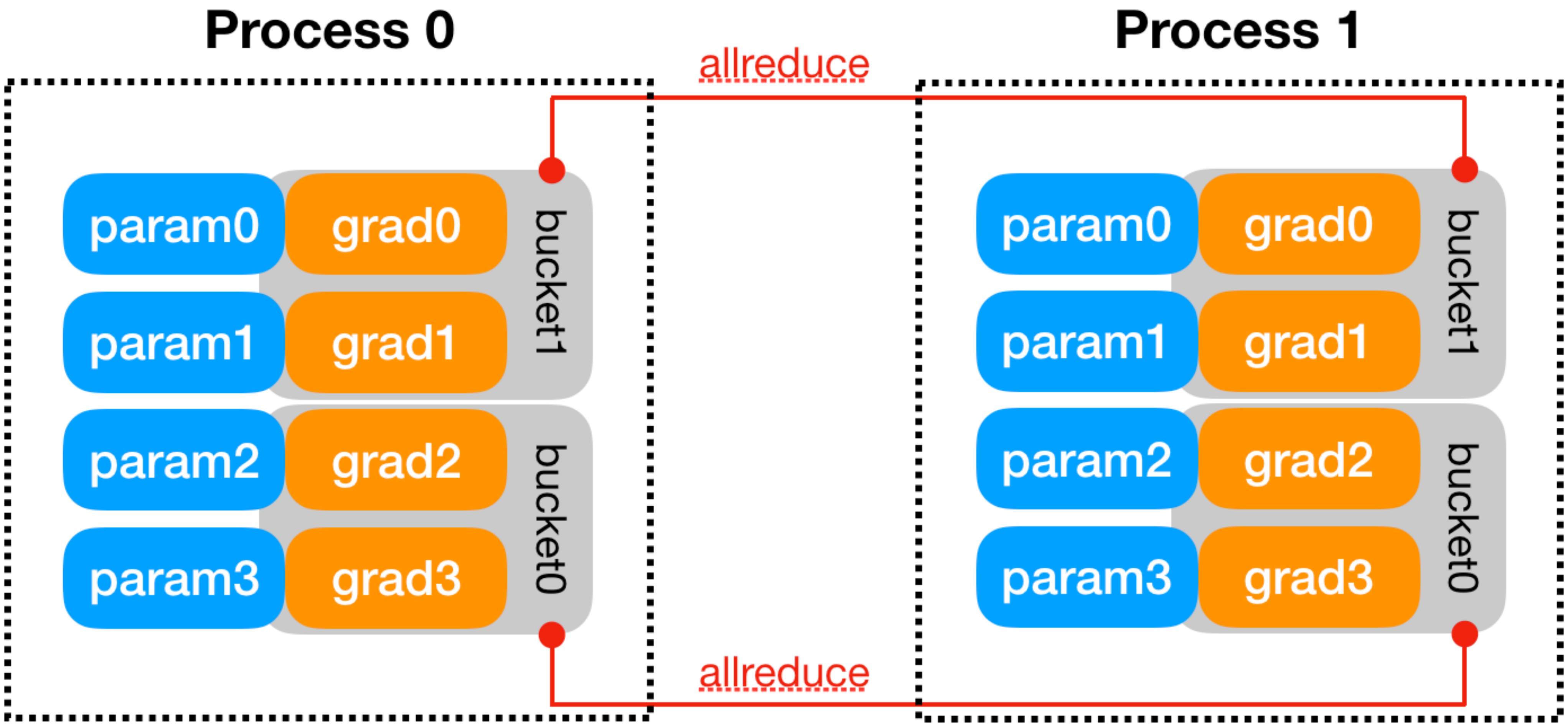
Implementing Distributed Data Parallel

- Naïve solution: synchronize gradients after the *entire* backward pass finishes
  - We can overlap gradient computation and synchronization!
- But how often should we synchronize? Per parameter?
  - Too much synchronization slows down execution



