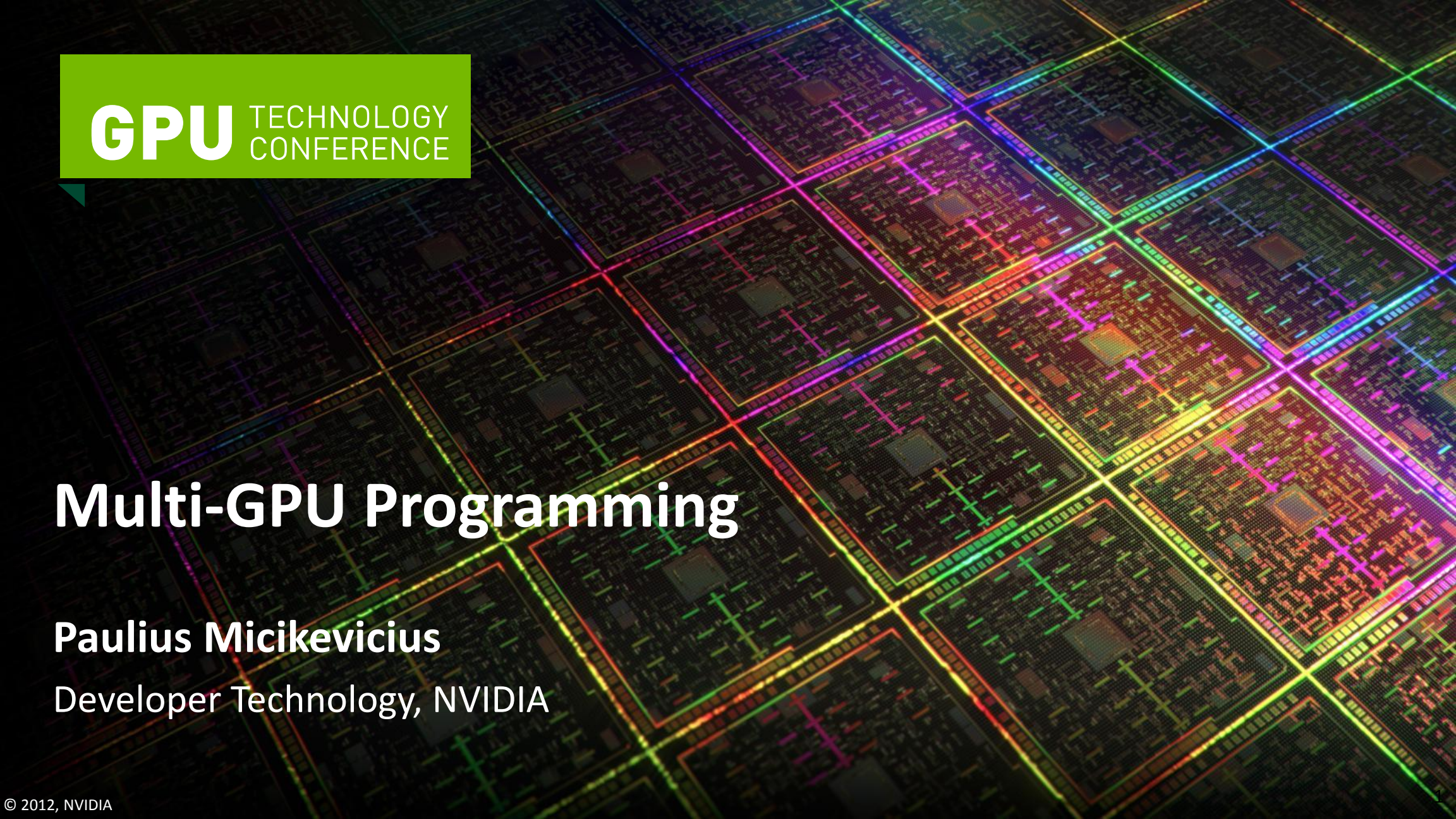


Multi-GPU Programming

Paulius Micikevicius

Developer Technology, NVIDIA

Outline

- **Usecases and a taxonomy of scenarios**
- **Inter-GPU communication:**
  - Single host, multiple GPUs
  - Multiple hosts
- **Case study**
- **Multiple GPUs, streams, and events**
- **Additional APIs:**
  - GPU-aware MPI, cudaIpc*
- **NUMA effect on GPU-CPU communication**

- **Why multi-GPU?**
  - To further speedup computation
  - Working set exceeds a single GPU's memory
  - Having multiple GPUs per node improves perf/W
    - Amortize the CPU server power among more GPUs
    - Same goes for the cost
- **Inter-GPU communication may be needed**
  - Two general cases:
    - GPUs within a single network node
    - GPUs across network nodes

Taxonomy of Inter-GPU Communication Cases

		Network nodes	
		Single	Multiple
Single process	Single-threaded		N/A
	Multi-threaded		N/A
Multiple processes			

GPUs can communicate via P2P or shared host memory

GPUs communicate via host-side message passing

Minimal Review of Streams and Async API

Overlap kernel and memory copy

- **Requirements:**
  - D2H or H2D memcopy from pinned memory
  - Device with compute capability ≥ 1.1 (G84 and later)
  - Kernel and memcopy in different, non-0 streams
- **Code:**

```
cudaStream_t   stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

} **potentially overlapped**

Communication for Single Host, Multiple GPUs

Managing multiple GPUs from a single CPU thread

- **CUDA calls are issued to the *current* GPU**
  - Exception: peer-to-peer memcopies
- **cudaSetDevice() sets the current GPU**
- **Current GPU can be changed while async calls (kernels, memcopies) are running**
  - The following code will have both GPUs executing concurrently:

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```


UVA and Multi-GPU Programming

- **Two interesting aspects:**
  - Peer-to-peer (P2P) memcopies
  - Accessing another GPU's addresses
- **Both require peer-access to be enabled:**
  - cudaDeviceEnablePeerAccess(peer_device, 0)
    - Enables current GPU to access addresses on *peer_device* GPU
  - cudaDeviceCanAccessPeer(&accessible, dev_X, dev_Y)
    - Checks whether dev_X can access memory of dev_Y
    - Returns 0/1 via the first argument
    - Peer-access is not available if:
      - One of the GPUs is pre-Fermi
      - GPUs are connected to different IOH chips on the motherboard
        - » QPI and PCIe protocols disagree on P2P

Peer-to-peer memcopy

- **cudaMemcpyPeerAsync(void* dst_addr, int dst_dev,**
  void* src_addr, int src_dev,
  size_t num_bytes, cudaStream_t stream)
  - Copies the bytes between two devices
  - Currently data is “pushed”: source GPU’s DMA engine carries out the copy
  - There is also a blocking (as opposed to Async) version
- **If peer-access is enabled:**
  - Bytes are transferred along the shortest PCIe path
  - No staging through CPU memory
- **If peer-access is not available**
  - CUDA driver stages the transfer via CPU memory

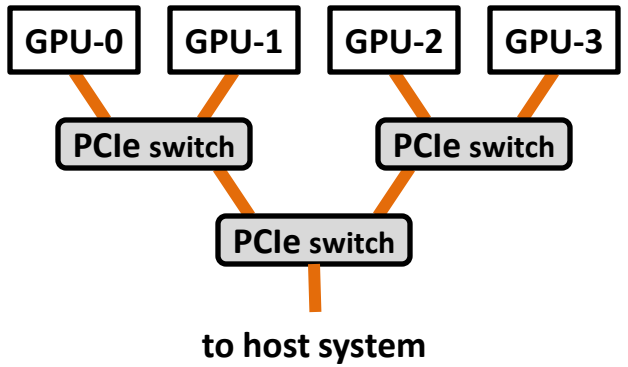
How Does P2P Memcopy Help Multi-GPU?

- **Ease of programming**
  - No need to manually maintain memory buffers on the host for inter-GPU exchanges
- **Increased throughput**
  - Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
    - Single-directional transfers achieve up to ~6.6 GB/s (~12 GB/s for gen3)
    - Duplex transfers achieve ~12.2 GB/s (~22 GB/s for gen3)
      - ~5 GB/s if going through the host
  - GPU-pairs can communicate concurrently if paths don't overlap

Example: 1D Domain Decomposition and P2P

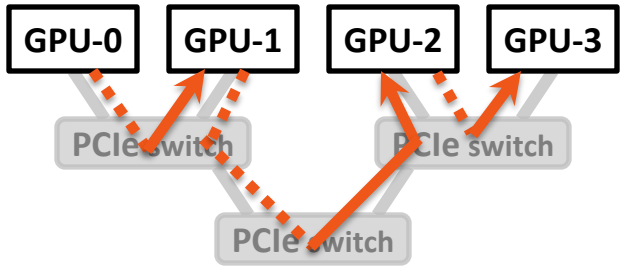
- **Each subdomain has at most two neighbors**
  - "left"/"right"
  - Communication graph = path
- **GPUs are physically arranged into a tree(s)**
  - GPUs can be connected to a PCIe switch
  - PCIe switches can be connected to another switch
- **A path can be efficiently mapped onto a tree**
  - Multiple exchanges can happen without contending for the same PCIe links
  - Aggregate exchange throughput:
    - Approaches (PCIe bandwdith) * (number of GPU pairs)
    - Typical achieved PCIe gen2 simplex bandwidth on a single link: 6 GB/s

Example: 4-GPU Topology

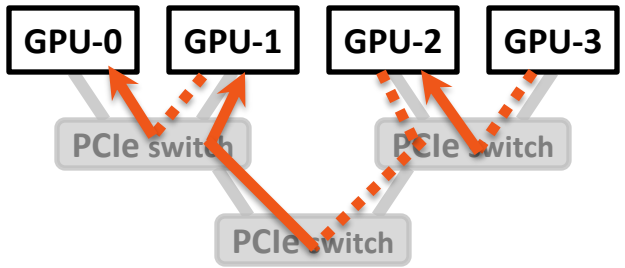


- **Two ways to implement 1D exchange**
  - Left-right approach
  - Pairwise approach
  - Both require two stages

Example: Left-Right Approach for 4 GPUs



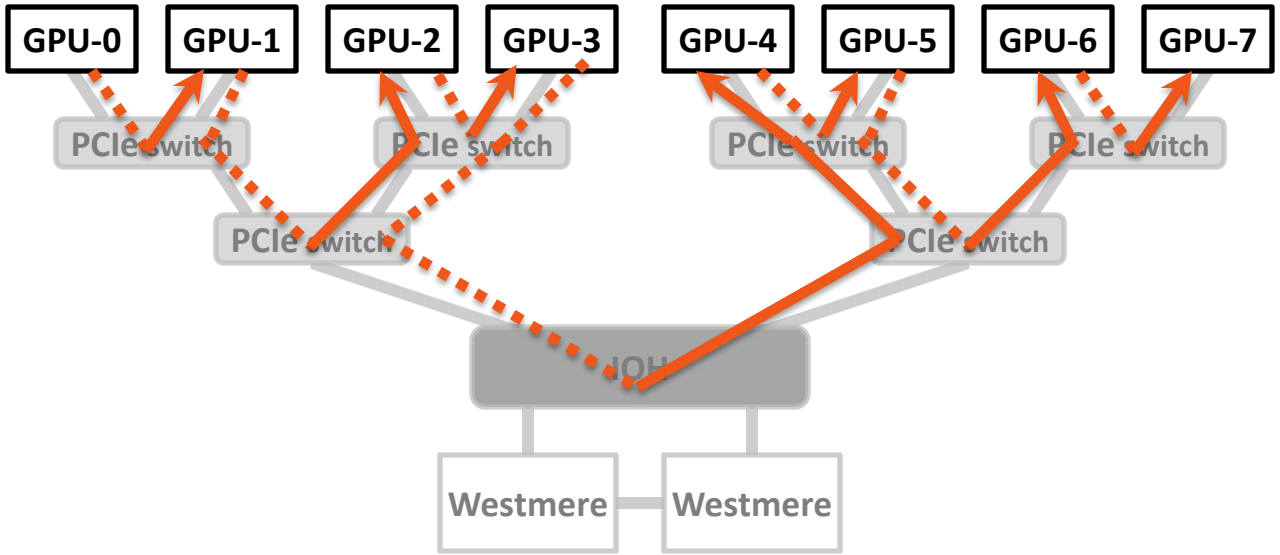
Stage 1: send “right” / receive from “left”



Stage 2: send “left” / receive from “right”

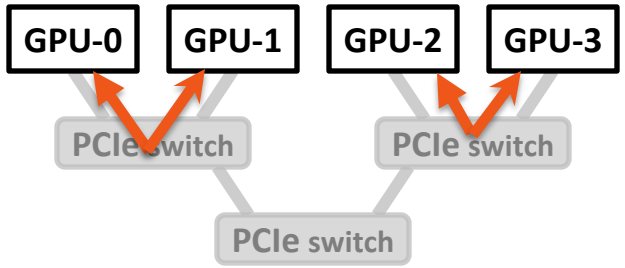
- **The 3 transfers in a stage happen concurrently**
  - Achieved throughput: ~15 GB/s (4-MB messages)
- **No contention for PCIe links**
  - PCIe links are duplex
  - Note that no link has 2 communications in the same “direction”

Example: Left-Right Approach for 8 GPUs

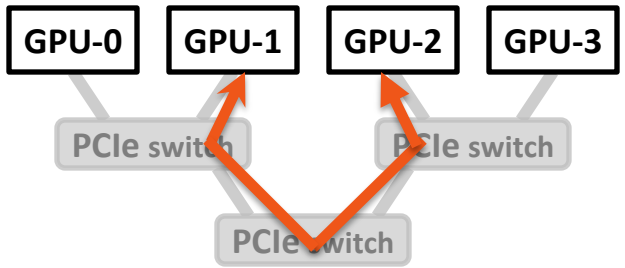


- Stage 1 shown above (Stage 2 is basically the same)
- Achieved aggregate throughput: ~34 GB/s

Example: Pairwise Approach for 4 GPUs



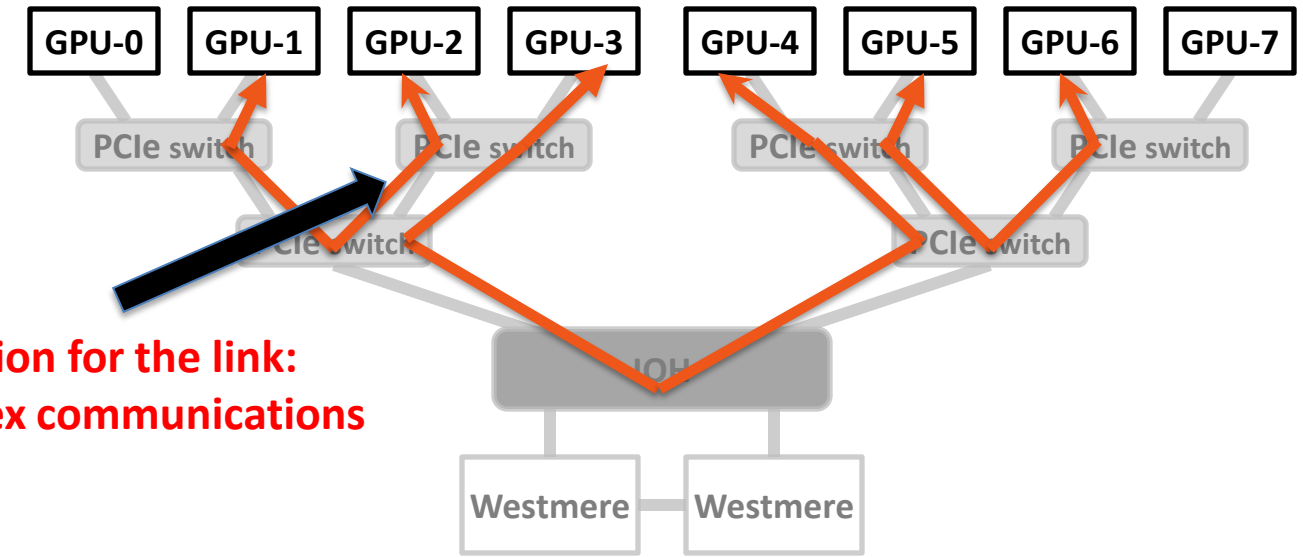
Stage 1: even-odd pairs



Stage 2: odd-even pairs

- **No contention for PCIe links**
  - All transfers are duplex, PCIe links are duplex
  - Note that no link has more than 1 exchange
    - Not true for 8 or more GPUs

Example: Even-Odd Stage of Pairwise Approach for 8 GPUs



Contention for the link:
2 duplex communications

- **Odd-even stage:**
  - Will always have contention for 8 or more GPUs
- **Even-odd stage:**
  - Will not have contention

1D Communication

- **Pairwise approach slightly better for 2-GPU case**
- **Left-Right approach better for the other cases**

Code for the Left-Right Approach

```
for( int i=0; i<num_gpus-1; i++ )          // “right” stage
   cudaMemcpyPeerAsync( d_a[i+1], gpu[i+1], d_a[i], gpu[i], num_bytes, stream[i] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream[i] );

for( int i=1; i<num_gpus; i++ )          // “left” stage
   cudaMemcpyPeerAsync( d_b[i-1], gpu[i-1], d_b[i], gpu[i], num_bytes, stream[i] );
```

- **Code assumes that addresses and GPU IDs are stored in arrays**
- **The middle loop isn’t necessary for correctness**
  - Improves performance by preventing the two stages from interfering with each other (15 vs 11 GB/s for the 4-GPU example)

Communication for Multiple Host, Multiple GPUs

Communication Between GPUs in Different Nodes

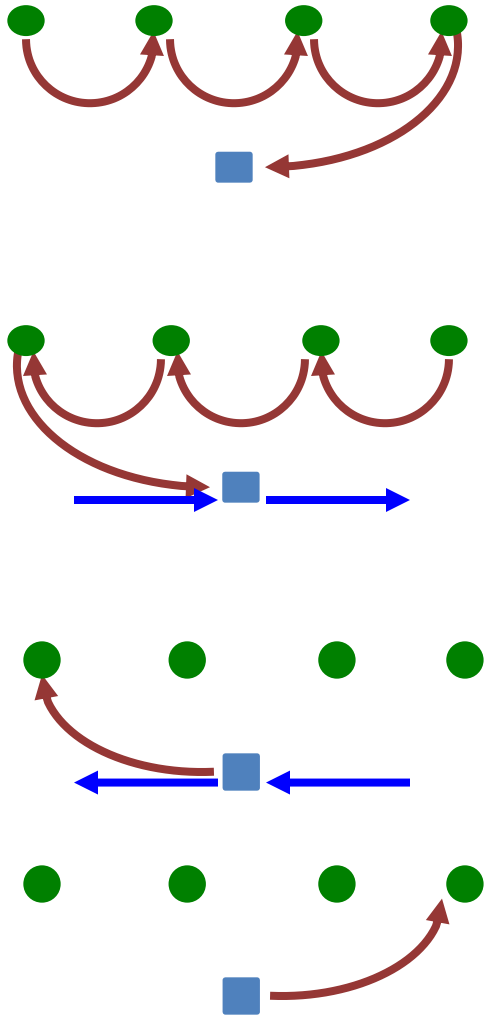
- **Requires network communication**
  - Currently requires data to first be transferred to host
- **Steps for an exchange:**
  - GPU->CPU transfer
  - CPU exchanges via network
    - For example, MPI_Sendrecv
    - Just like you would do for non-GPU code
  - CPU->GPU transfer
- **If each node also has multiple GPUs:**
  - Can continue using P2P within the node, netw outside the node
  - Can overlap some PCIe transfers with network communication
    - In addition to kernel execution

Code Pattern

```
cudaMemcpyAsync( ..., stream_halo[i] );
cudaStreamSynchronize( stream_halo[i] );
MPI_Sendrecv( ... );
cudaMemcpyAsync( ..., stream_halo[i] );
```

Overlapping MPI and PCIe Transfers

