Agenda

What is NCCL?

Why use NCCL?

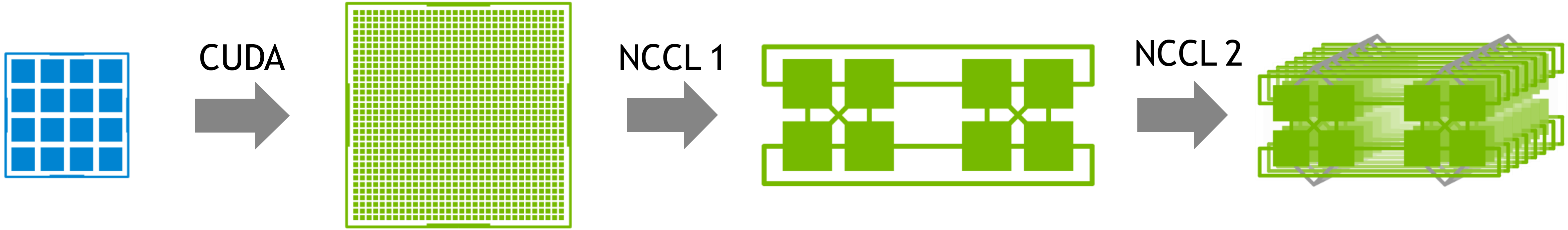
How to use NCCL?

What is NCCL?

DEEP LEARNING ON GPUS

Making DL training times shorter

Deeper neural networks, larger data sets … training is a very, very long operation !



CUDA

NCCL 1

NCCL 2

Multi-core CPU

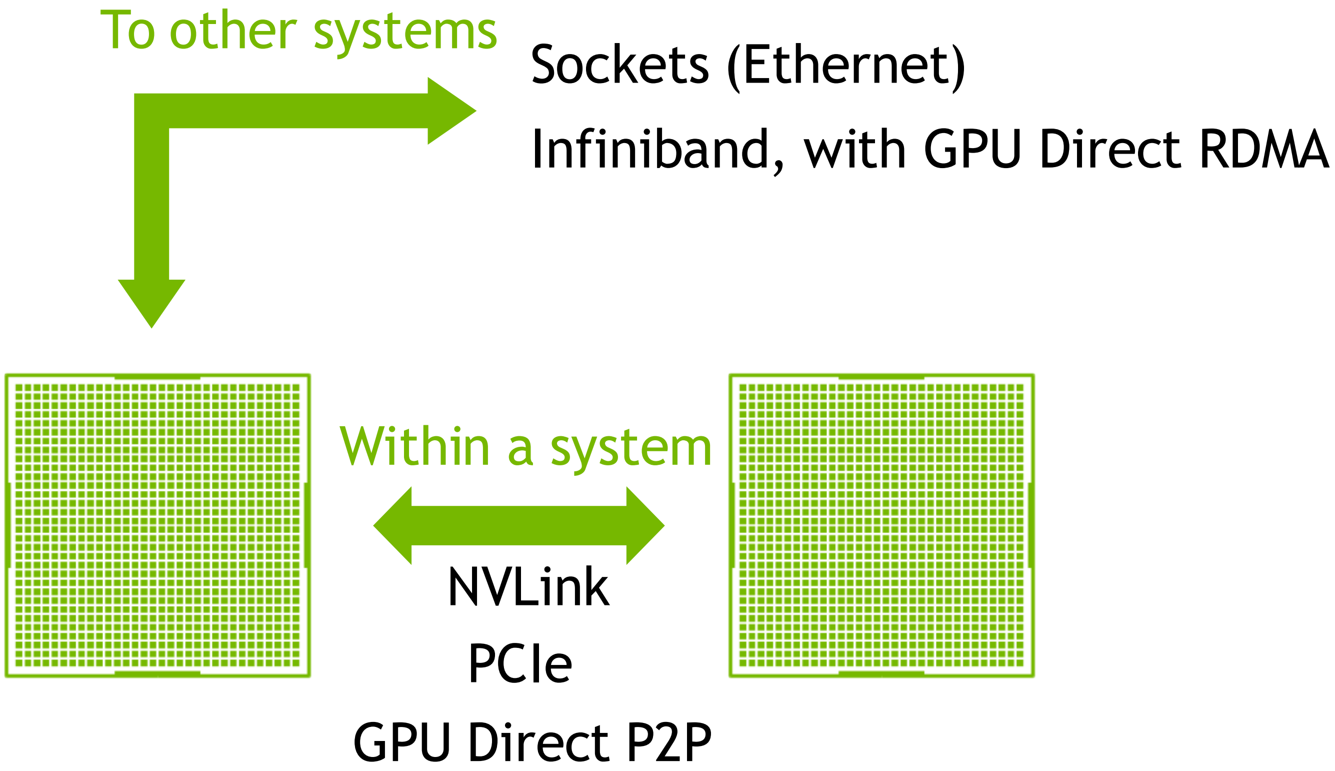
GPU

Multi-GPU

Multi-GPU
Multi-node

NCCL

A multi-GPU communication library



To other systems

Sockets (Ethernet)

Infiniband, with GPU Direct RDMA

Within a system

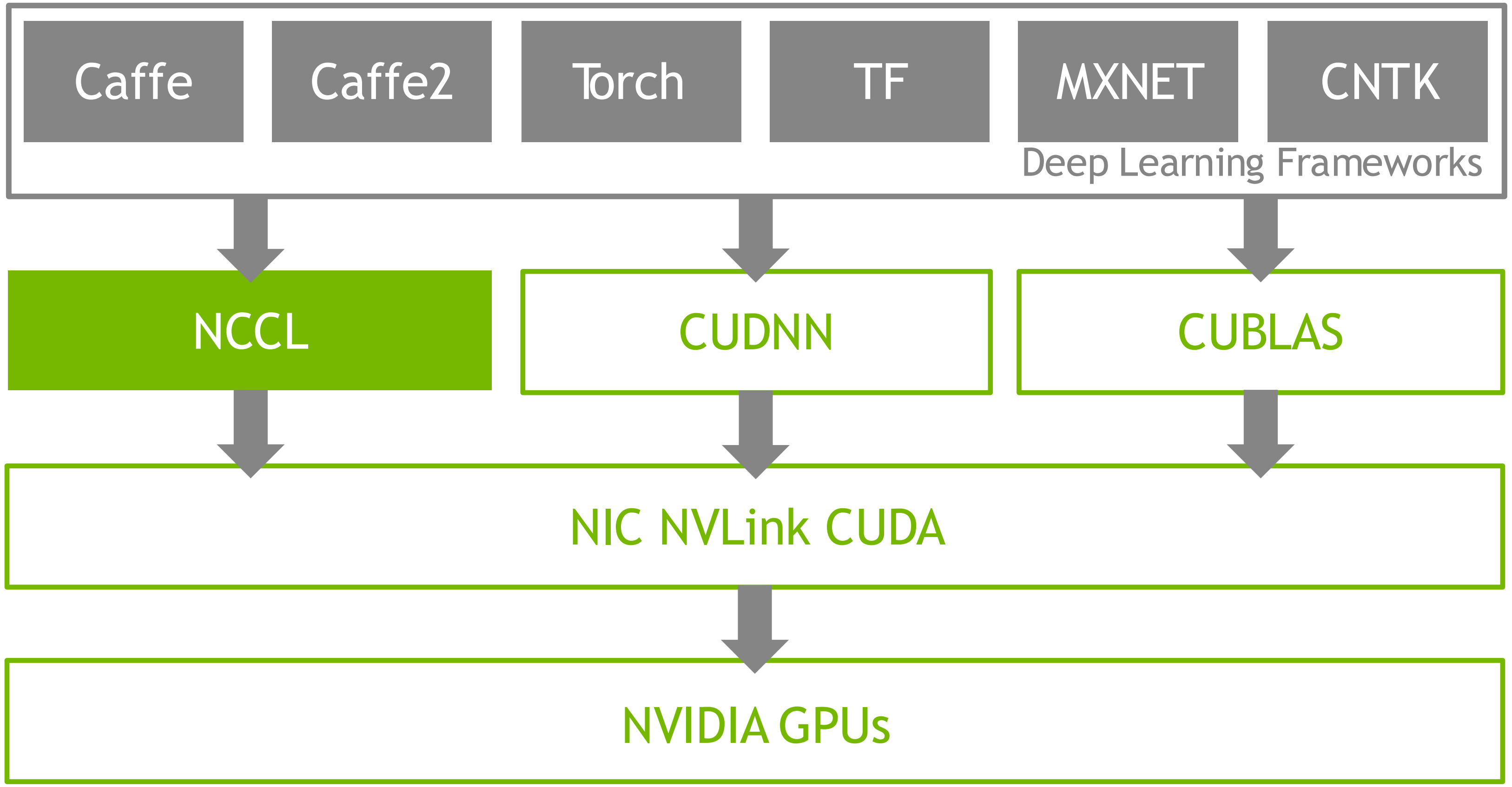
NVLink

PCIe

GPU Direct P2P

NCCL

Architecture



Caffe

Caffe2

Torch

TF

MXNET

CNTK

Deep Learning Frameworks

NCCL

CUDNN

CUBLAS

NIC NVLink CUDA

NVIDIA GPUs

Why use NCCL?

DESIGN

Optimized collective communication library between CUDA devices.

Easy to integrate into any DL framework, as well as MPI.

Runs on the GPU using asynchronous CUDA kernels, for faster access to GPU traditional HPC apps using memory, parallel reductions, NVLink usage.

Operates on CUDA pointers. Operations are tied to a CUDA stream.

Uses as little threads as possible to permit other computation to progress simultaneously.

What is collective communication?

Common Communication Patterns

Point-to-point communication

- Single sender, single receiver
- Relatively easy to implement efficiently

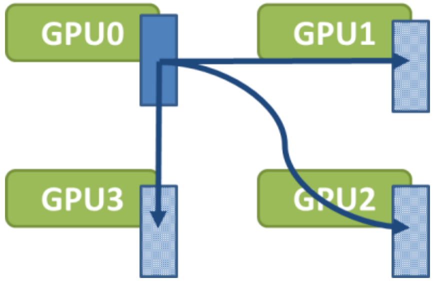
Collective communication

- Multiple senders and/or receivers
- Patterns include broadcast, scatter, gather, reduce, all-to-all, …
- Difficult to implement efficiently

