A Practical Guide to Communication Libraries on AMD GPUs

Aurelien Bouteiller
Corey Derochie
Edgar Gabriel
Nusrat Islam

Collectives.AI

AMD
together we advance_

Outline

- Introduction to AMD GPUs
- RCCL
- Break
- rocSHMEM
- UCX and Open MPI

Part 1: Introduction to AMD GPUs

Overview

- AMD Instinct™ GPUs overview
  - MI200 series
  - MI300 series
- AMD Workstation and Client GPUs
- ROCm software stack

CDNA vs RDNA Architecture

CDNA Design Philosophy

- **Compute-First**
  Maximizes die area for compute units and HBM. No display engine, no rasterization.

- **Matrix Cores**
  Hardware matrix multiply-accumulate units (Matrix Cores) for GEMM-heavy AI/ML and HPC.

- **Infinity Fabric**
  High-bandwidth inter-die and inter-GPU links enabling unified memory pools across GPUs.

- **HBM Memory**
  High-Bandwidth Memory stacked directly on package for maximum bandwidth and capacity.

- **ROCm Software**
  Open-source GPU computing stack: HIP, rocBLAS, rocFFT, MIOpen, PyTorch, JAX support.

RDNA Design Philosophy

- **Graphics-First**
  Full rasterization pipeline, display engine, and ray tracing hardware. Optimized for real-time rendering and visual output.

- **Ray Tracing Cores**
  Dedicated RT accelerator per CU (2nd-gen in RDNA 3, 3rd-gen inRDNA 4) for real-time ray/box and ray/triangle intersection.

- **PCIe based multi-GPU connectivity**

- **GDDR6 Memory**
  High-speed GDDR6 on a wide bus suited for graphics workloads.

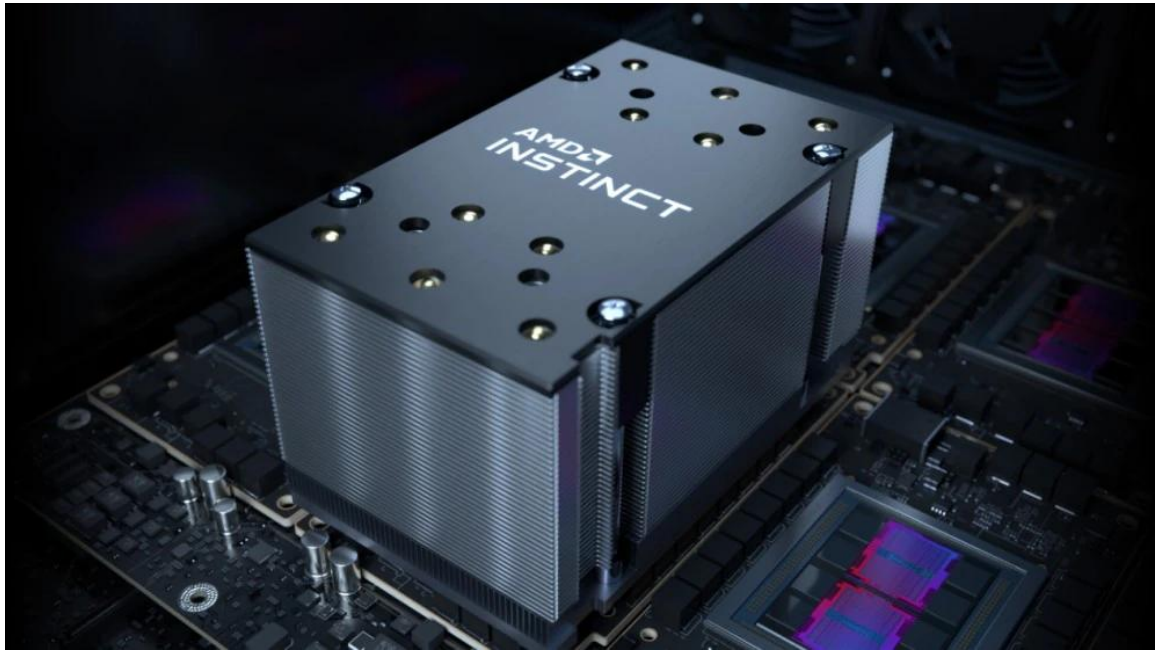
- **AI Accelerators**
  Dedicated AI engines per CU (2 per CU in RDNA 4) for inference, upscaling (FSR), and AI-assisted creative workflows.

- **Display & Software**
  Supports ROCm and Vulkan software stacks

AMD Instinct™ MI250X

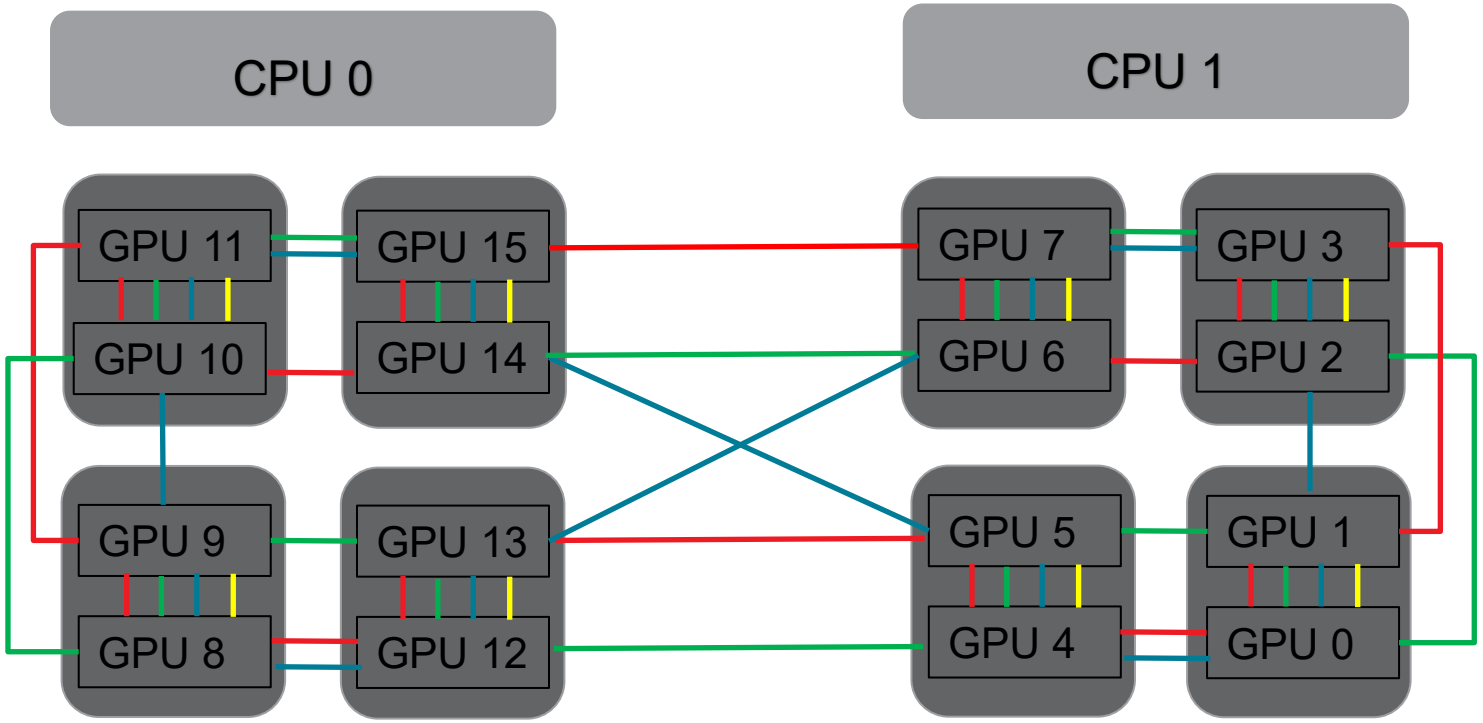
- CDNA 2 based architecture
- Used in ORNLs Frontier, Lumi, and many other systems in the Top 500
- Two Graphic Compute Dies (GCDs) on a single MI250X Chip

Feature	
Compute Units	220
Stream Processors	14,080
FP32 single / matrix peak performance	47.9 TFLOPS / 95.7 TFLOPS
FP64 single / matrix peak performance	47.9 TFLOPS / 95.7 TFLOPS
Memory Size	128 GB
Peak Memory Bandwidth	3276.9 GB/s



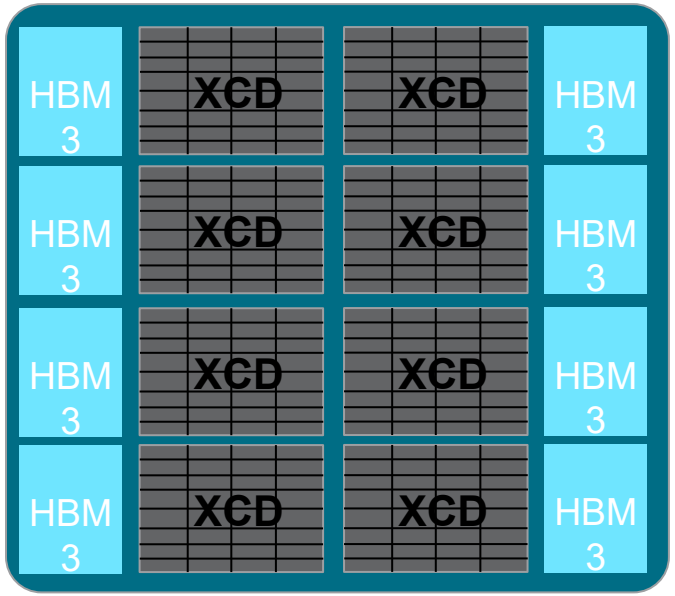
Source: https://www.amd.com/en/products/server-accelerators/instinct-mi250x