```
for( int i=0; i<num_gpus-1; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

for( int i=1; i<num_gpus; i++ )
    cudaMemcpyPeerAsync( ..., stream_halo[i] );
cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream_halo[i] );

cudaSetDevice( gpu[0] );
cudaMemcpyAsync( ..., stream_halo[0] );
MPI_Sendrecv( ... );

cudaSetDevice( gpu[num_gpus-1] );
cudaMemcpyAsync( ..., stream_halo[num_gpus-1] );
```



GPU-Aware MPI

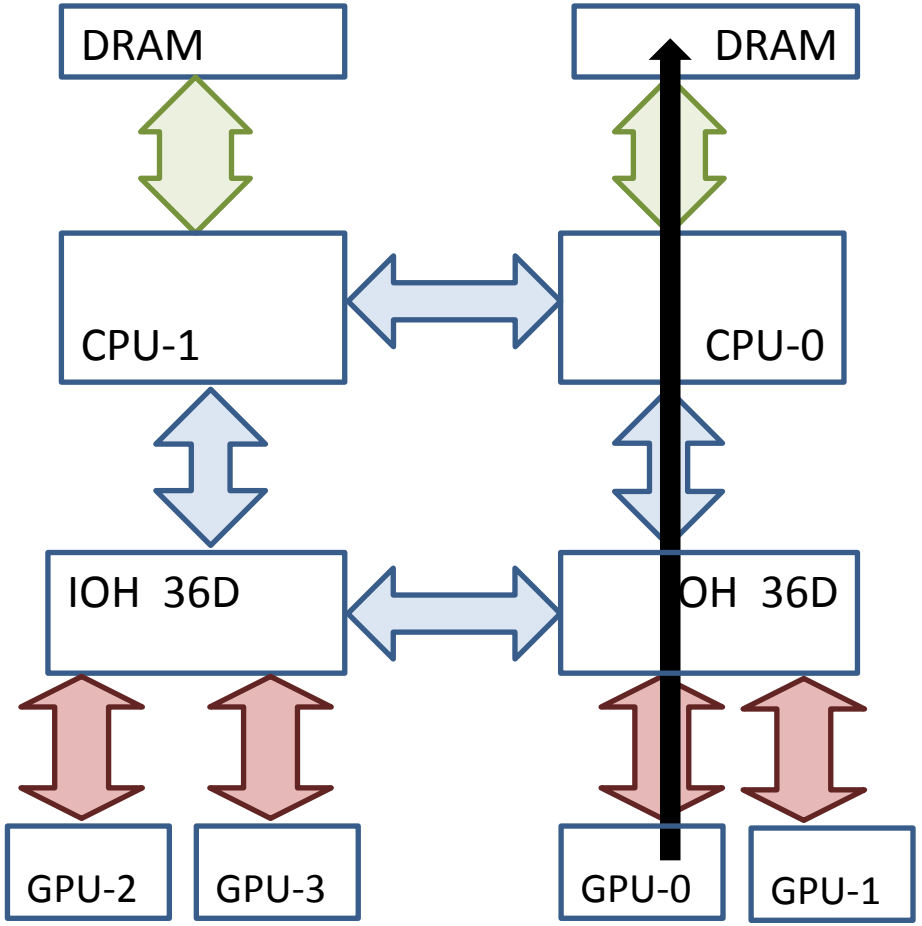
- **MPI calls can take GPU pointers**
  - mvapich, openmpi
  - Works with C/C++, Fortran, CUDA C, CUDA Fortran, directives-based code
- **Benefits:**
  - Simplifies code (no need to explicitly copy GPU<->CPU)
  - Can pipeline transfers for better performance:
    - Break the transfer into smaller pieces
    - Pipeline the transfer of pieces: overlap PCIe and Netw for all but the first and last piece
- **Not yet available:**
  - P2P path when MPI ranks are on the same node

Host (CPU) NUMA and CPU/GPU Transfers

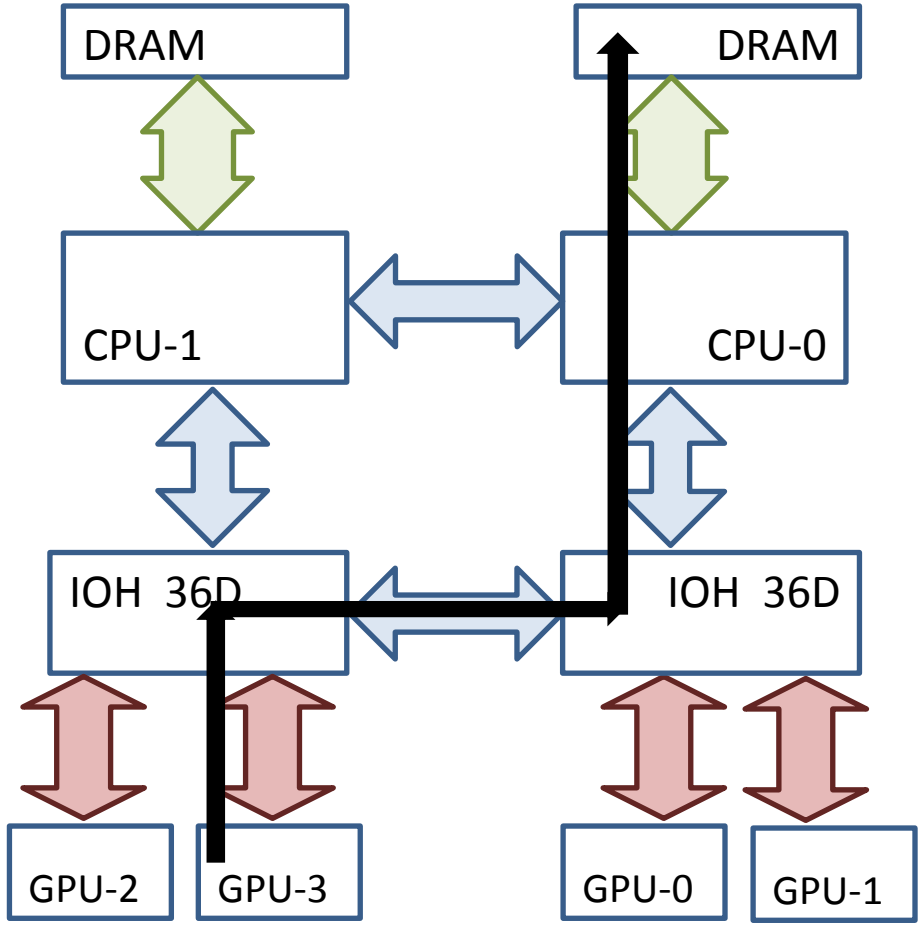
Additional System Issues to Consider

- **CPU NUMA affects PCIe transfer throughput in dual-IOH systems**
  - Transfers to “remote” GPUs achieve lower throughput
    - One additional QPI hop
  - This affects any PCIe device, not just GPUs
    - Network cards, for example
  - When possible, lock CPU threads to a socket that’s “closest” to the GPU
    - For example, by using numactl, GOMP_CPU_AFFINITY, KMP_AFFINITY, etc.
- **Dual-IOH systems prevent PCIe P2P across the IOH chips**
  - QPI link between the IOH chips isn’t compatible with PCIe P2P
  - P2P copies will still work, but will get staged via host memory

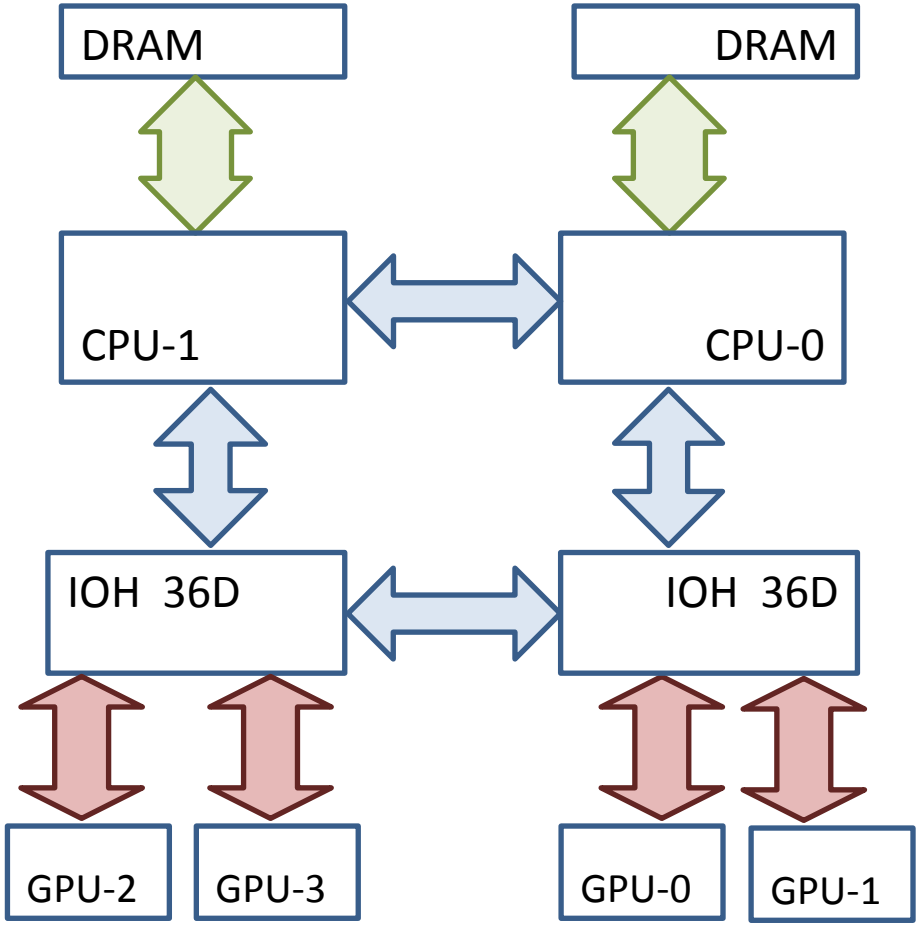
“Local” D2H Copy: 6.3 GB/s



“Remote” D2H Copy: 4.3 GB/s



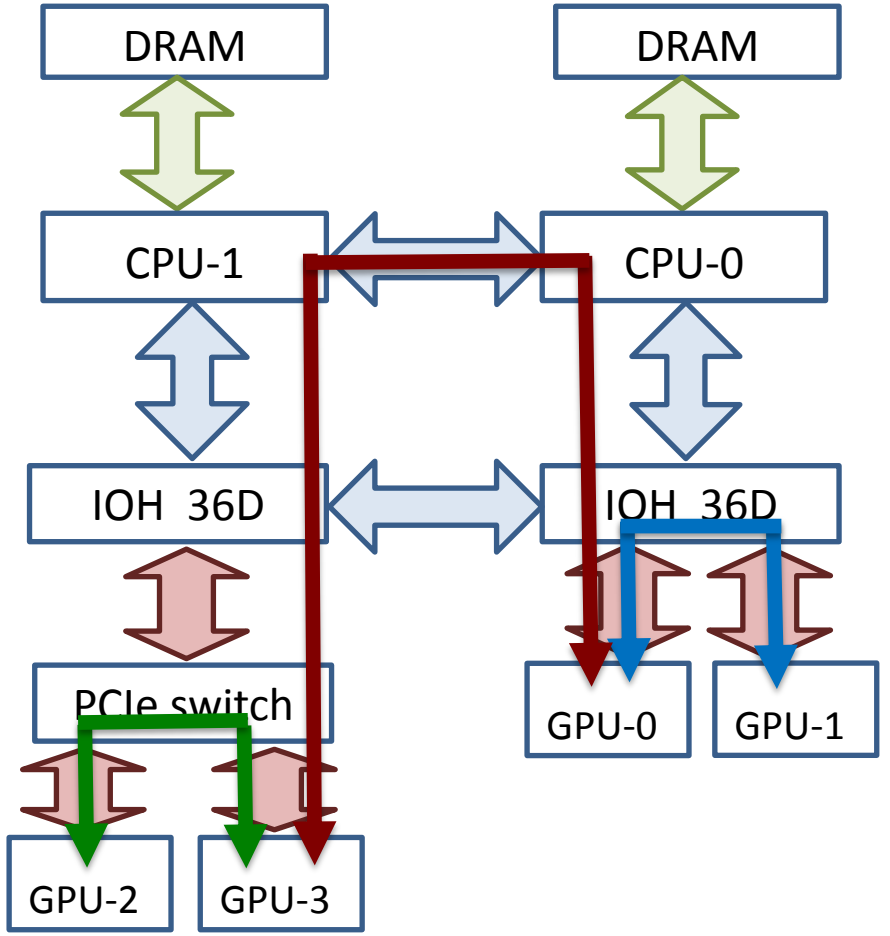
Summary of CPU-GPU Copy Throughputs on One System



- **Note that these vary among different systems**
  - Different BIOS settings
  - Different IOH chips