(a) NCCL

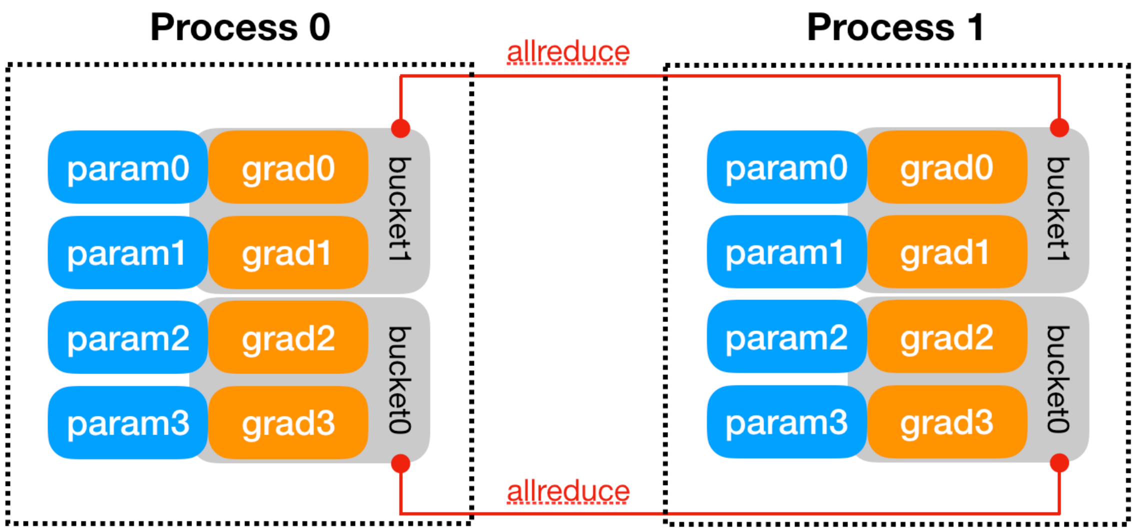
Gradient Bucketing



Gradient Bucketing

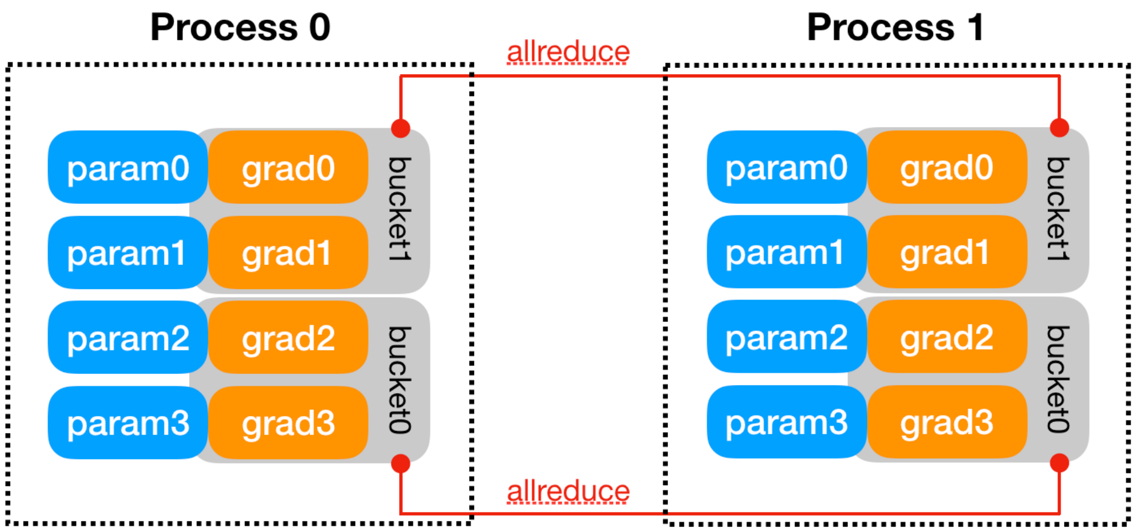
- Bucket size can be configured by setting the **bucket_cap_mb** argument in DDP constructor.