AMD Instinct™ MI250X Multi-GPU configuration



AMD Instinct™ MI300X and MI300A accelerators

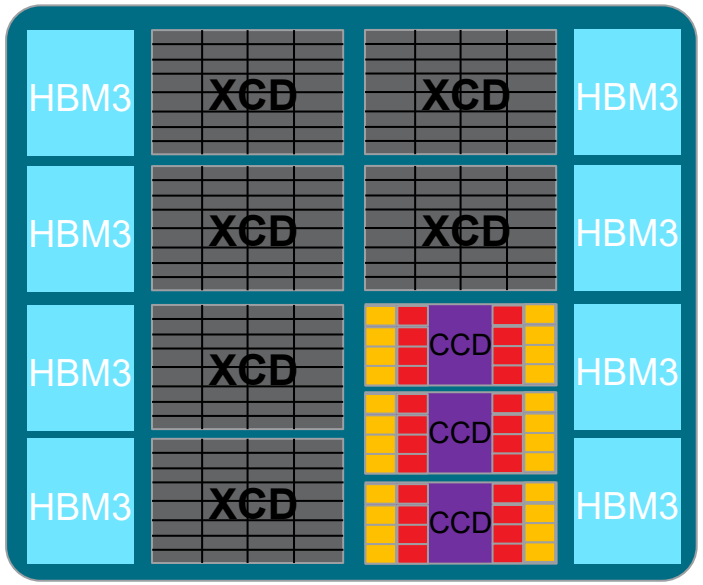
MI300X



Key-specs:

- 304 Compute Units
- 1216 Matrix Cores
- 192 GB HBM3 memory
- 5.3 TB/s memory bandwidth

MI300A



Key-specs:

- 228 Compute Units
- 912 Matrix Cores
- 128 GB HBM3 memory
- 5.3 TB/s memory bandwidth
- 24 ‘Zen 4’ CPU cores

AMD Instinct™ MI300X: Compute and Memory Partitioning

Compute Partition Modes (CPM) — 8 XCDs logically divided into independent GPU instances

SPX	DPX	QPX	CPX
Single Partition X-celerator	**Dual Partition X-celerator**	**Quad Partition X-celerator**	**Core Partition X-celerator**
1 logical GPU 8 XCDs unified 304 CUs 192 GB HBM (default mode)	2 logical GPUs 4 XCDs each 152 CUs each 96 GB HBM each	4 logical GPUs 2 XCDs each 76 CUs each 48 GB HBM each	8 logical GPUs 1 XCD each 38 CUs each 24 GB HBM each
8 XCDs	**4 | 4**	**2|2|2|2**	**1×8**

Memory Partition Modes (NPS — NUMA Per Socket) — Controls number of HBM NUMA domains exposed

NPS1 — 1 NUMA Domain

192 GB unified
All HBM interleaved across all 8 stacks

Large models needing full unified VRAM; default for most HPC

NPS2 — 2 NUMA Domains

96 GB per domain
4 HBM stacks each

Compatible: QPX, CPX

Dual-partition workloads; better memory locality per partition pair

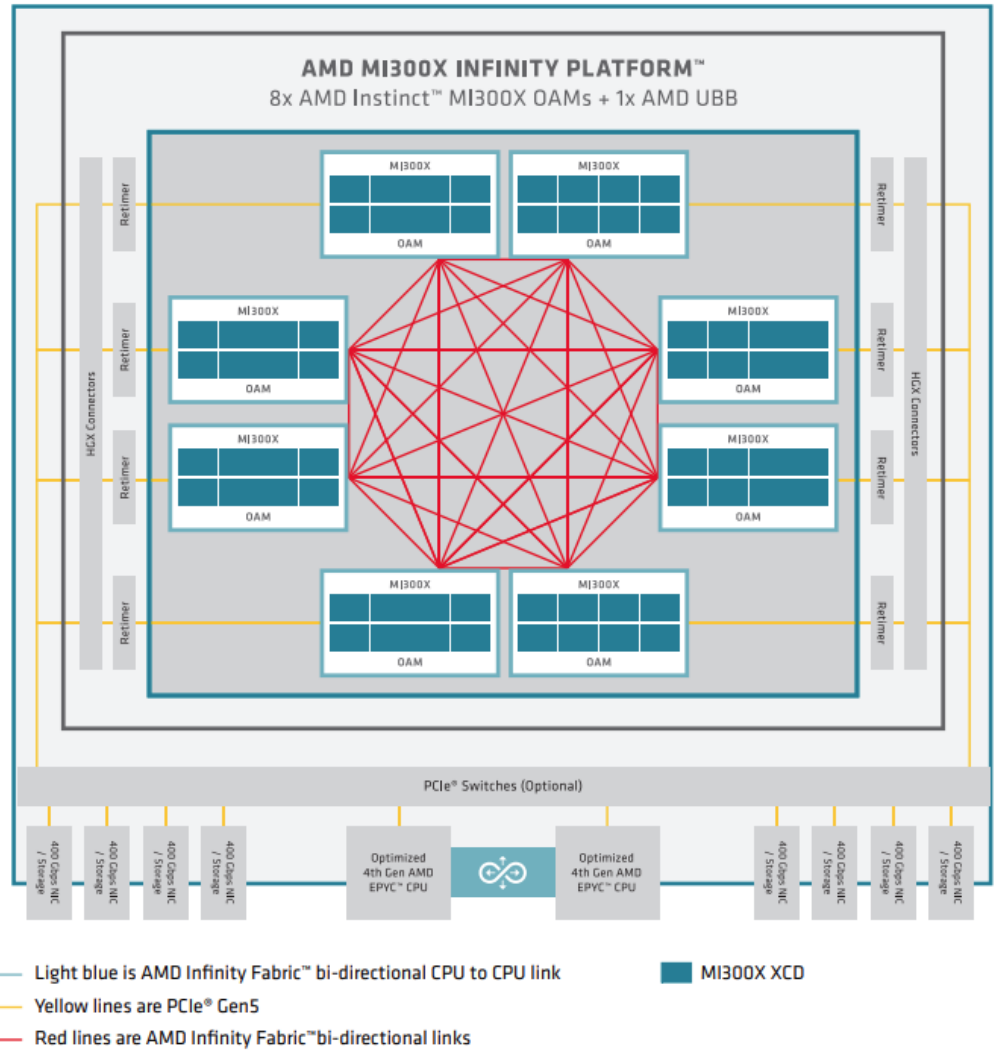
Compute partitioning can be changed at runtime with the amd-smi tool
Certain limitations for combinations of Compute and Memory partitioning exist

AMD Instinct™ MI300X: Compute and Memory Partitioning (II)

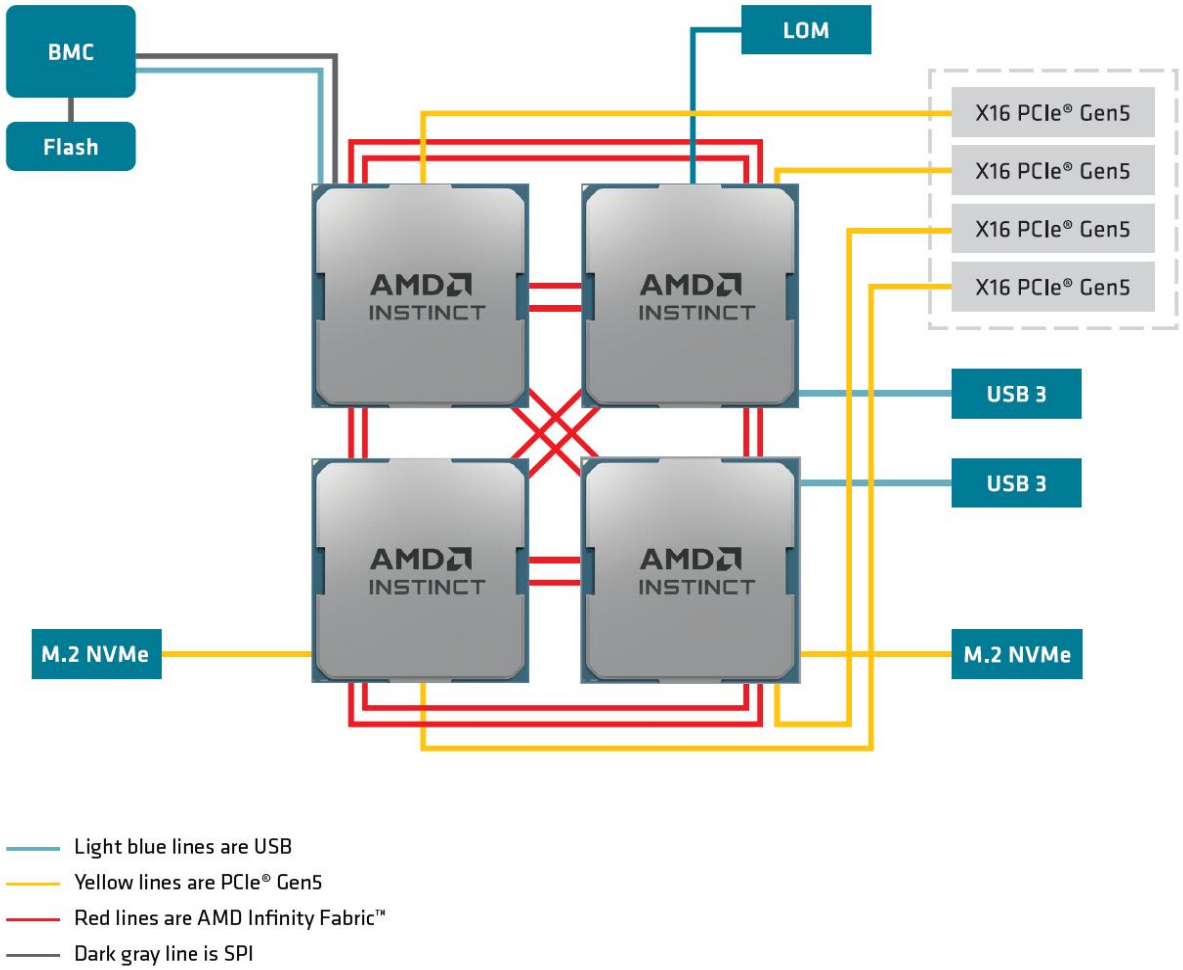
```
$ amd-smi partition
CURRENT_PARTITION:
GPU_ID  MEMORY  ACCELERATOR_TYPE  ACCELERATOR_PROFILE_INDEX  PARTITION_ID
0       NPS1    SPX               0                          0
1       NPS1    SPX               0                          0
2       NPS1    SPX               0                          0
3       NPS1    SPX               0                          0
4       NPS1    SPX               0                          0
5       NPS1    SPX               0                          0
6       NPS1    SPX               0                          0
7       NPS1    SPX               0                          0

MEMORY_PARTITION:
GPU_ID  MEMORY_PARTITION_CAPS  CURRENT_MEMORY_PARTITION
0       NPS1,NPS2             NPS1
1       NPS1,NPS2             NPS1
2       NPS1,NPS2             NPS1
3       NPS1,NPS2             NPS1
4       NPS1,NPS2             NPS1
5       NPS1,NPS2             NPS1
6       NPS1,NPS2             NPS1
7       NPS1,NPS2             NPS1
```