- **Local:**
  - D2H: 6.3 GB/s
  - H2D: 5.7 GB/s
- **Remote:**
  - D2H: 4.3 GB/s
  - H2D: 4.9 GB/s

Summary of P2P Throughputs, PCIe gen2

- **Via PCIe switch:**
  - GPUs attached to the same PCIe switch
  - Simplex: 6.3 GB/s (12 GB/s gen3)
  - Duplex: 12.2 GB/s (22 GB/s gen3)
- **Via IOH chip:**
  - GPUs attached to the same IOH chip
  - Simplex: 5.3 GB/s
  - Duplex: 9.0 GB/s
- **Via host:**
  - GPUs attached to different IOH chips
  - Simplex: 2.2 GB/s
  - Duplex: 3.9 GB/s



Determining Topology/Locality of a System

- **Hardware Locality tool:**
  - http://www.open-mpi.org/projects/hwloc/
  - Cross-OS, cross-platform

Multi-GPU Systems

GPU becoming more specialized

Modern GPU “Processing Block”

- 32 Threads
- 16 INT
- 16 single-precision FP
- 8 double-precision FP
- 4 SFU (sin, cos, log)
- 2 Tensor units for DNN
- 64KB RF

GPU Streaming Multiprocessor

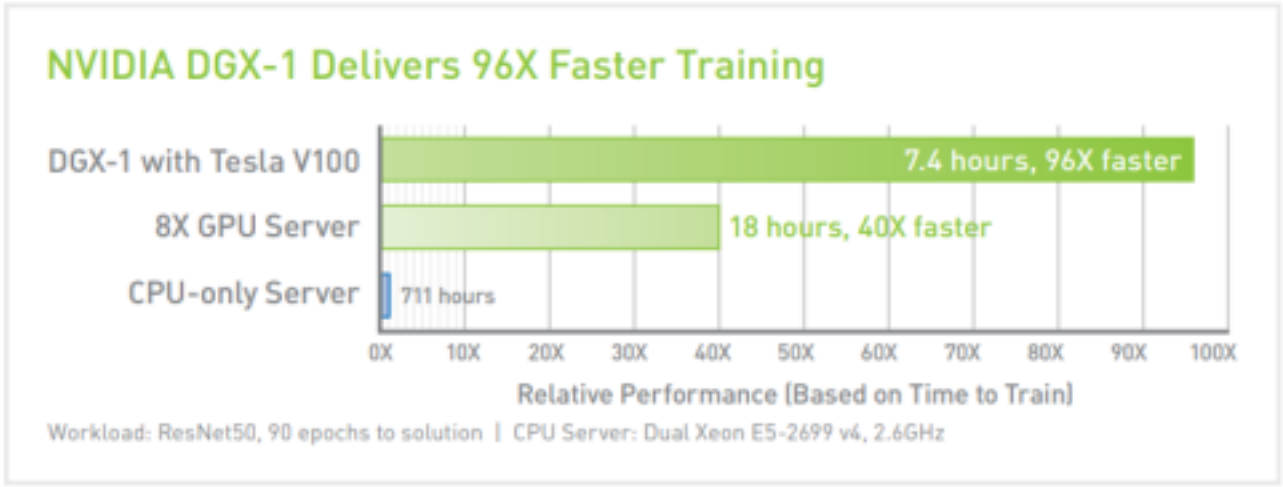
- Contains 4 “Processing Blocks”
- Each independently schedules a set of 32 threads called a warp
- Share L1 Cache between blocks

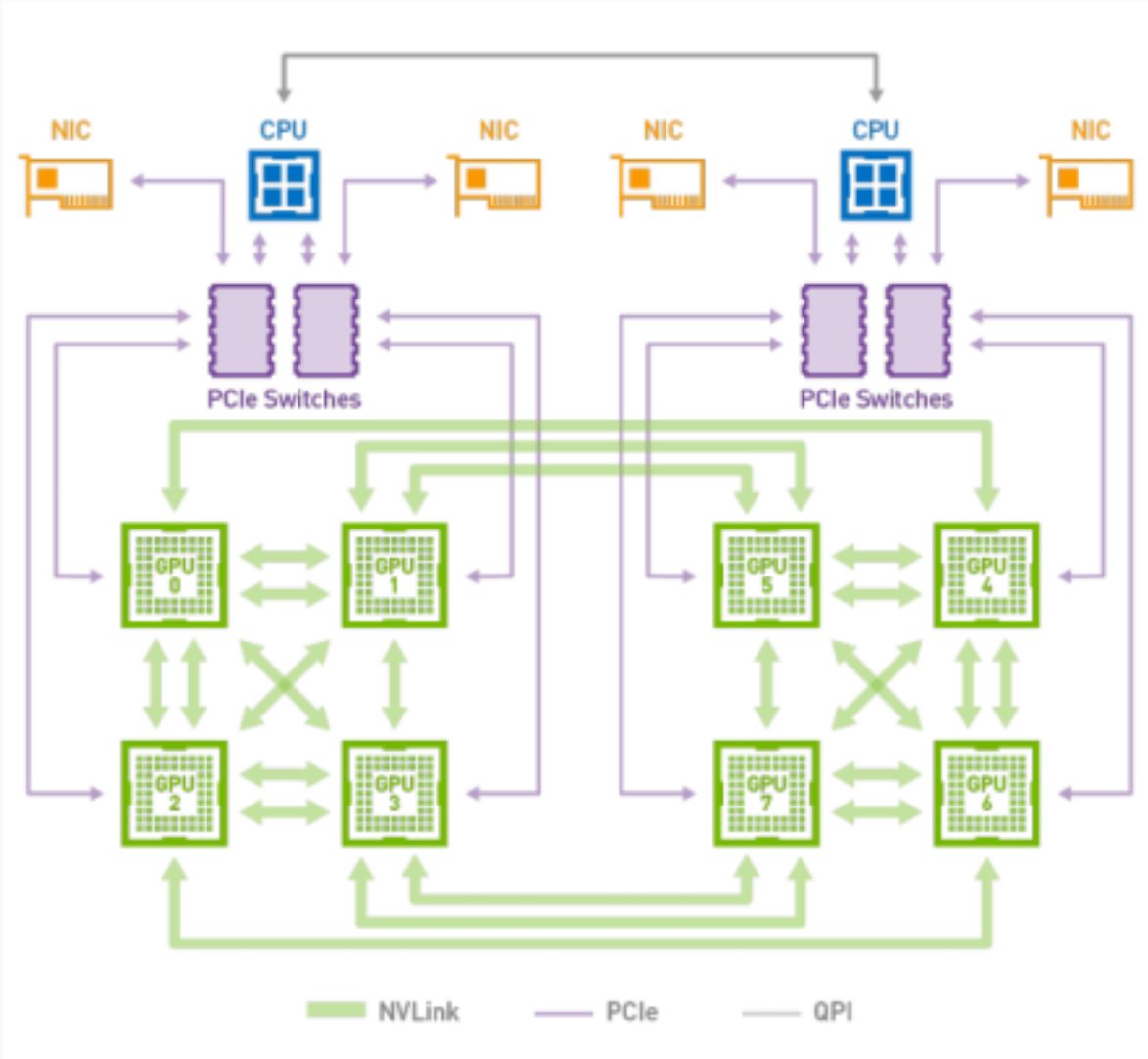
GPU Hardware

- V100 has 80 SM
- 5376 FPU
- Peak 15.7 TFLOPS

GPU “Data center in a box”

- DGX
  - A Multi-GPU “Node”
  - 300GB/s NVlink 2.0 cube mesh
  - 1 PFLOPS
  - Faster Machine Learning





NVIDIA

DATA CENTER PRODUCTS SOLUTIONS APPS FOR DEVELOPERS TECHNOLOGIES

DGX-1 OVERVIEW TOUR SPECS ORDER NOW

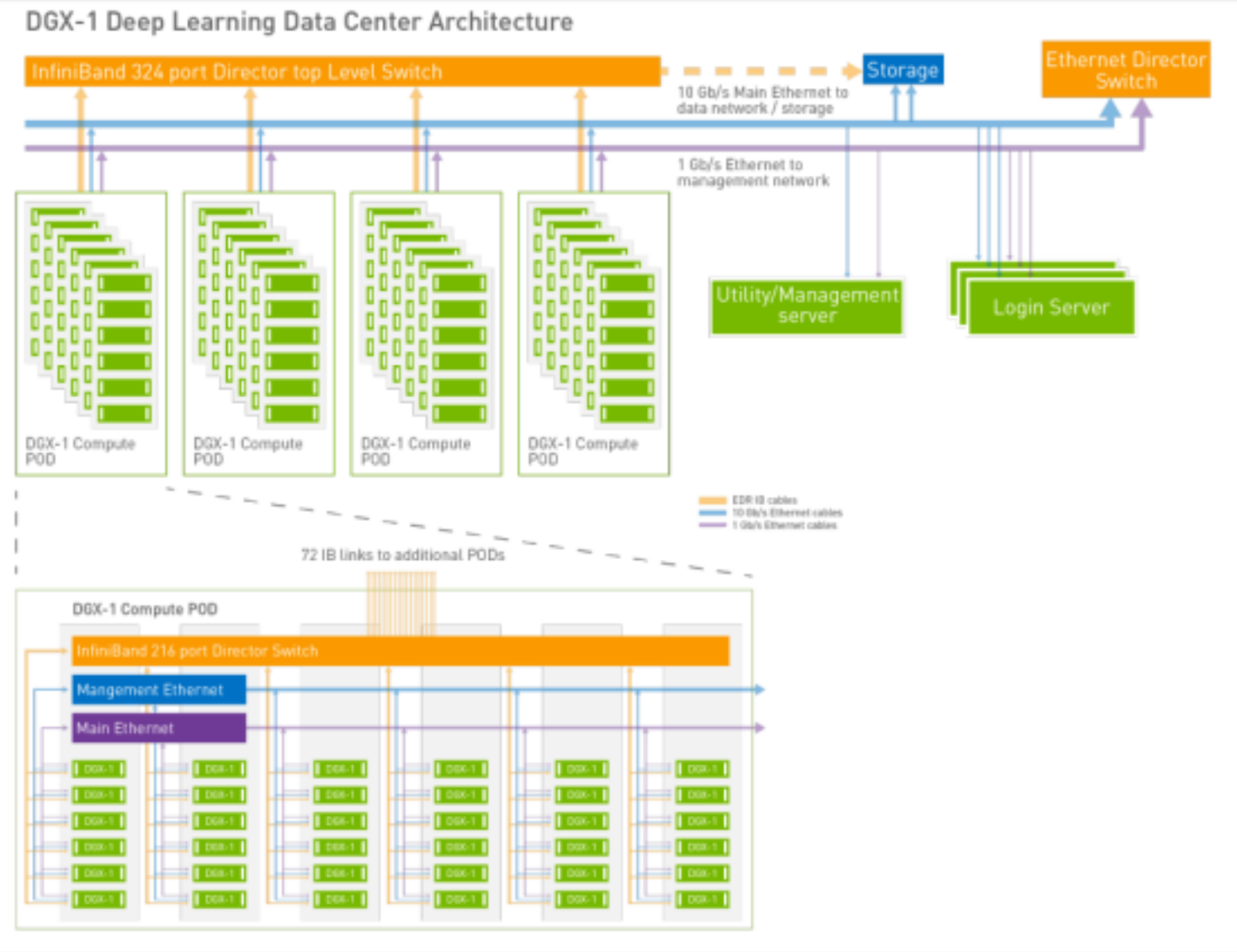
NVIDIA DGX-1

Essential Instrument of AI Research

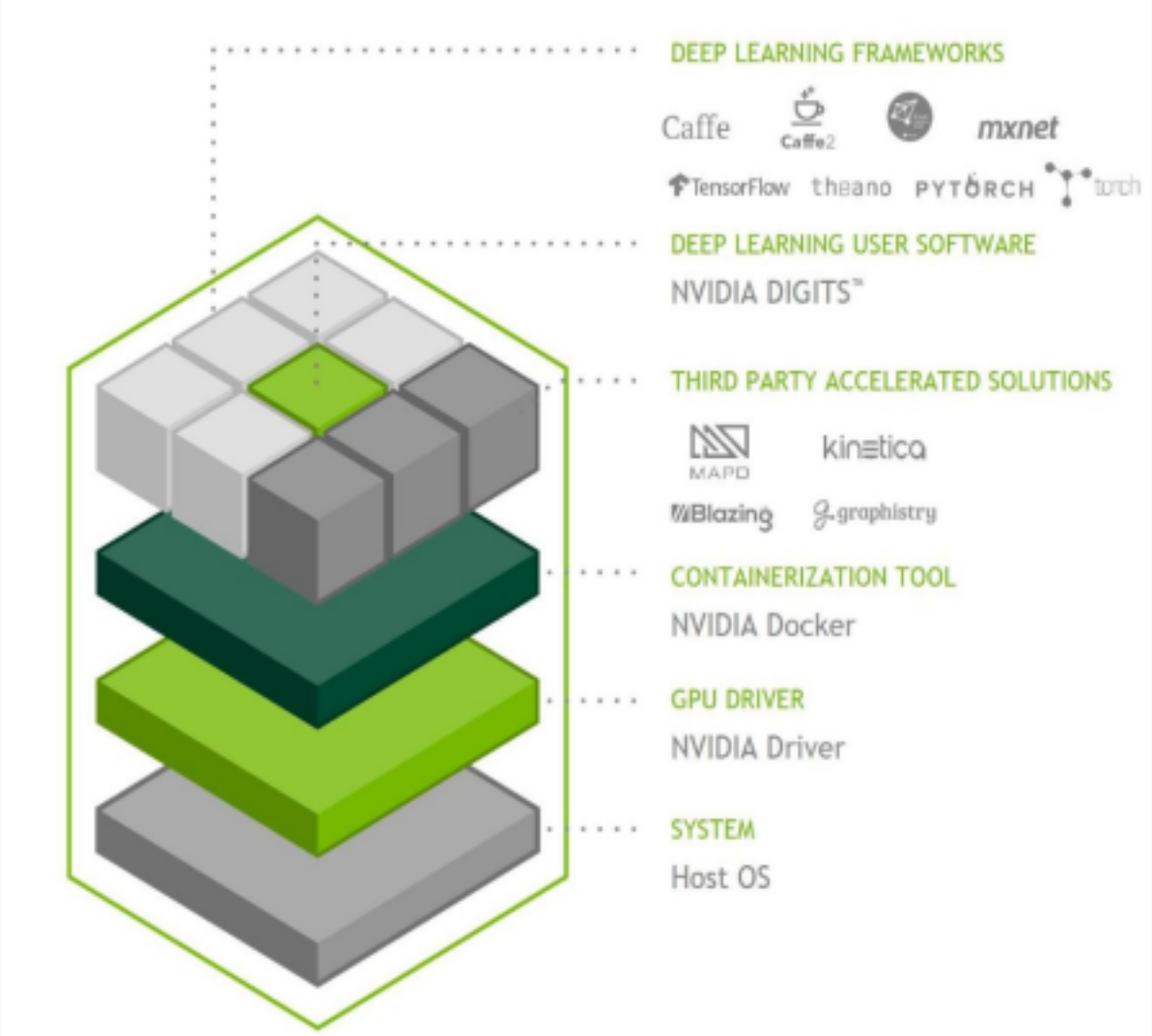
THE FASTEST PATH TO DEEP LEARNING

Building a platform for deep learning goes well beyond selecting a server and GPUs. A commitment to implementing AI in your business involves carefully selecting and integrating complex software with hardware. NVIDIA® DGX-1™ fast-tracks your initiative with a solution that works right out of the box, so you can gain insights in hours instead of weeks or months.