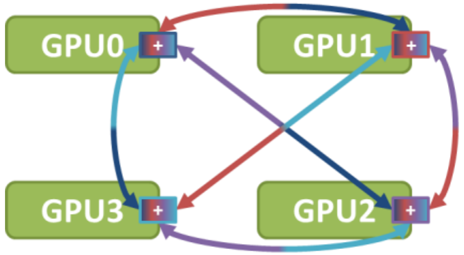
COLLECTIVE COMMUNICATION

Multiple senders and/or receivers

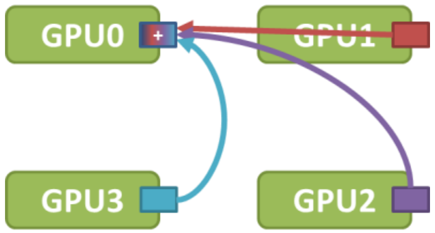
Broadcast



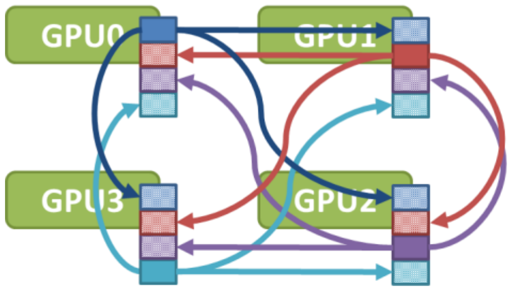
All-Reduce



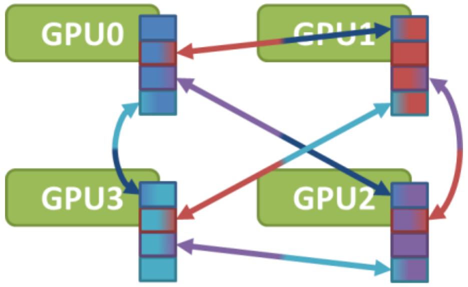
Reduce



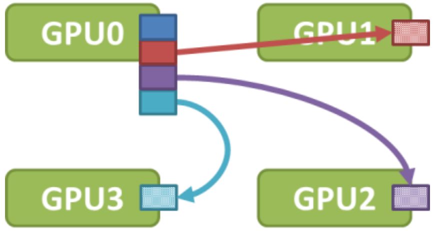
All-Gather



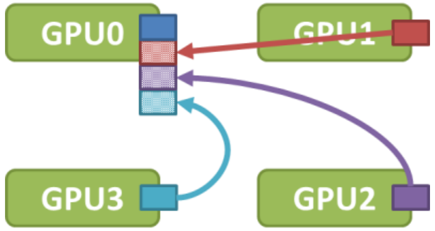
All-to-All



Scatter

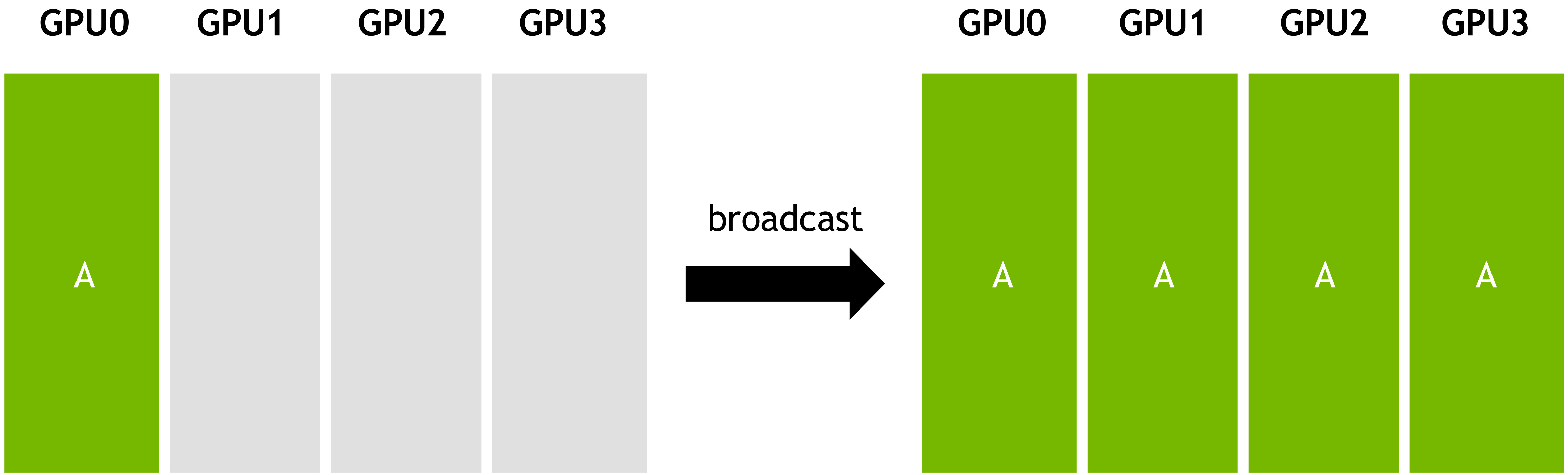


Gather



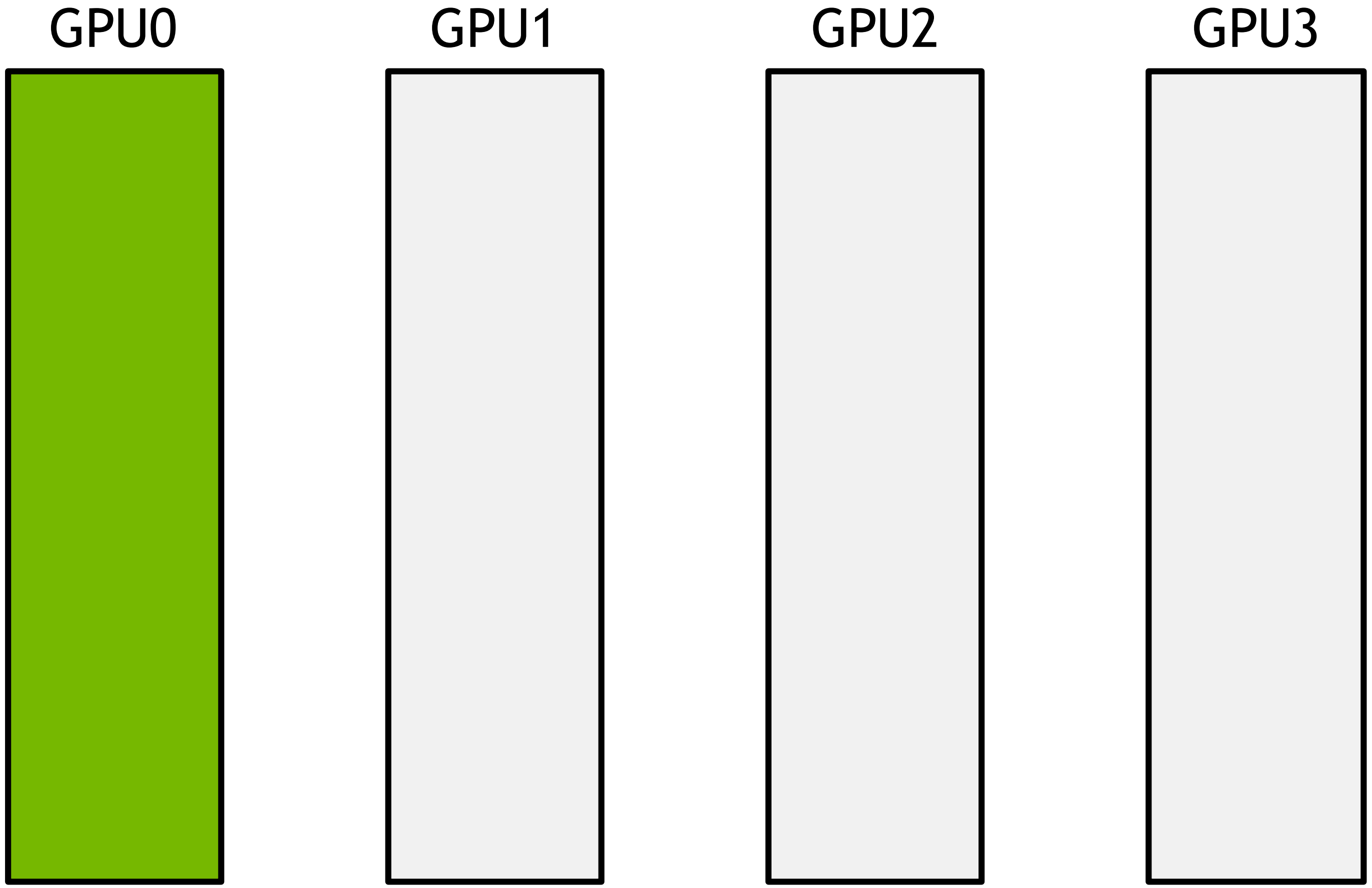
BROADCAST

One sender, multiple receivers



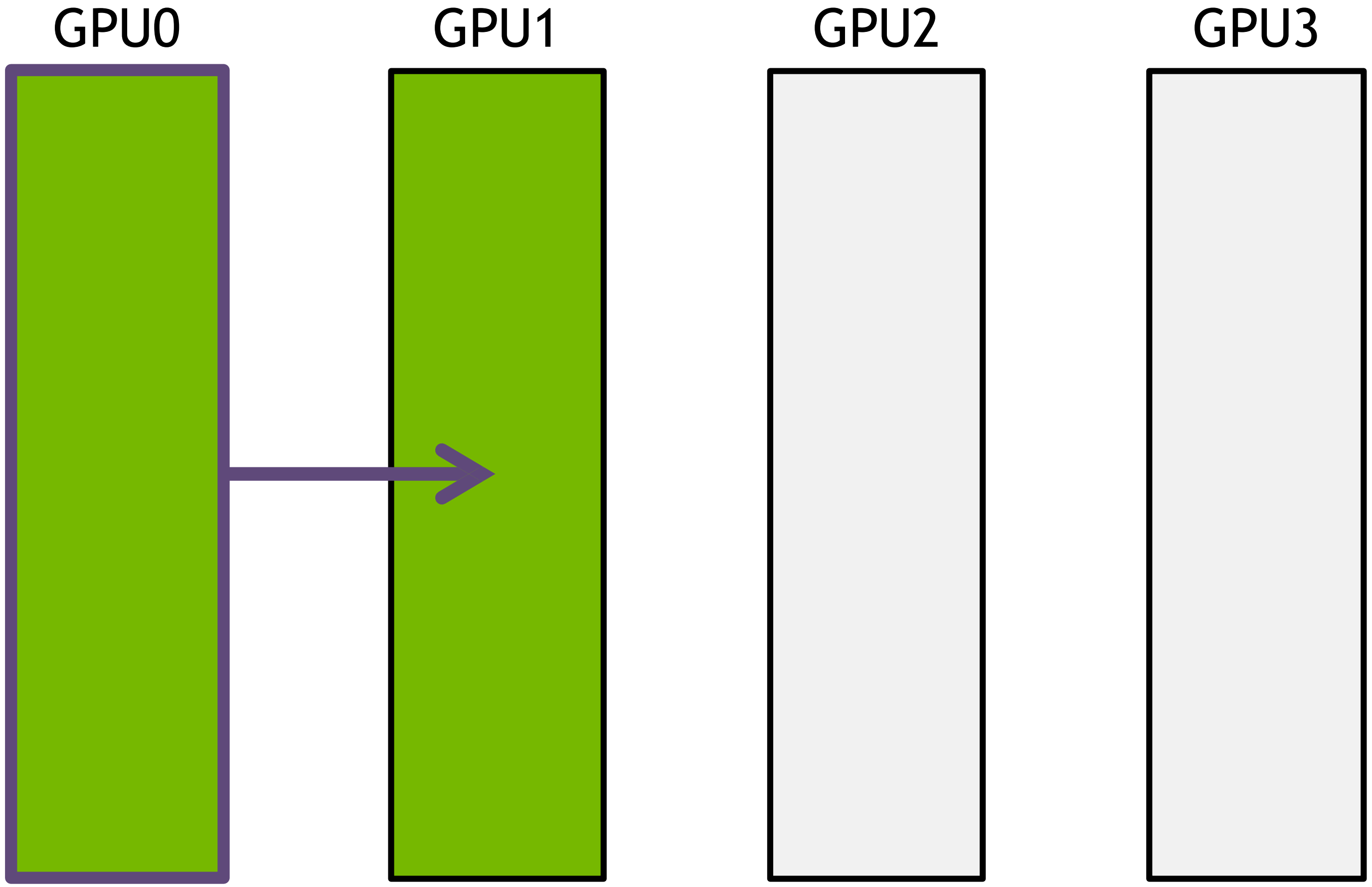
BROADCAST

with unidirectional ring



BROADCAST

with unidirectional ring



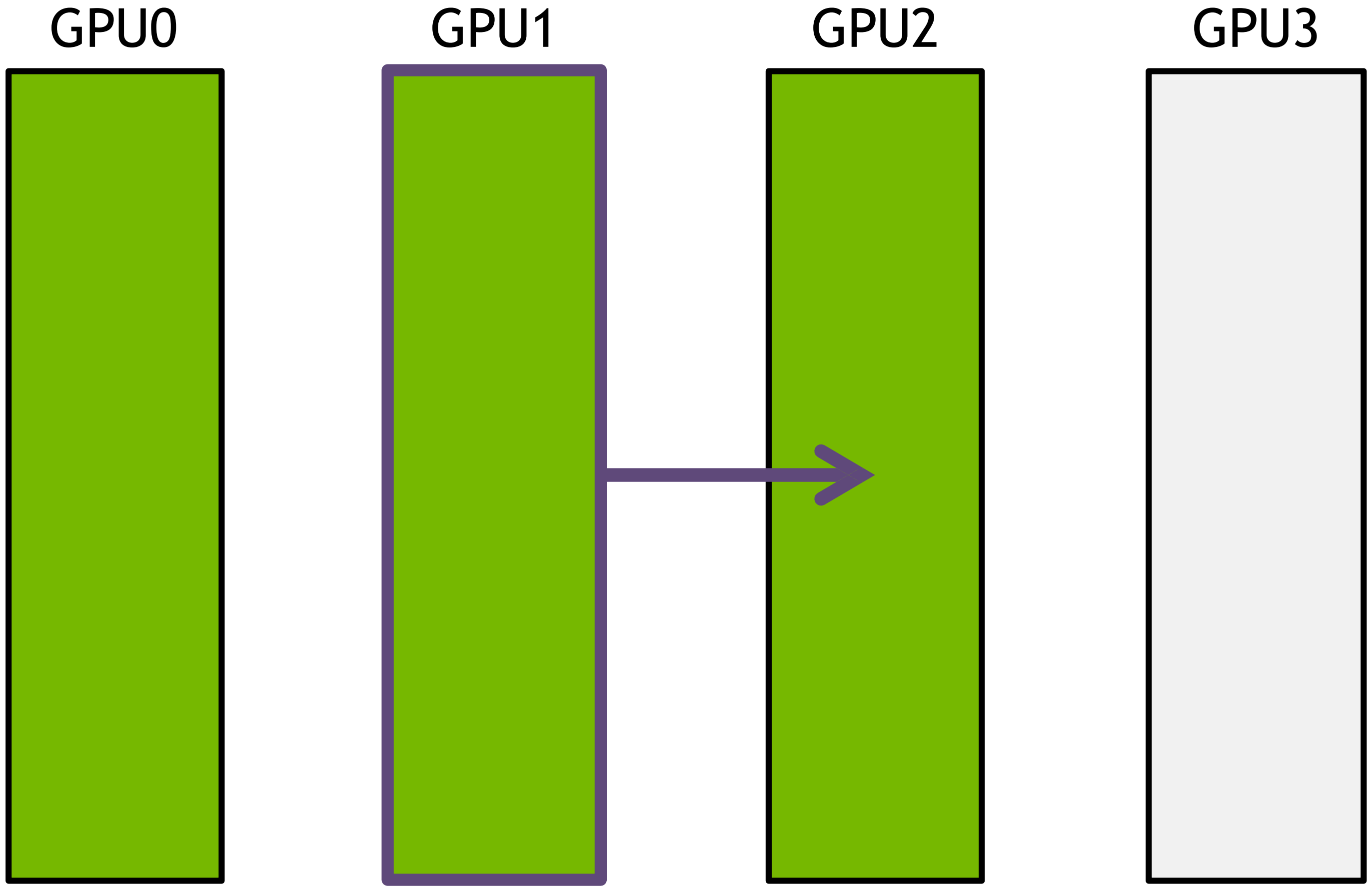
Step 1: $\Delta t = N/B$

N: bytes to broadcast

B: bandwidth of each link

BROADCAST

with unidirectional ring



Step 1: $\Delta t = N/B$

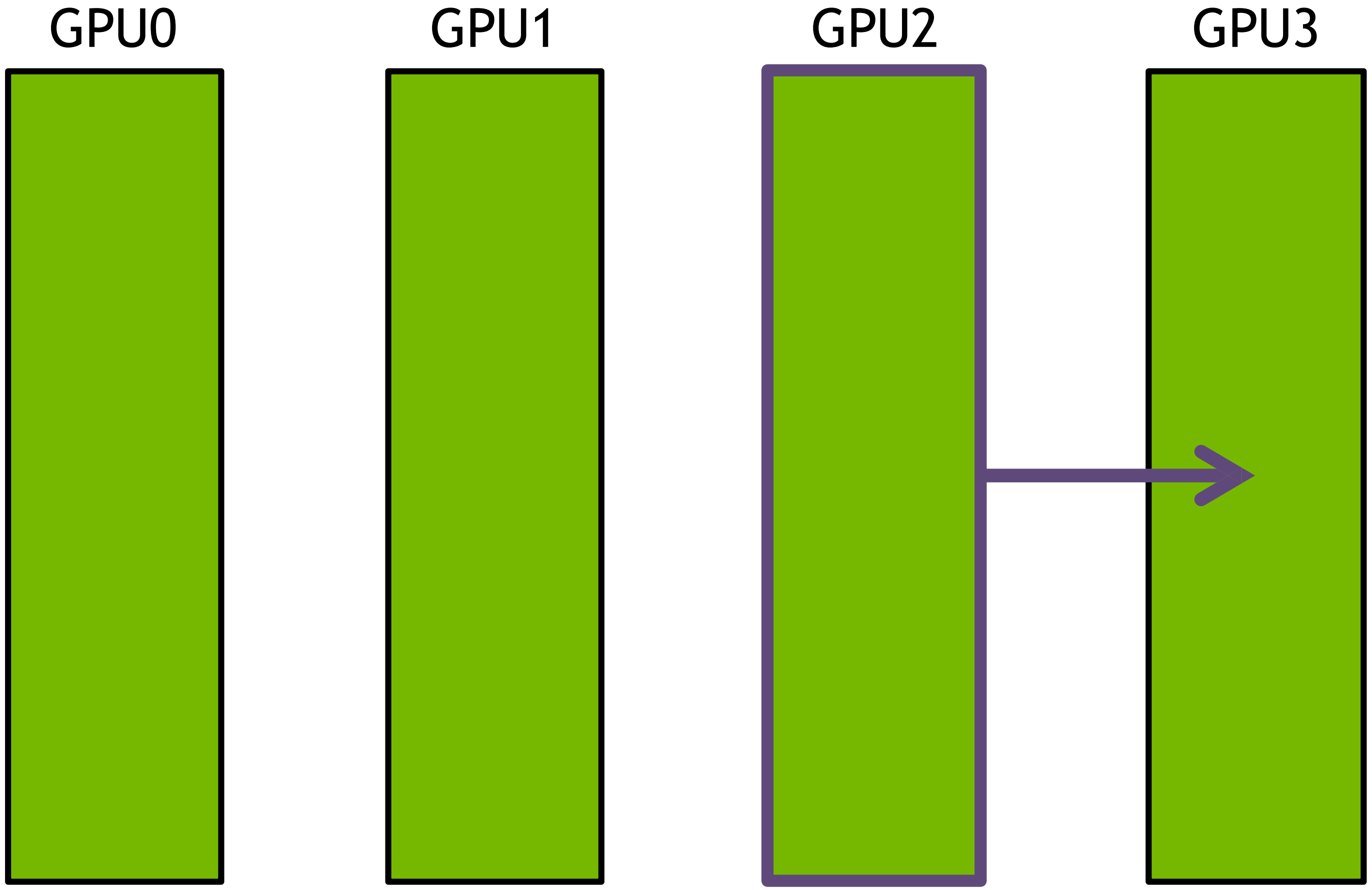
Step 2: $\Delta t = N/B$

N: bytes to broadcast

B: bandwidth of each link

BROADCAST

with unidirectional ring



Step 1: $\Delta t = N/B$

Step 2: $\Delta t = N/B$

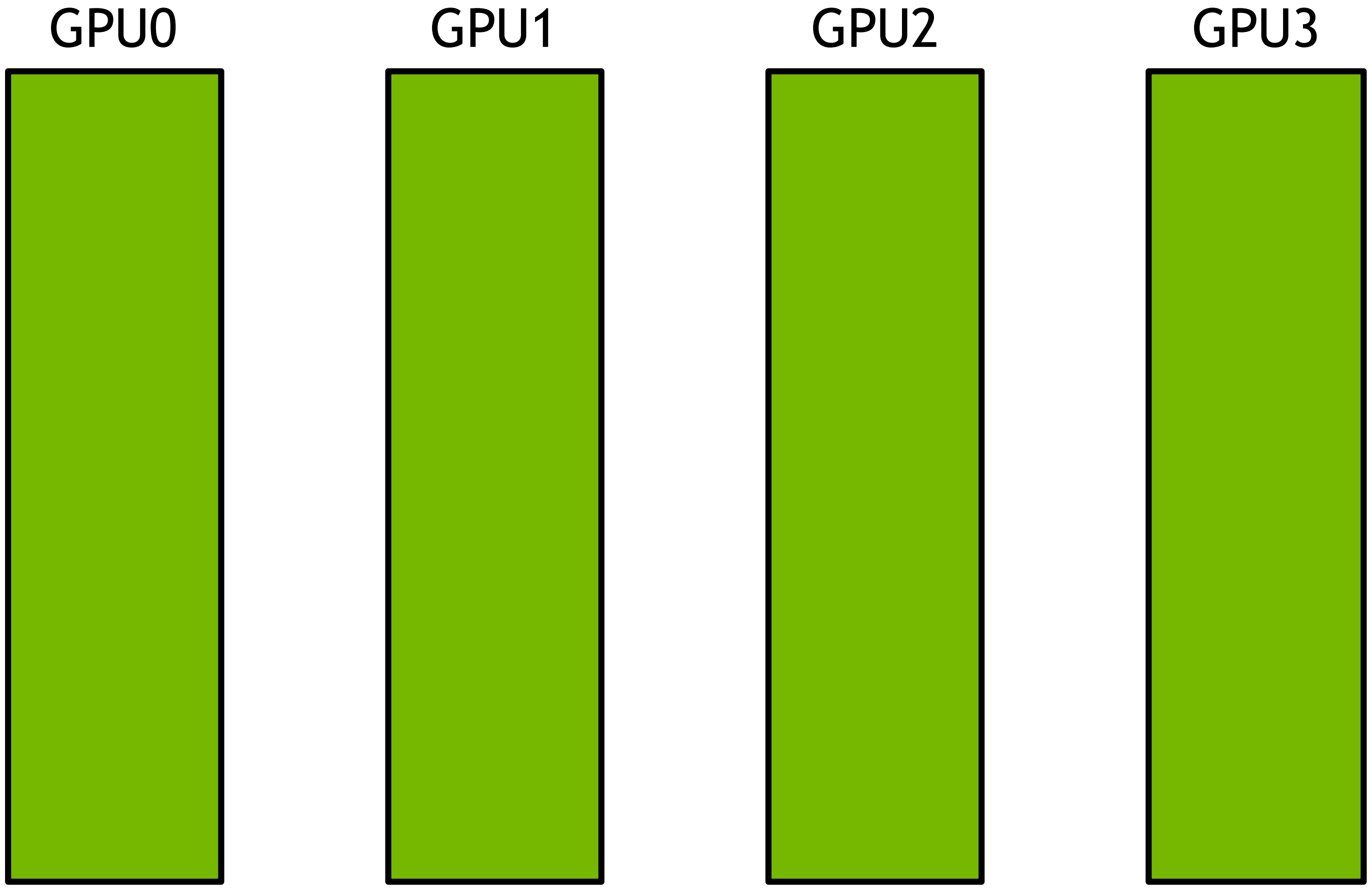
Step 3: $\Delta t = N/B$

N: bytes to broadcast

B: bandwidth of each link

BROADCAST

with unidirectional ring



Step 1: $\Delta t = N/B$

Step 2: $\Delta t = N/B$

Step 3: $\Delta t = N/B$

Total time: $(k-1)N/B$

N: bytes to broadcast

B: bandwidth of each link

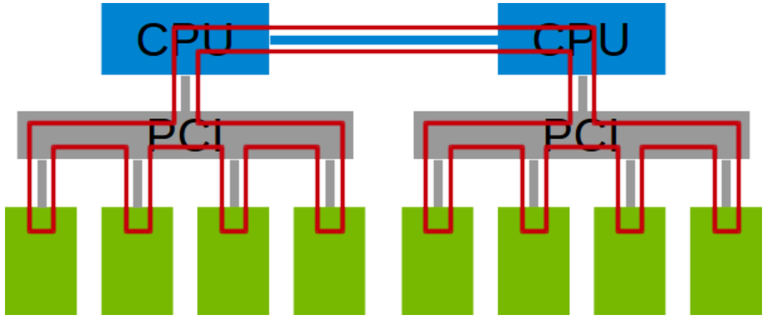
k: number of GPUs

How NCCL optimize it?

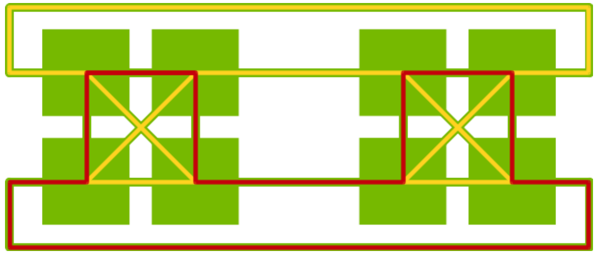
DESIGN

Rings

NCCL uses rings to move data across all GPUs.