AMD Instinct™ MI300X node configuration

- 8 MI300X GPUs on a single UUB Board
- 1 direct InfinityFabric™ Link to each other GPU



Example AMD Instinct™ MI300A node configuration



AMD Instinct™ MI350 Series: CDNA 4 Architecture

MI350X

Cooling: Air-cooled | TBP: ~1,000–1,200W

Broader deployment, standard OCP racks

MI355X

Cooling: Liquid-cooled | TBP: 1,400W

Maximum performance, hyperscale / HPC

MI350 Series — Shared Specifications

- **Architecture:** CDNA 4
- **Memory:** 288 GB | 8.0 TB/s bandwidth
- **Precision:** FP4, FP6, FP8, BF16, FP16, FP32, FP64 — FP4/FP6 NEW in CDNA 4
- **Interconnect:** 7× InfinityFabric™ links

AMD InfinityFabric™ Link Bandwidth by GPU Generation

GPU	IF Gen	Lane Rate	Sustained Uni BW (per link)	Links to each peer
MI250X (CDNA 2)	3rd Gen	25 GT/s	**~36 GB/s**	1, 2, or 4 (topology dependent)
MI300X (CDNA 3)	4th Gen	32 Gbps/lane	**~48 GB/s**	1
MI325X (CDNA 3)	4th Gen	32 Gbps/lane	**~48 GB/s**	1
MI350X (CDNA 4)	5th Gen	~38.4 Gbps/lane	**~60 GB/s**	1
MI355X (CDNA 4)	5th Gen	~38.4 Gbps/lane	**~60 GB/s**	1

AMD Radeon™ Workstation & Gaming GPUs

Radeon Pro W7900 — Professional Workstation GPU (RDNA 3)

96 CUs | 6,144 SPs | 48 GB GDDR6 | 864 GB/s | 61 TFLOPS FP32 | 295 W | RDNA 3 / Navi 31

▸ Professional 3D & Visualization

ISV-certified for SolidWorks, Maya, Houdini, CATIA, ParaView and 100+ apps. 3-slot design with 4× DisplayPort 2.1

▸ Massive VRAM for large datasets

48 GB GDDR6 holds full simulation outputs, large BIM models, and volumetric medical imaging datasets in GPU memory without paging.

▸ RDNA 3 Chiplet design

GCD with 384 MB Infinity Cache. 2nd-gen ray tracing and dedicated AI accelerators per CU.

▸ Compute + display on one card

Full OpenCL/HIP compute alongside display output — eliminates the need for a separate display GPU in mixed visualization + simulation workflows.

Radeon AI PRO R9700 — AI Workstation GPU (RDNA 4)

64 CUs | 4,096 SPs | 32 GB GDDR6 | 640 GB/s | 47.8 TFLOPS FP32 | 300 W | RDNA 4 / Navi 48

▸ Purpose-built for local AI inference

128 second-gen AI Accelerators deliver 1,531 TOPS INT4 (sparse) — enabling local inference of 32–70B quantized LLMs (Llama 3, DeepSeek) fully in VRAM.

▸ RDNA 4: next-gen architecture

3rd-gen ray tracing, FP8 matrix (383 TFLOPS dense / 766 TFLOPS sparse), and full ROCm + PyTorch support.

▸ AI + display in one card

4× DisplayPort 2.1 outputs — run interactive AI-assisted design tools, real-time upscaling (FSR 4), and GPU-accelerated video production simultaneously.

▸ 4-GPU scalability

Driver-level support for up to 4× R9700 per workstation → 128 GB pooled GDDR6. PCIe 5.0 for maximum host-to-GPU bandwidth.

Large BAR Mapping in AMD GPUs

What Is a BAR?

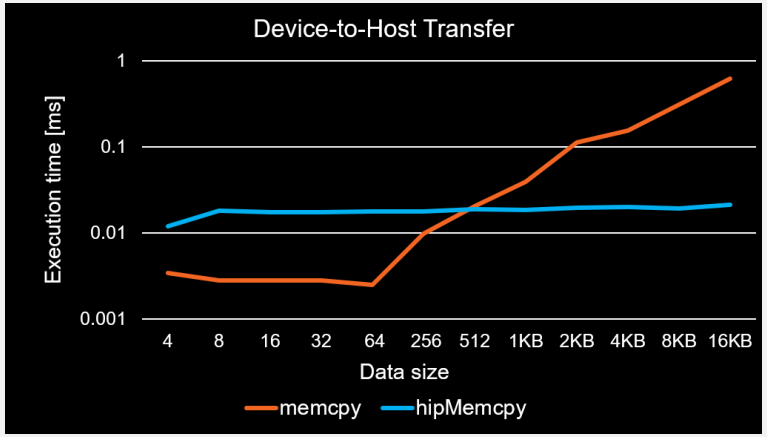
A Base Address Register (BAR) is a PCIe mechanism that maps a device's memory (e.g., GPU VRAM) into the CPU's address space via Memory-Mapped I/O (MMIO). The CPU accesses GPU memory by reading and writing to this mapped region.

The Default BAR Problem

Traditional PCIe limits the CPU-visible VRAM window to 256 MB, regardless of how much VRAM the GPU has. To access data beyond this window, the OS must remap the BAR — an expensive operation incurring 5–10 µs of latency per remap.

Why Large BAR Matters

- memory allocated with hipMalloc() can be directly accessed from CPU side, e.g. using memcpy()
- memcpy vs hipMemcpy() have different performance characteristics
  ⟹ can be exploited for performance optimizations