DGX Data Center

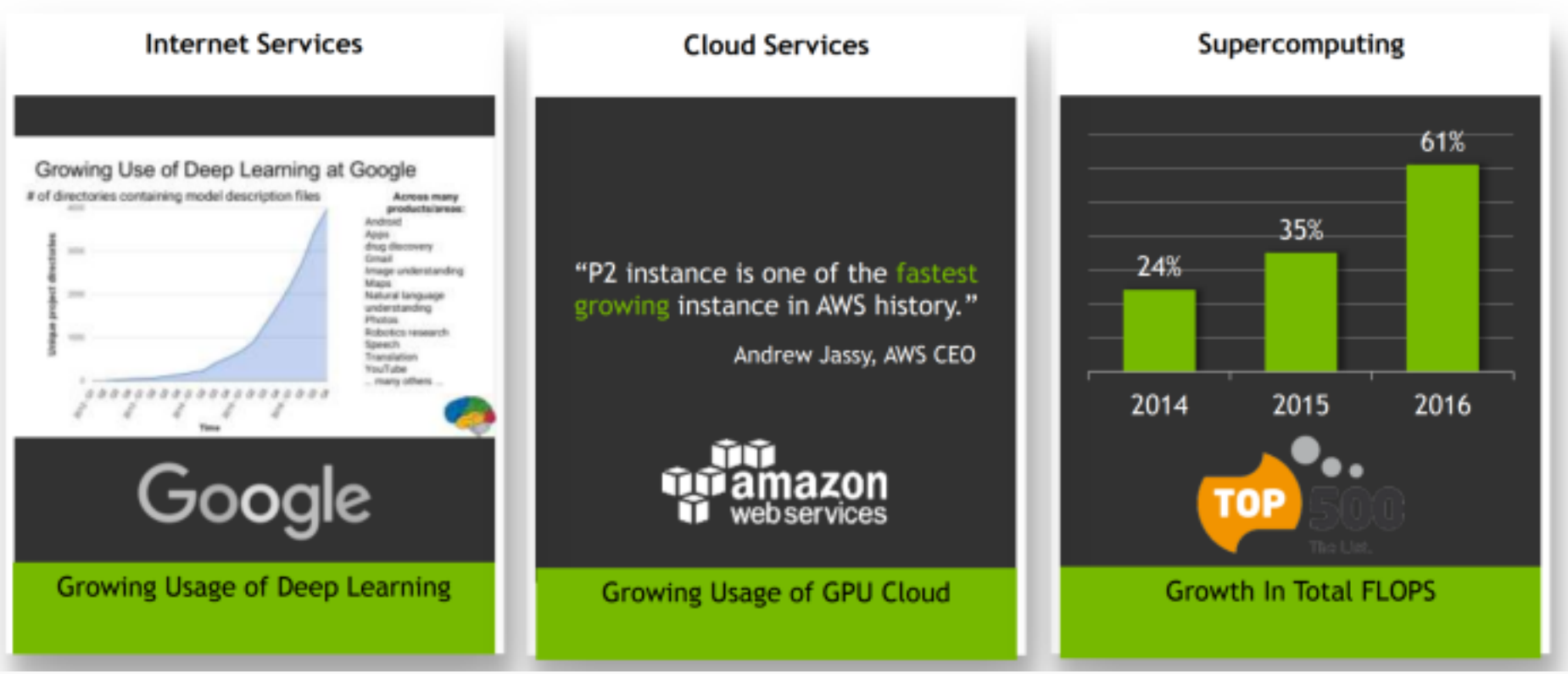


GPU Support in Cloud Computing Stack

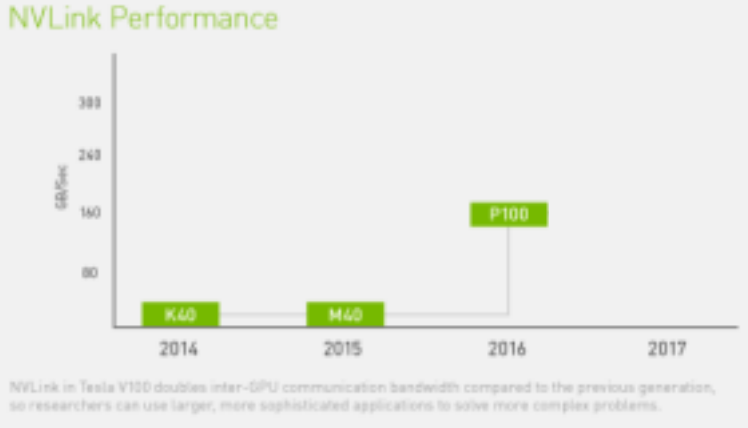


GPUs in the Cloud

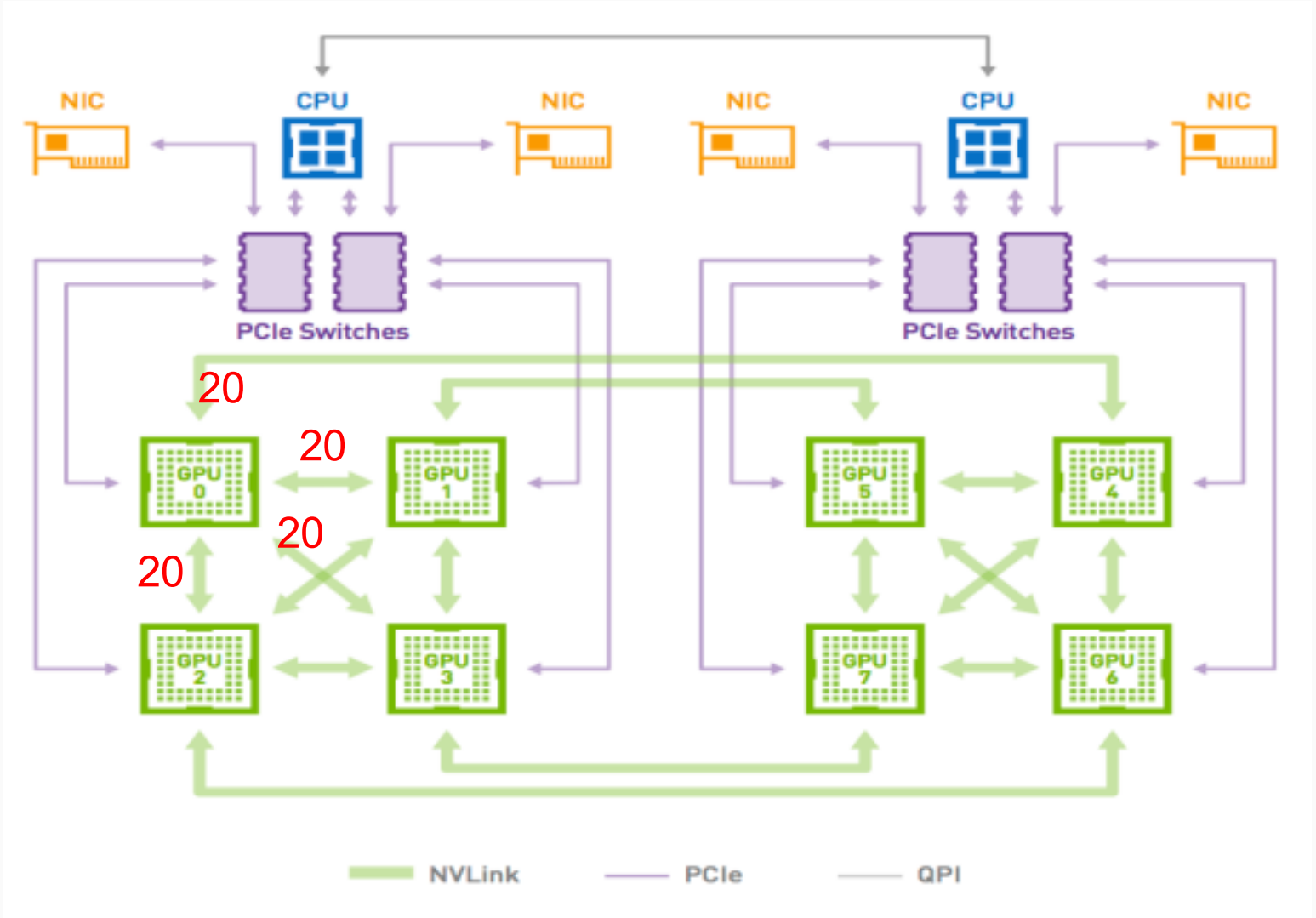
- Exponential demand for more compute power



GPU inter-connection is getting complex

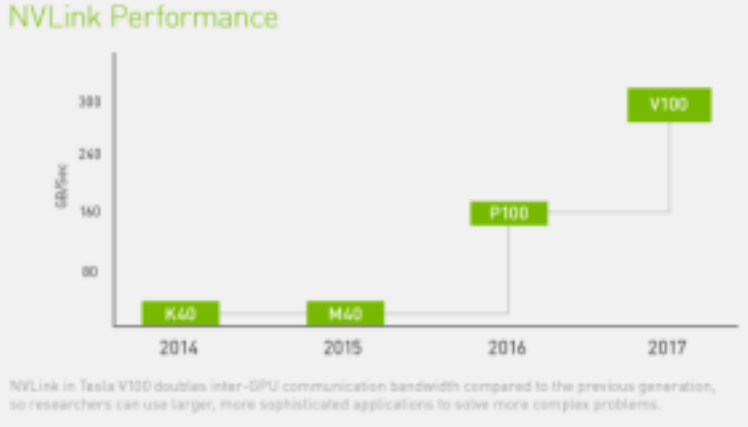


80(in)+80(out)=160GB/s(Total)

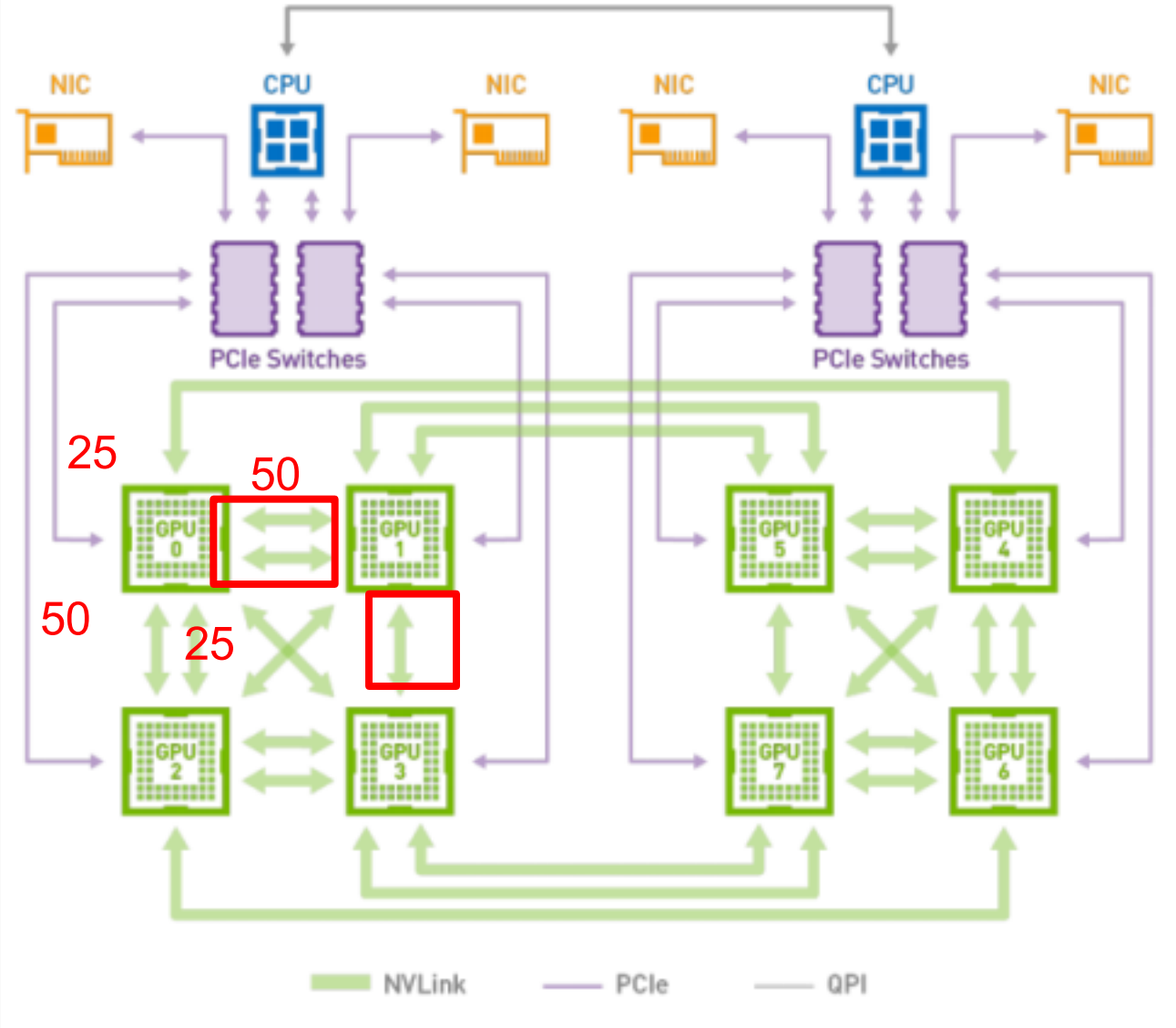


Picture sources: NVlink whitepaper

GPU inter-connection is getting complex

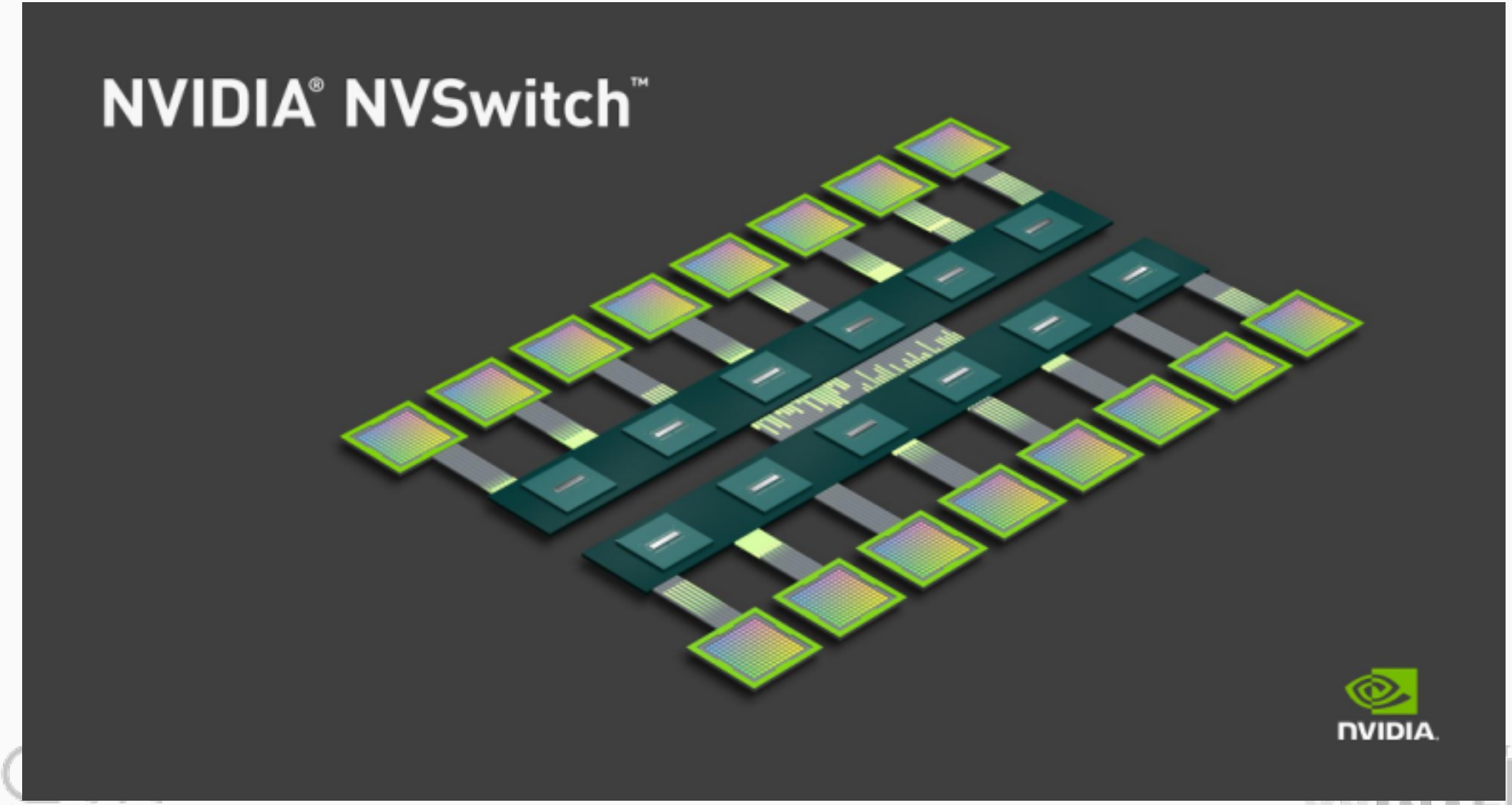


150(in)+150(out)=300GB/s(Total)

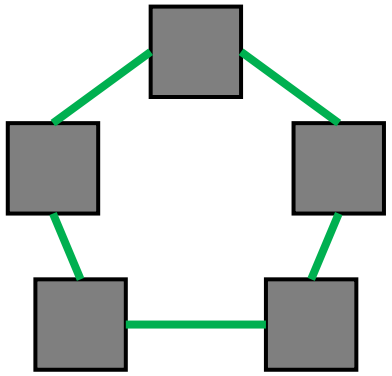


Picture sources: NVlink whitepaper

GPU inter-connection is getting complex



Accelerated workload exhibit diverse inter-accelerator communication patterns



Collective communication (Ring/Tree)

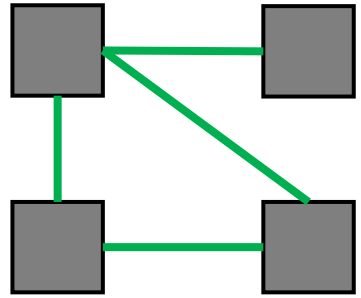
ML training
Ex. RCCL, NCCL

Point-to-point / Grid (MPI-like)

HPC workloads
Ex. Aluminum (LLNL)

Pipelined

Data science
Ex. RAPIDS

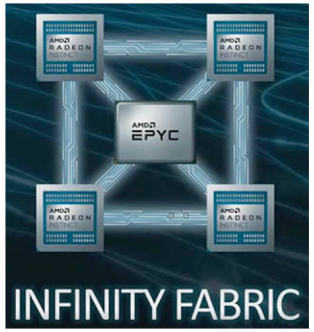


Irregular

Partitioned Global Address Space
Ex. OpenSHMEM

Accelerator fabric galore!

- AMD Infinity Fabric/Architecture



- Nvidia NVLink / NVSwitch

NVIDIA NVLINK™

- Compute Express Link (CXL)

CXL Compute Express Link™

Xe LINK

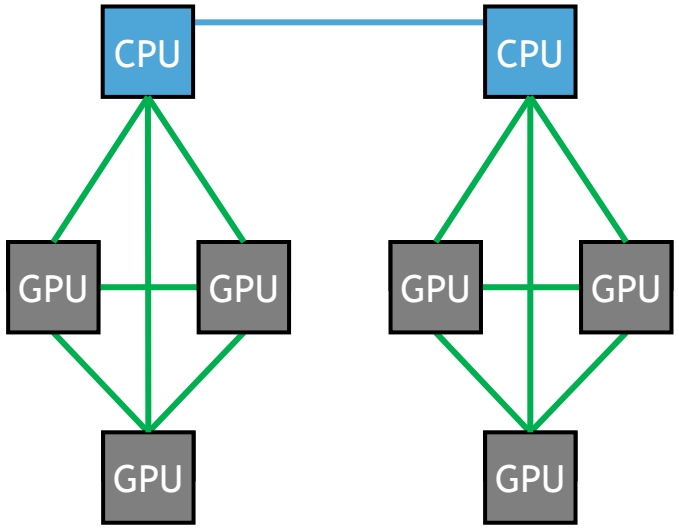
- and more!

CCIX®

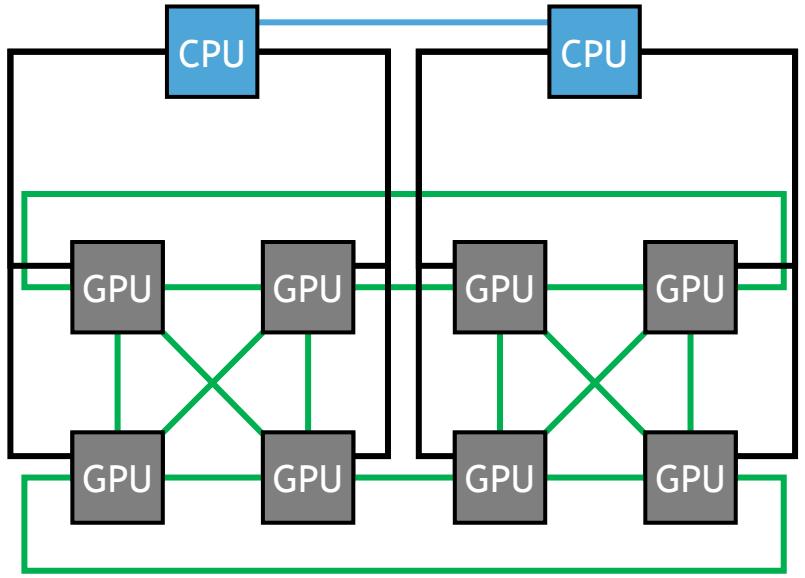
OpenCAPI™

GEN Z

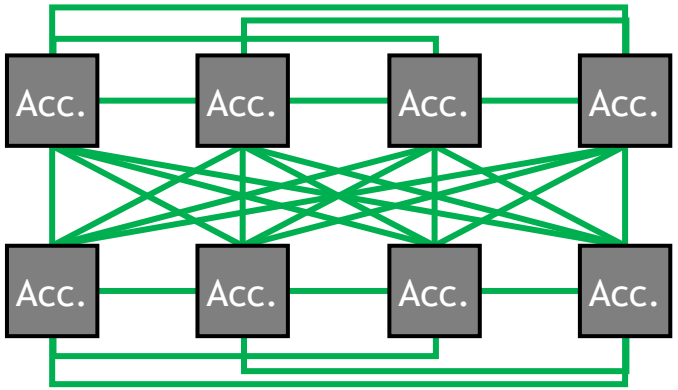
Accelerator topology is diverse



Summit (ORNL)



DGX-1 / Big Basin

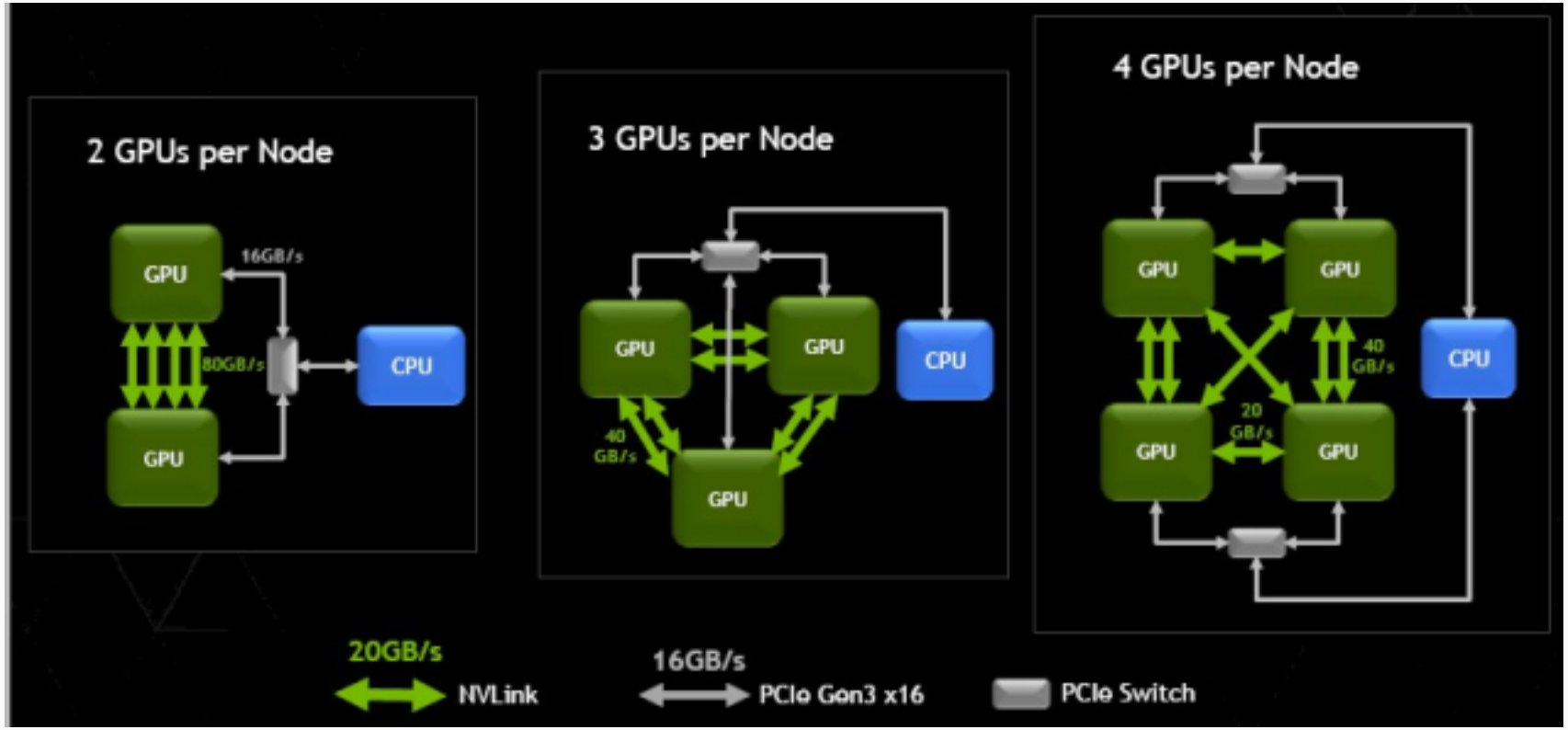


Facebook Zion
(Open Compute Project)

CPU-CPU PCIe Accelerator fabric

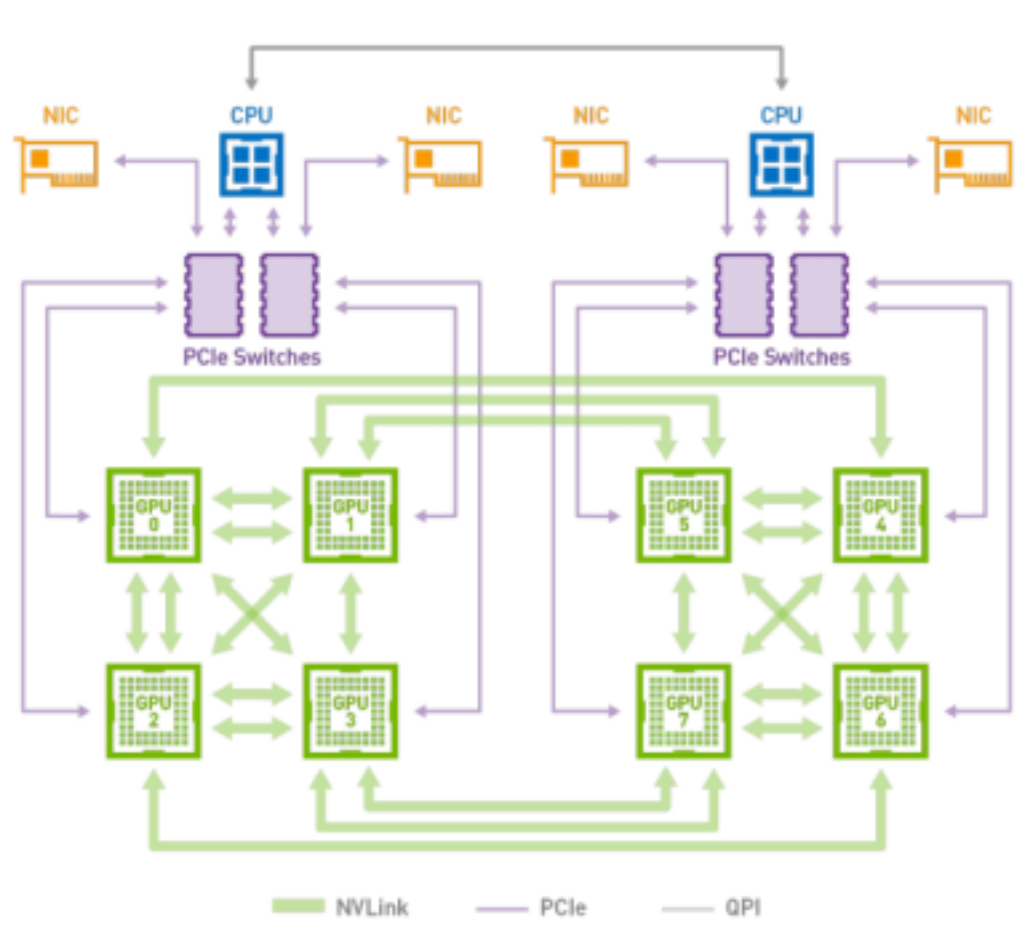
How can we make efficient use of GPU inter-connects?

NVLink: Fast communication between multi-GPUs



Challenges of complex GPU inter-connects

- Programming Multi-GPU applications is hard



NCCL: ACCELERATED MULTI-GPU COLLECTIVE COMMUNICATIONS

Cliff Woolley, Sr. Manager, Developer Technology Software, NVIDIA