- The mapping from parameter gradients to buckets is determined at the construction time, based on the bucket size limit and parameter sizes.



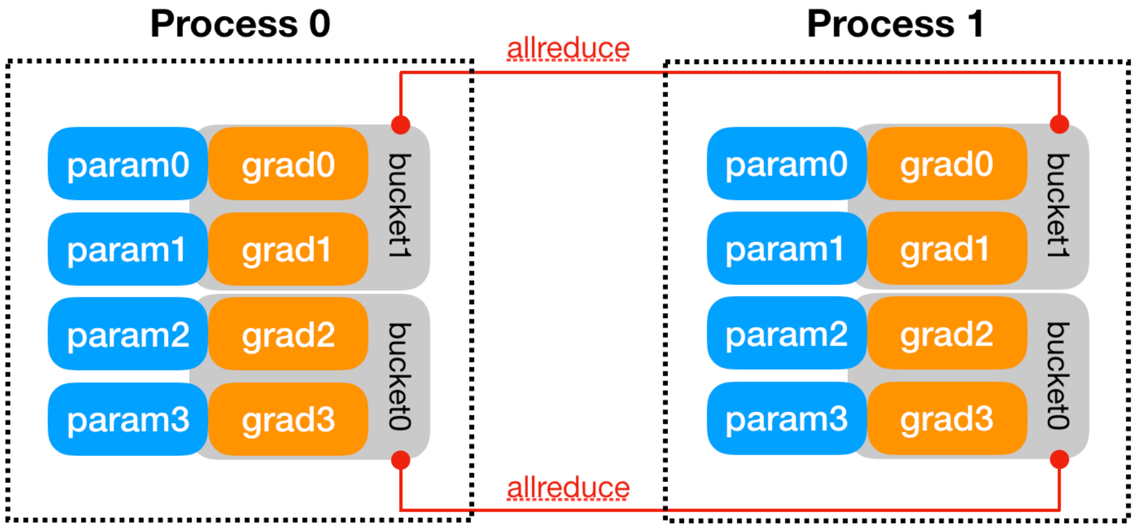
Gradient Bucketing

- Model parameters are allocated into buckets in (roughly) the reverse order of Model.parameters() from the given model.
- DDP expects gradients to become ready during the backward pass in approximately that order.