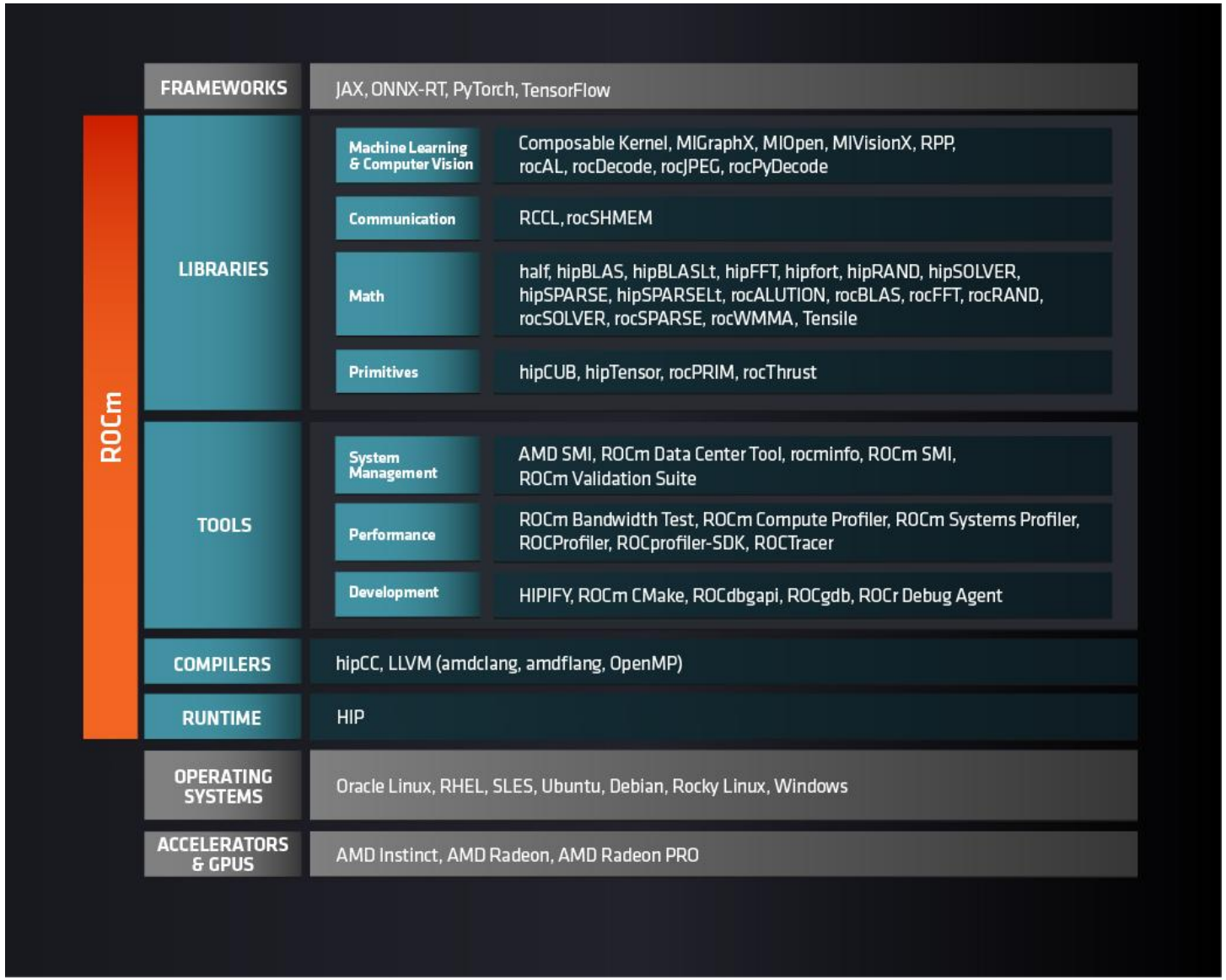


- Larbe BAR is always enabled on Instinct™ class GPUs, but might have to be enabled manually in BIOS for Workstation and Client class GPUs

BIOS Setting / Feature	Required State
Above 4G Decoding	Enabled
Resizable BAR (ReBAR)	Enabled

Source: AMD ROCm Docs — BAR Memory; AMD MI300X System Optimization Guide

AMD ROCm Open-Source GPU Computing Stack



Introduction to HIP: Heterogeneous-compute Interface for Portability

What is HIP?

HIP (Heterogeneous-compute Interface for Portability) is a C++ runtime API and kernel language that allows developers to write GPU code that runs on both AMD and NVIDIA GPUs from a single source.

Key design goals

- Single source — one codebase, two GPU vendors
- CUDA-familiar syntax — minimal learning curve for CUDA developers
- Open source — part of AMD ROCm (github.com/ROCm/HIP)
- Portable runtime API — streams, events, memory, peer access

HIP Programming Model: Kernels, Execution & Key Differences

Execution Hierarchy

Grid	All blocks for a kernel launch
Block (CTA)	Shared LDS; __syncthreads()
Wavefront	64 threads on Instinct™ GPUs 32 threads on Radeon™ GPUs
Thread	threadIdx.x/y/z — a single SIMD lane

Always query warpSize at runtime for portable code — do not hardcode 64 or 32.

Essential HIP API Categories

Memory	hipMalloc / hipFree / hipMemcpy / hipMemset
Kernels	<<<grid,block,smem,stream>>> or hipLaunchKernelGGL()
Streams	hipStreamCreate / hipStreamSynchronize / hipStreamDestroy
Events	hipEventCreate / hipEventRecord / hipEventElapsedTime
Peer access	hipDeviceEnablePeerAccess
Device info	hipGetDeviceProperties → warpSize, sharedMemPerBlock …

HIP Code Sample

```
// HIP kernel — identical syntax to CUDA __global__
__global__ void saxpy(float a, float* x, float* y, int n) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
       if (i < n)  y[i] = a * x[i] + y[i];
}

int main() {
    float *dx, *dy;
    hipMalloc(&dx, n * sizeof(float));     // allocate GPU memory
    hipMalloc(&dy, n * sizeof(float));
    hipMemcpy(dx, hx, n*sizeof(float),     // H→D transfer
                      hipMemcpyHostToDevice);

    // Launch: 64 threads/block = 1 AMD wavefront
    dim3 block(64);
    dim3 grid((n + 63) / 64);
    saxpy<<<grid, block>>>(a, dx, dy, n);
    // or: hipLaunchKernelGGL(saxpy, grid, block, 0, 0, a, dx, dy);

    hipDeviceSynchronize();                // wait for GPU
    hipMemcpy(hy, dy, n*sizeof(float),     // D→H transfer
                      hipMemcpyDeviceToHost);
    hipFree(dx); hipFree(dy);
}
```

AMD GPU Portfolio: Key Takeaways

1. Two GPU Families for Different Needs

CDNA (Instinct™): compute-only, HBM, Infinity Fabric — for AI training, large-scale inference, HPC.
RDNA (Radeon™): display + compute, GDDR6, visualization, hybrid workflows.

2. CDNA 3 → MI300X / MI325X: Unified APU Option + Massive Memory

MI300X integrates 8 XCDs + HBM3 on one package (192 GB).
MI325X upgrades to 256 GB HBM3E — both share the same CDNA 3 ISA and ROCm ecosystem for drop-in compatibility.

3. CDNA 4 → MI350X/MI355X: Next Frontier in AI Performance

Newest generation of AMD Datacenter GPUs. FP4/FP6 precision for ultra-efficient inference, 288 GB HBM3E. Significant performance improvements over MI300X.

4. Radeon Pro W7900/ 9700 AI PRO: Professional Visualization + AI

RDNA 3/4 architecture with 48/32 GB GDDR6, DP 2.1, AI accelerators and hardware ray tracing. The bridge between compute and professional graphics.

5. Open Software Ecosystem: ROCm & HIP

AMD's open-source ROCm stack provides PyTorch, TensorFlow, JAX, and HIP support. Runs across the entire AMD GPU portfolio.

Part 2: ROCm Collective Communication Library (RCCL)

Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

Introduction

- Collective communication involves a group of processes
  - Point-to-point communication has one-to-one communication between two processes
  - Collective communication supports one-to-many, many-to-one, many-to-many patterns among a group of processes (can be more than two)

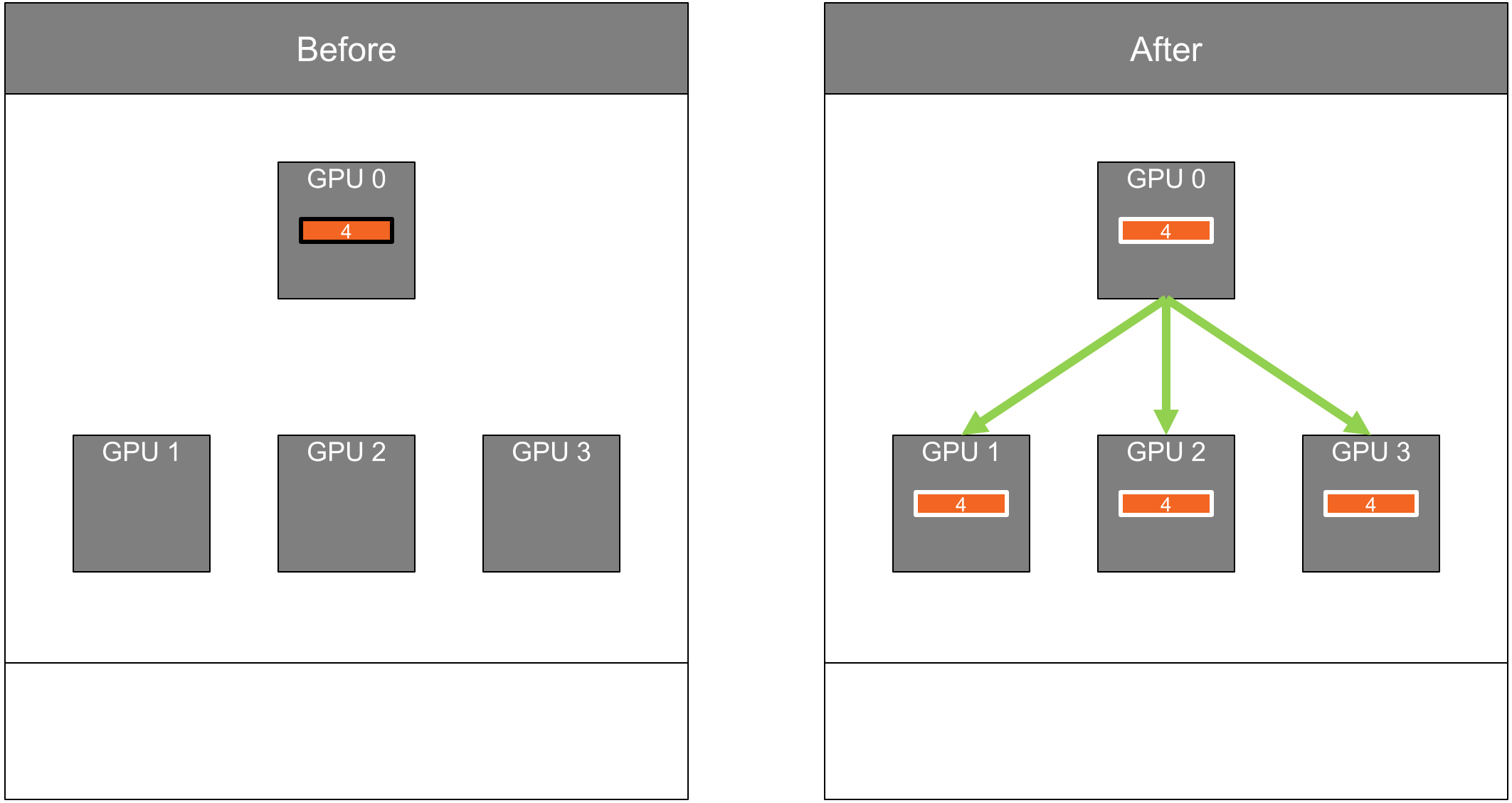
- Message Passing Interface (MPI) library supports collective communications
  - Other communication libraries such as, RCCL, NCCL, oneCCL also have support for collectives