BACKGROUND

What limits the scalability of parallel applications?

Efficiency of parallel computation tasks

- Amount of exposed parallelism
- Amount of work assigned to each processor

Expense of communications among tasks

- Amount of communication
- Degree of overlap of communication with computation

COMMON COMMUNICATION PATTERNS

COMMUNICATION AMONG TASKS

What are common communication patterns?

Point-to-point communication

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

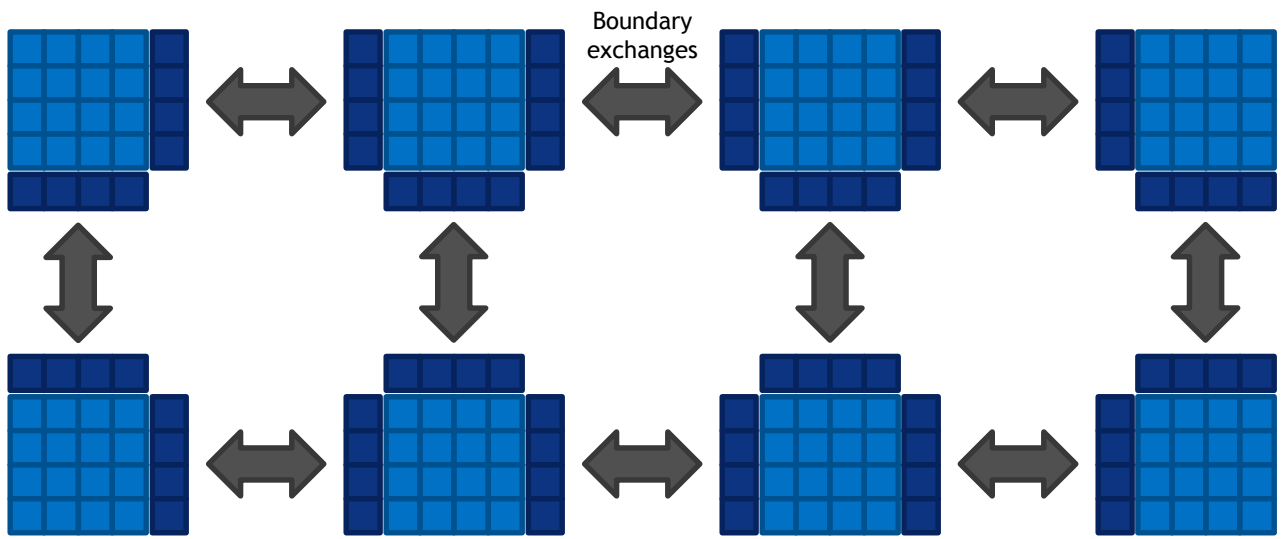
Collective communication

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

POINT-TO-POINT COMMUNICATION

Single-sender, single-receiver per instance

Most common pattern in HPC, where communication is usually to nearest neighbors

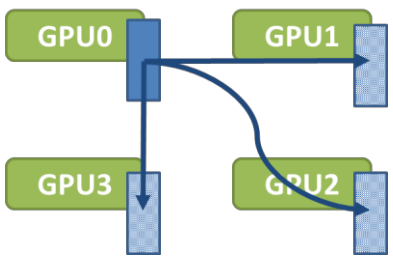


2D Decomposition

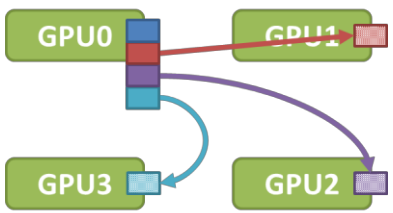
COLLECTIVE COMMUNICATION

Multiple senders and/or receivers

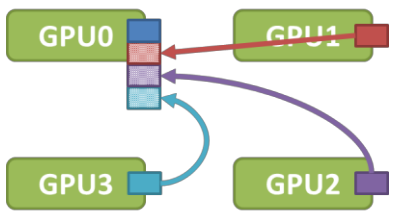
Broadcast



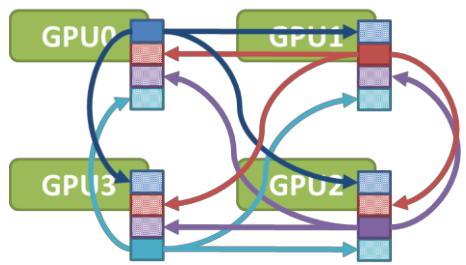
Scatter



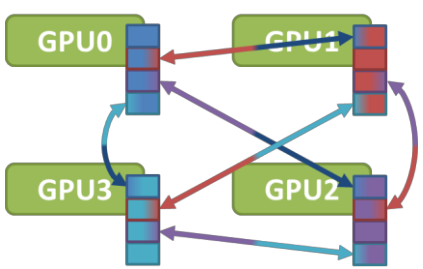
Gather



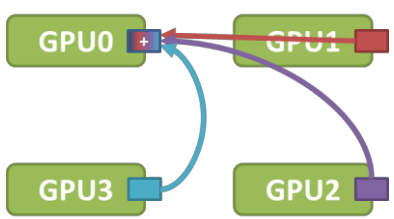
All-Gather



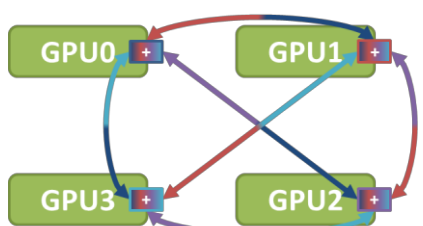
All-to-All



Reduce

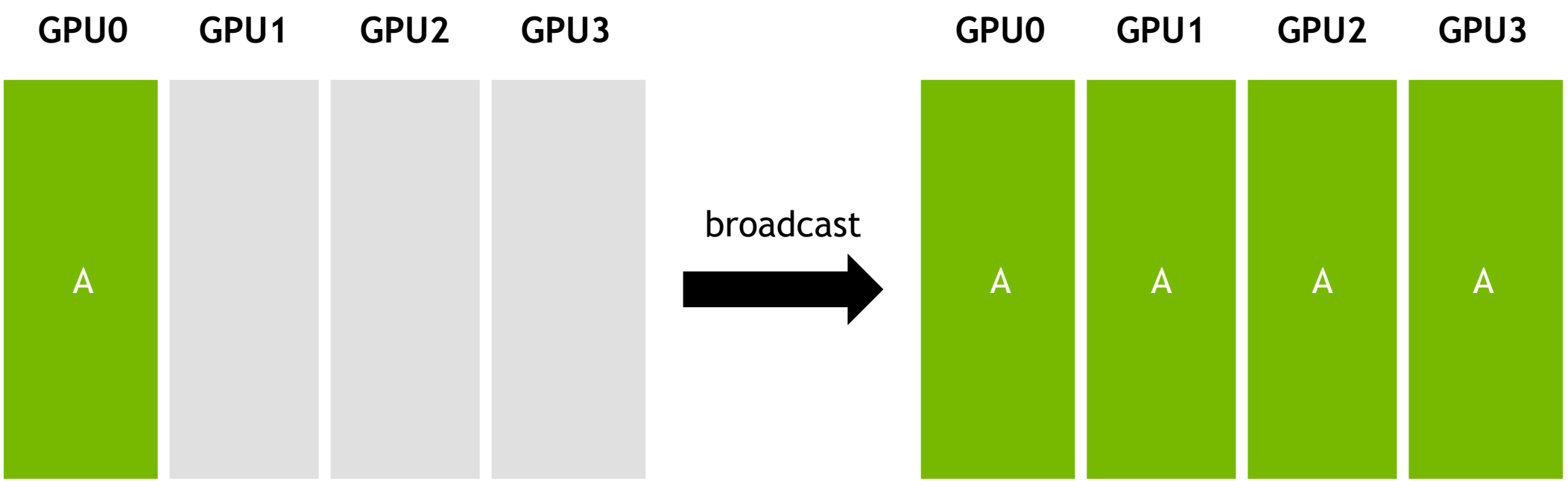


All-Reduce



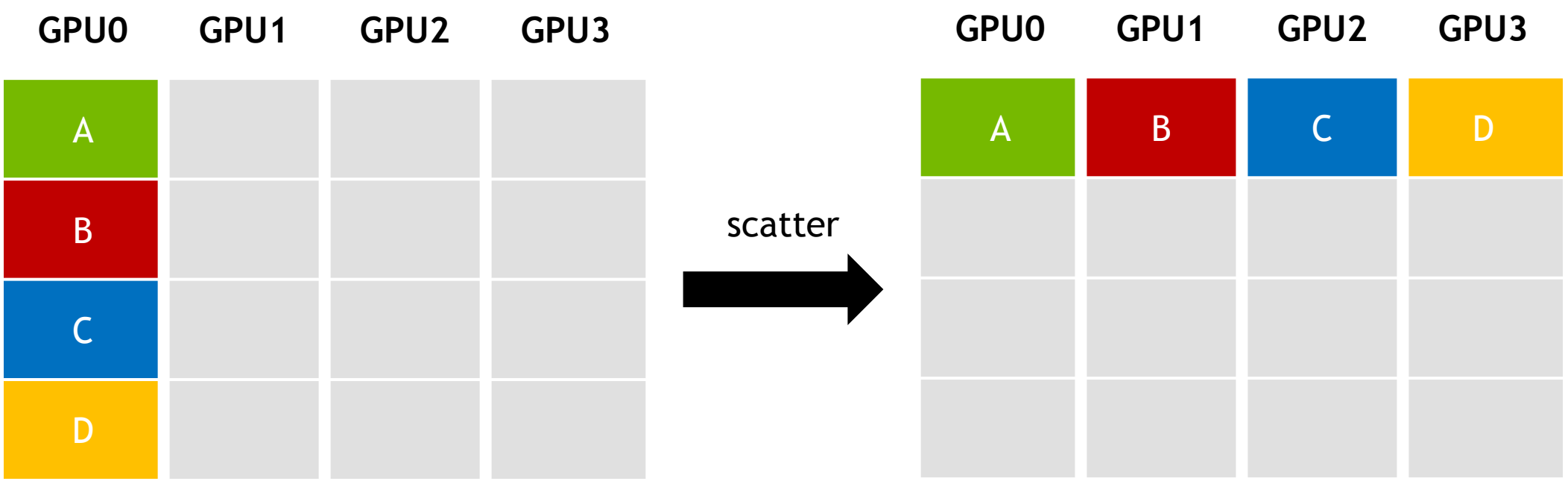
BROADCAST

One sender, multiple receivers



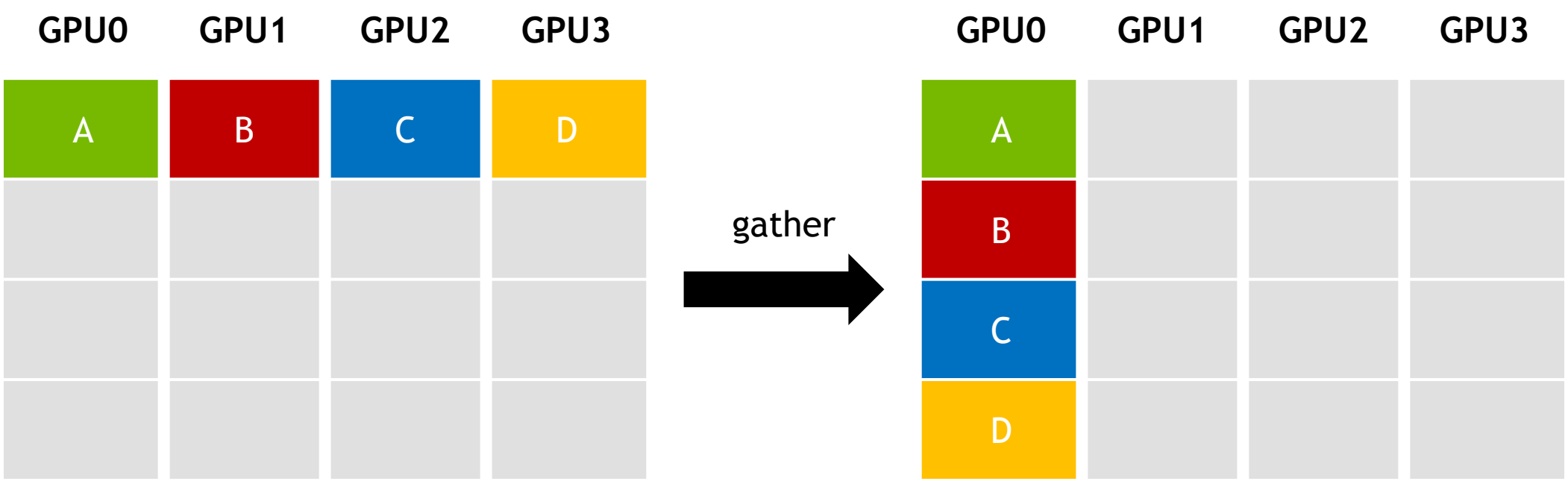
SCATTER

One sender; data is distributed among multiple receivers



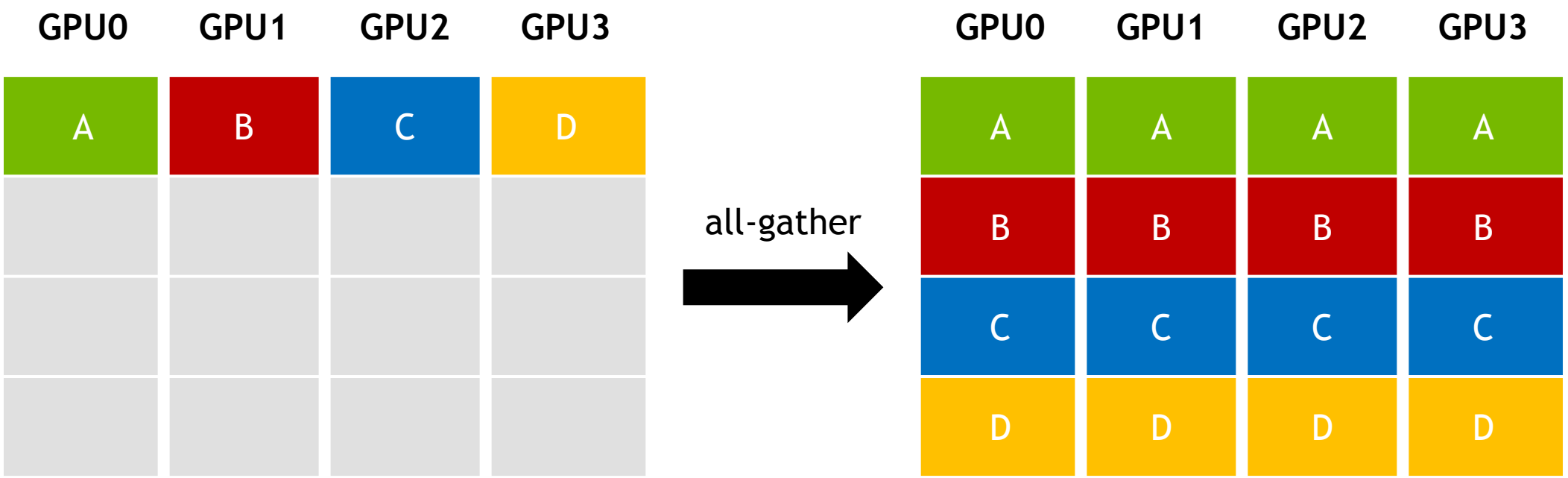
GATHER

Multiple senders, one receiver



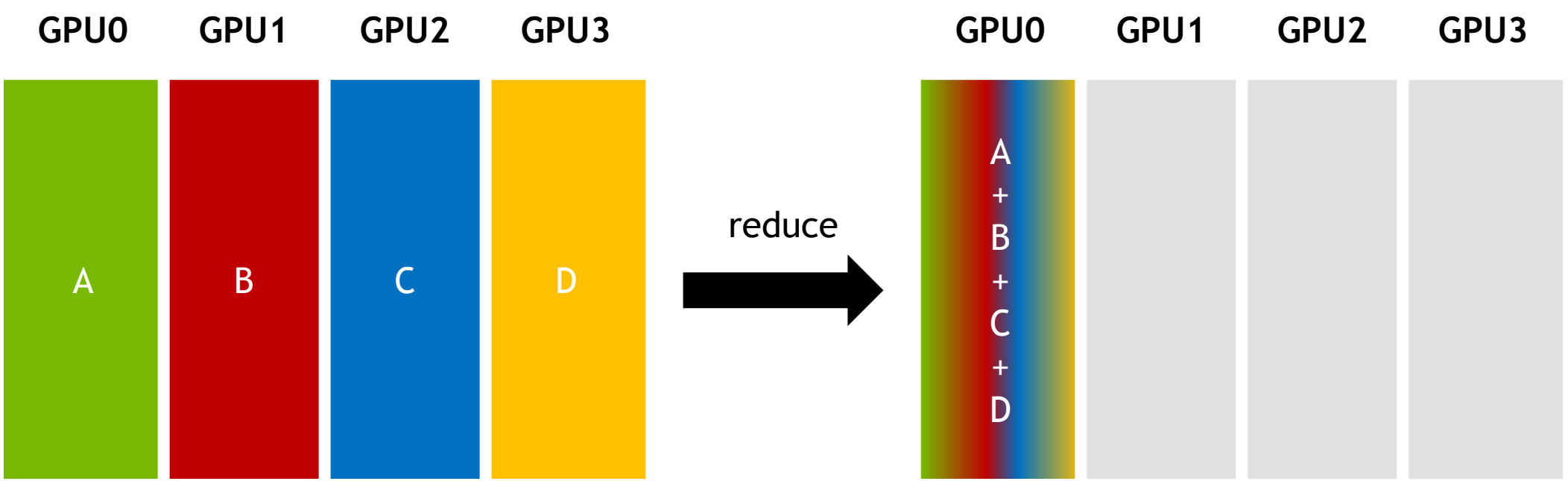
ALL-GATHER

Gather messages from all; deliver gathered data to all participants



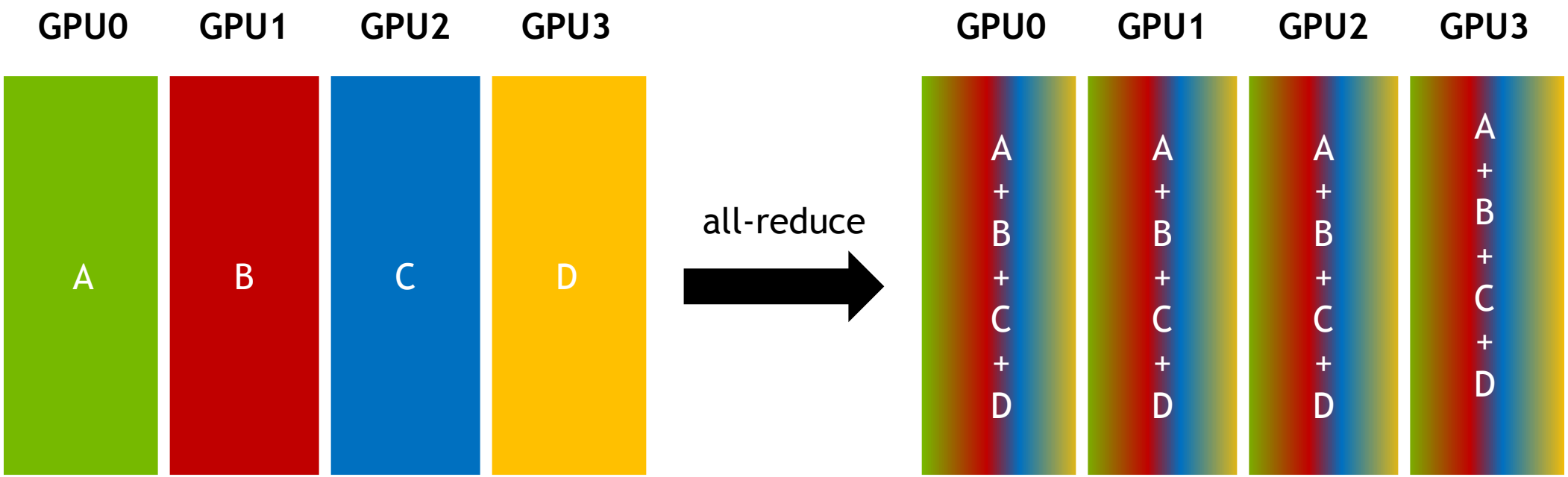
REDUCE

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



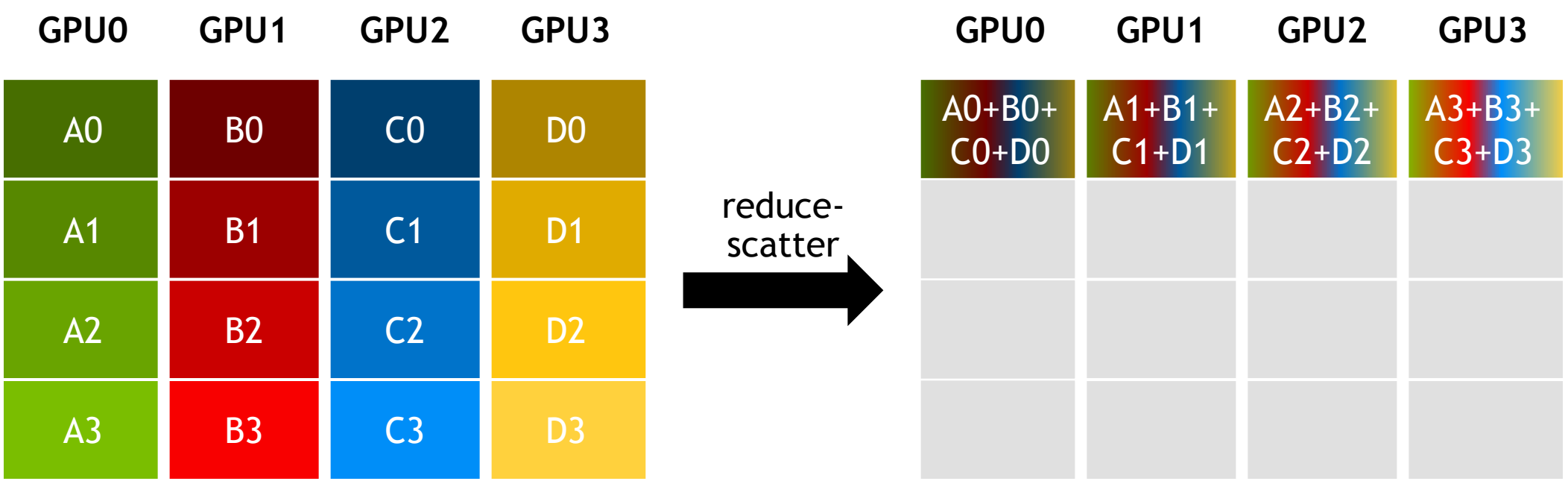
ALL-REDUCE

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



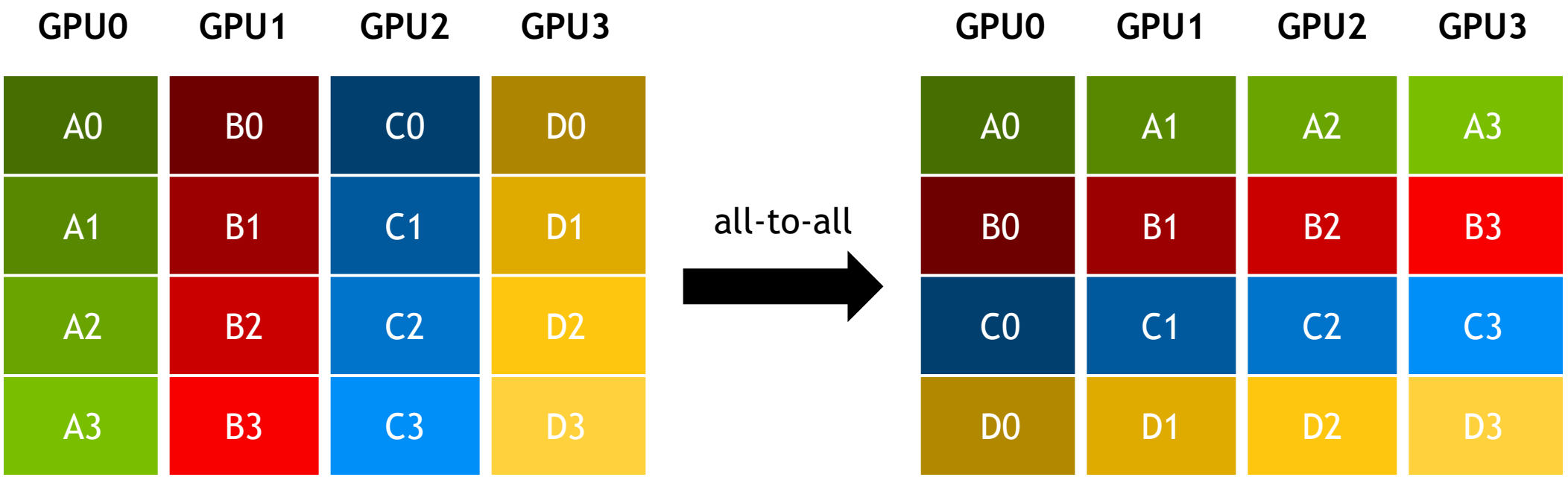