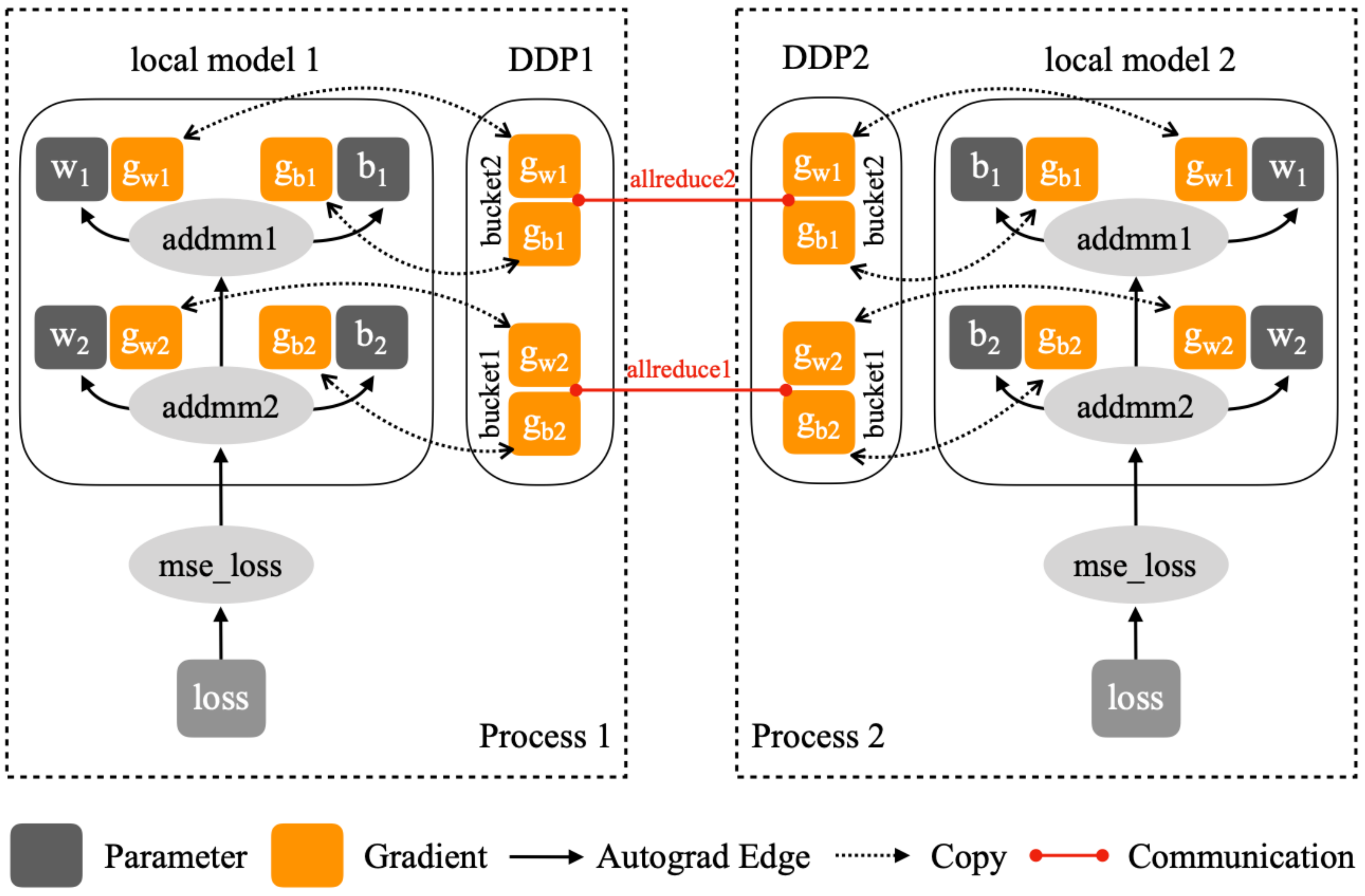


Gradient Bucketing

- When gradients in one bucket are all ready, the Reducer kicks off an asynchronous allReduce on that bucket to calculate average of gradients across all processes.
- Overlapping computation (backward) with communication (AllReduce)