- Collective communication is a critical part of HPC applications
  - DL/ML applications require collectives for distributed training and inference

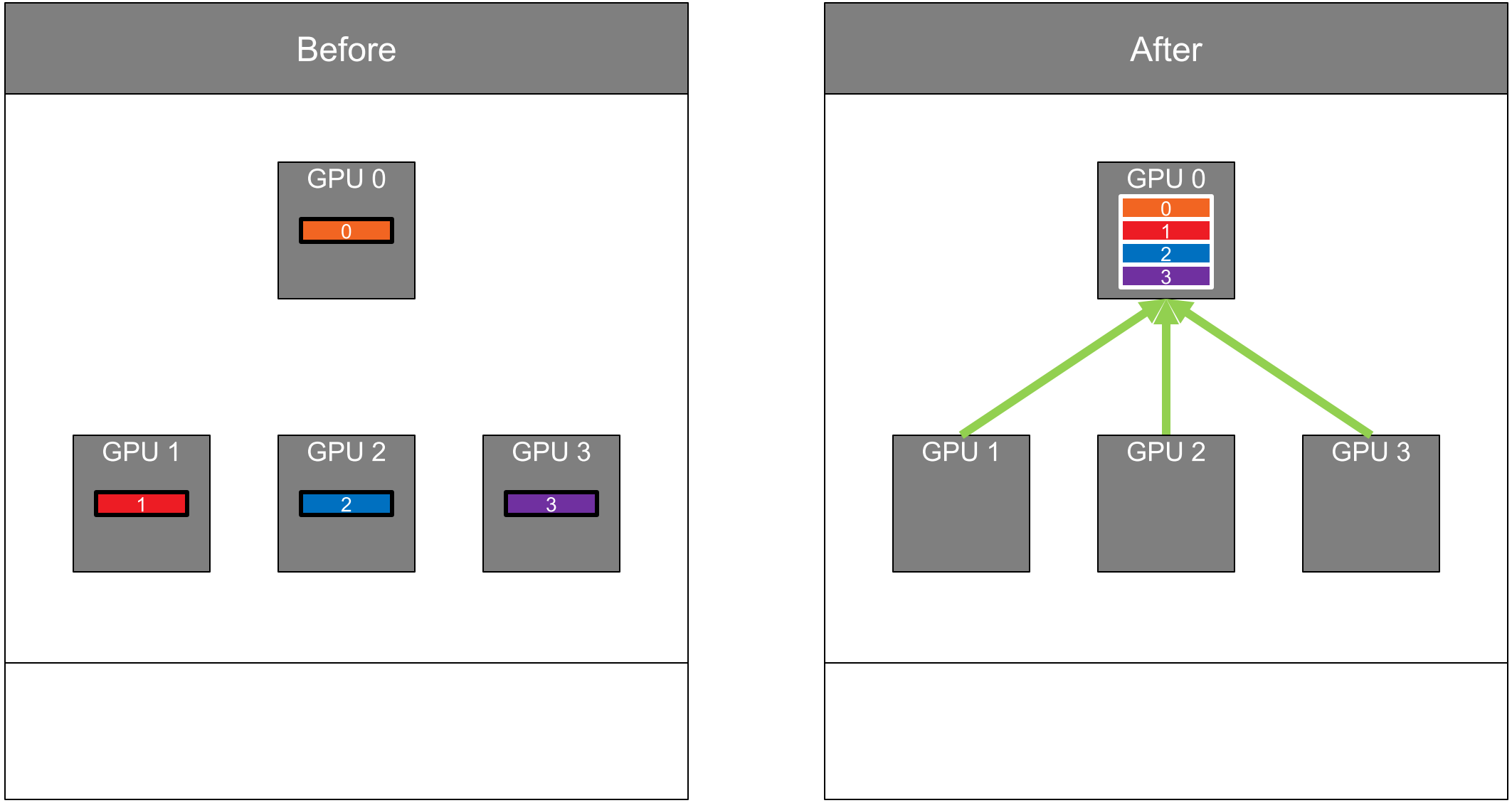
Overview

- One to all
  - Broadcast, Scatter
- All to one
  - Reduce, Gather
- All to all
  - Allreduce, Alltoall, Allgather, Barrier