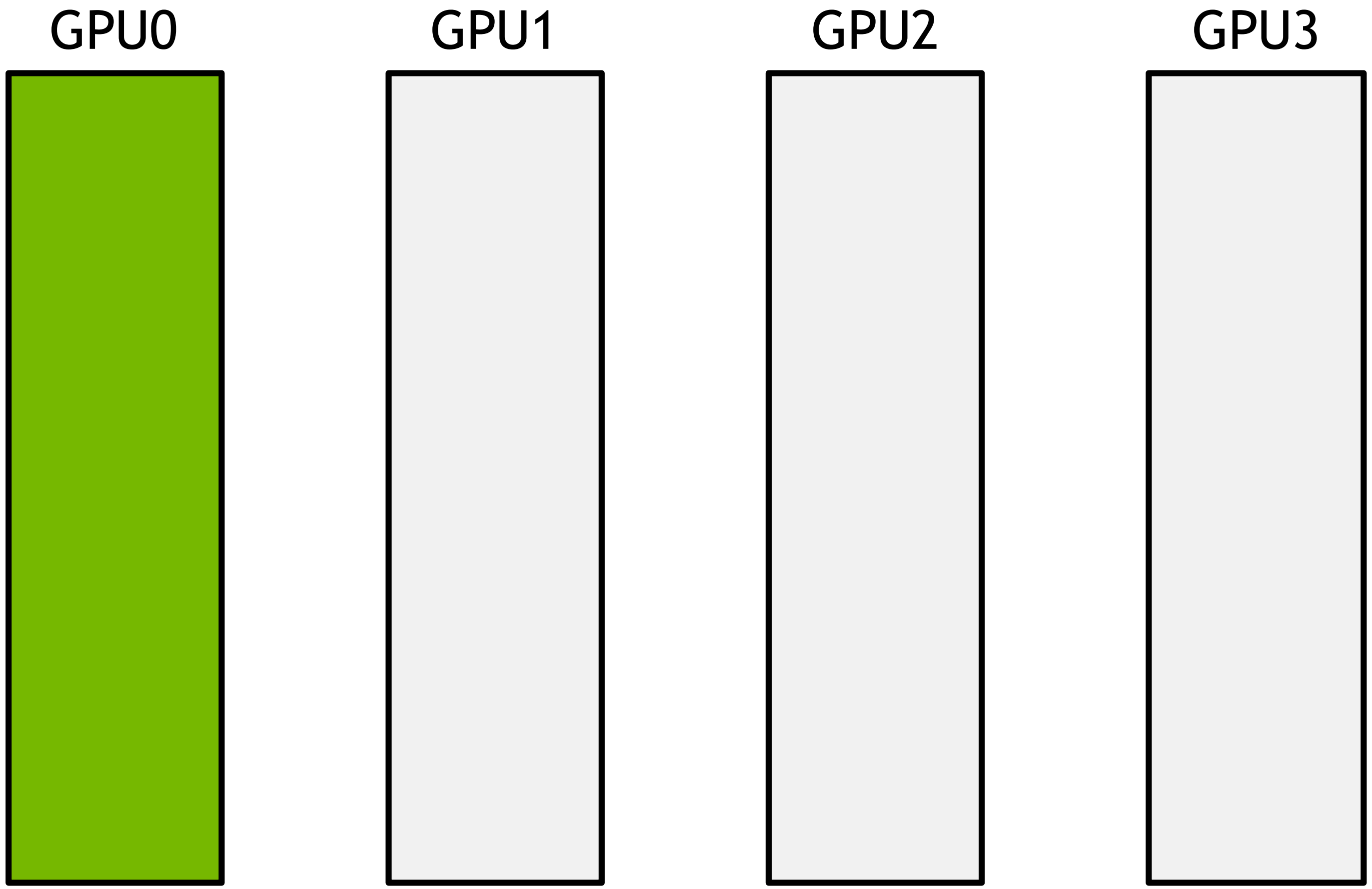
PCIe / QPI : 1 unidirectional ring



DGX-1 : 4 unidirectional rings

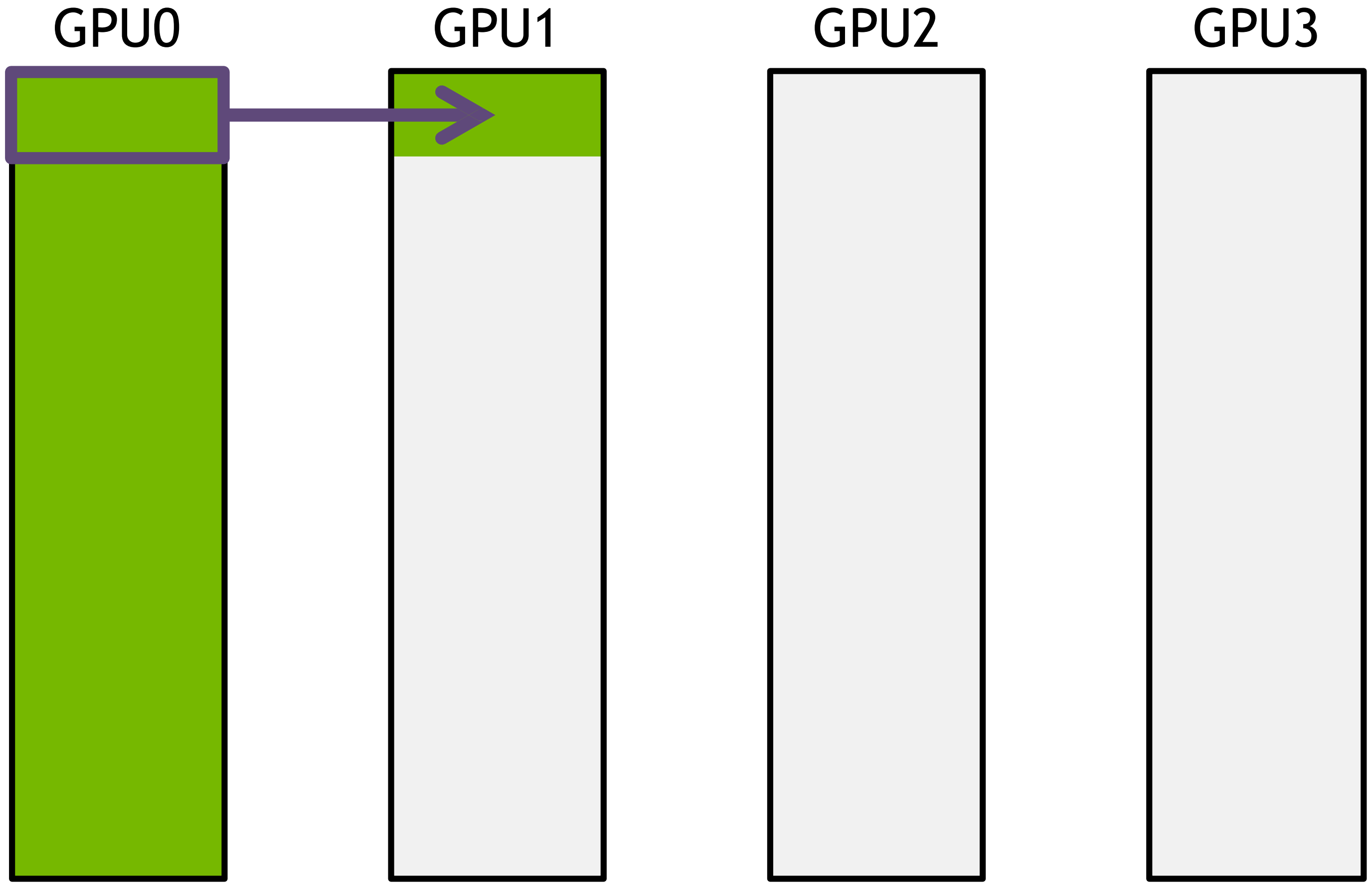
BROADCAST

with unidirectional ring



BROADCAST

with unidirectional ring

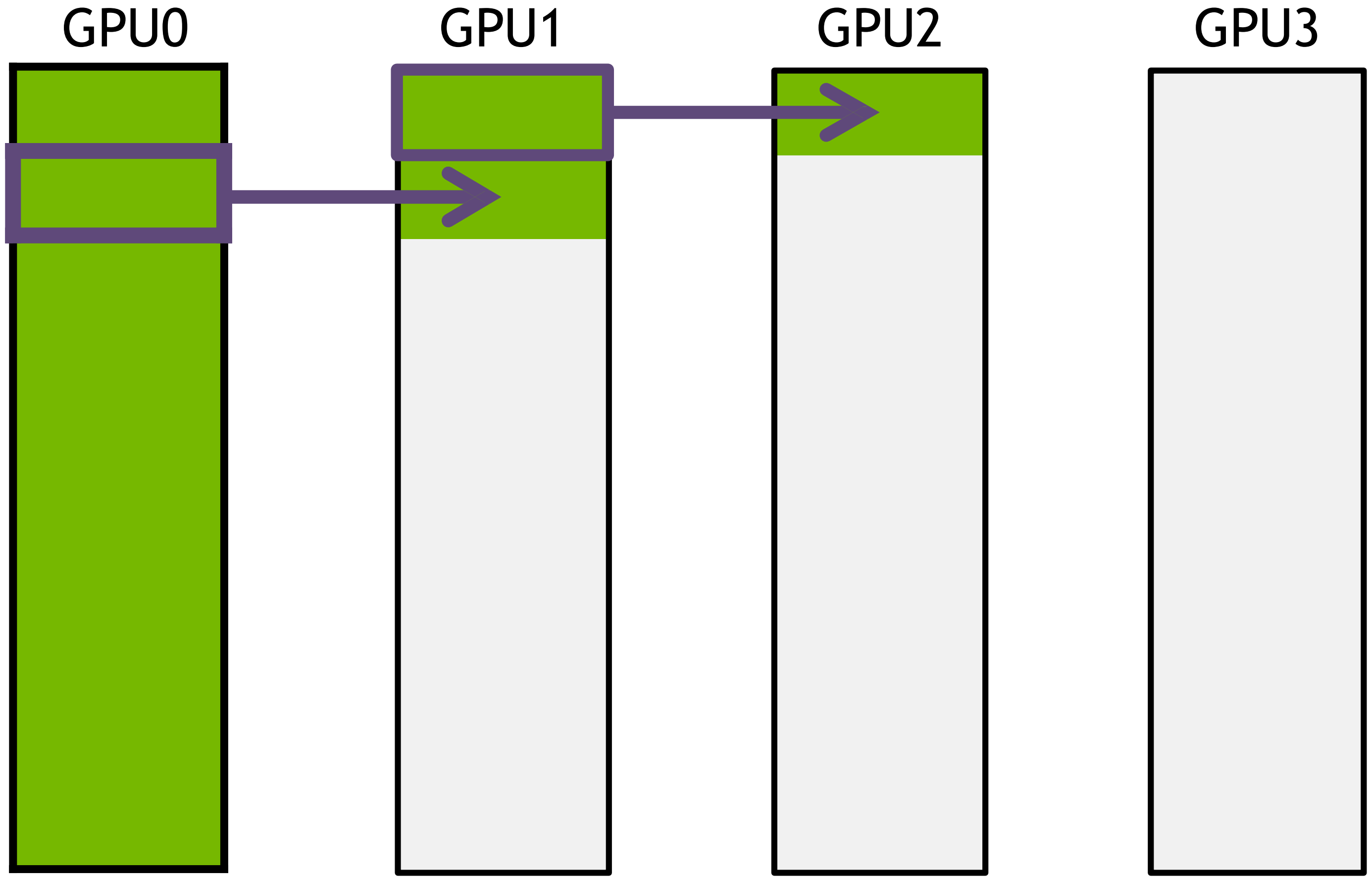


Split data into S messages

Step 1: $\Delta t = N/(SB)$

BROADCAST

with unidirectional ring



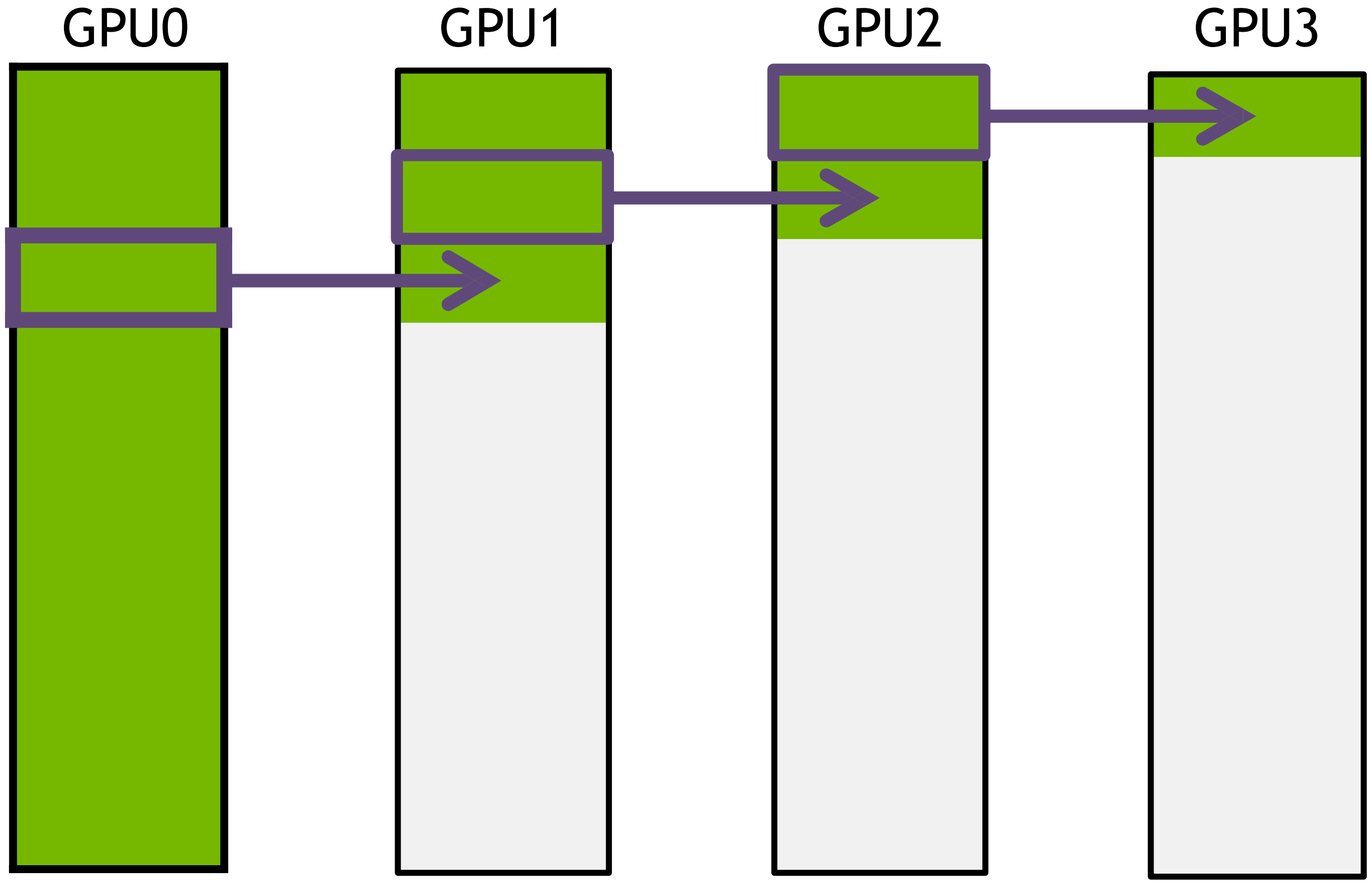
Split data into S messages

Step 1: $\Delta t = N/(SB)$

Step 2: $\Delta t = N/(SB)$

BROADCAST

with unidirectional ring



Split data into S messages

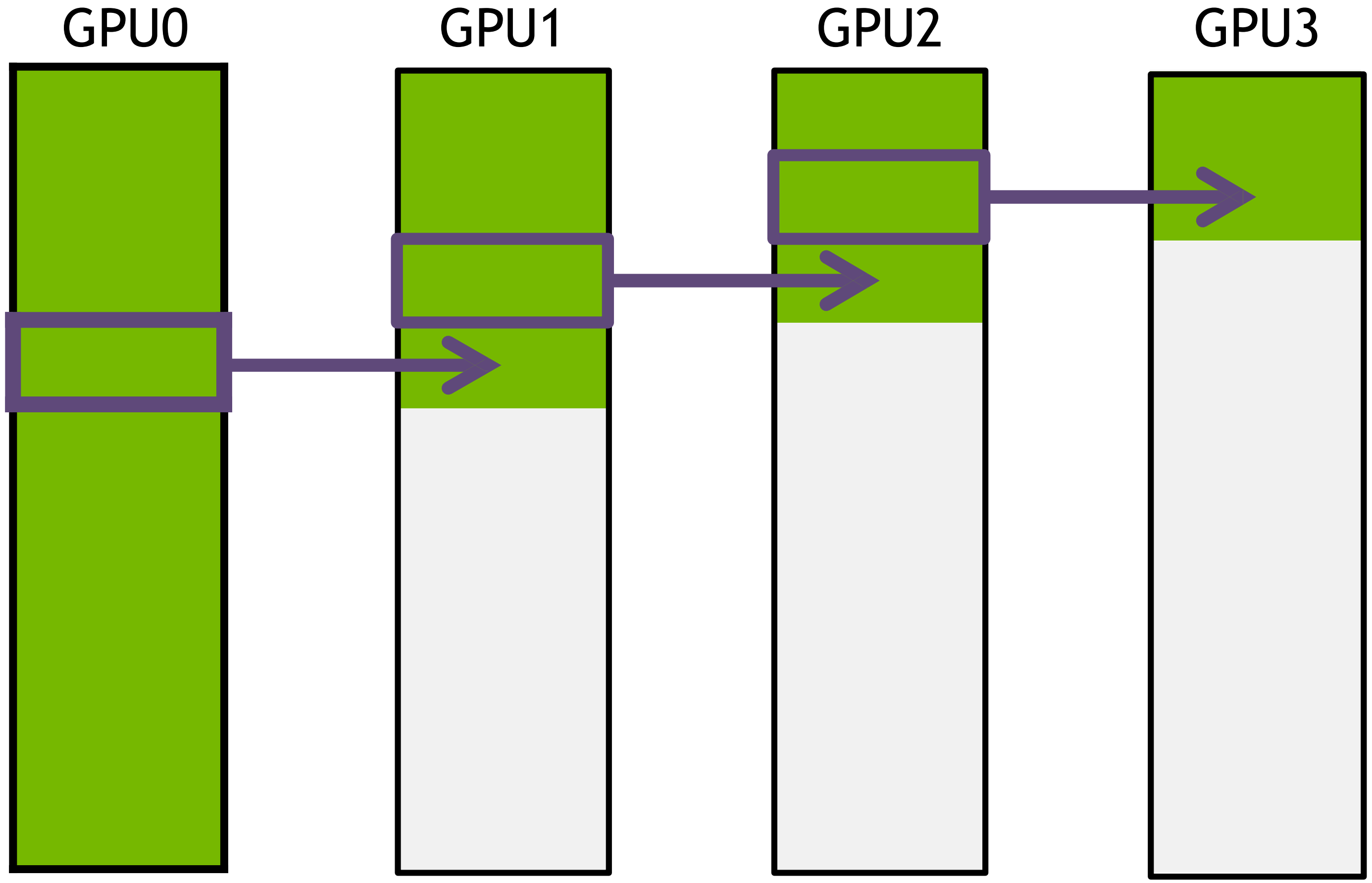
Step 1: $\Delta t = N/(SB)$

Step 2: $\Delta t = N/(SB)$

Step 3: $\Delta t = N/(SB)$

BROADCAST

with unidirectional ring



Split data into S messages

Step 1: $\Delta t = N/(SB)$

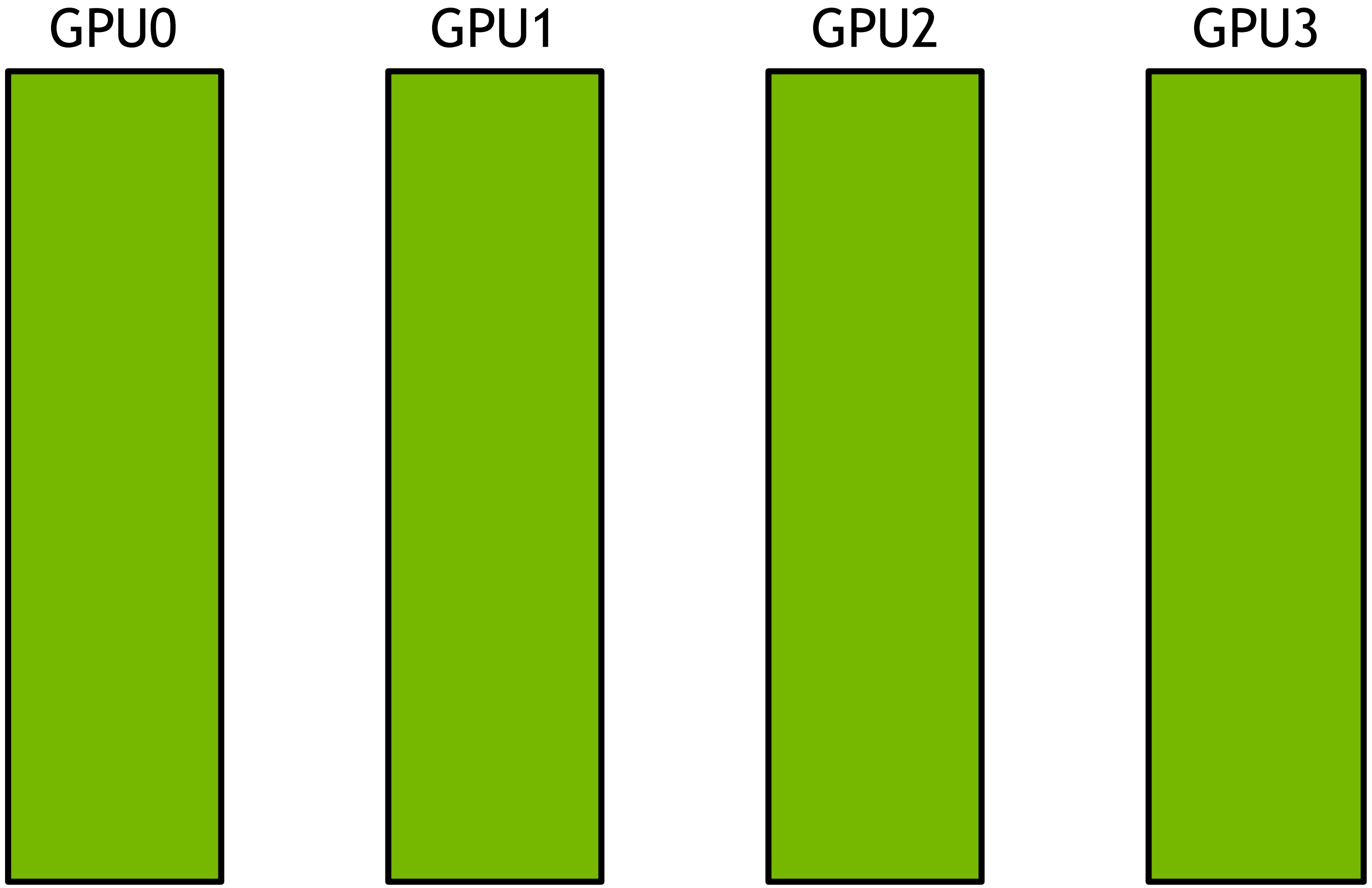
Step 2: $\Delta t = N/(SB)$

Step 3: $\Delta t = N/(SB)$

Step 4: $\Delta t = N/(SB)$

BROADCAST

with unidirectional ring



Split data into S messages

Step 1: $\Delta t = N/(SB)$

Step 2: $\Delta t = N/(SB)$

Step 3: $\Delta t = N/(SB)$

Step 4: $\Delta t = N/(SB)$

...

Total time:
$SN/(SB) + (k-2)\,N/(SB)$
$= N(S+k-2)/(SB) \rightarrow N/B$

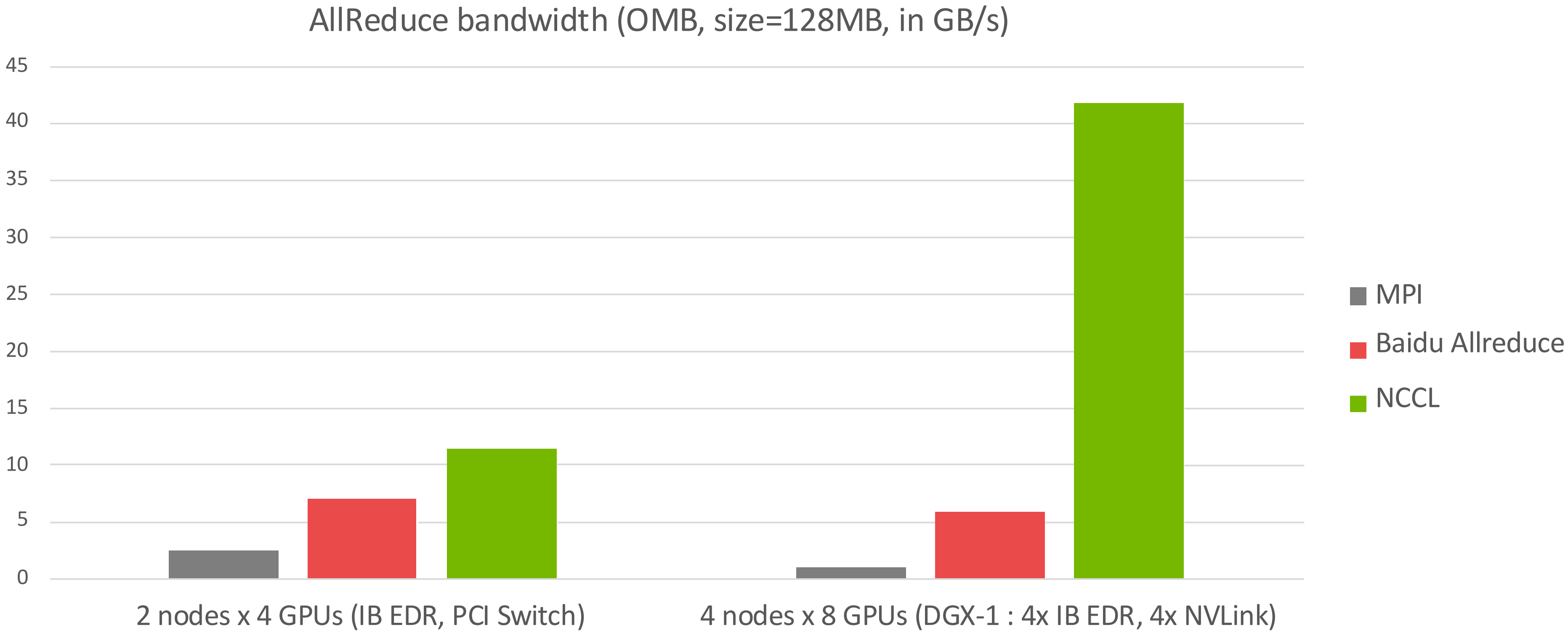