REDUCE-SCATTER

Combine data from all senders; distribute result across participants



ALL-TO-ALL

Scatter/Gather distinct messages from each participant to every other



THE CHALLENGE OF COLLECTIVES

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

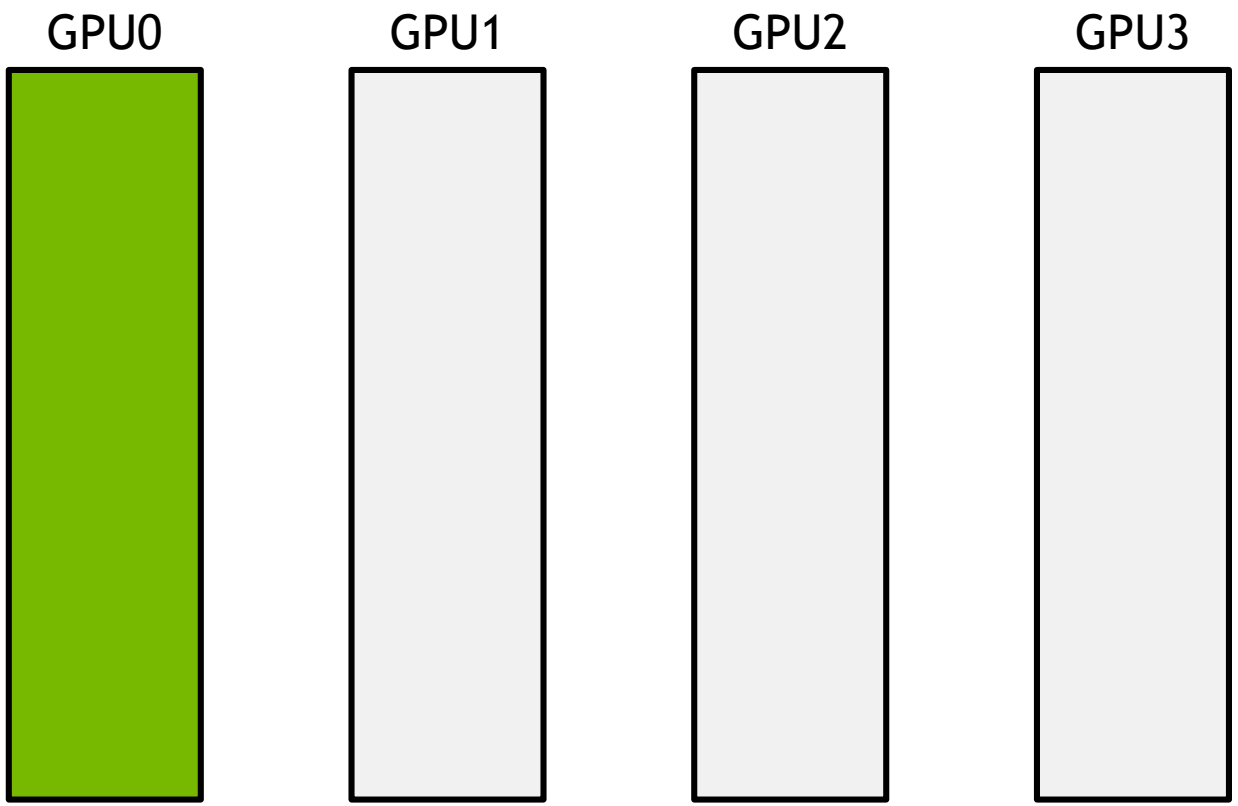
Topologies can be complex – not every system is a fat tree

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]

RING-BASED COLLECTIVES: A PRIMER

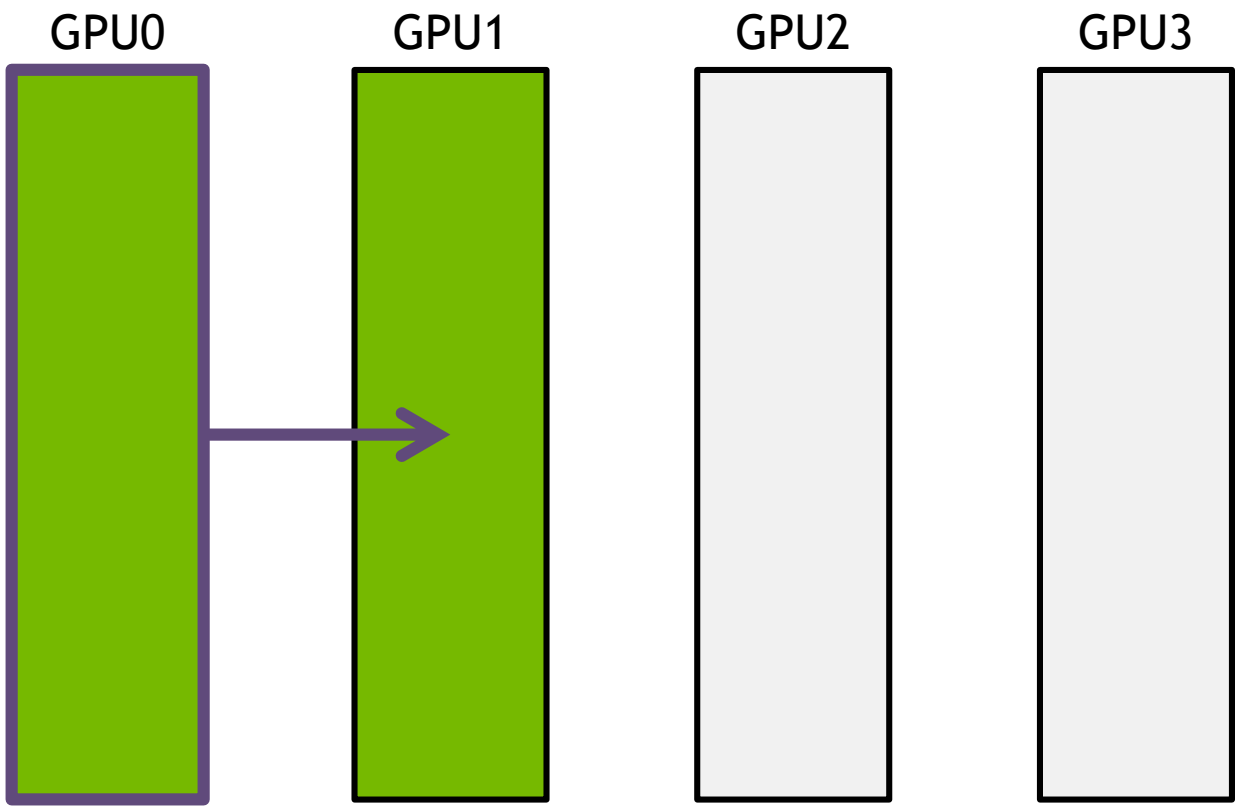
BROADCAST

with unidirectional ring



BROADCAST

with unidirectional ring



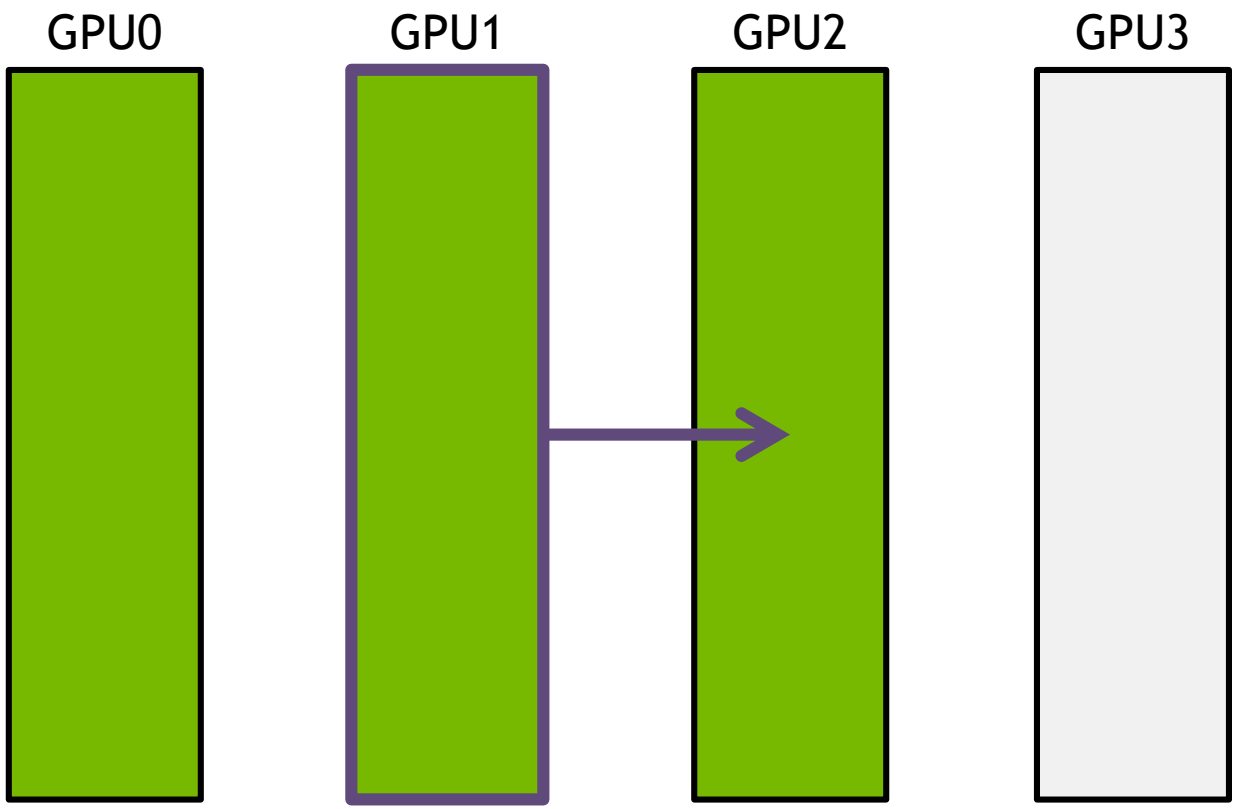
Step 1: $\Delta t = N/B$

N: bytes to broadcast

B: bandwidth of each link

BROADCAST

with unidirectional ring



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

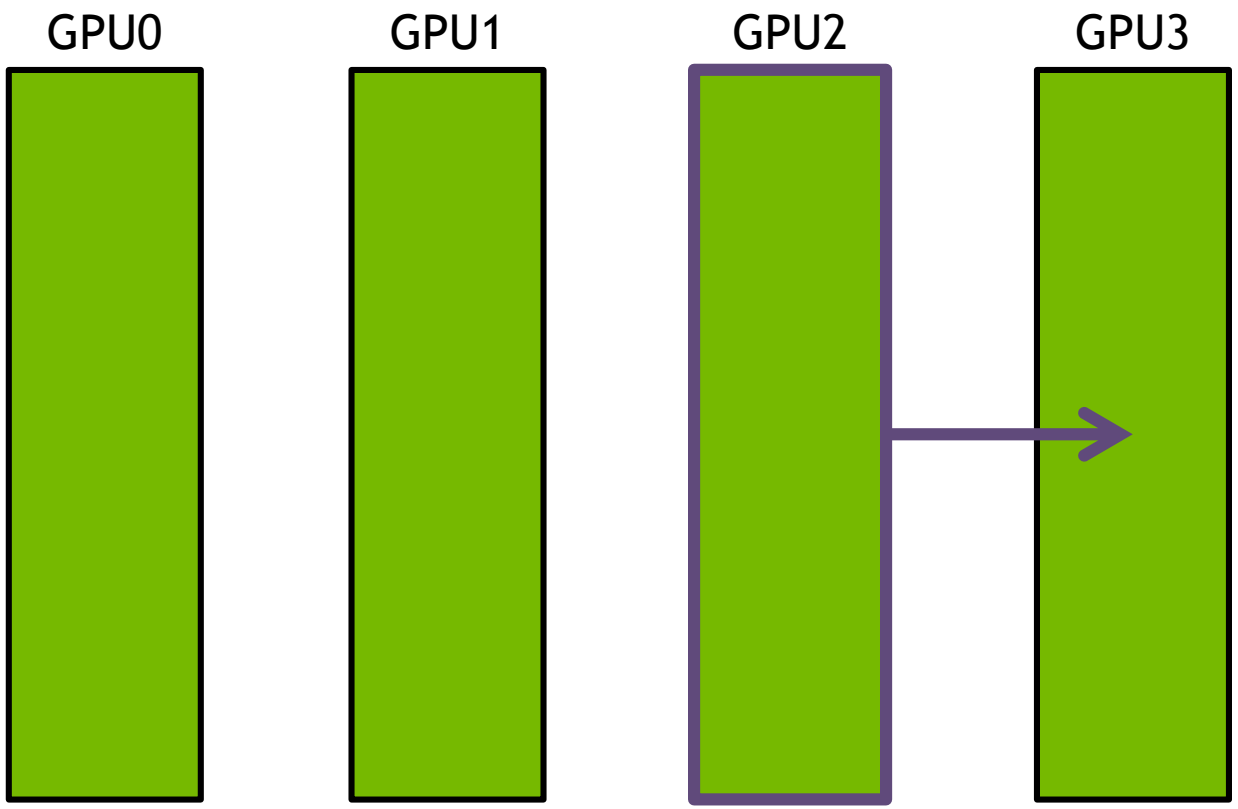
Step 2: $\Delta t = N/B$

N: bytes to broadcast

B: bandwidth of each link

BROADCAST

with unidirectional ring



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

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

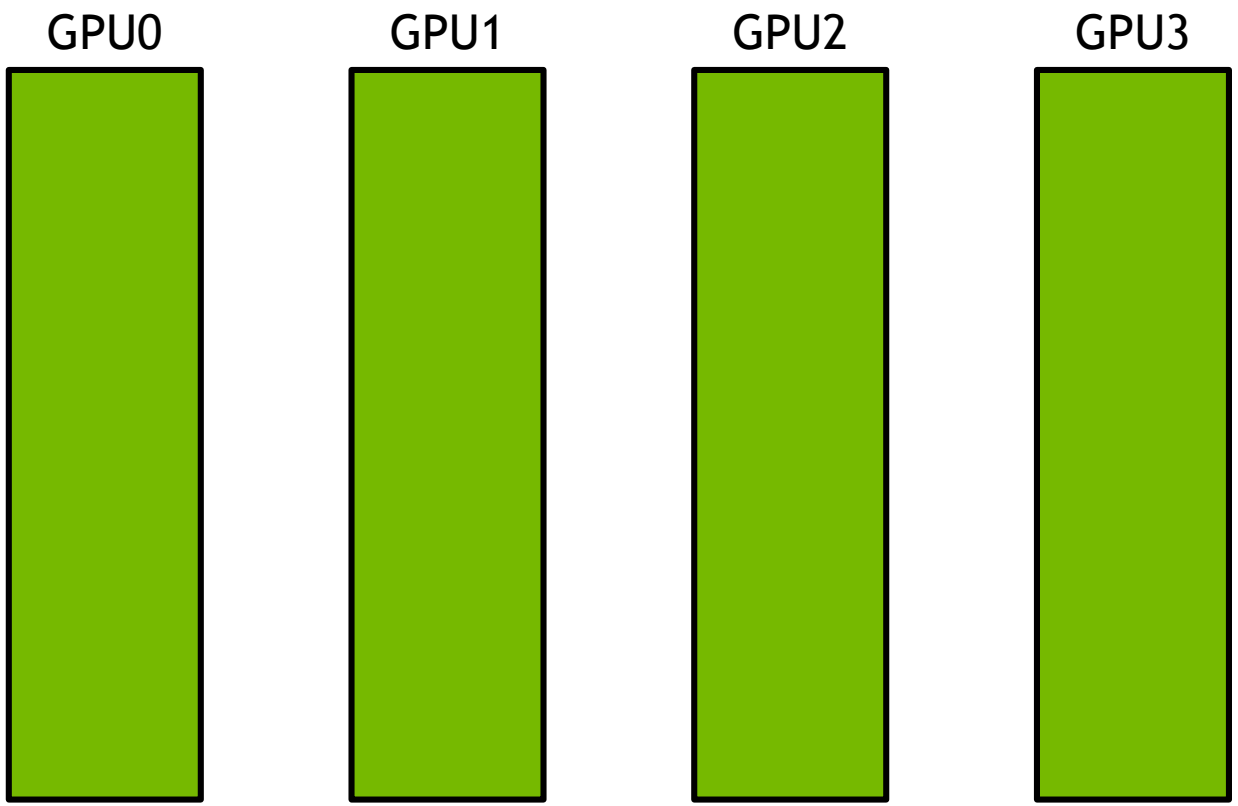
Step 3: $\Delta t = N/B$

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: $(k-1)N/B$

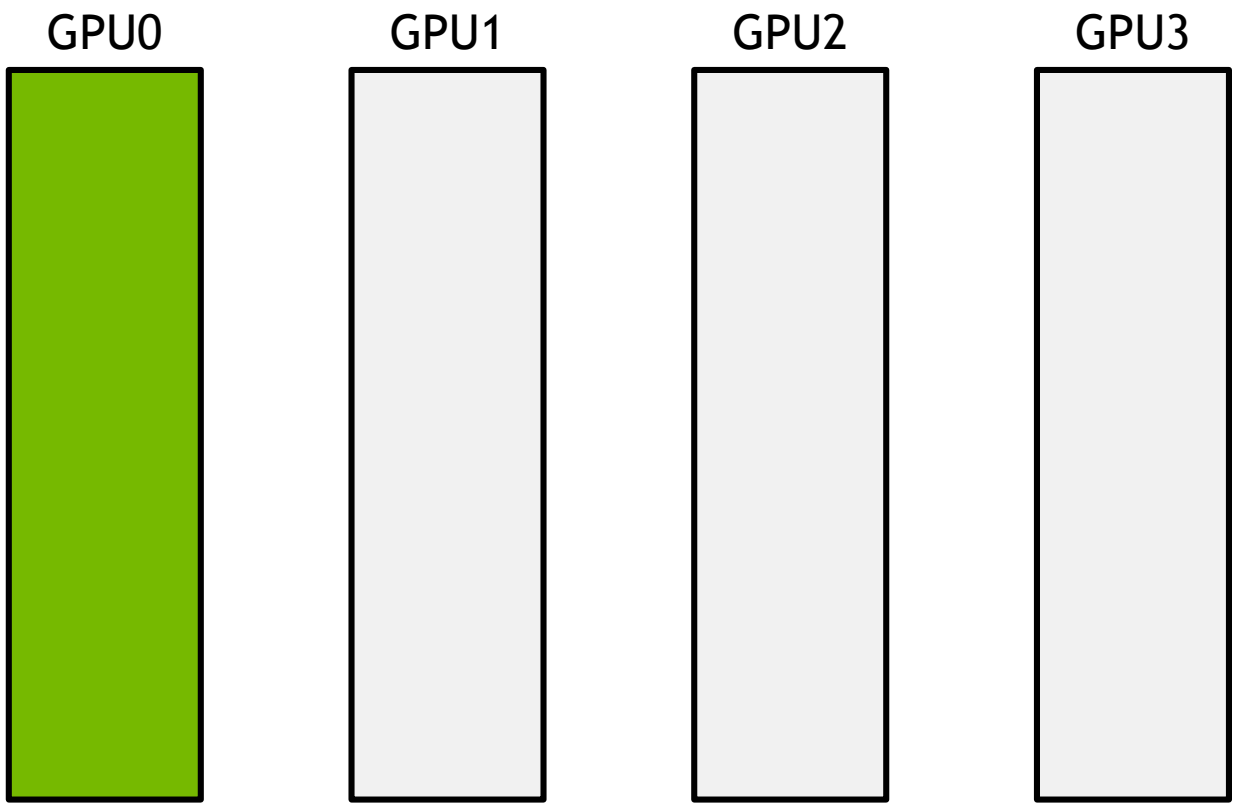
N: bytes to broadcast

B: bandwidth of each link

k: number of GPUs

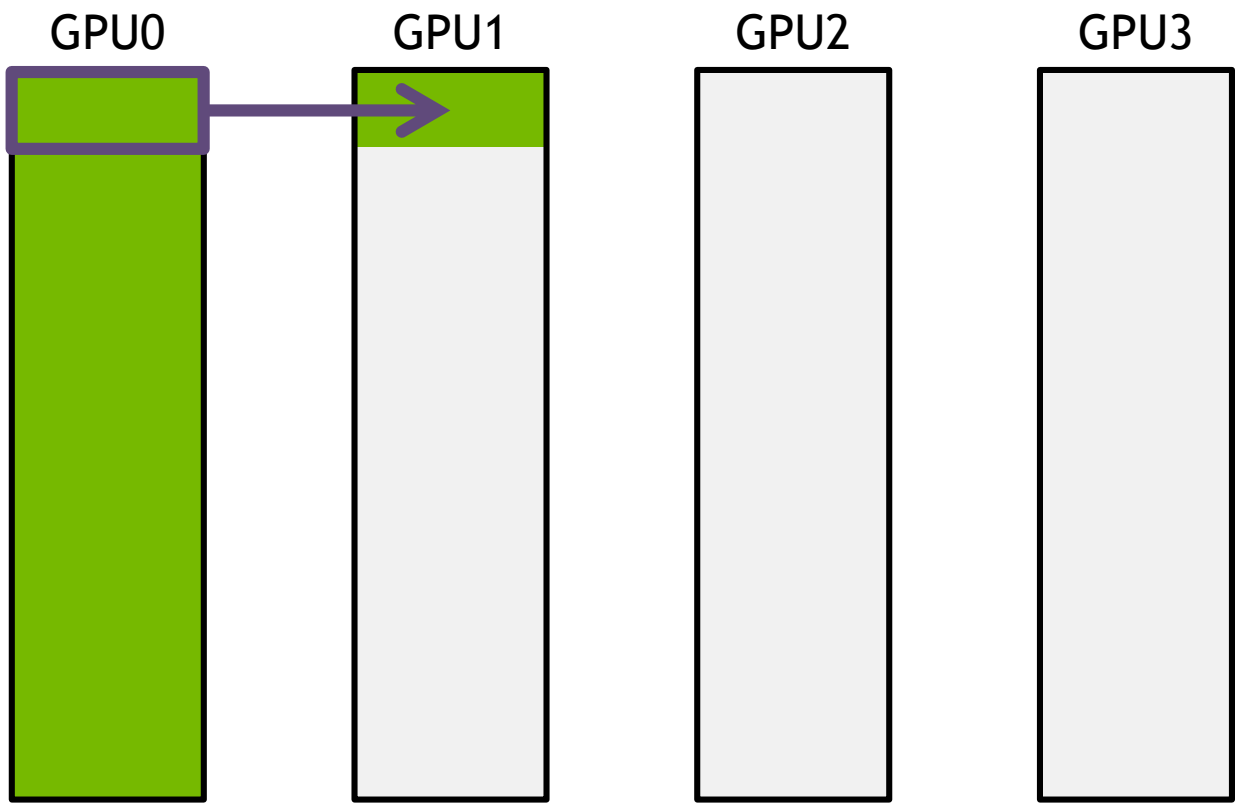
BROADCAST

with unidirectional ring



BROADCAST

with unidirectional ring

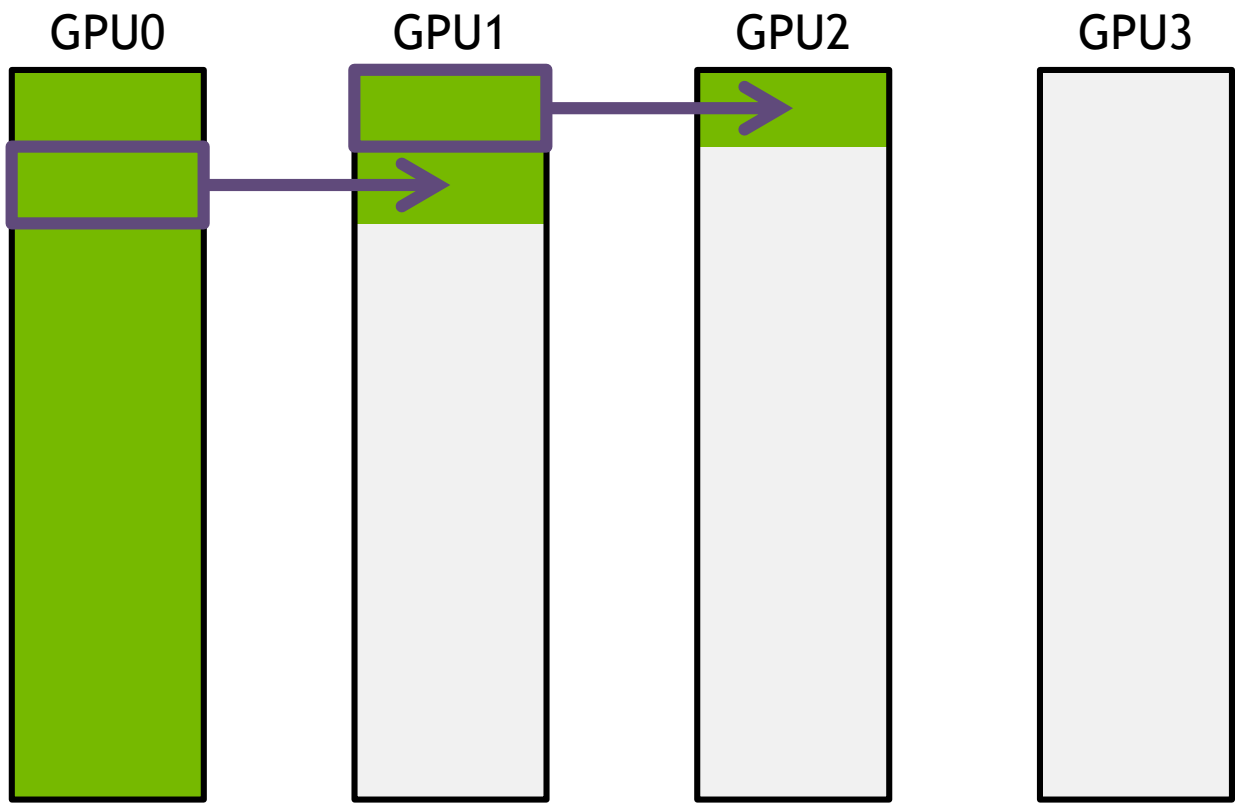


Split data into S messages

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

BROADCAST

with unidirectional ring



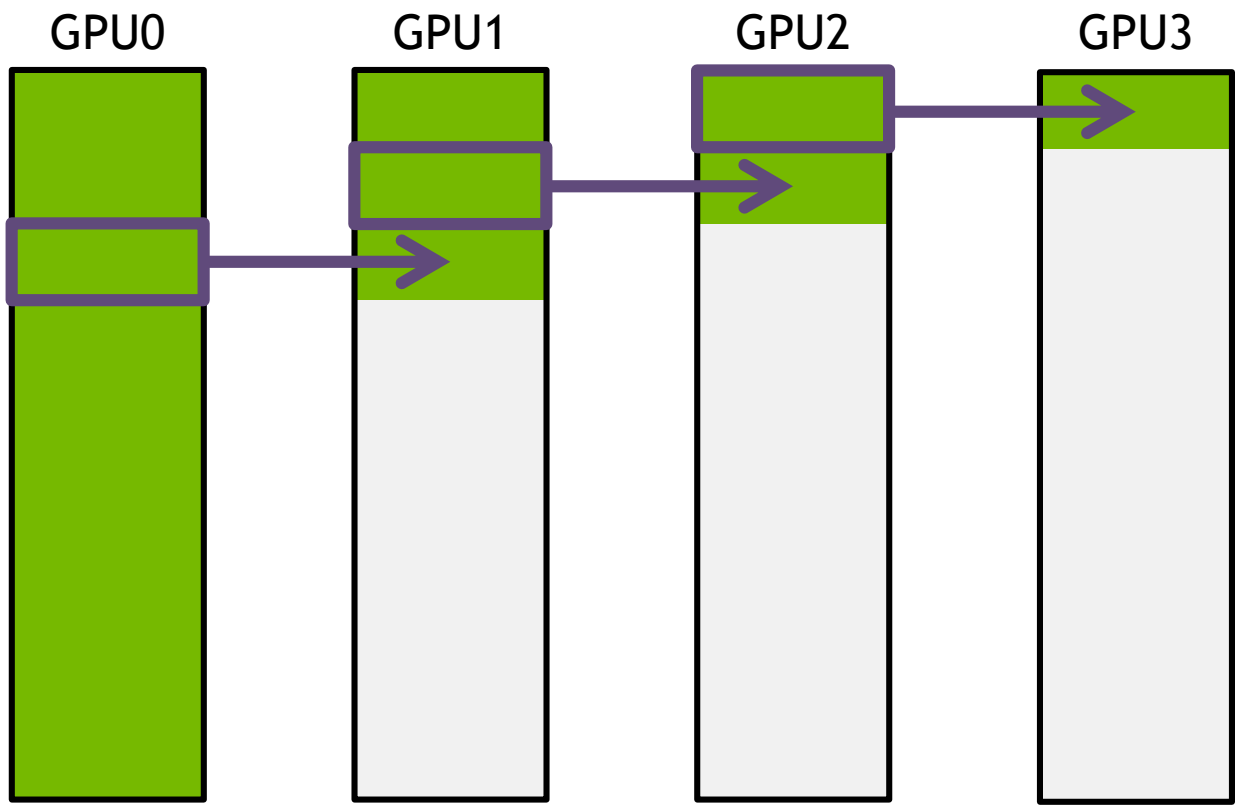
Split data into S messages

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

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

BROADCAST

with unidirectional ring



Split data into S messages

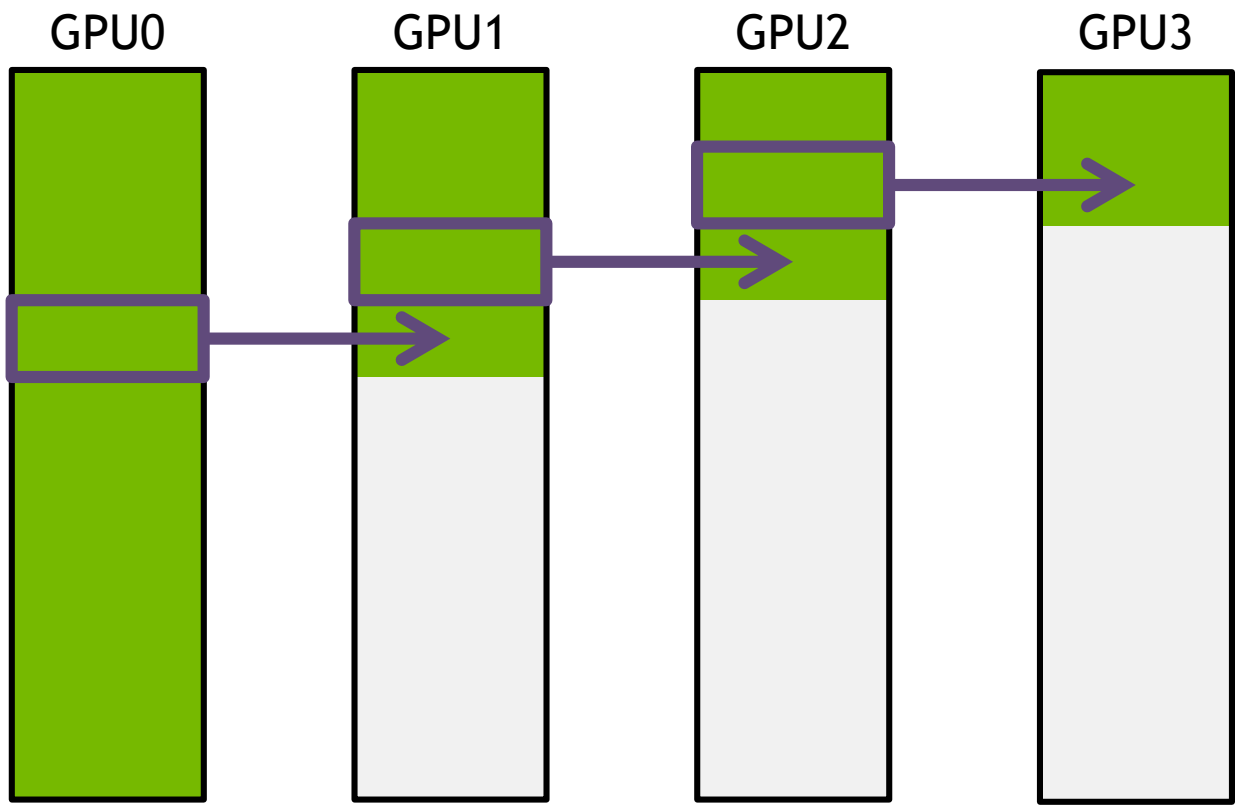
Step 1: $\Delta t = N/(SB)$

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