BROADCAST

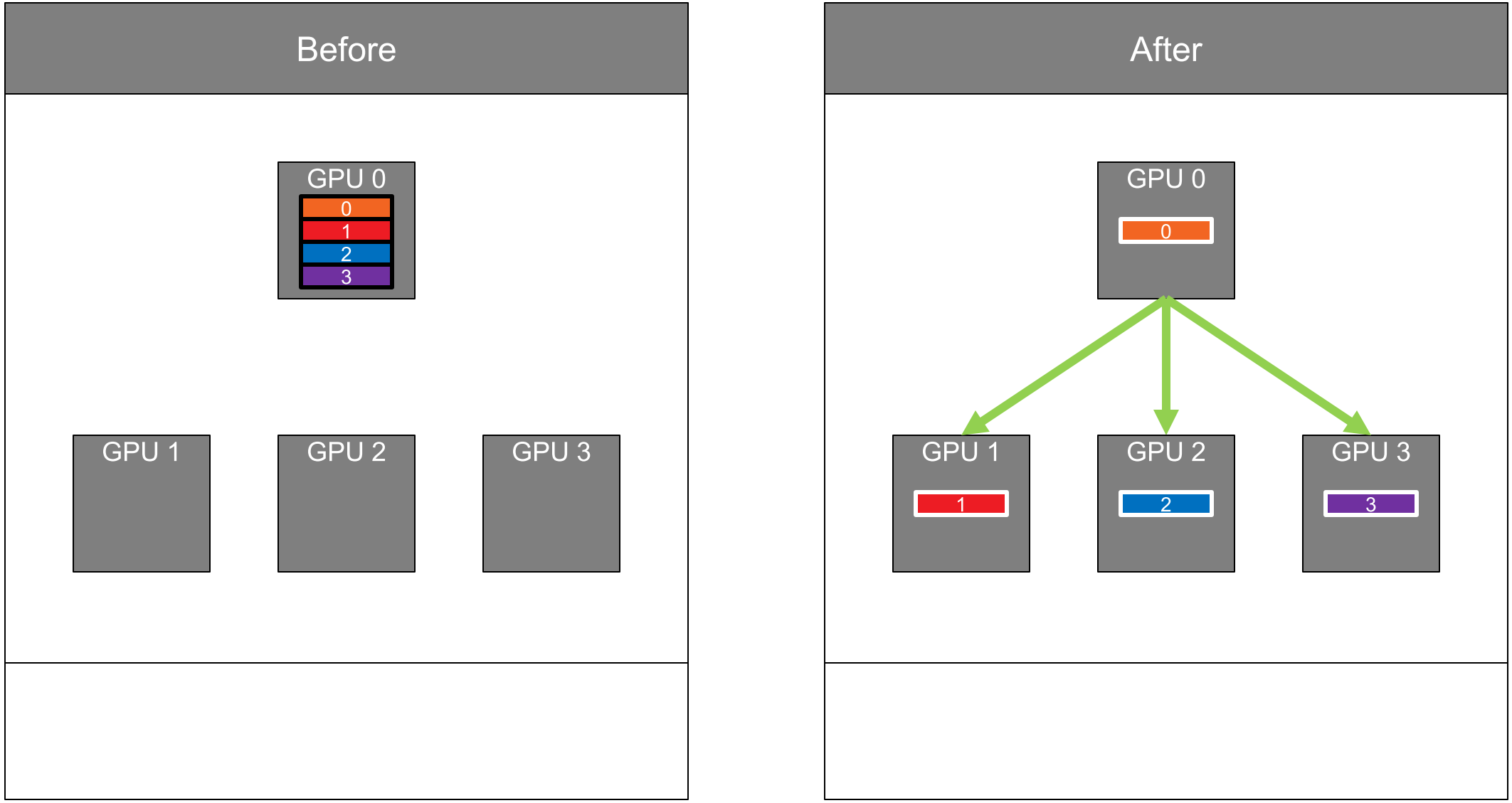


GATHER

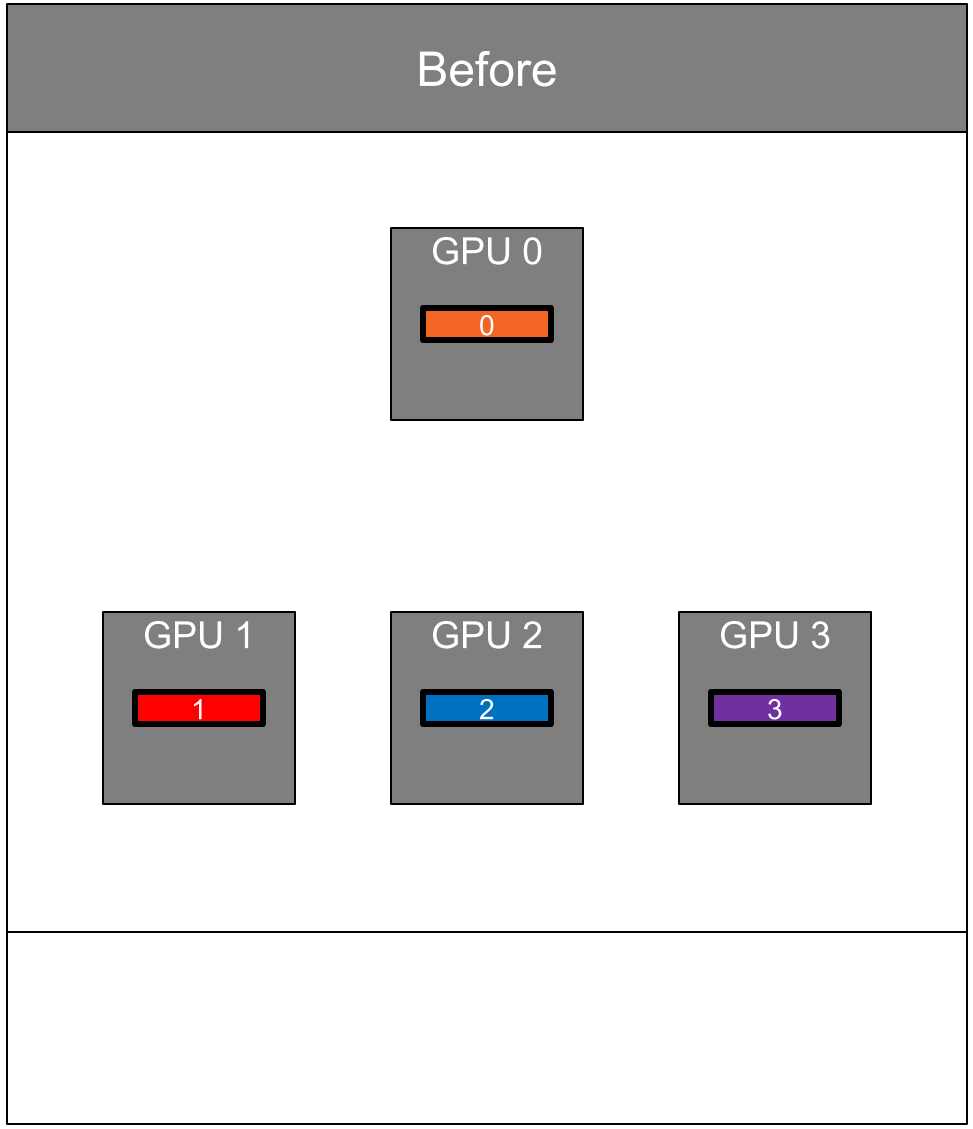


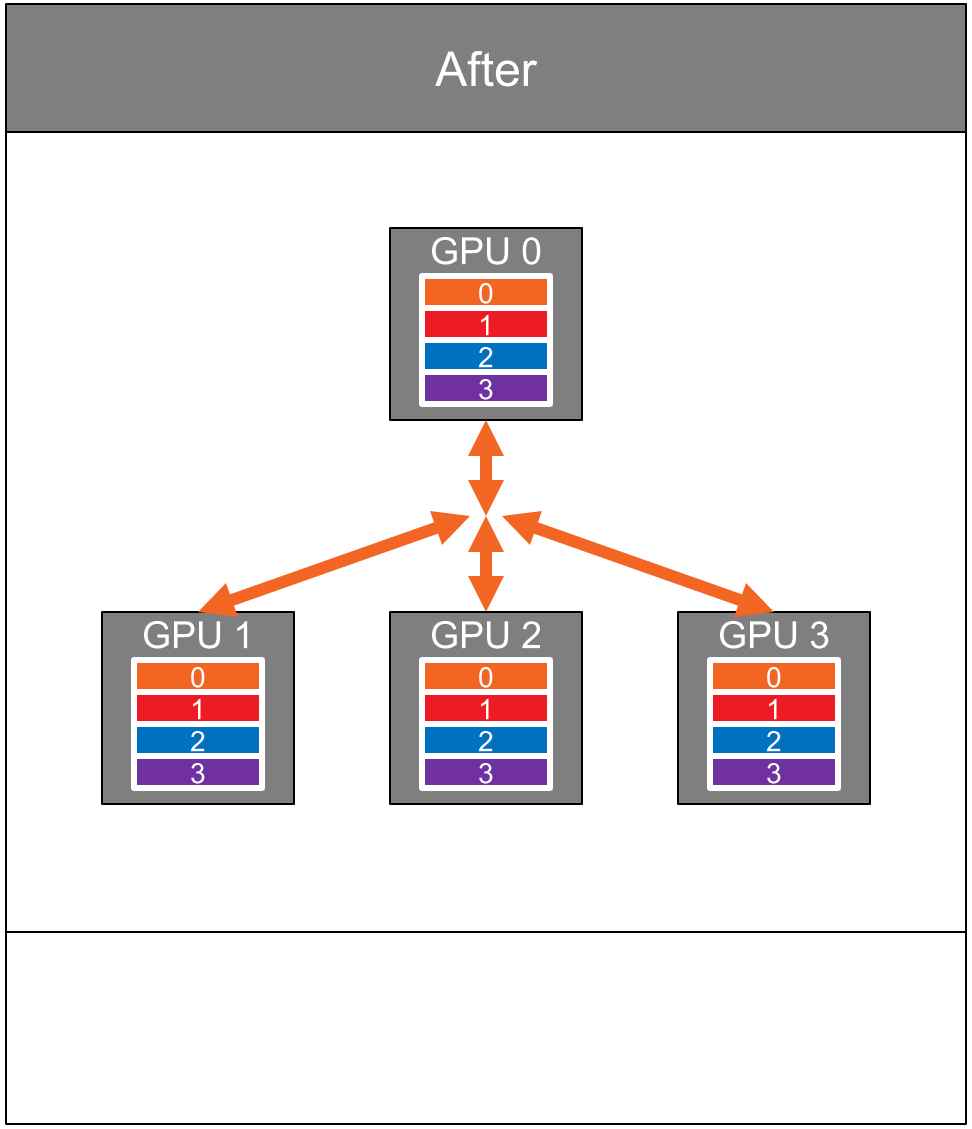
REDUCE

SCATTER

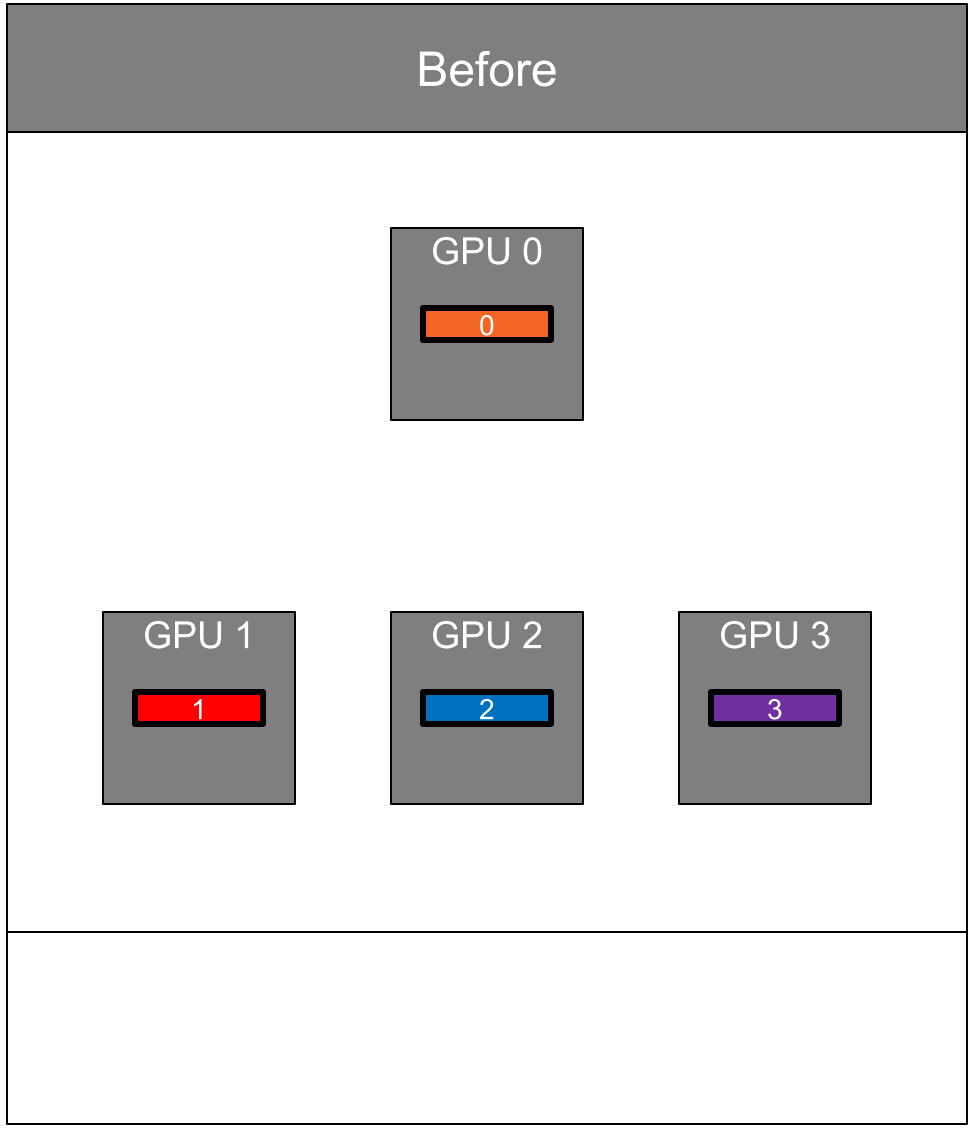


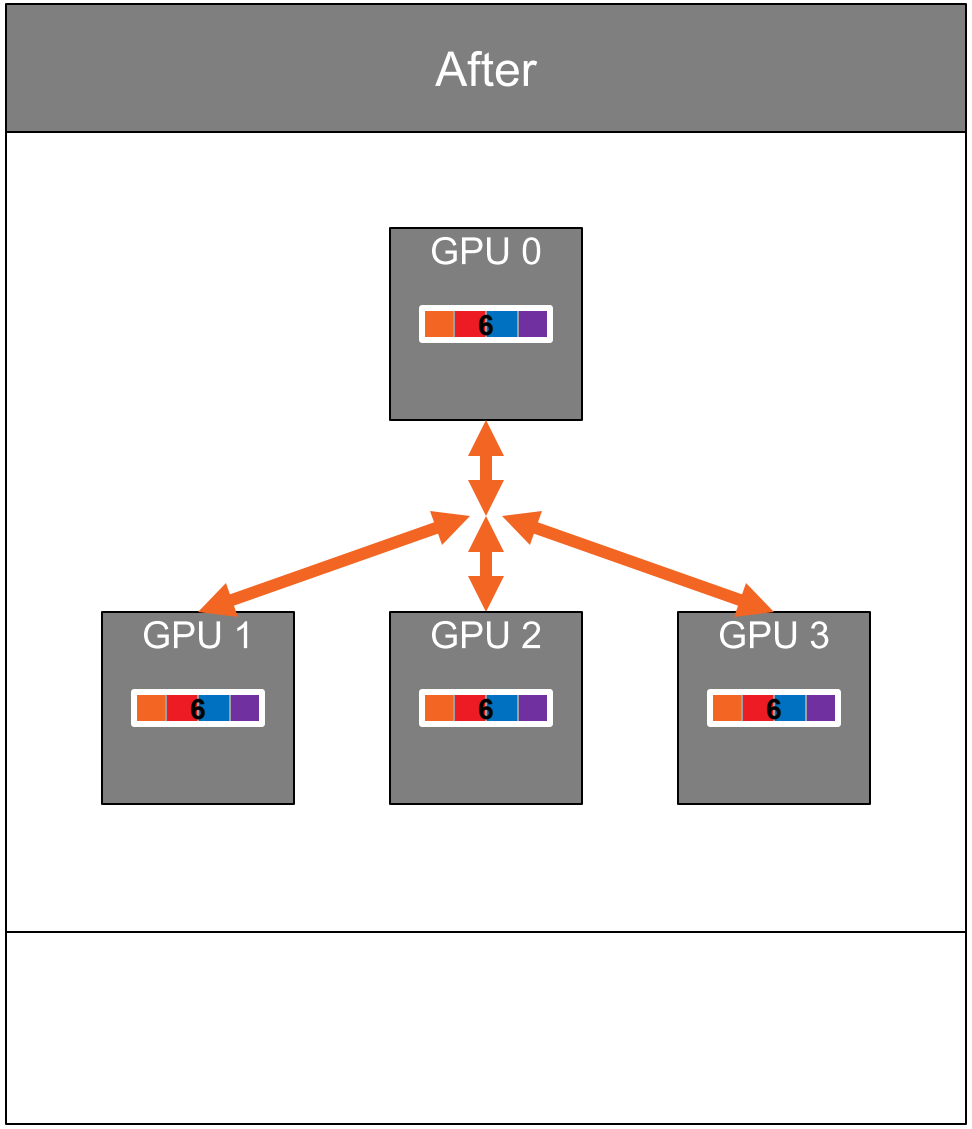
ALLGATHER



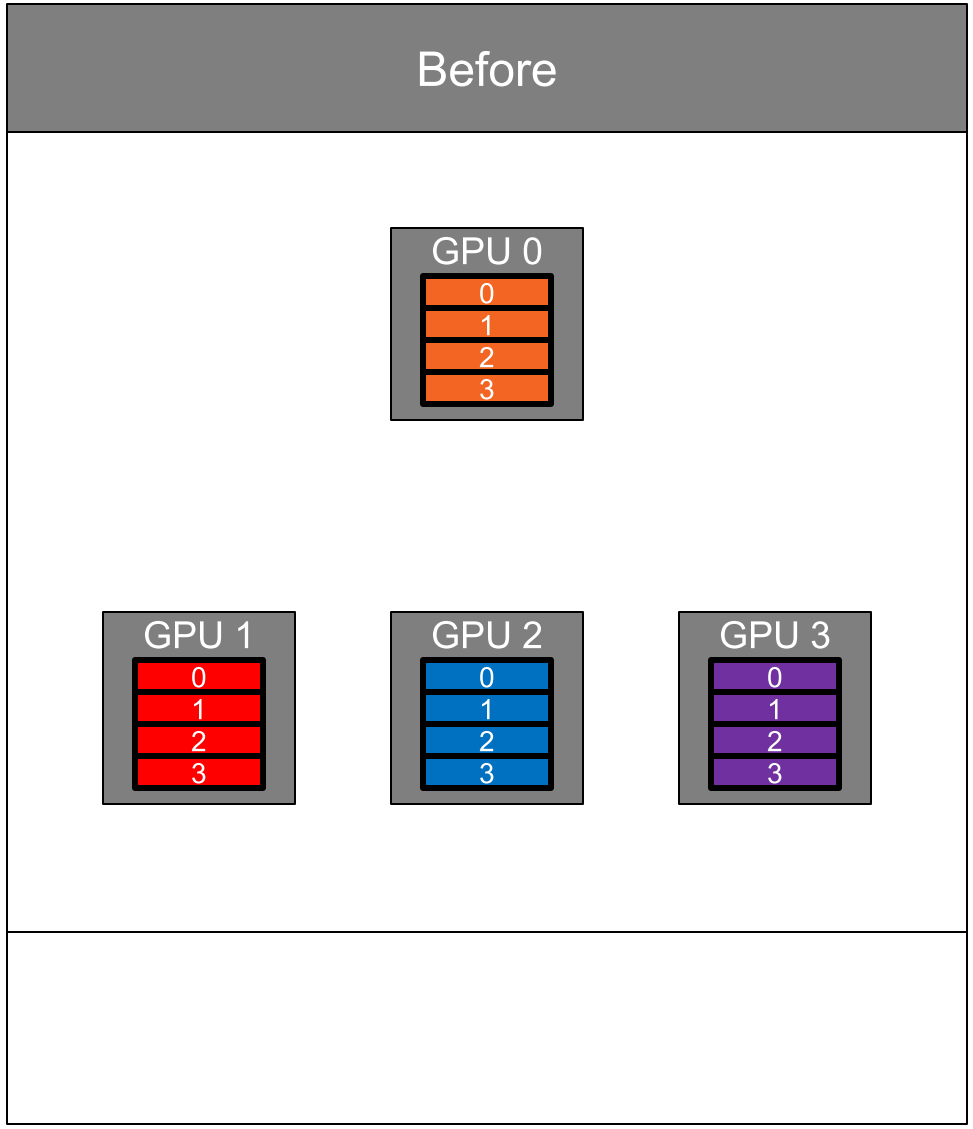


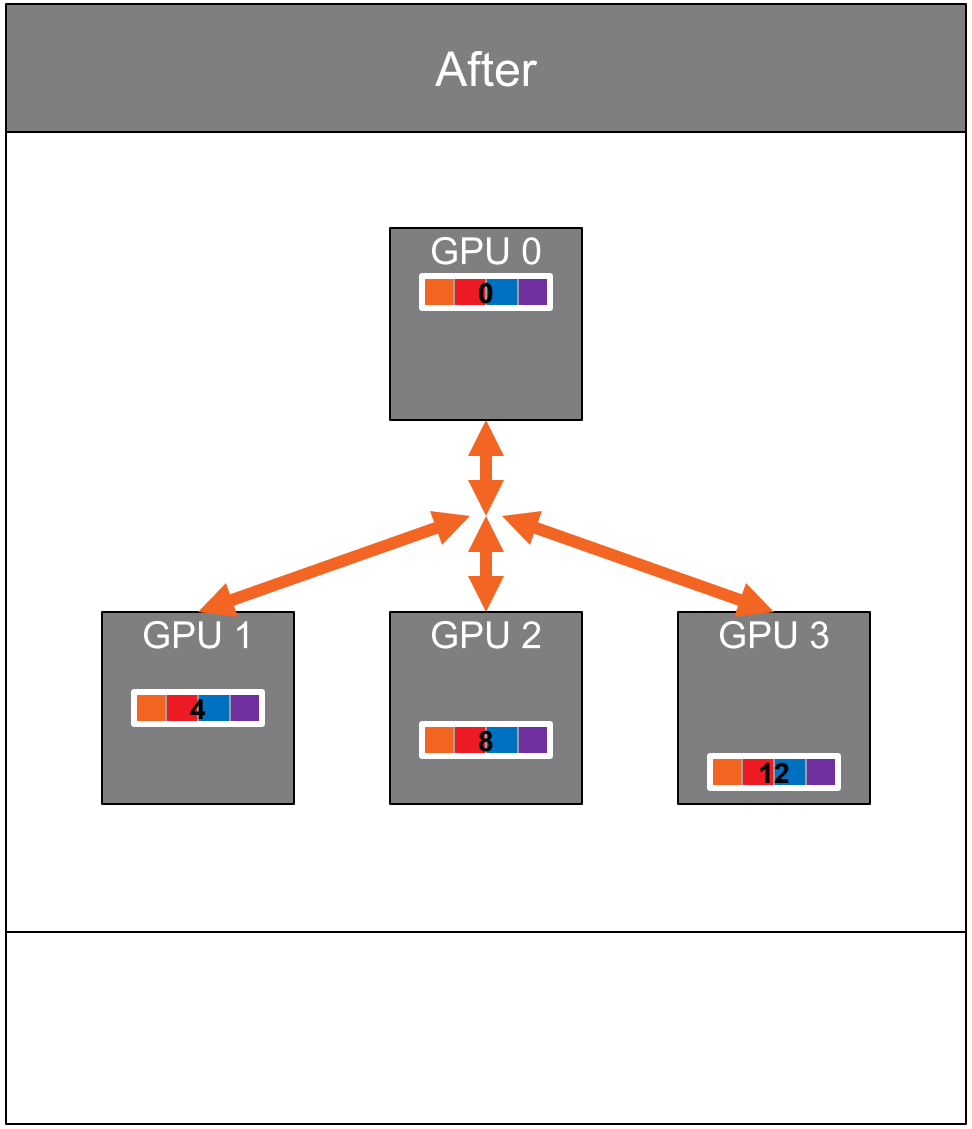
ALLREDUCE





REDUCE SCATTER





Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

Introduction to RCCL

- RCCL: **R**OCm **C**ollective **C**ommunication **L**ibrary
  - Pronounced “rickle” (rhymes with nickel)
- Open-source host-initiated library enabling collective communications executed via GPU as well as direct send/receive operations
- Supports collective algorithms across multiple processes / nodes via networking using Infiniband Verbs or TCP/IP sockets
- Actively worked on to optimize performance for AMD hardware
- Forked from NCCL (NVIDIA Collective Communication Library)
  - Maintains identical API (drop-in replacement)