NCCL

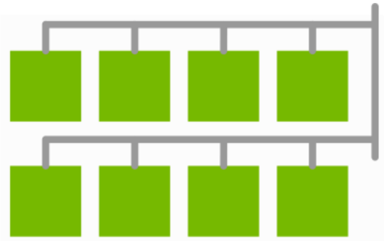
- Automatic topology detection.
  - Better utilization of NVLink topology

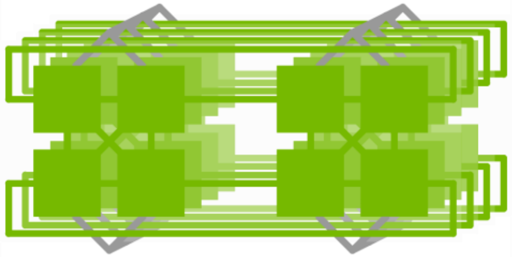
- Automatic algorithm selection and optimization.
  - Ring, Multi-ring.
  - Tree.

PERFORMANCE

Inter-node performance



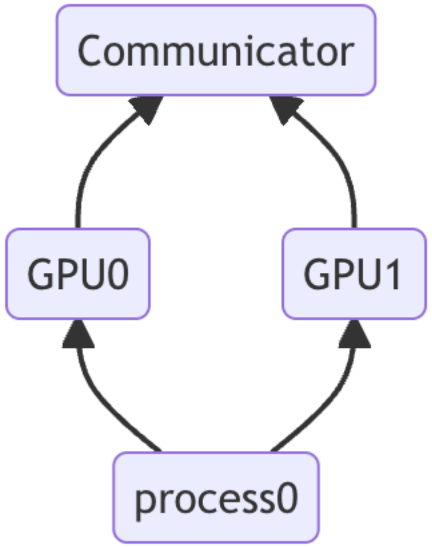


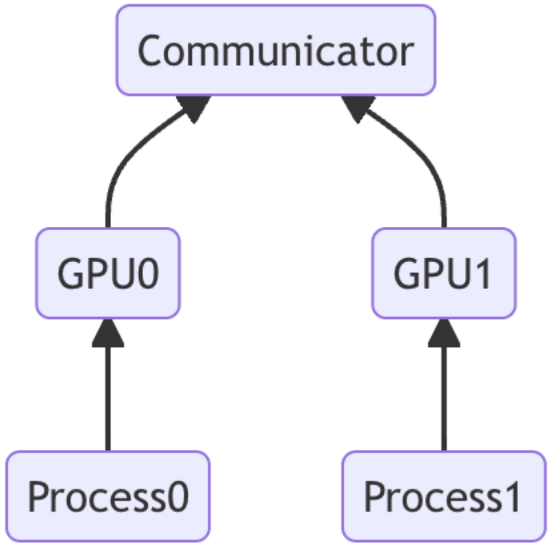


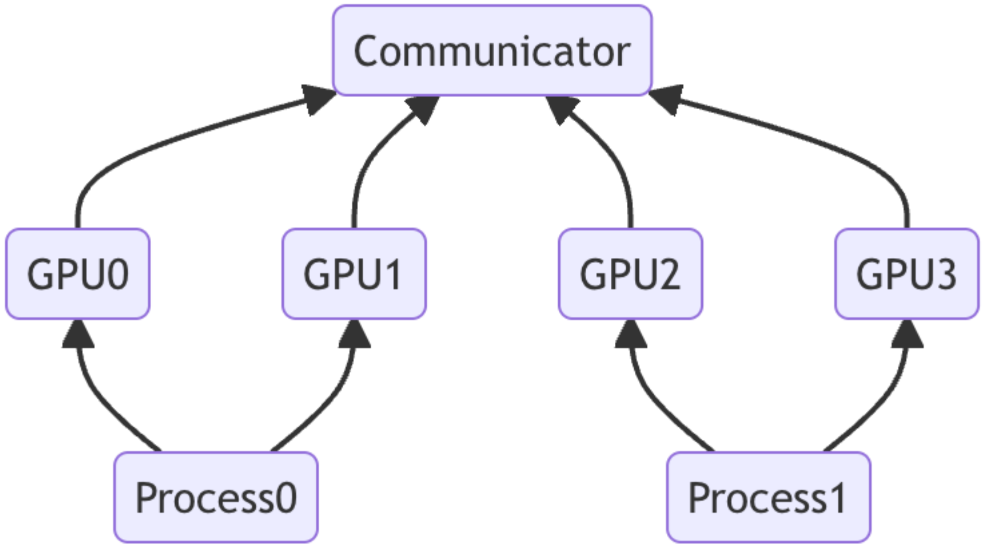
How to use NCCL?

Common Cases

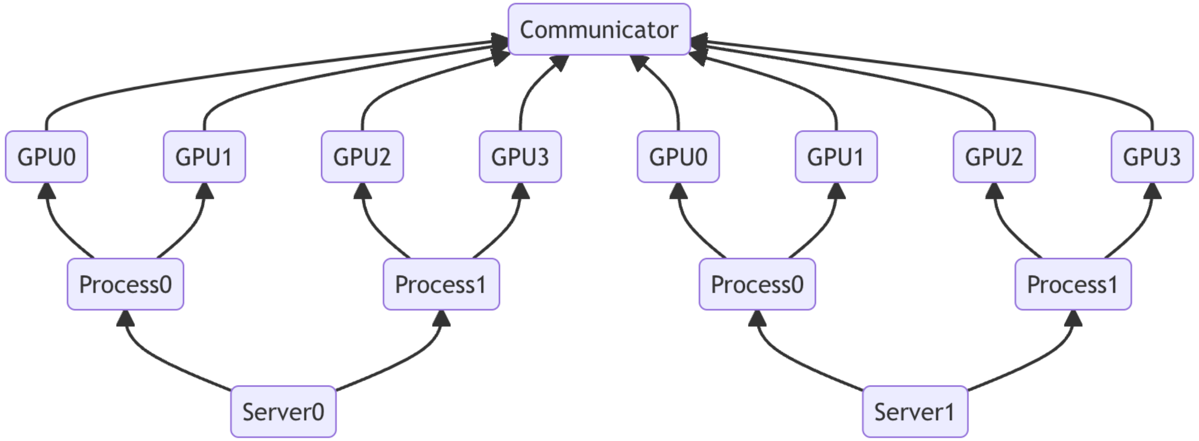
1. Single process/thread multiple devices.
2. One Device per process/thread.
3. Multiple devices per process/thread.

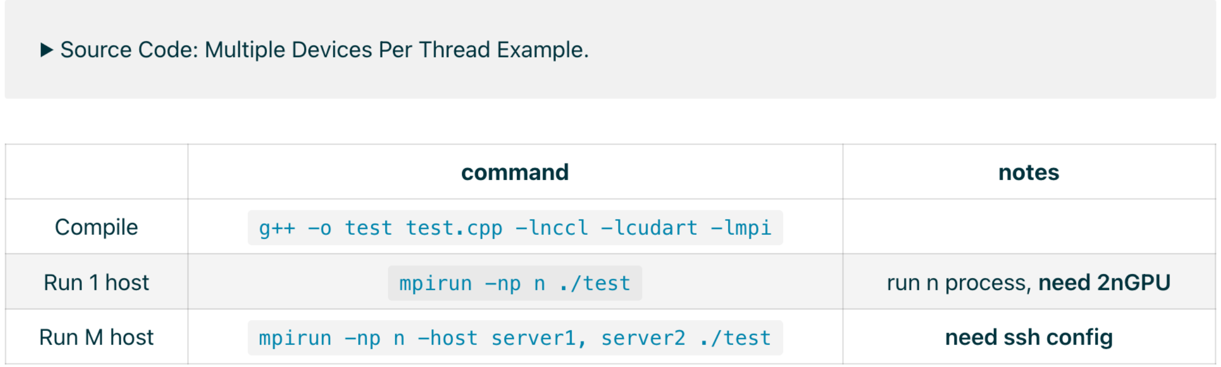






M servers, M process each with M devices

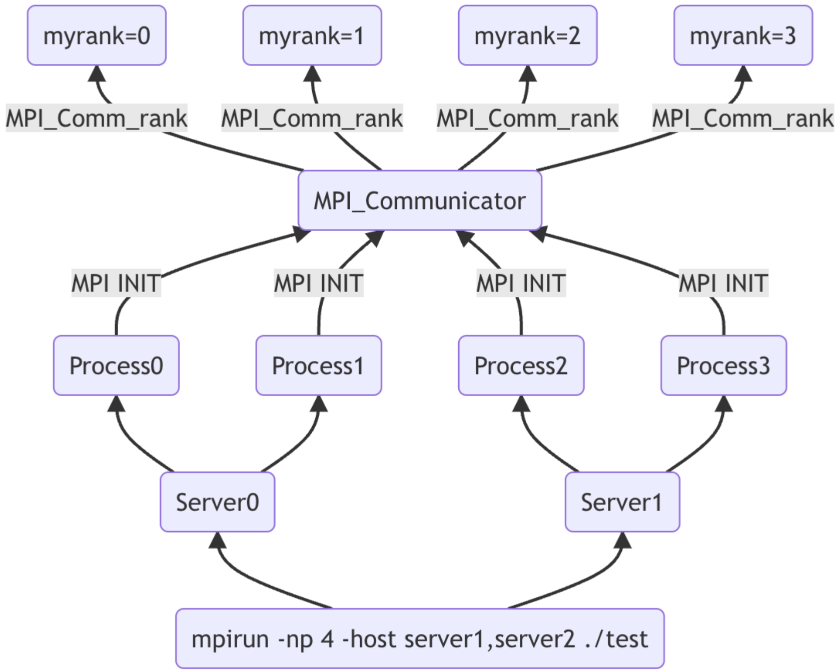


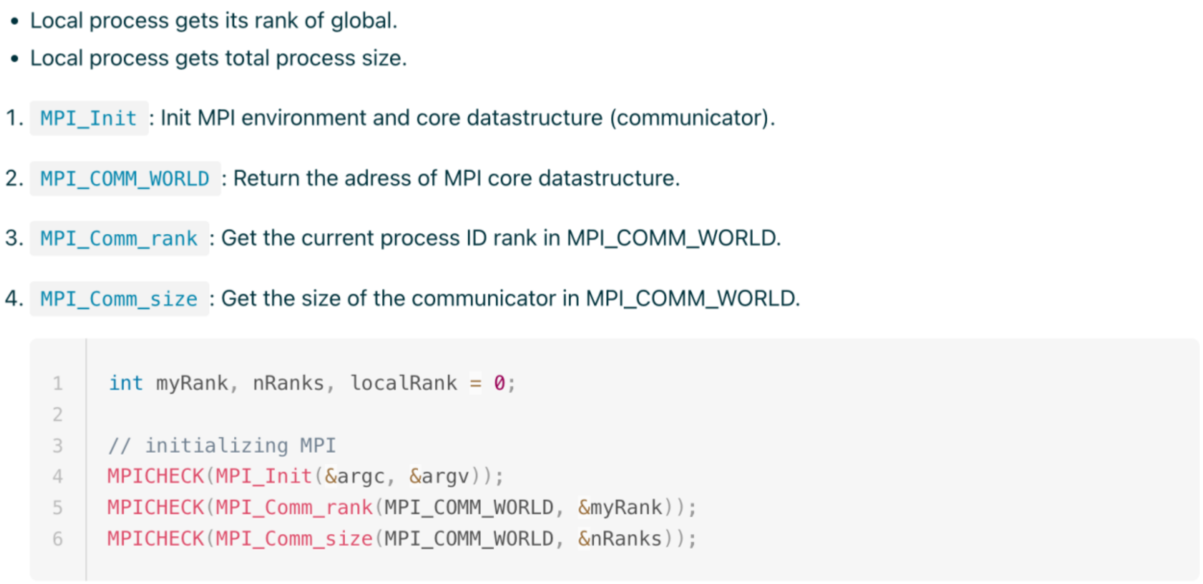


▶ Source Code: Multiple Devices Per Thread Example.

	command	notes
Compile	`g++ -o test test.cpp -lnccl -lcudart -lmpi`	
Run 1 host	`mpirun -np n ./test`	run n process, **need 2nGPU**
Run M host	`mpirun -np n -host server1, server2 ./test`	**need ssh config**

Step1: Initialize MPI



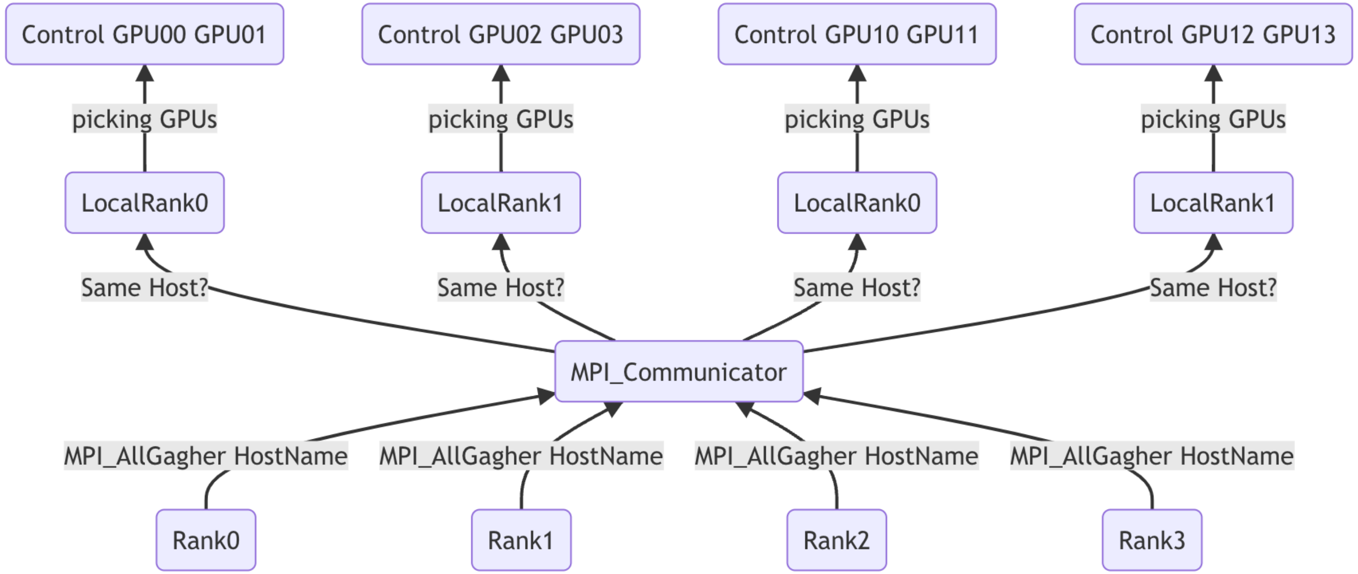


- Local process gets its rank of global.
- Local process gets total process size.