Gradient Reduction

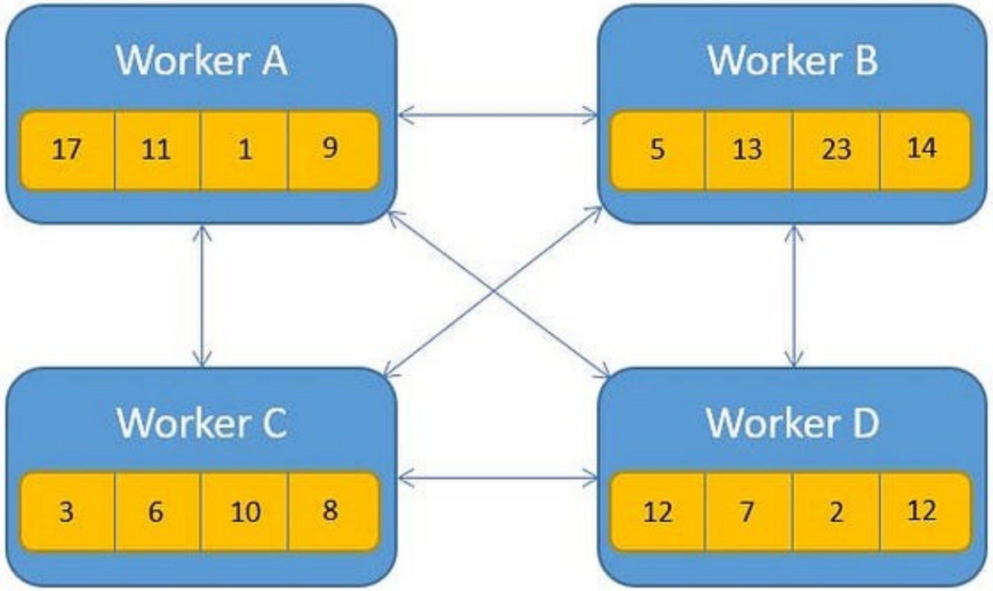


DDP Implementation

```
// The function `autograd_hook` is called after the gradient for a
// model parameter has been accumulated into its gradient tensor.
// This function is only to be called from the autograd thread.
void Reducer::autograd_hook(size_t index) {
      mark_variable_ready(index);
}

void Reducer::mark_variable_ready(size_t variable_index) {
      const auto& bucket_index = variable_locators_[variable_index];
      auto& bucket = buckets_[bucket_index.bucket_index];

      if (--bucket.pending == 0) {
            mark_bucket_ready(bucket_index.bucket_index);
      }
}

void Reducer::mark_bucket_ready(size_t bucket_index) {
      for (; next_bucket_ < buckets_.size() && buckets_[next_bucket_].pending == 0; next_bucket_++) {
            num_buckets_ready_++;
            auto& bucket = buckets_[next_bucket_];
            all_reduce_bucket(bucket);
      }
}

void Reducer::all_reduce_bucket(Bucket& bucket) {
      auto variables_for_bucket = get_variables_for_bucket(next_bucket_, bucket);
      const auto& tensor = bucket.gradients;
      GradBucket grad_bucket(next_bucket_, buckets_.size(), tensor, bucket.offsets,
            bucket.lengths, bucket.sizes_vec, variables_for_bucket);
      bucket.future_work = run_comm_hook(grad_bucket);
}
```

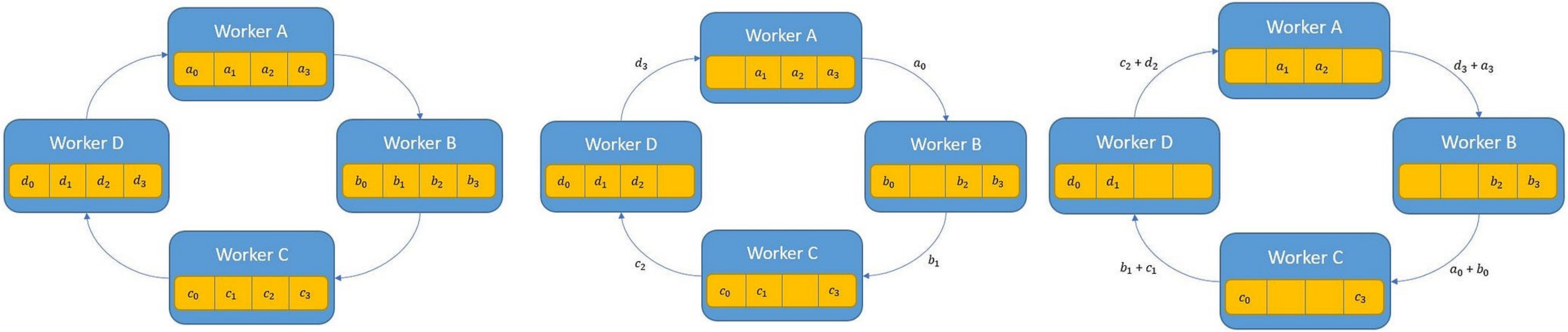
How to Synchronize Gradients?