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

BROADCAST

with unidirectional ring



Split data into S messages

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

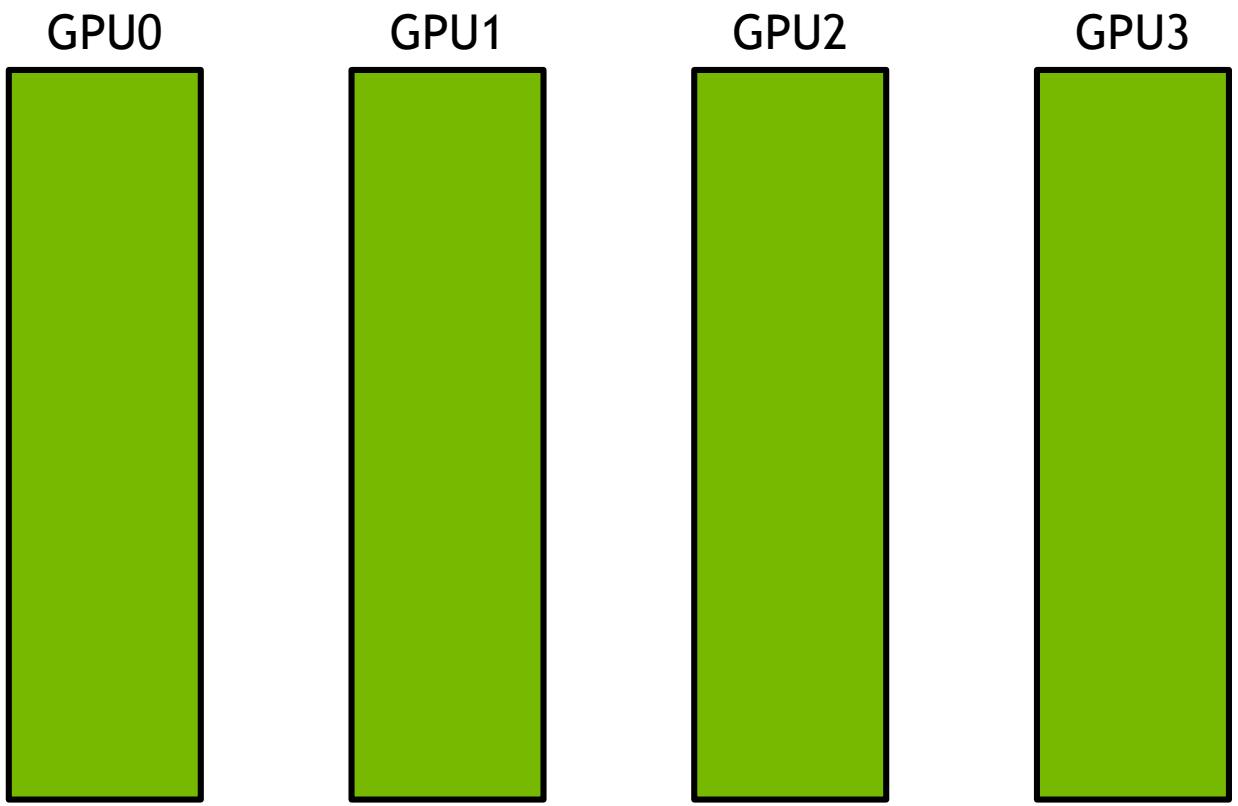
Step 2: $\Delta t = N/(SB)$

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

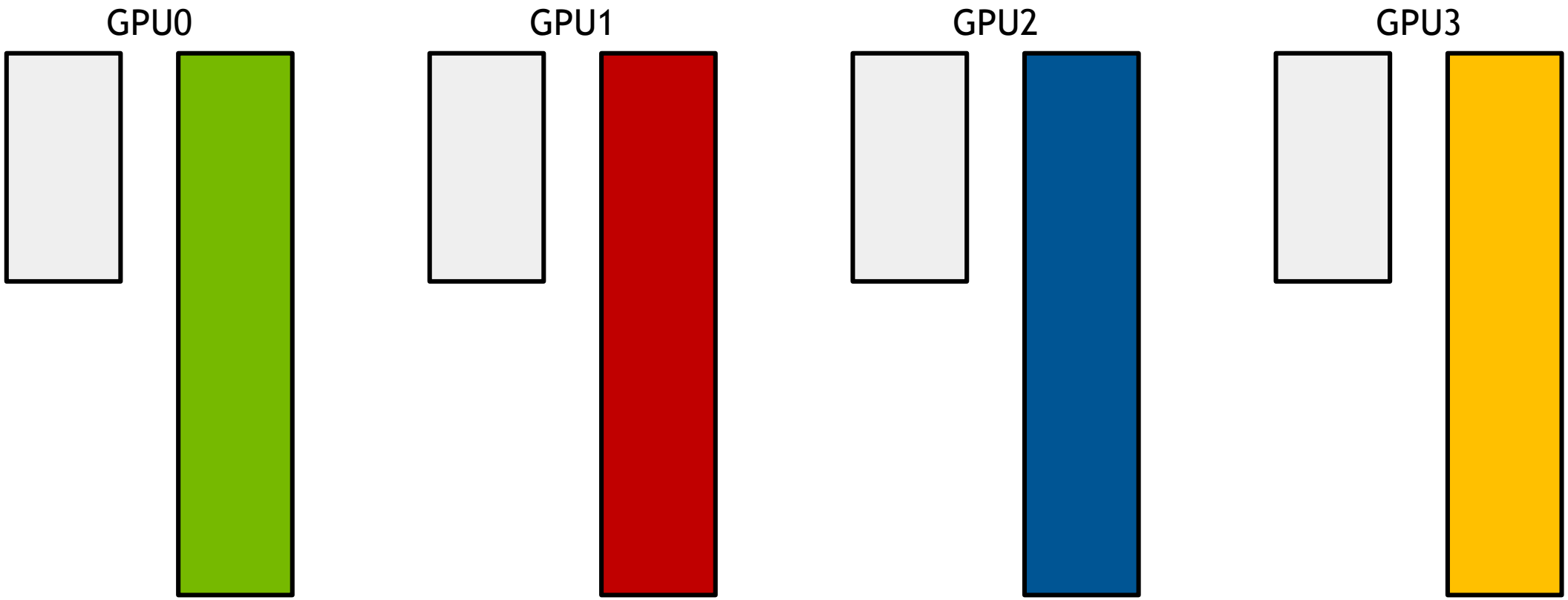
...

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

ALL-REDUCE

with unidirectional ring

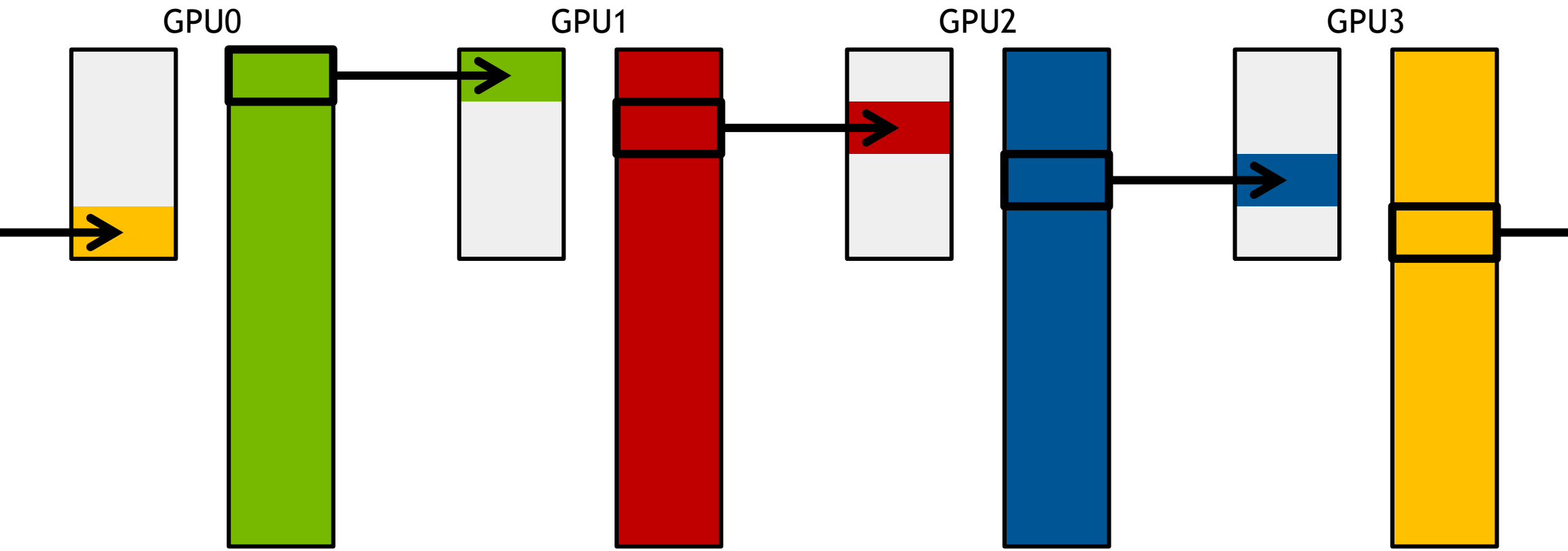
Chunk: 1
Step:



ALL-REDUCE

with unidirectional ring

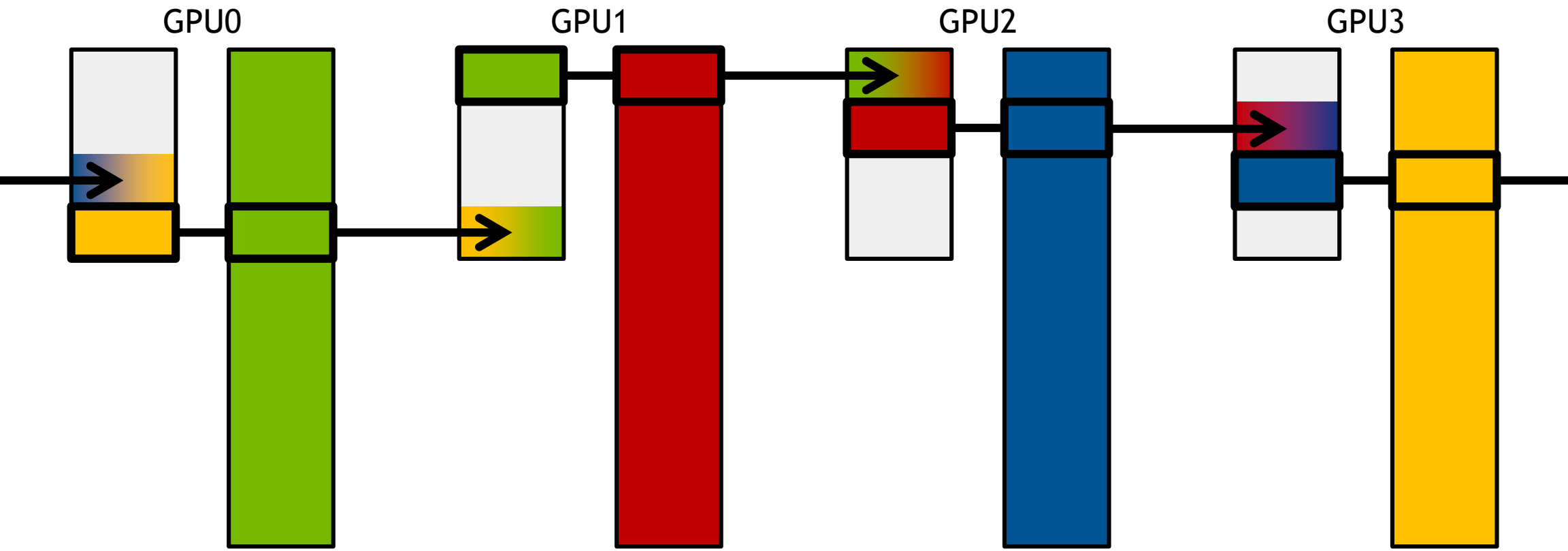
Chunk: 1
Step: 1



ALL-REDUCE

with unidirectional ring

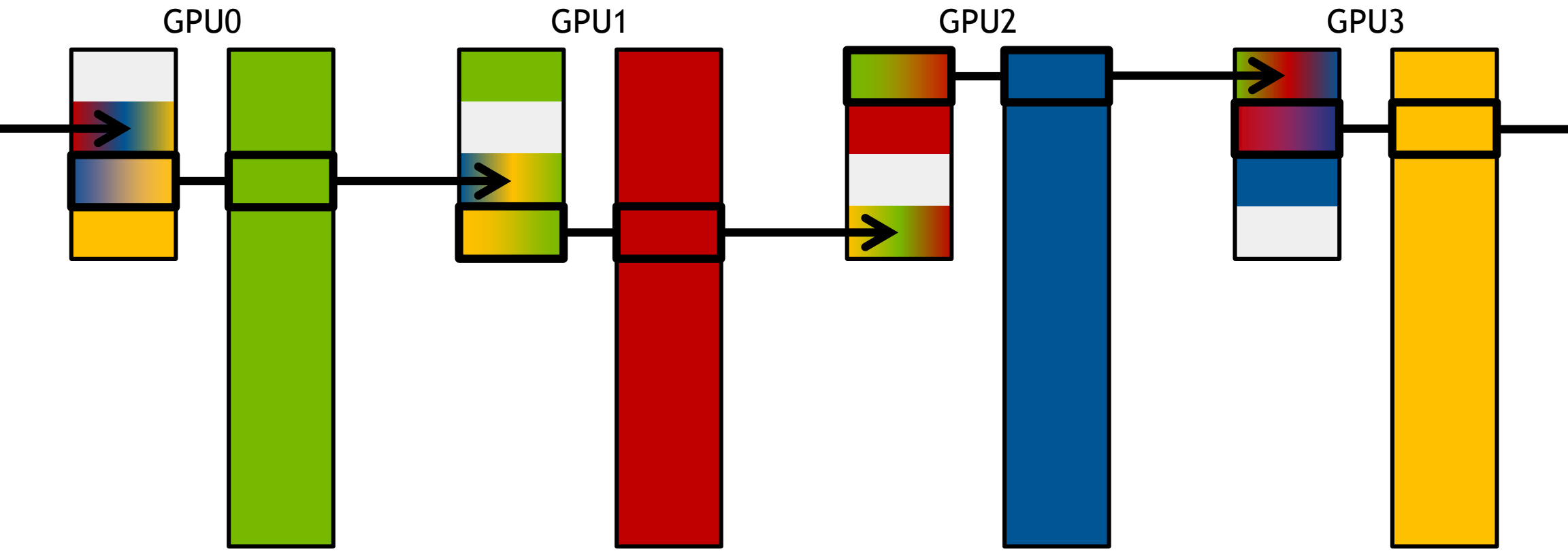
Chunk: 1
Step: 2



ALL-REDUCE

with unidirectional ring

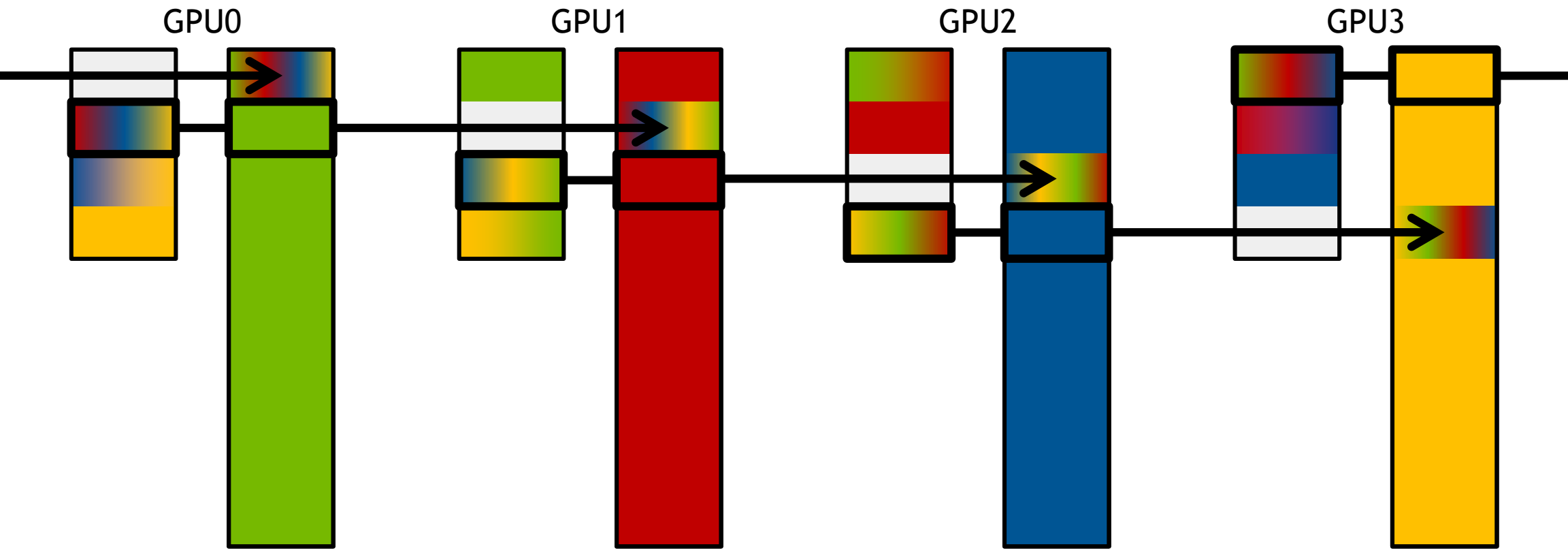
Chunk: 1
Step: 3



ALL-REDUCE

with unidirectional ring

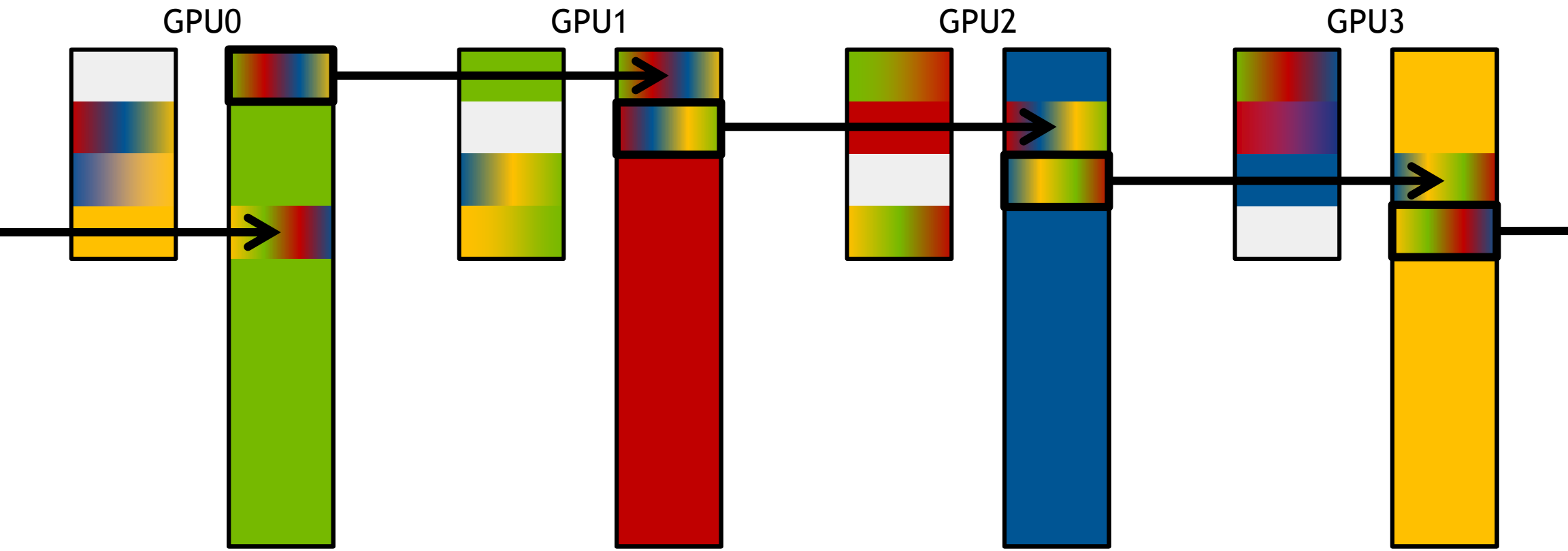
Chunk: 1
Step: 4



ALL-REDUCE

with unidirectional ring

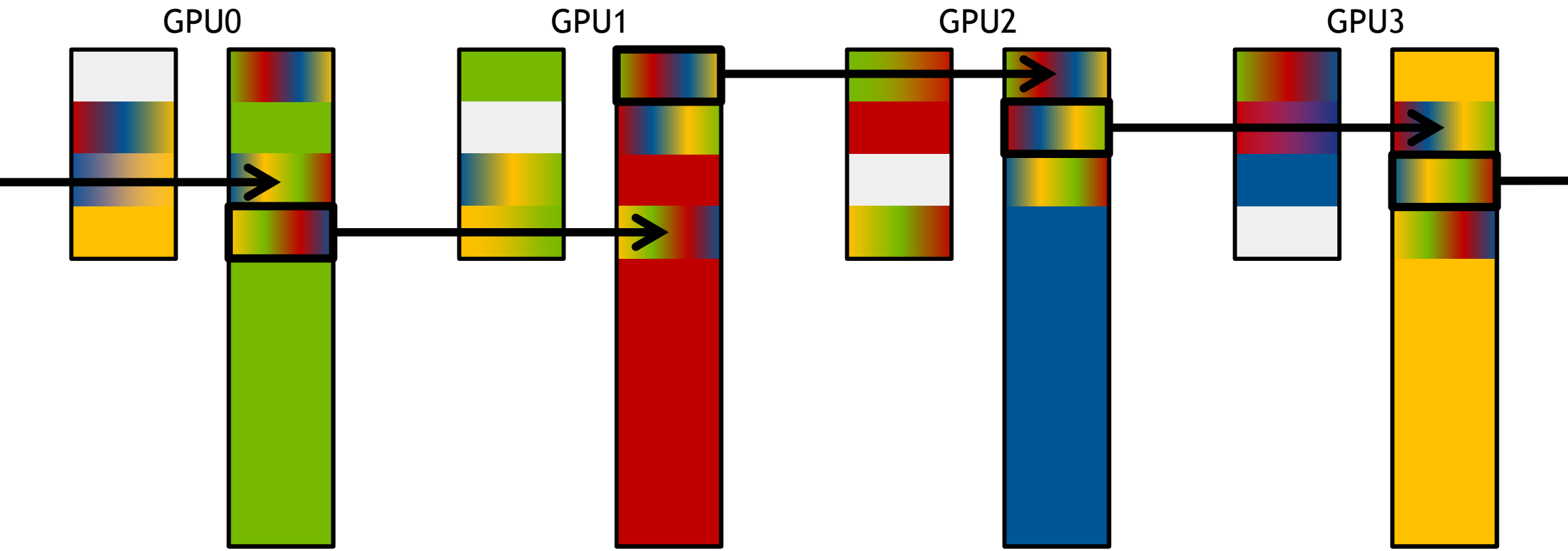
Chunk: 1
Step: 5



ALL-REDUCE

with unidirectional ring

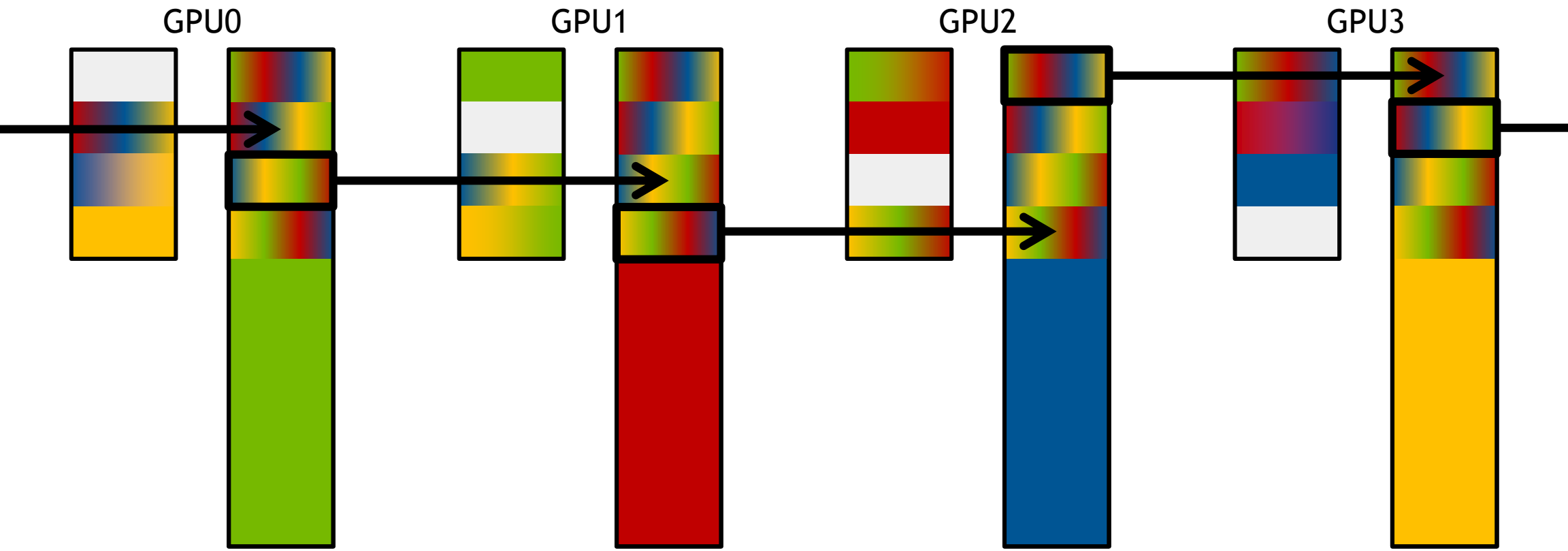
Chunk: 1
Step: 6



ALL-REDUCE

with unidirectional ring

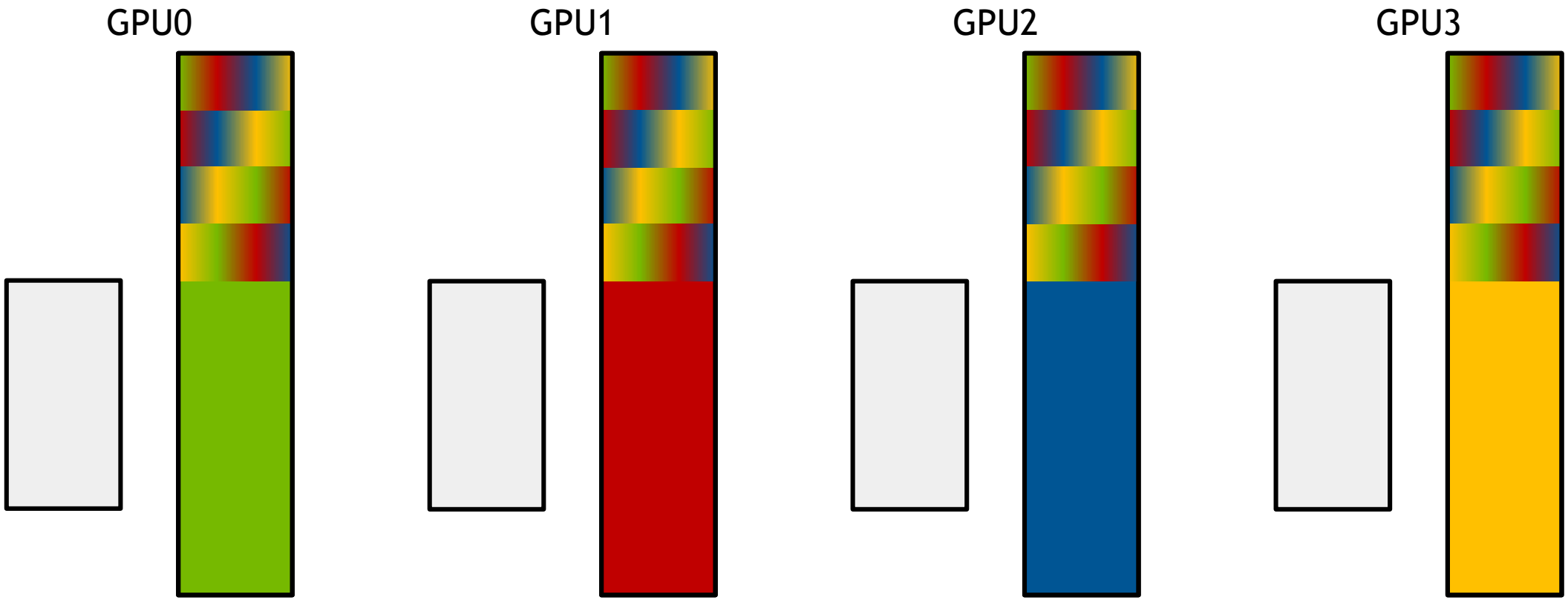
Chunk: 1
Step: 7



ALL-REDUCE

with unidirectional ring

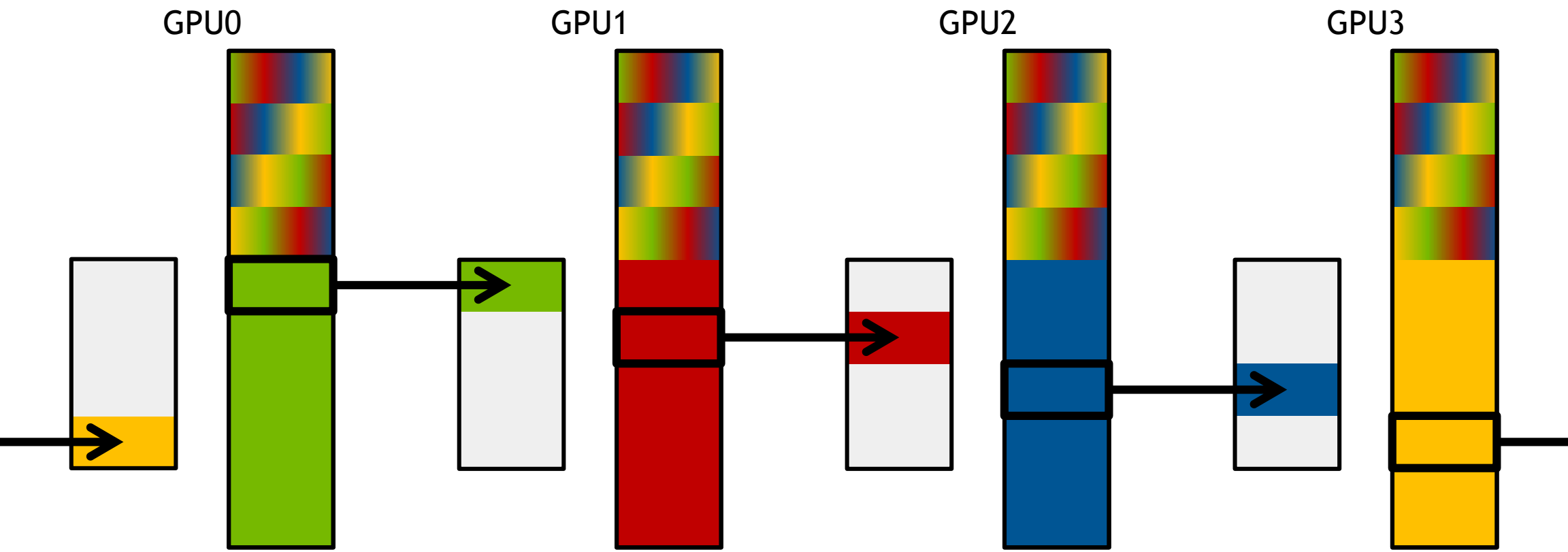
Chunk: 2
Step:



ALL-REDUCE

with unidirectional ring

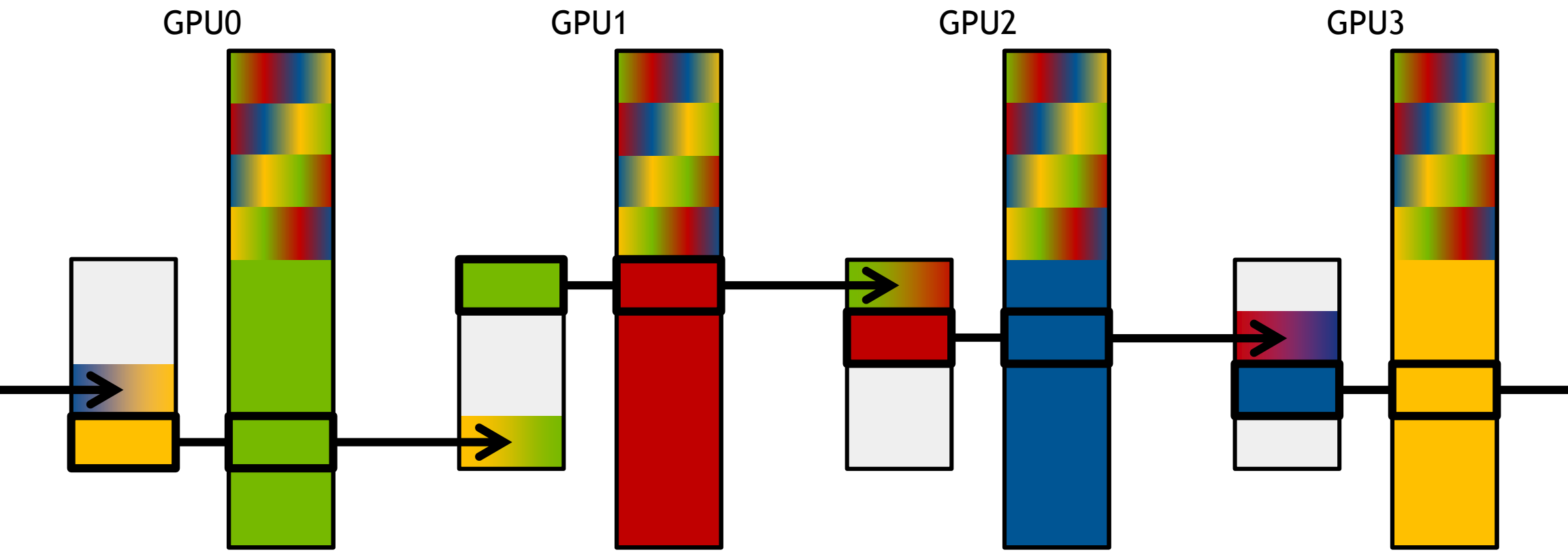
Chunk: 2
Step: 1



ALL-REDUCE

with unidirectional ring

Chunk: 2
Step: 2

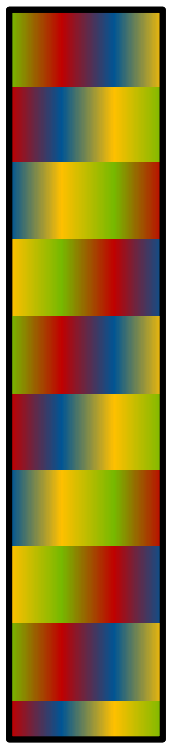


ALL-REDUCE

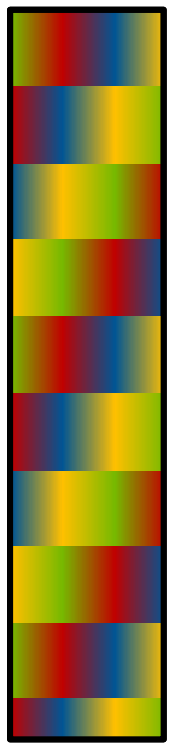
with unidirectional ring

done

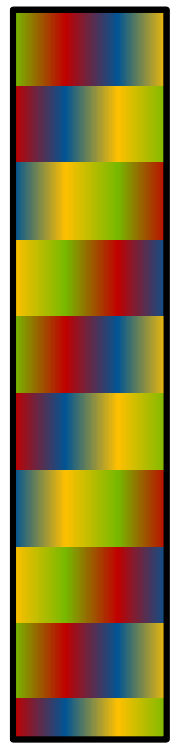
GPU0



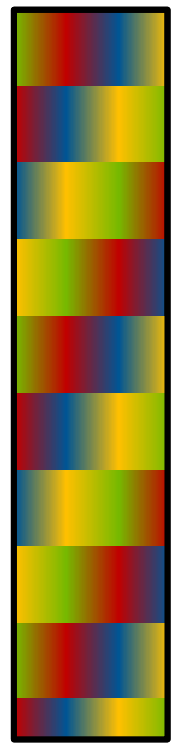
GPU1



GPU2

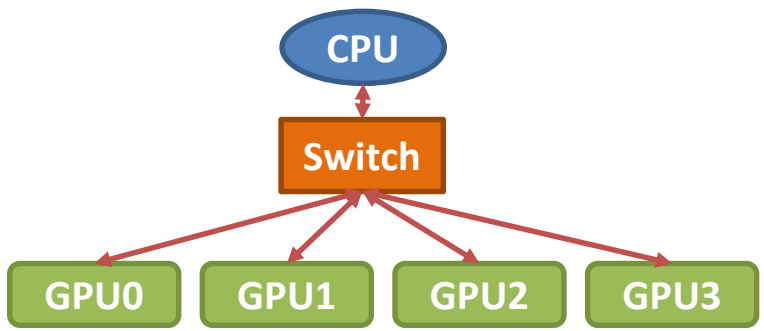


GPU3

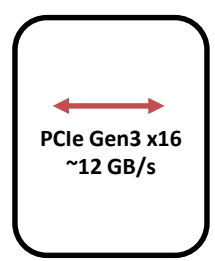


RING-BASED COLLECTIVES

A primer

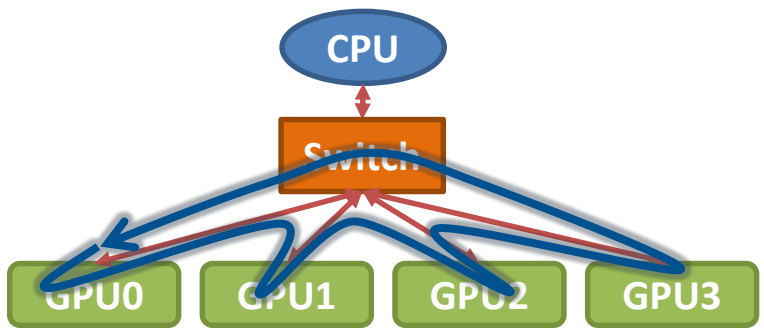


4-GPU-PCIe

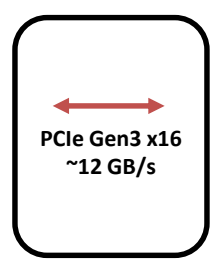


RING-BASED COLLECTIVES