1. `MPI_Init` : Init MPI environment and core datastructure (communicator).
2. `MPI_COMM_WORLD` : Return the adress of MPI core datastructure.
3. `MPI_Comm_rank` : Get the current process ID rank in MPI_COMM_WORLD.
4. `MPI_Comm_size` : Get the size of the communicator in MPI_COMM_WORLD.

```
int myRank, nRanks, localRank = 0;

// initializing MPI
MPICHECK(MPI_Init(&argc, &argv));
MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));
```

Step2: Get Local Rank



Control GPU00 GPU01
Control GPU02 GPU03
Control GPU10 GPU11
Control GPU12 GPU13

picking GPUs
picking GPUs
picking GPUs
picking GPUs

LocalRank0
LocalRank1
LocalRank0
LocalRank1

Same Host?
Same Host?
Same Host?
Same Host?

MPI_Communicator

MPI_AllGagher HostName
MPI_AllGagher HostName
MPI_AllGagher HostName
MPI_AllGagher HostName

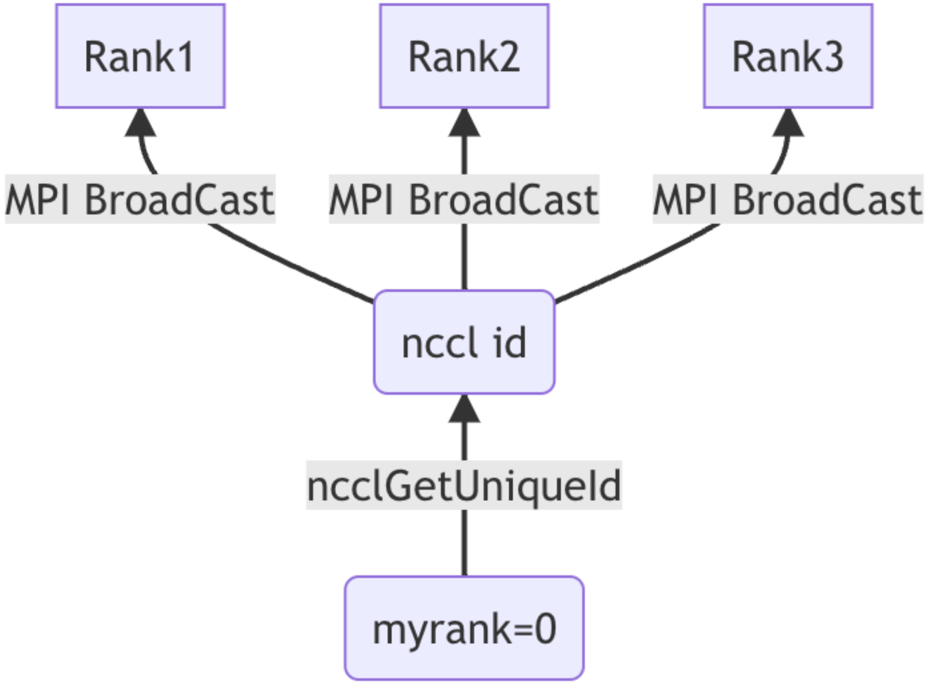
Rank0
Rank1
Rank2
Rank3

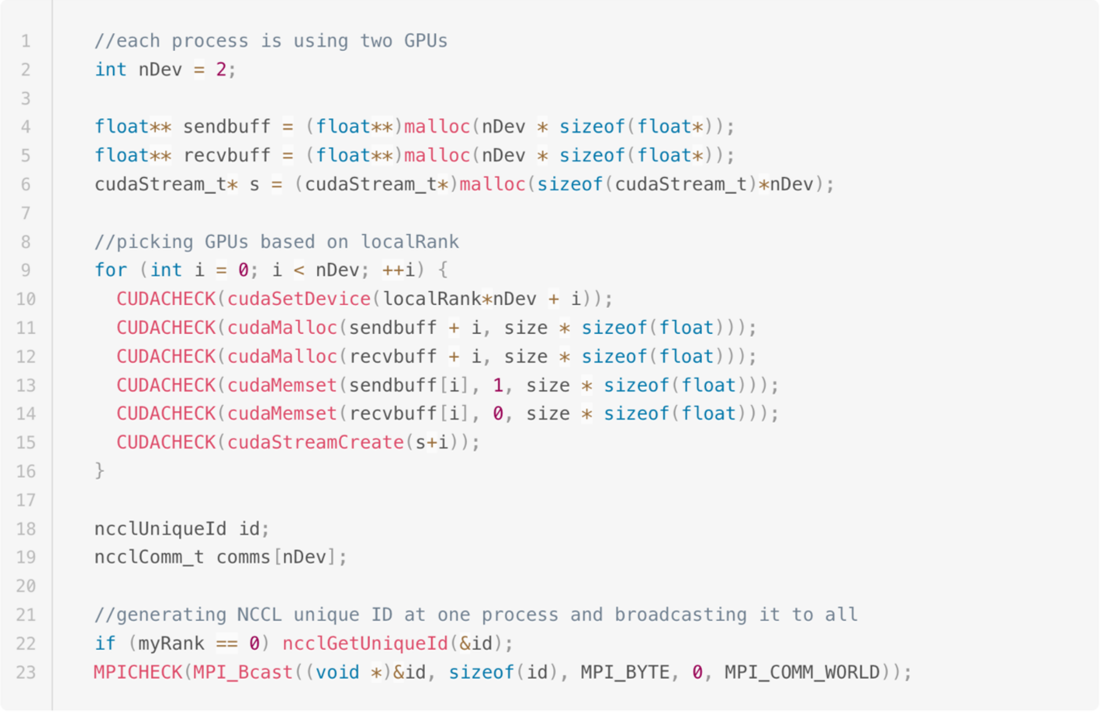
- Local process uses MPI All_gather to get global hostHashs.
- Local process gets localRank of the same host.

1. `MPI_Allgather` : Gather data from all processes.
2. `MPI_IN_PLACE` : Use input buffer as output buffer.
3. `MPI_DATATYPE_NULL` : No datatype specified.
4. `MPI_BYTE` : Byte-sized data type.

```
int size = 32*1024*1024;

int myRank, nRanks, localRank = 0;

// calculating localRank based on hostname which is used in selecting a GPU
uint64_t hostHashs[nRanks];
char hostname[1024];
getHostName(hostname, 1024);
hostHashs[myRank] = getHostHash(hostname);
MPICHECK(MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, hostHashs,
                       sizeof(uint64_t), MPI_BYTE, MPI_COMM_WORLD));
for (int p = 0; p < nRanks; p++) {
    if (p == myRank) break;
    if (hostHashs[p] == hostHashs[myRank]) localRank++;
}
```

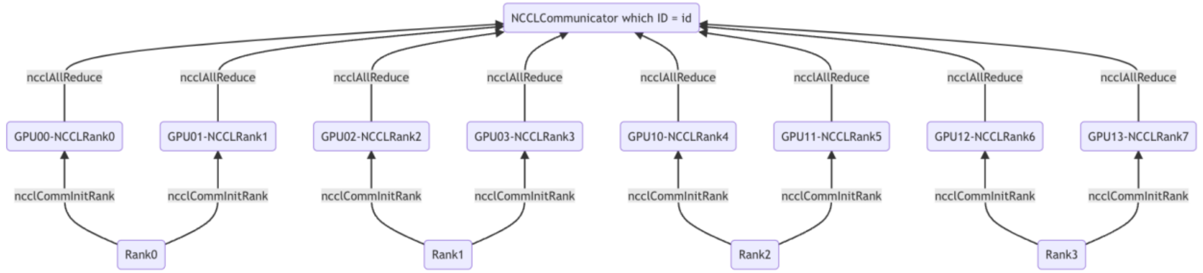
Step3: Broadcast NCCL ID

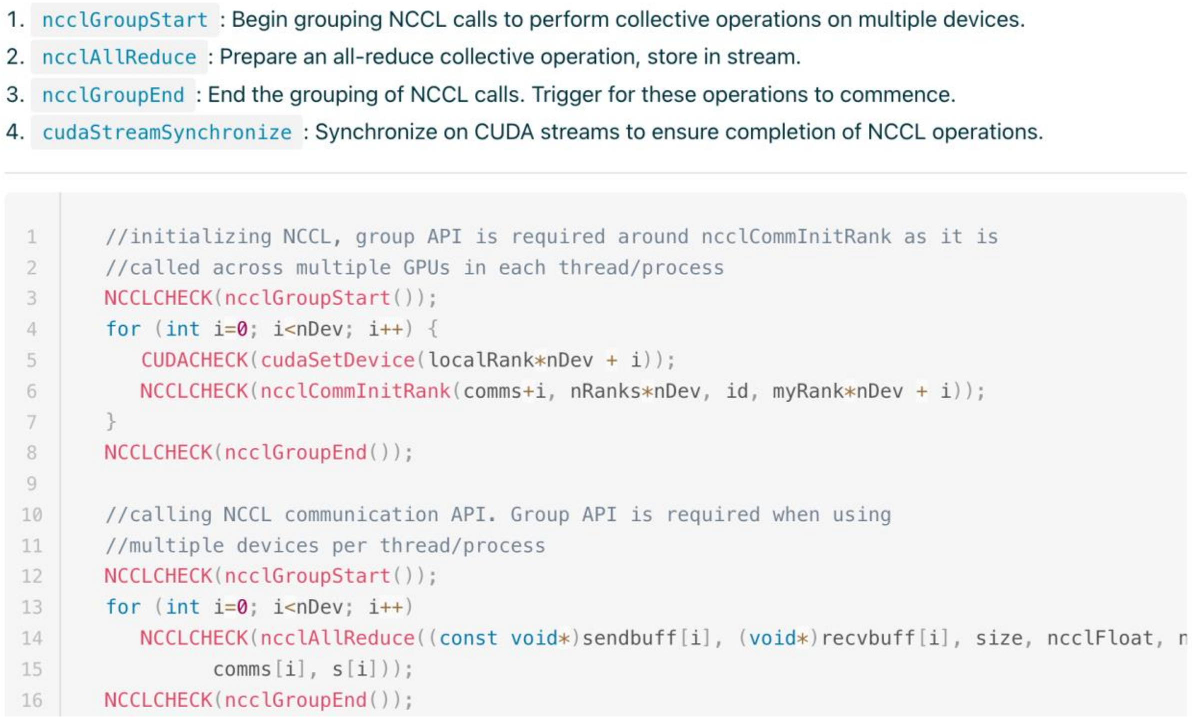




```
//each process is using two GPUs
int nDev = 2;

float** sendbuff = (float**)malloc(nDev * sizeof(float*));
float** recvbuff = (float**)malloc(nDev * sizeof(float*));
cudaStream_t* s = (cudaStream_t*)malloc(sizeof(cudaStream_t)*nDev);

//picking GPUs based on localRank
for (int i = 0; i < nDev; ++i) {
  CUDACHECK(cudaSetDevice(localRank*nDev + i));
  CUDACHECK(cudaMalloc(sendbuff + i, size * sizeof(float)));
  CUDACHECK(cudaMalloc(recvbuff + i, size * sizeof(float)));
  CUDACHECK(cudaMemset(sendbuff[i], 1, size * sizeof(float)));
  CUDACHECK(cudaMemset(recvbuff[i], 0, size * sizeof(float)));
  CUDACHECK(cudaStreamCreate(s+i));
}

ncclUniqueId id;
ncclComm_t comms[nDev];

//generating NCCL unique ID at one process and broadcasting it to all
if (myRank == 0) ncclGetUniqueId(&id);
MPICHECK(MPI_Bcast((void *)&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD));
```

Step4: NCCL Communicate





1. `ncclGroupStart` : Begin grouping NCCL calls to perform collective operations on multiple devices.
2. `ncclAllReduce` : Prepare an all-reduce collective operation, store in stream.
3. `ncclGroupEnd` : End the grouping of NCCL calls. Trigger for these operations to commence.
4. `cudaStreamSynchronize` : Synchronize on CUDA streams to ensure completion of NCCL operations.

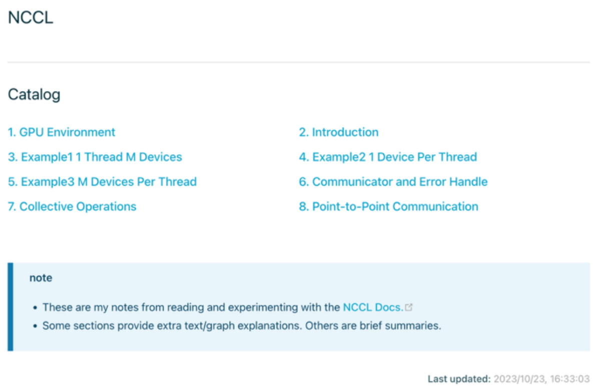
```
//initializing NCCL, group API is required around ncclCommInitRank as it is
//called across multiple GPUs in each thread/process
NCCLCHECK(ncclGroupStart());
for (int i=0; i<nDev; i++) {
   CUDACHECK(cudaSetDevice(localRank*nDev + i));
   NCCLCHECK(ncclCommInitRank(comms+i, nRanks*nDev, id, myRank*nDev + 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, n
         comms[i], s[i]));
NCCLCHECK(ncclGroupEnd());
```

For more information, take a look at my notes! :)

https://notes.wwei.one/nccl/

1. Create ENV with suitable Driver, CUDA, OpenMPI, NCCL and PyTorch version.
2. CloudLab matters and my profile with environment disk image.
3. Summary of NCCL Docs, point-to-point and Collective operations.
4. Analysis of NCCL examples with NCCL and CUDA source code.

TODO: NCCL source code analysis.



NCCL

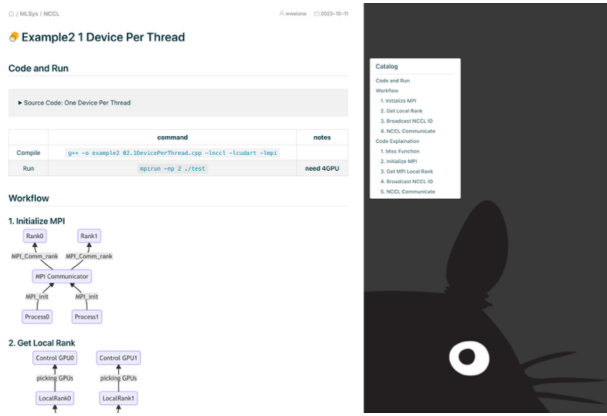
Catalog

1. GPU Environment
2. Introduction
3. Example1 1 Thread M Devices
4. Example2 1 Device Per Thread
5. Example3 M Devices Per Thread
6. Communicator and Error Handle
7. Collective Operations
8. Point-to-Point Communication

note

- These are my notes from reading and experimenting with the NCCL Docs.
- Some sections provide extra text/graph explanations. Others are brief summaries.

Last updated: 2023/10/23, 16:33:03



/ MLSys / NCCL   wweione   2023-10-11

Example2 1 Device Per Thread

Code and Run

▸ Source Code: One Device Per Thread

	command	notes
Compile	`g++ -o example2 02.1DevicePerThread.cpp -lnccl -lcudart -lmpi`	
Run	`mpirun -np 2 ./test`	need 4GPU

Workflow

1. Initialize MPI

2. Get Local Rank

Catalog

Code and Run

Workflow

1. Initialize MPI
2. Get Local Rank
3. Broadcast NCCL ID
4. NCCL Communicate

Code Explaination

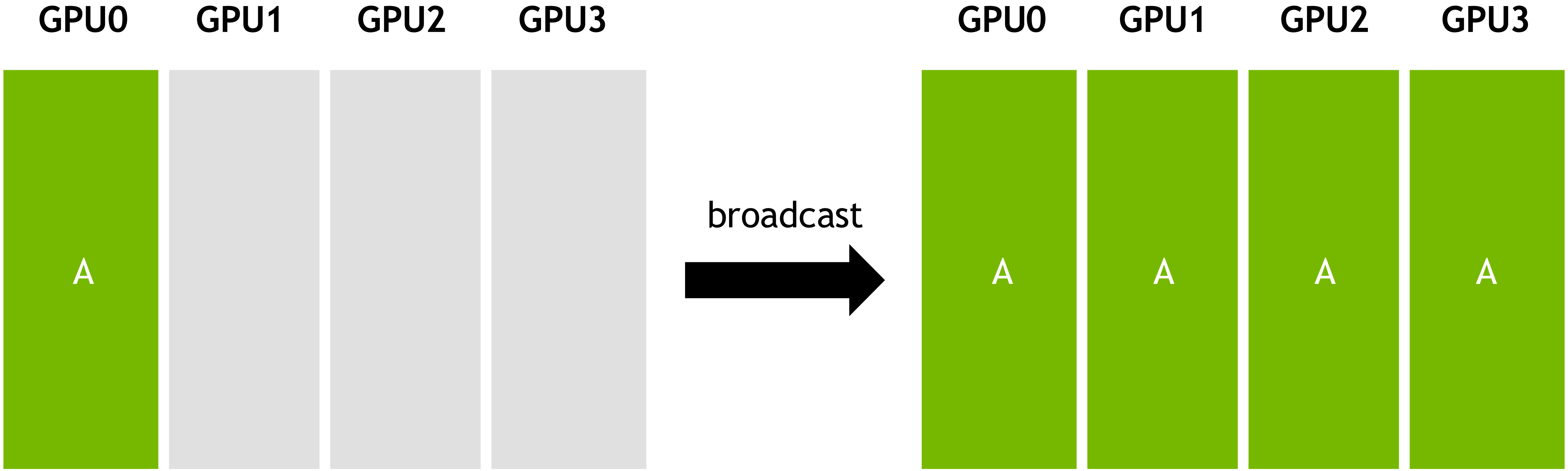
1. Misc Function
2. Initialize MPI
3. Get MPI Local Rank
4. Broadcast NCCL ID
5. NCCL Communicate

Thanks!