- Naïve all-reduce

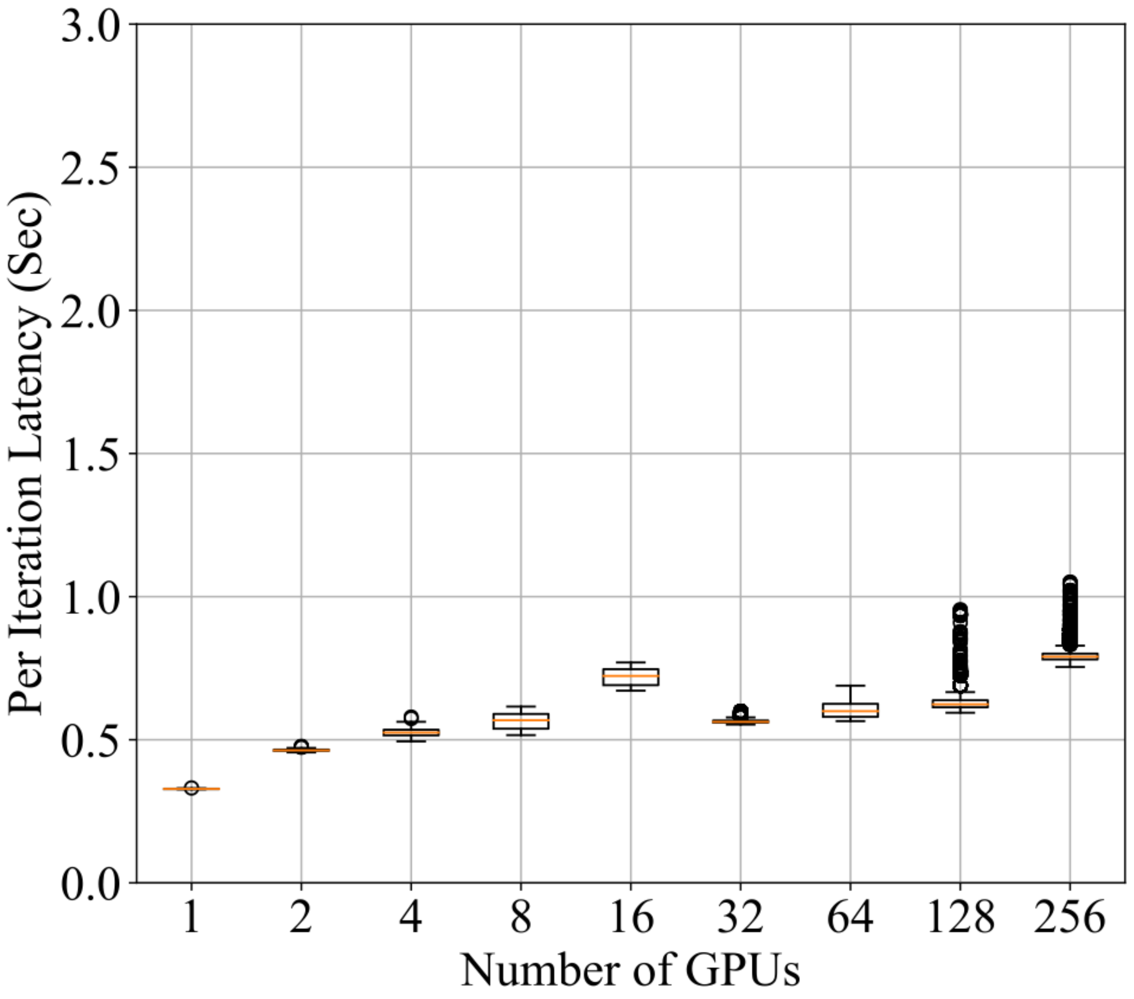


How to Synchronize Gradients?