A primer

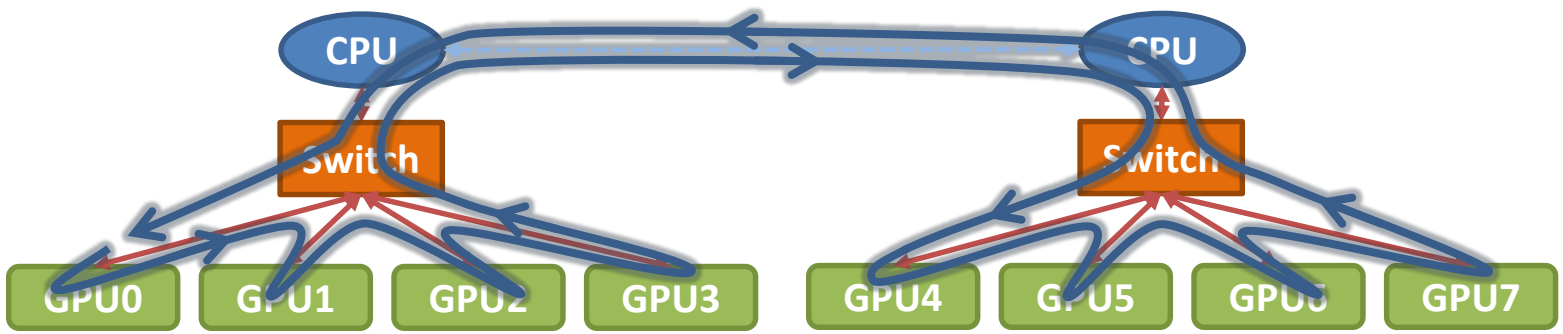


4-GPU-PCIe



RING-BASED COLLECTIVES

...apply to lots of possible topologies



SMP Connection
(e.g., QPI)

PCIe Gen3 x16
~12 GB/s

INTRODUCING NCCL (“NICKEL”): ACCELERATED COLLECTIVES FOR MULTI-GPU SYSTEMS

INTRODUCING NCCL

Accelerating multi-GPU collective communications

GOAL:

- Build a research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

APPROACH:

- Pattern the library after MPI's collectives
- Handle the intra-node communication in an optimal way
- Provide the necessary functionality for MPI to build on top to handle inter-node

NCCL FEATURES AND FUTURES

(Green = Currently available)

Collectives

- Broadcast
- All-Gather
- Reduce
- All-Reduce
- Reduce-Scatter
- Scatter
- Gather
- All-To-All
- Neighborhood

Key Features

- Single-node, up to 8 GPUs
- Host-side API
- Asynchronous/non-blocking interface
- Multi-thread, multi-process support
- In-place and out-of-place operation
- Integration with MPI
- Topology Detection
- NVLink & PCIe/QPI* support

NCCL IMPLEMENTATION

Implemented as monolithic CUDA C++ kernels combining the following:

- GPUDirect P2P Direct Access
- Three primitive operations: Copy, Reduce, ReduceAndCopy
- Intra-kernel synchronization between GPUs
- One CUDA thread block per ring-direction

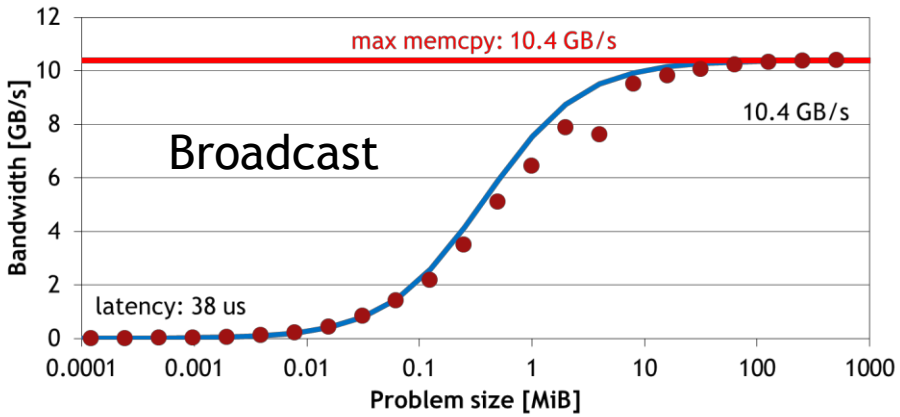
NCCL EXAMPLE

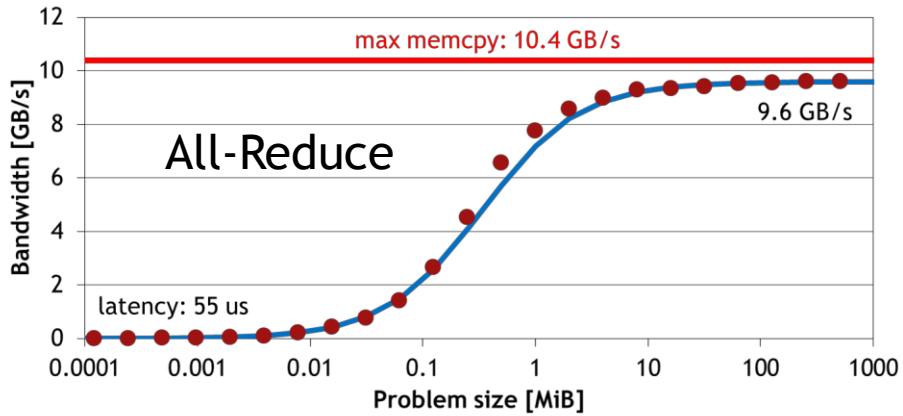
All-reduce

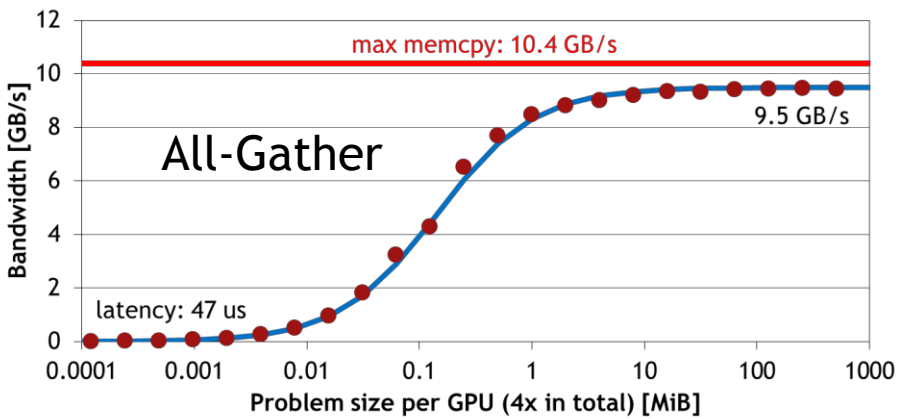
```
#include <nccl.h>
ncclComm_t comm[4];
ncclCommInitAll(comm, 4, {0, 1, 2, 3});
foreach g in (GPUs) { // or foreach thread
  cudaSetDevice(g);
  double *d_send, *d_recv;
  // allocate d_send, d_recv; fill d_send with data
  ncclAllReduce(d_send, d_recv, N, ncclDouble, ncclSum, comm[g], stream[g]);
  // consume d_recv
}
```

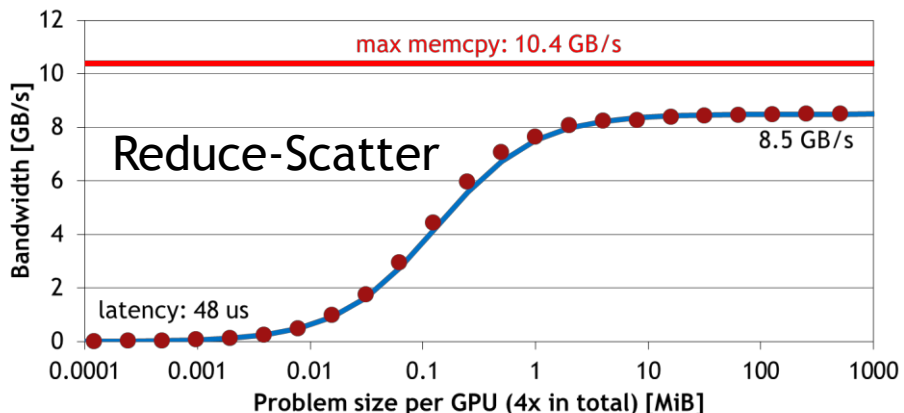
NCCL PERFORMANCE

Bandwidth at different problem sizes (4 Maxwell GPUs)









AVAILABLE NOW

github.com/NVIDIA/nccl