Backup

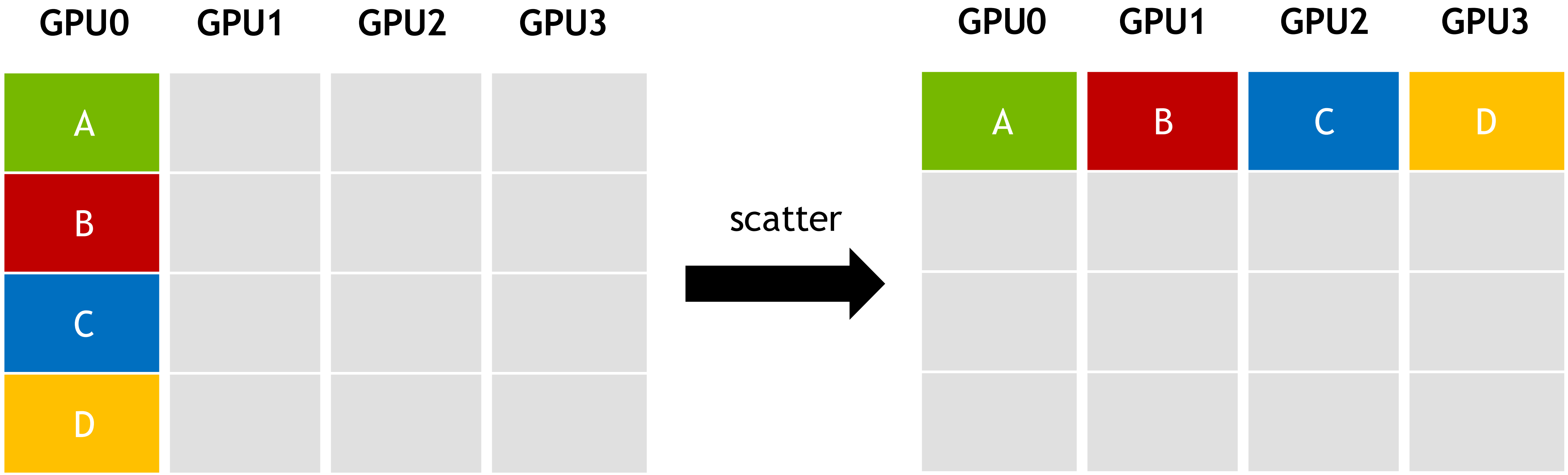
BROADCAST

One sender, multiple receivers



SCATTER

One sender; data is distributed among multiple receivers

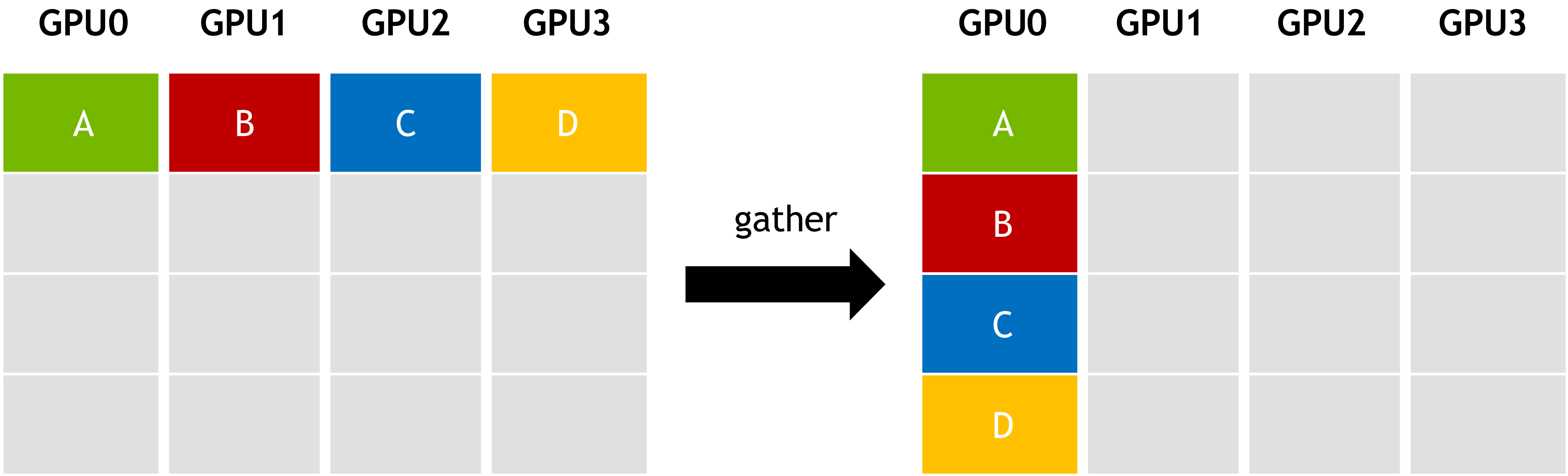


GPU0	GPU1	GPU2	GPU3
A			
B			
C			
D			

GPU0	GPU1	GPU2	GPU3
A	B	C	D

GATHER

Multiple senders, one receiver



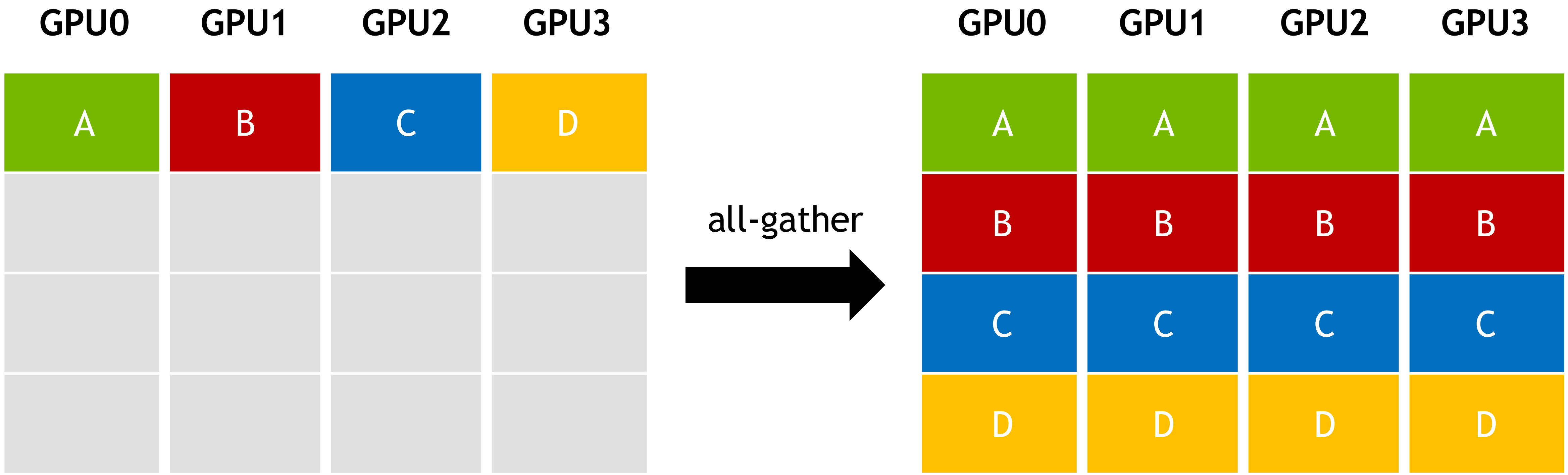
GPU0	GPU1	GPU2	GPU3
A	B	C	D

gather

GPU0	GPU1	GPU2	GPU3
A			
B			
C			
D			

ALL-GATHER

Gather messages from all; deliver gathered data to all participants

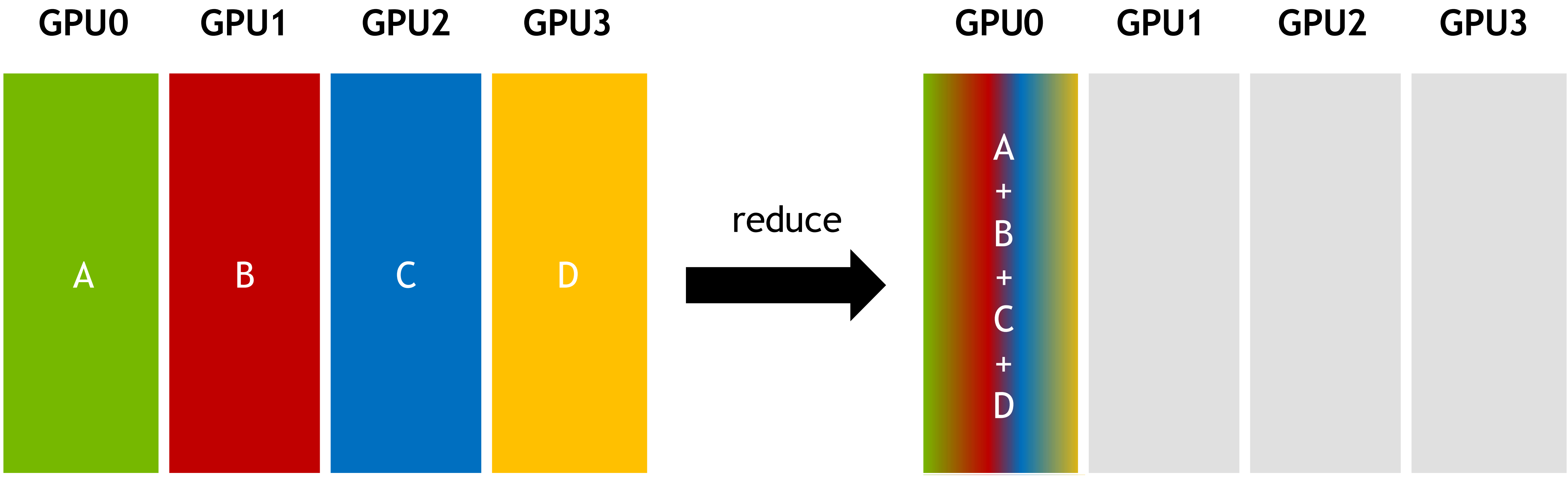


GPU0	GPU1	GPU2	GPU3
A	B	C	D

GPU0	GPU1	GPU2	GPU3
A	A	A	A
B	B	B	B
C	C	C	C
D	D	D	D

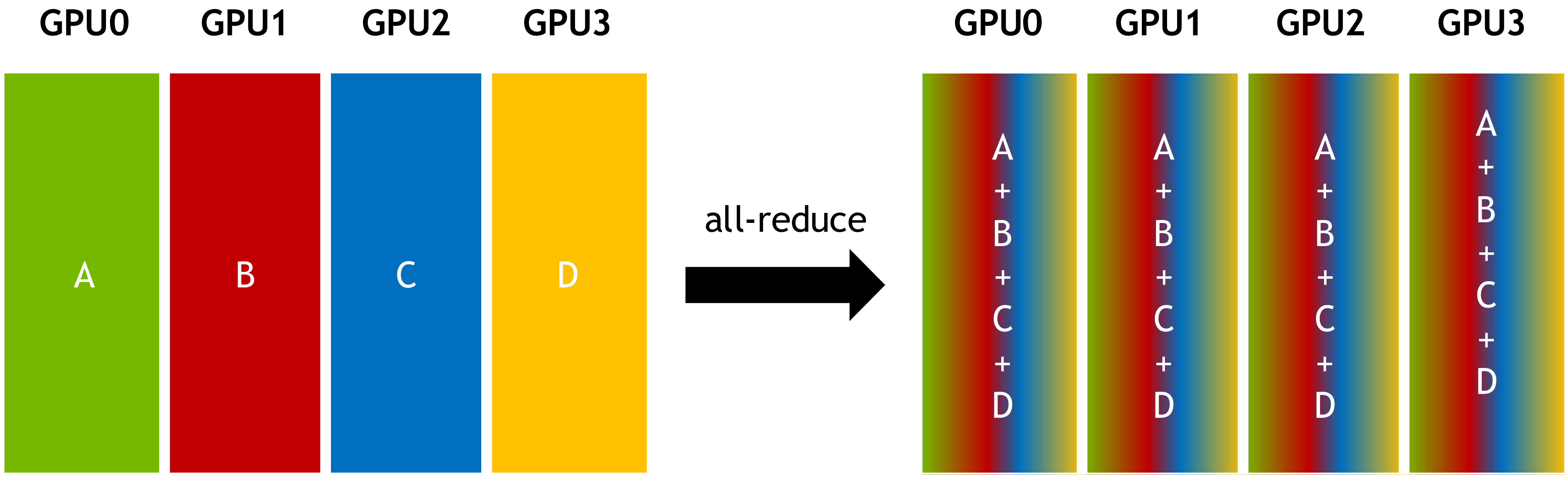
REDUCE

Combine data from all senders; deliver the result to one receiver



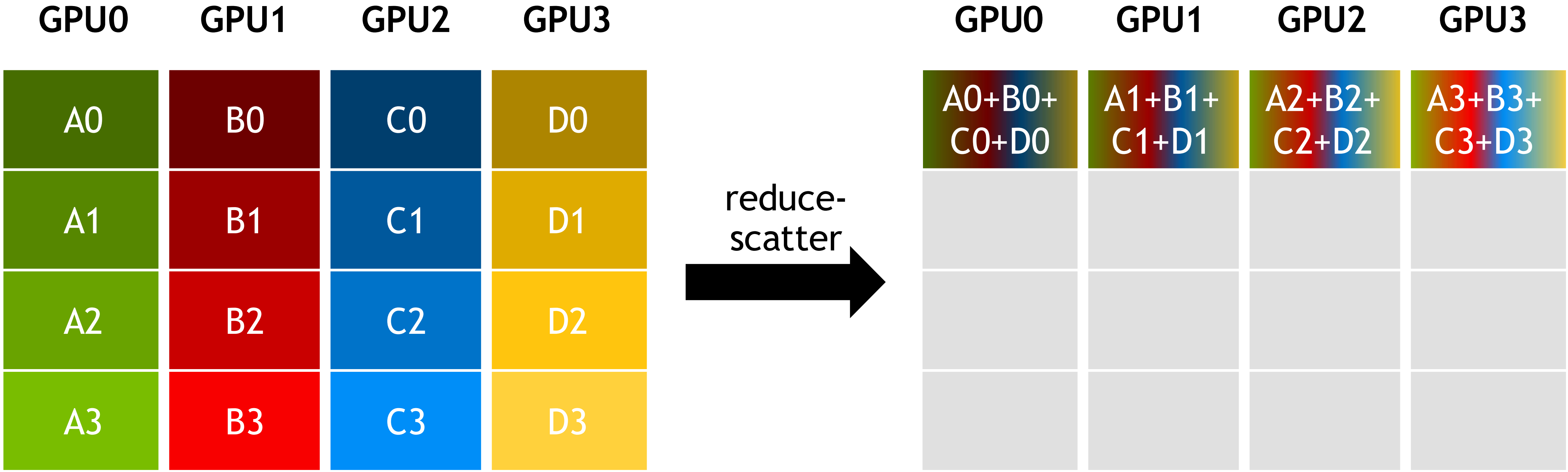
ALL-REDUCE

Combine data from all senders; deliver the result to all participants



REDUCE-SCATTER

Combine data from all senders; distribute result across participants



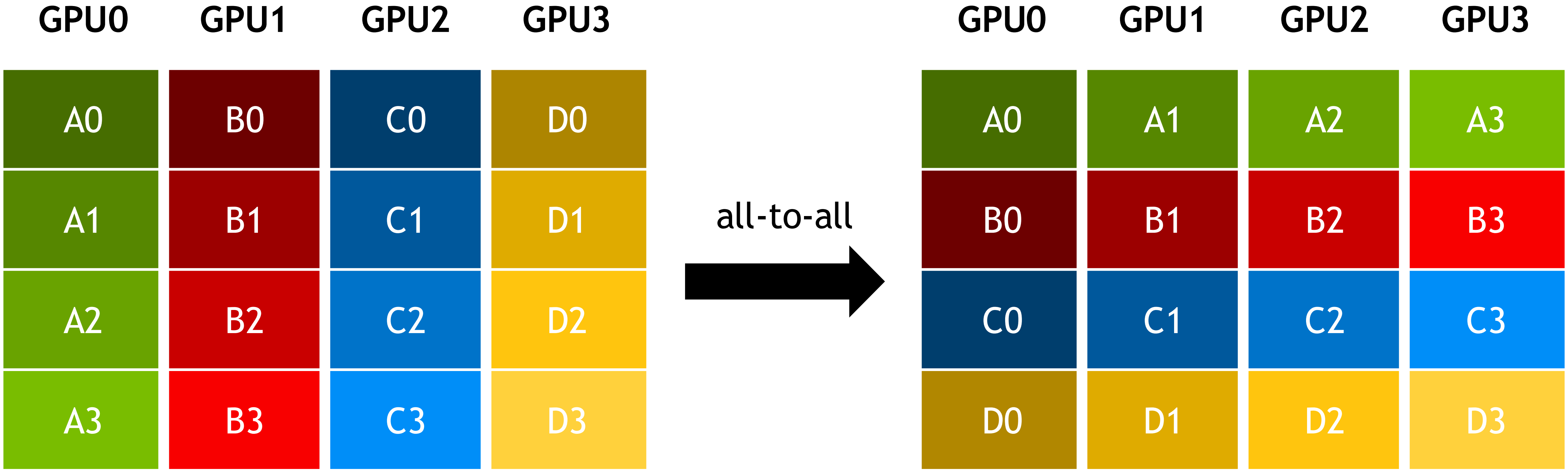
GPU0	GPU1	GPU2	GPU3
A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

reduce-scatter

GPU0	GPU1	GPU2	GPU3
A0+B0+C0+D0	A1+B1+C1+D1	A2+B2+C2+D2	A3+B3+C3+D3

ALL-TO-ALL

Scatter/Gather distinct messages from each participant to every other



GPU0	GPU1	GPU2	GPU3
A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

all-to-all

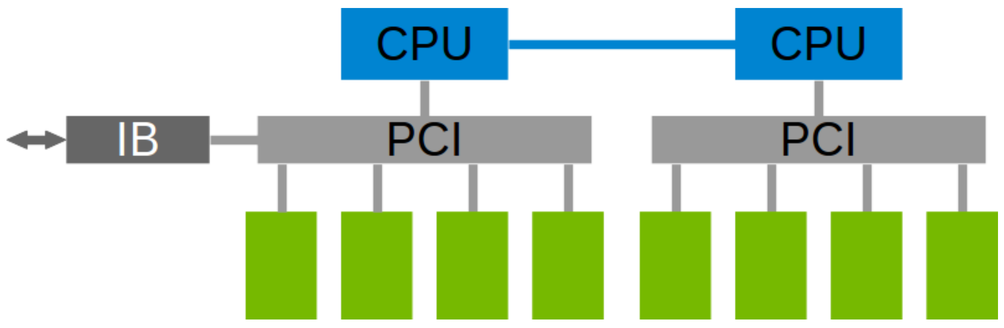
GPU0	GPU1	GPU2	GPU3
A0	A1	A2	A3
B0	B1	B2	B3
C0	C1	C2	C3
D0	D1	D2	D3

THE CHALLENGE OF COLLECTIVES

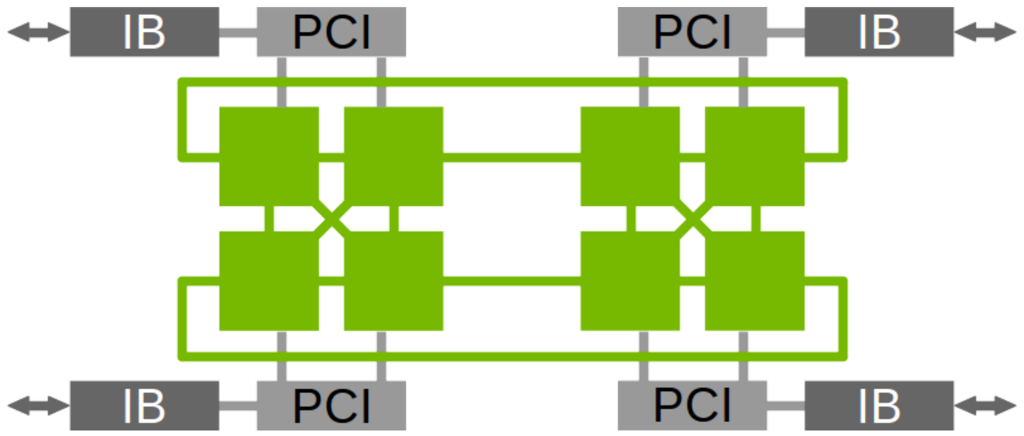
NCCL

Inter-node communication using Sockets or Infiniband verbs, with multi-rail support, topology detection and automatic use of GPU Direct RDMA.