- Ring all-reduce

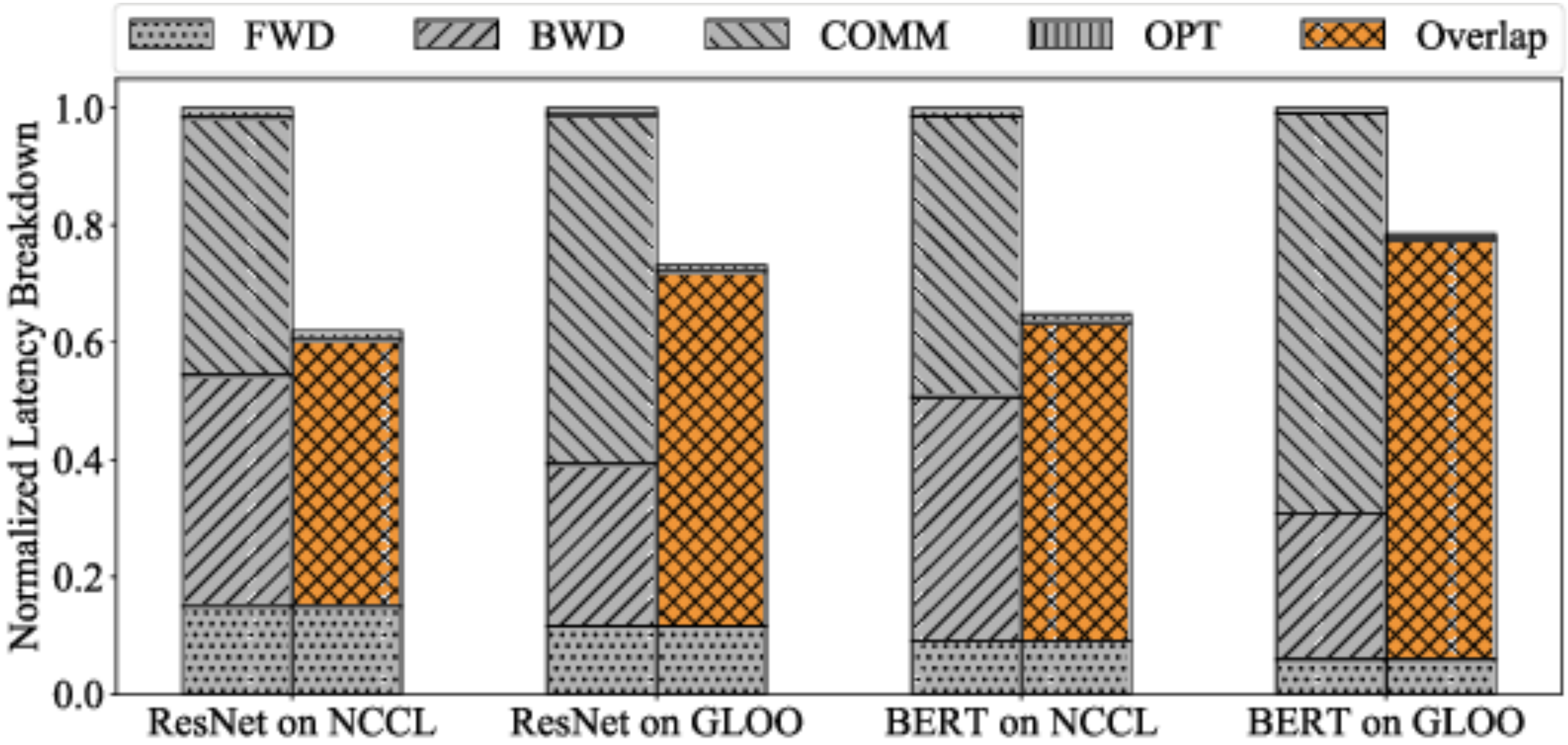


DDP Scalability



(c) BERT on NCCL

DDP Reduces Latency by Overlapping Communication and Computation

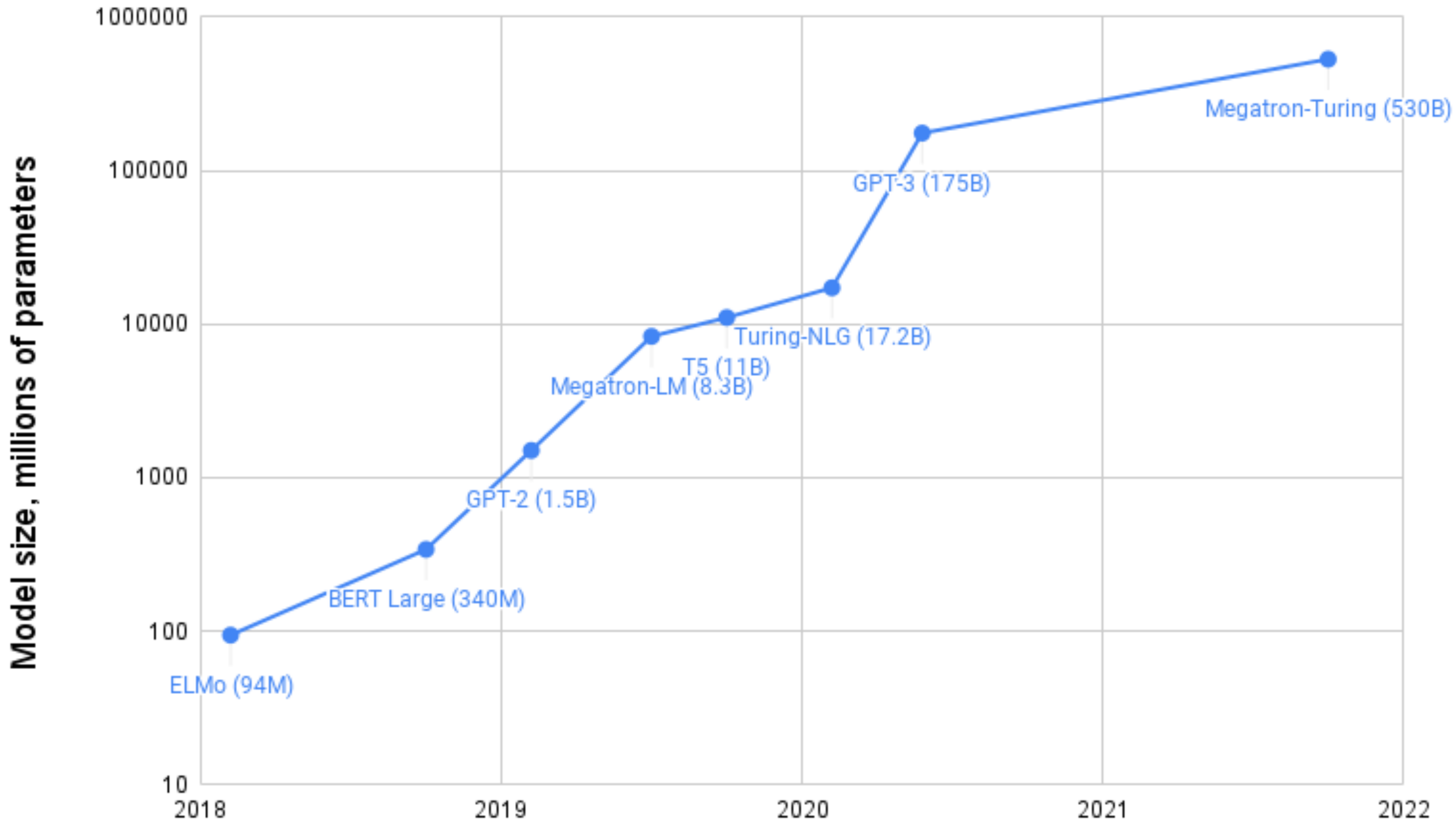


bwd & comm

Figure 6: Per Iteration Latency Breakdown

Fully Shared Data Parallel

- Motivation: Large models cannot fit into one GPU



Reading for next lecture

- Huang et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. 2018
- Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019
- Narayanan et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, SC 2021