Optimal. combination of NVLink, PCI and network interfaces to maximize bandwidth and create rings across nodes



PCIe, Infiniband



DGX-1 : NVLink, 4x Infiniband

THE CHALLENGE OF COLLECTIVES

Collectives are often avoided because they are expensive. Why?

Having multiple senders and/or receivers compounds communication inefficiencies

- For small transfers, latencies dominate; more participants increase latency
- For large transfers, bandwidth is key; bottlenecks are easily exposed
- May require topology-aware implementation for high performance
- Collectives are often blocking/non-overlapped

THE CHALLENGE OF COLLECTIVES

If collectives are so expensive, do they actually get used? YES!

Collectives are central to scalability in a variety of key applications:

- Deep Learning (All-reduce, broadcast, gather)
- Parallel FFT (Transposition is all-to-all)
- Molecular Dynamics (All-reduce)
- Graph Analytics (All-to-all)
- …

THE CHALLENGE OF COLLECTIVES

Many implementations seen in the wild are suboptimal

Scaling requires efficient communication algorithms and careful implementation

Communication algorithms are topology-dependent

Topologies can be complex – not every system is a fat tree

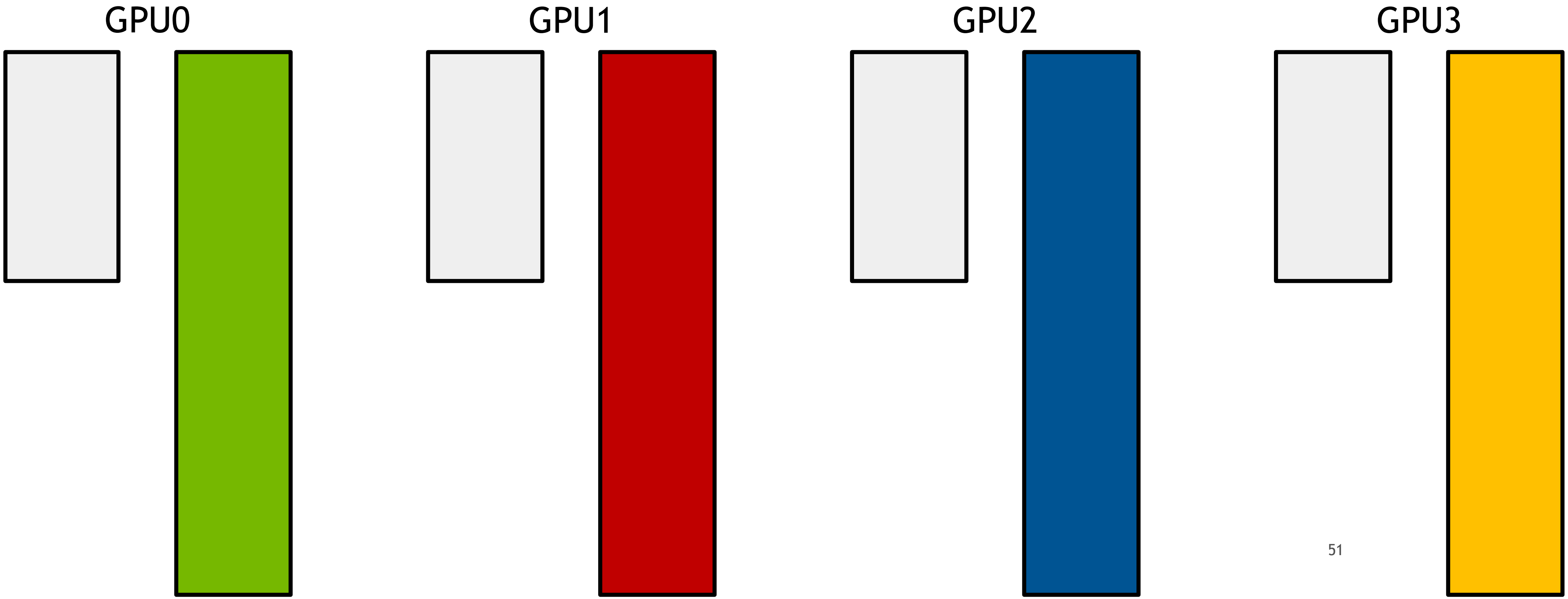
Most collectives amenable to bandwidth-optimal implementation on rings, and many topologies can be interpreted as one or more rings [P. Patarasuk and X. Yuan]

All-reduce

ALL-REDUCE

with unidirectional ring

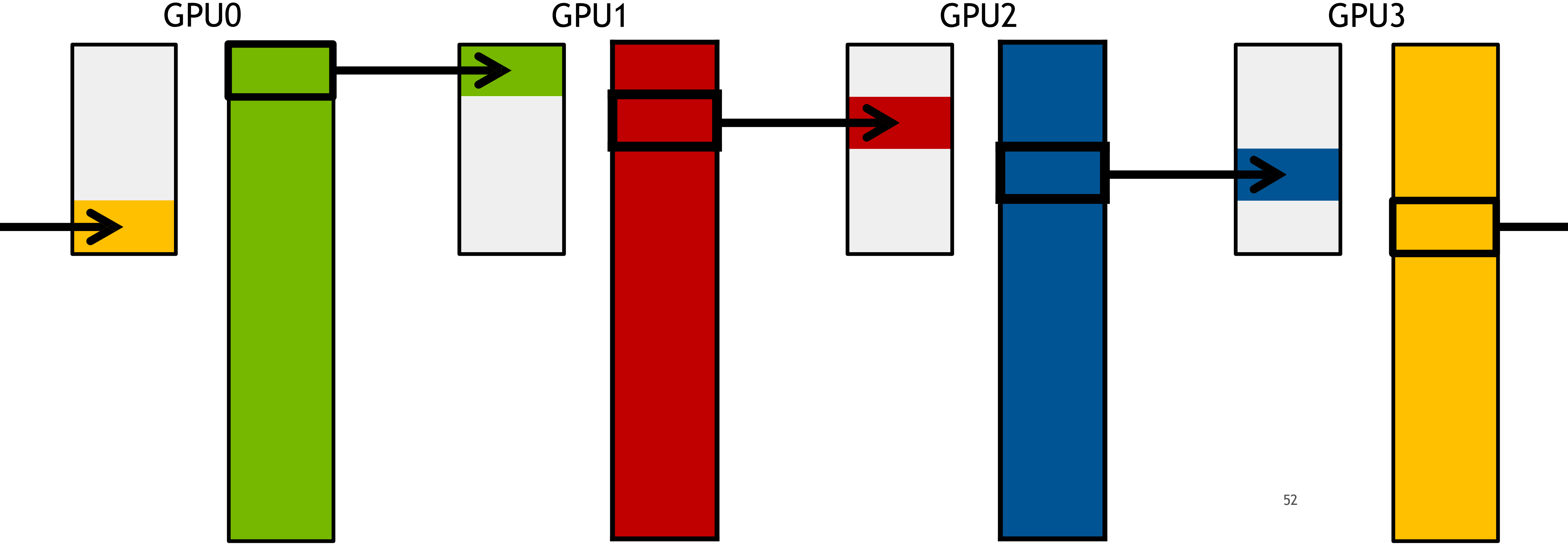
Chunk: 1
Step:



ALL-REDUCE

with unidirectional ring

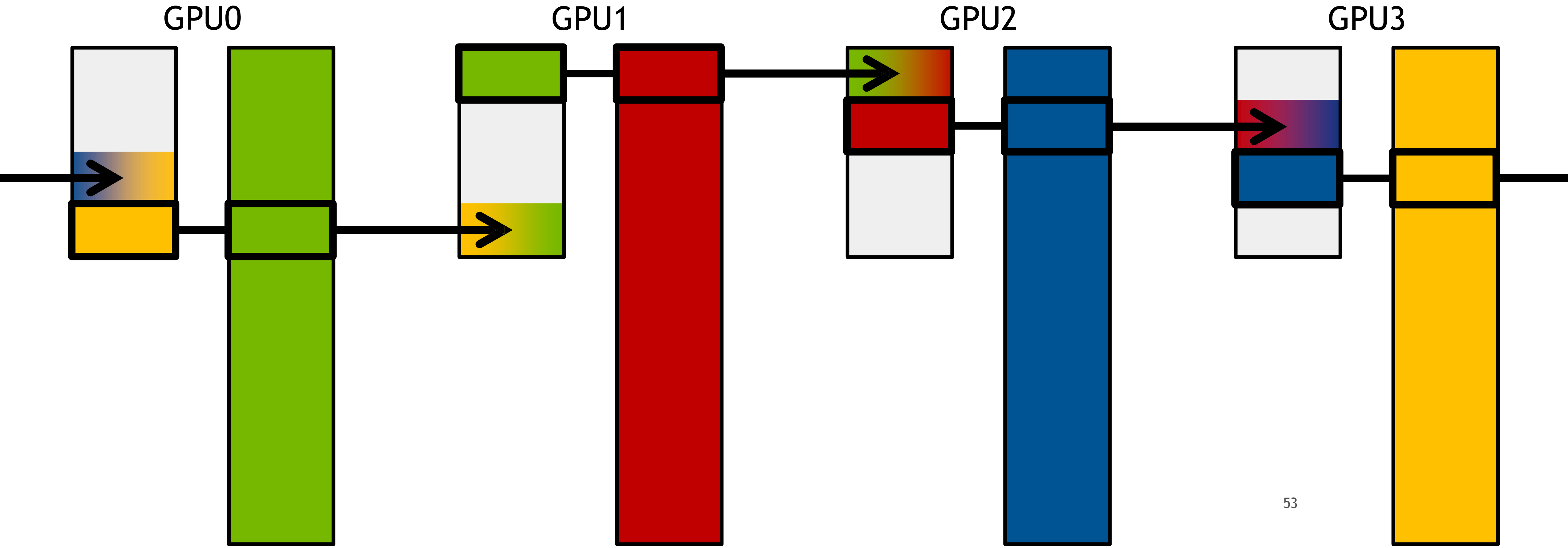
Chunk: 1
Step: 1



ALL-REDUCE

with unidirectional ring

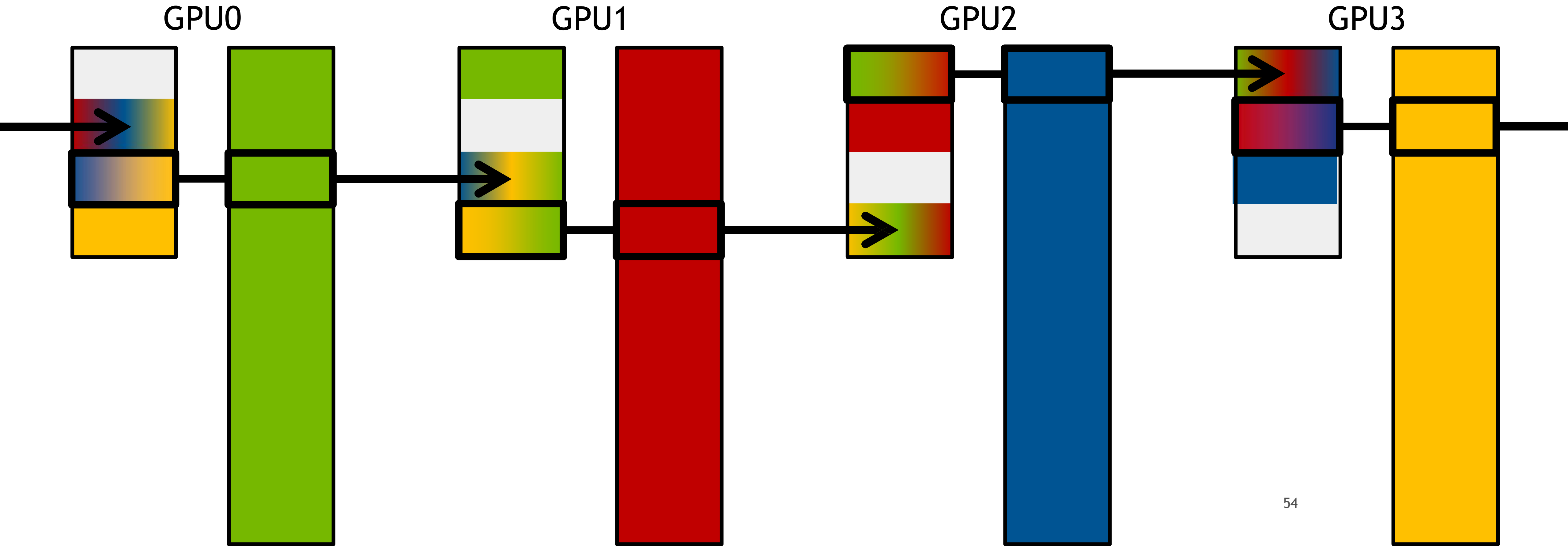
Chunk: 1
Step: 2



ALL-REDUCE

with unidirectional ring

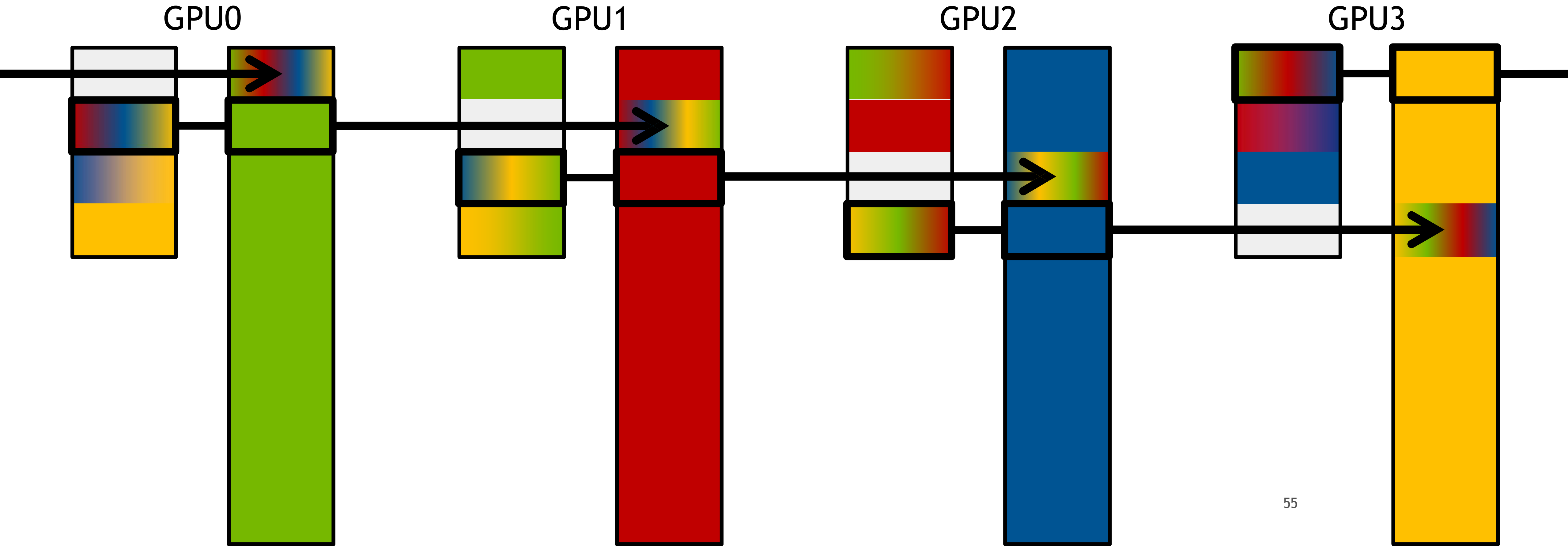
Chunk: 1
Step: 3



ALL-REDUCE

with unidirectional ring

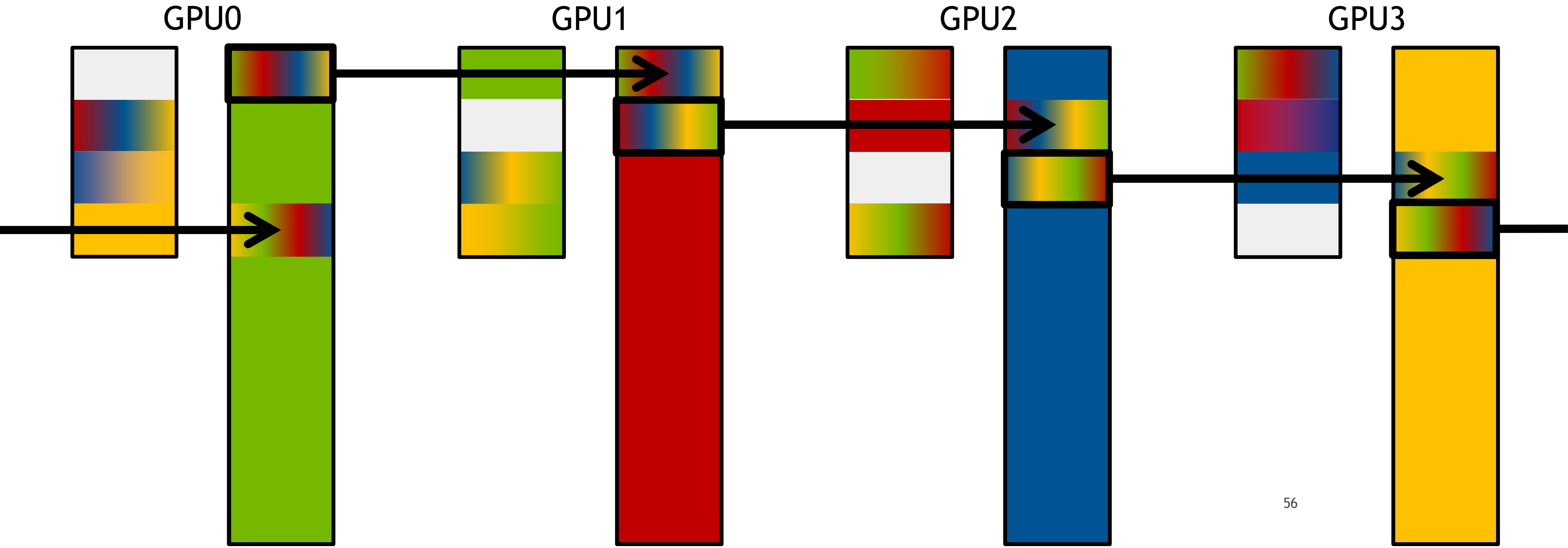
Chunk: 1
Step: 4



ALL-REDUCE

with unidirectional ring

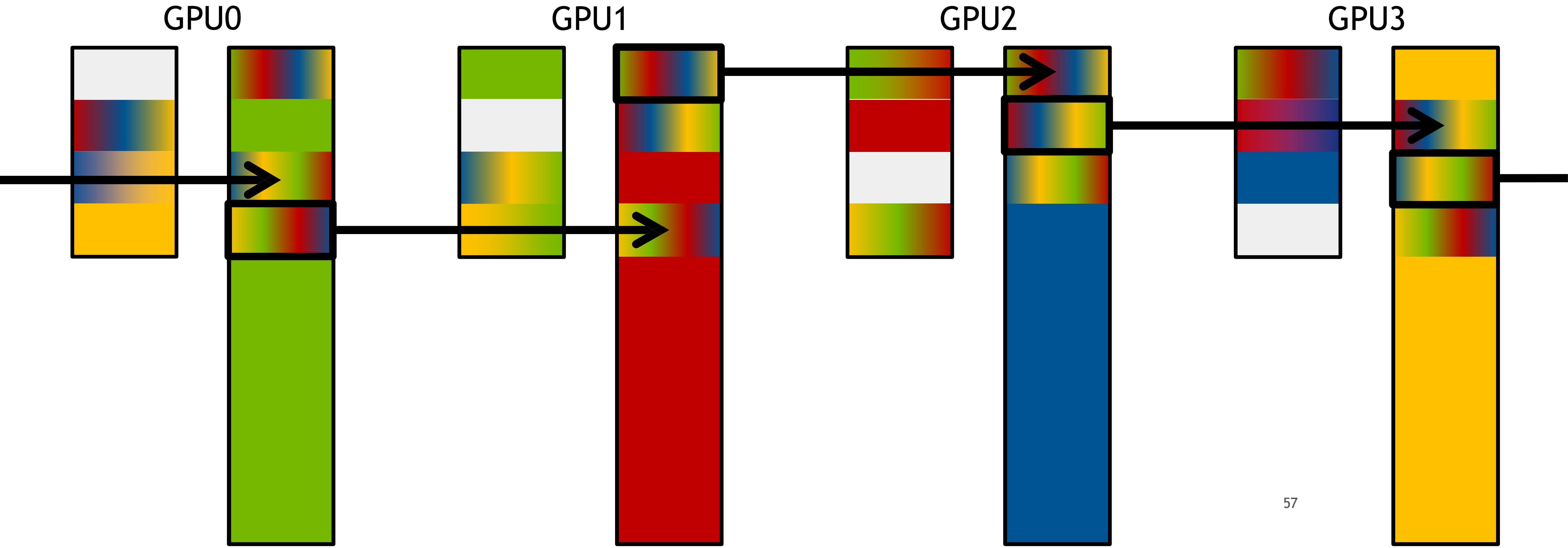
Chunk: 1
Step: 5



ALL-REDUCE

with unidirectional ring

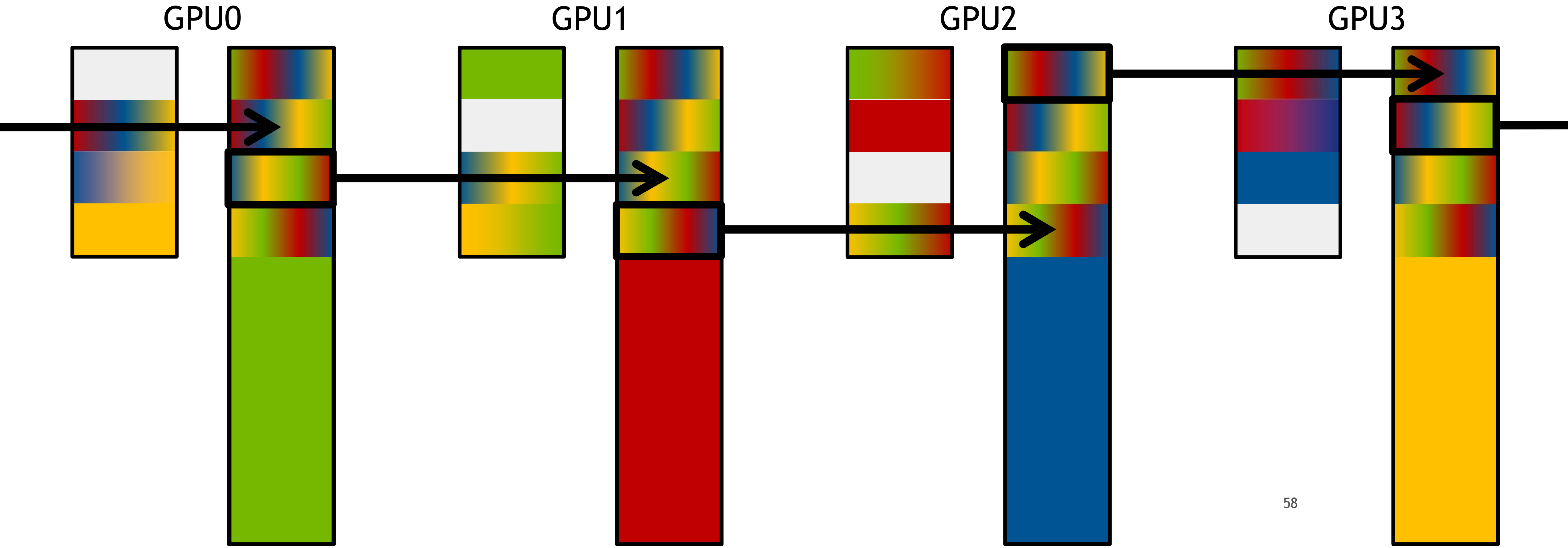
Chunk: 1
Step: 6



ALL-REDUCE

with unidirectional ring

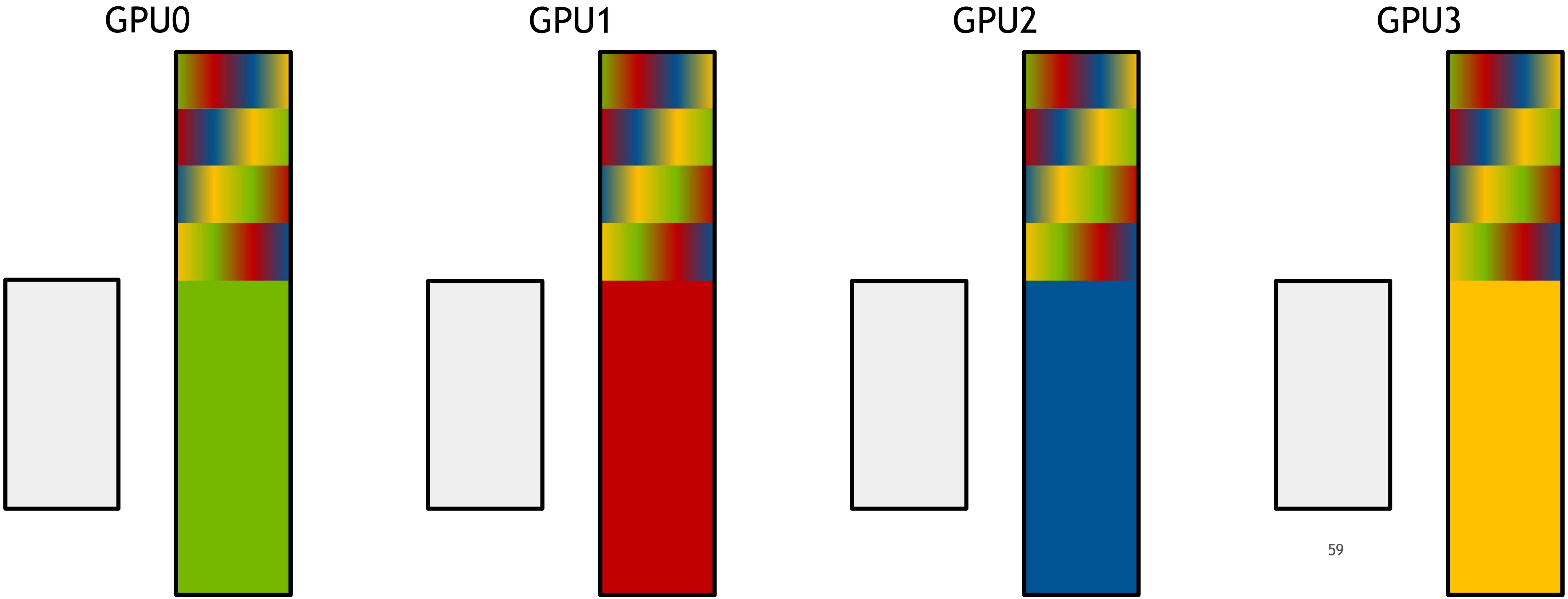
Chunk: 1
Step: 7



ALL-REDUCE

with unidirectional ring

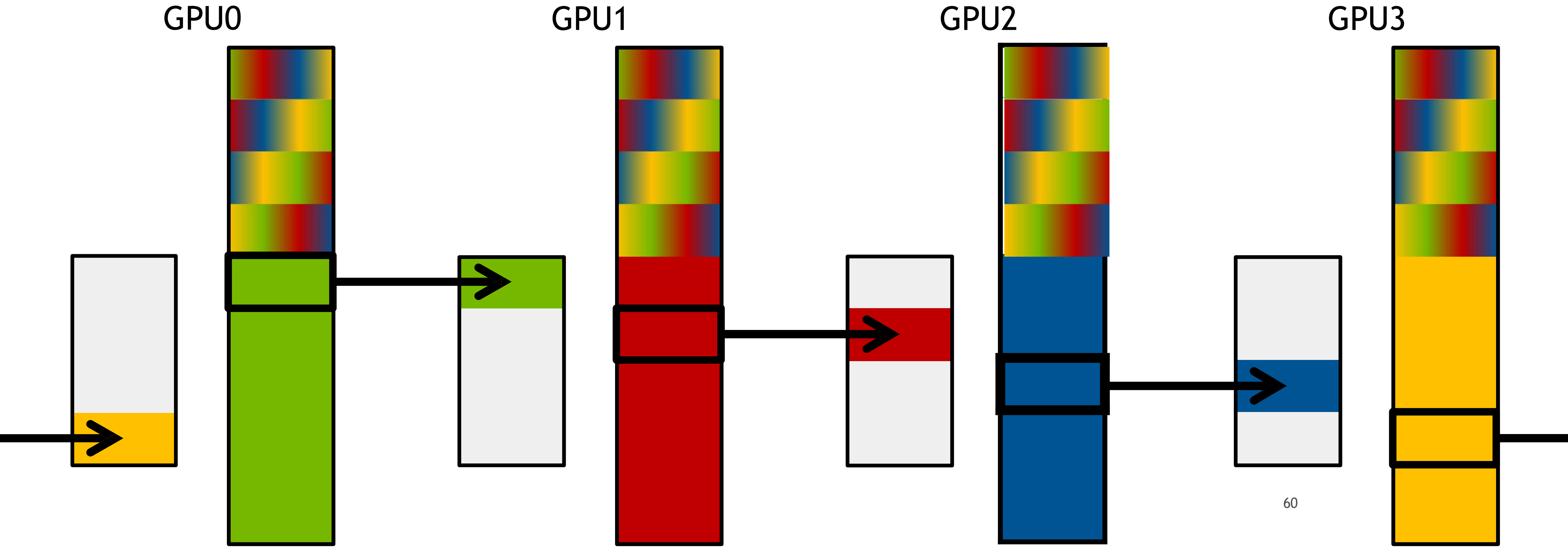
Chunk: 2
Step:



ALL-REDUCE

with unidirectional ring

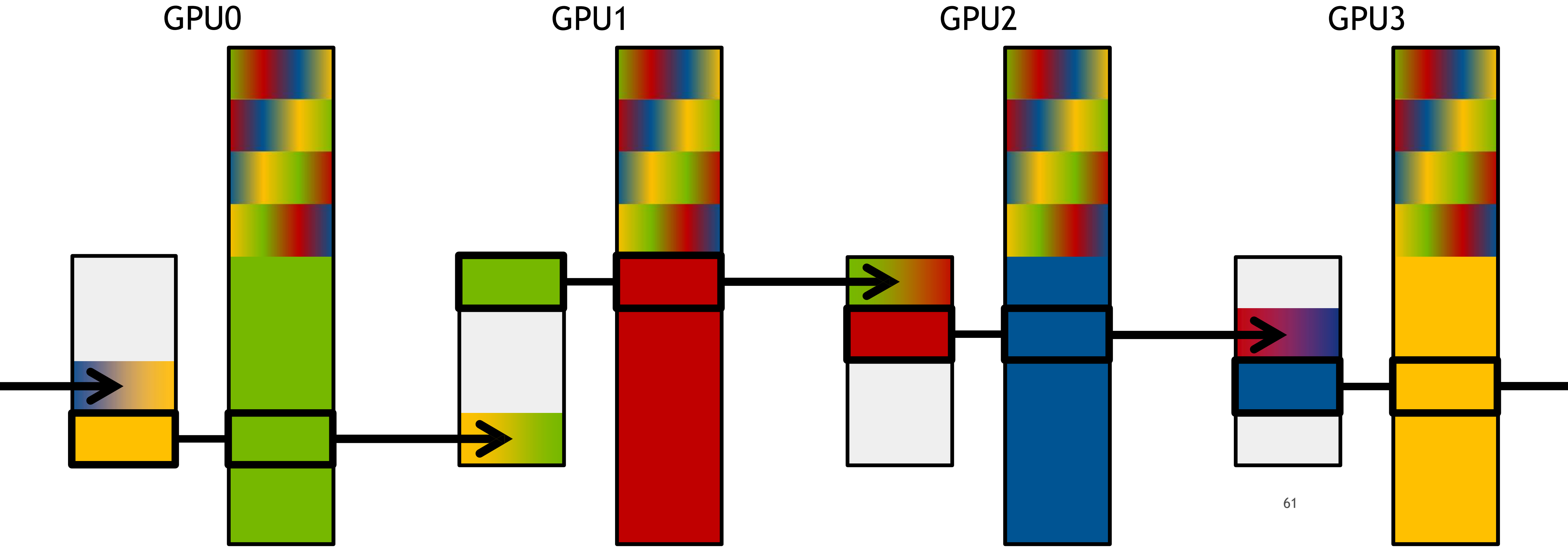
Chunk: 2
Step: 1



ALL-REDUCE

with unidirectional ring

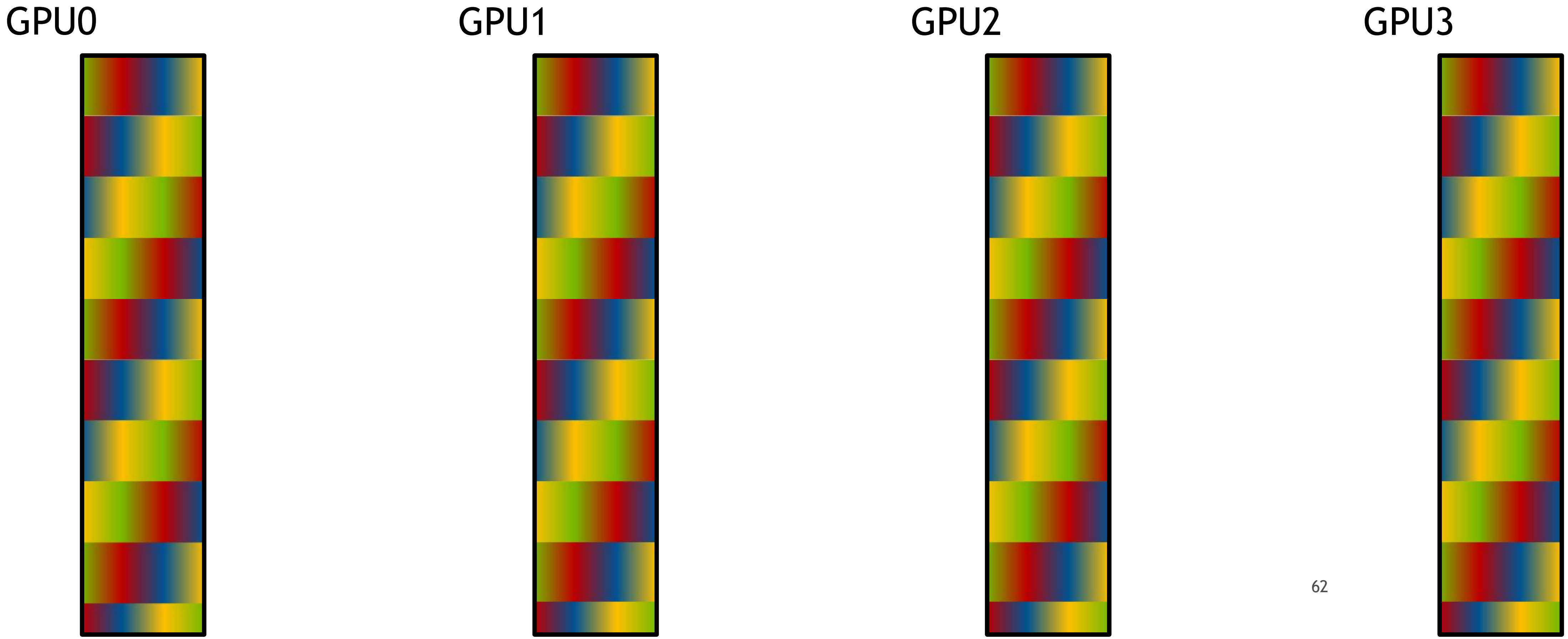
Chunk: 2
Step: 2



ALL-REDUCE

with unidirectional ring

done


GPU0
GPU1
GPU2
GPU3