RCCL Usage

- RCCL is provided as library and is not a standalone executable
  - Unit tests executables exist, but generally are not packaged and must be built from source
- Performance benchmarking can be done via `rccl-tests`
  - Fork of NCCL's nccl-tests
- RCCL is currently integrated in many machine learning frameworks, such as PyTorch, JAX, etc.
- RCCL is controlled primarily via environment variables

NCCL/RCCL API Example

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

- Reduces data arrays of length ***count*** in ***sendbuff*** using ***op*** operation and leaves identical copies of the result on each ***recvbuff***.
- In-place operation will happen if ***sendbuff*** == ***recvbuff***.
- Assumes ***sendbuff*** and ***recvbuff*** are GPU memory allocations

```cpp
#include <hip/hip_runtime.h>
#include <rccl/rccl.h>
#include <vector>

int main(int argc, char **argv)
{
  int numGpus;
  hipGetDeviceCount(&numGpus);

  // Initialize communicators for each GPU
  std::vector<ncclComm_t> comm(numGpus);
  ncclCommInitAll(comm.data(), numGpus, NULL);

  // Allocate memory and streams for each GPU
  size_t N = 1024;
  std::vector<float*> buffer(numGpus);
  std::vector<hipStream_t> streams(numGpus);
  for (int gpuIdx = 0; gpuIdx < numGpus; gpuIdx++) {
    hipSetDevice(gpuIdx);
    hipMalloc((void**)&buffer[gpuIdx], N * sizeof(float));
    hipStreamCreate(&streams[gpuIdx]);
  }

  // Call in-place AllReduce
  ncclGroupStart();
  for (int gpuIdx = 0; gpuIdx < numGpus; gpuIdx++)
    ncclAllReduce(buffer[gpuIdx], buffer[gpuIdx], N, ncclFloat, ncclSum, comm[gpuIdx], streams[gpuIdx]);
  ncclGroupEnd;

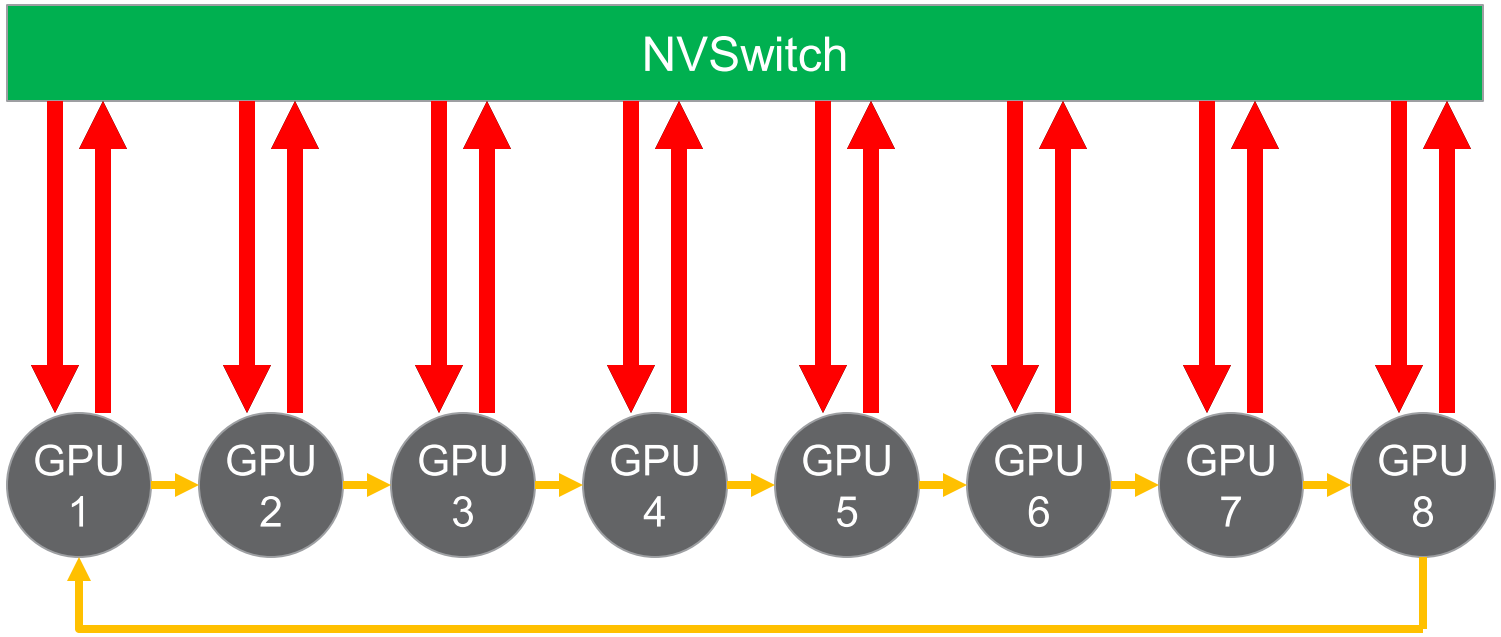
  // Wait for result
  for (int gpuIdx = 0; gpuIdx < numGpus; gpuIdx++)
    hipStreamSynchronize(streams[gpuIdx]);

  // Cleanup
  for (int gpuIdx = 0; gpuIdx < numGpus; gpuIdx++) {
    hipFree(buffer[gpuIdx]);
    hipStreamDestroy(streams[gpuIdx]);
    ncclCommDestroy(comm[gpuIdx]);
  }

  return 0;
}
```

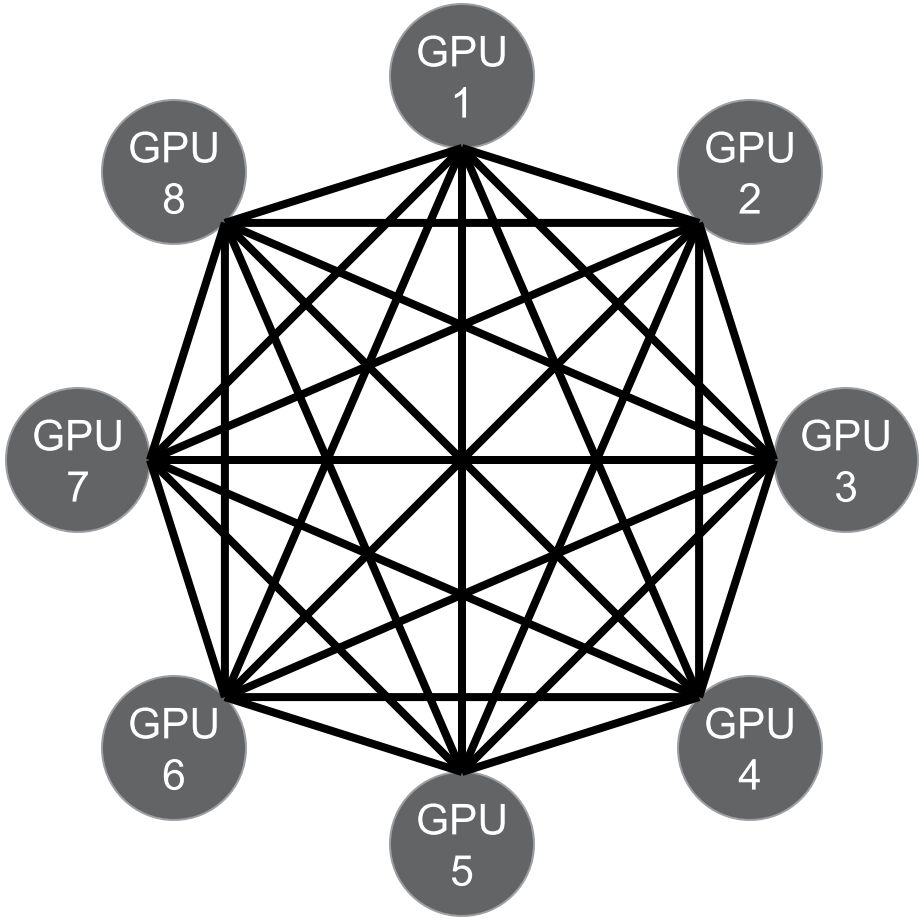
Ring Building on Switch-based Topology

- Any permutation of GPUs can form a single uni-directional ring that utilizes full bandwidth:



Hardware specific optimizations

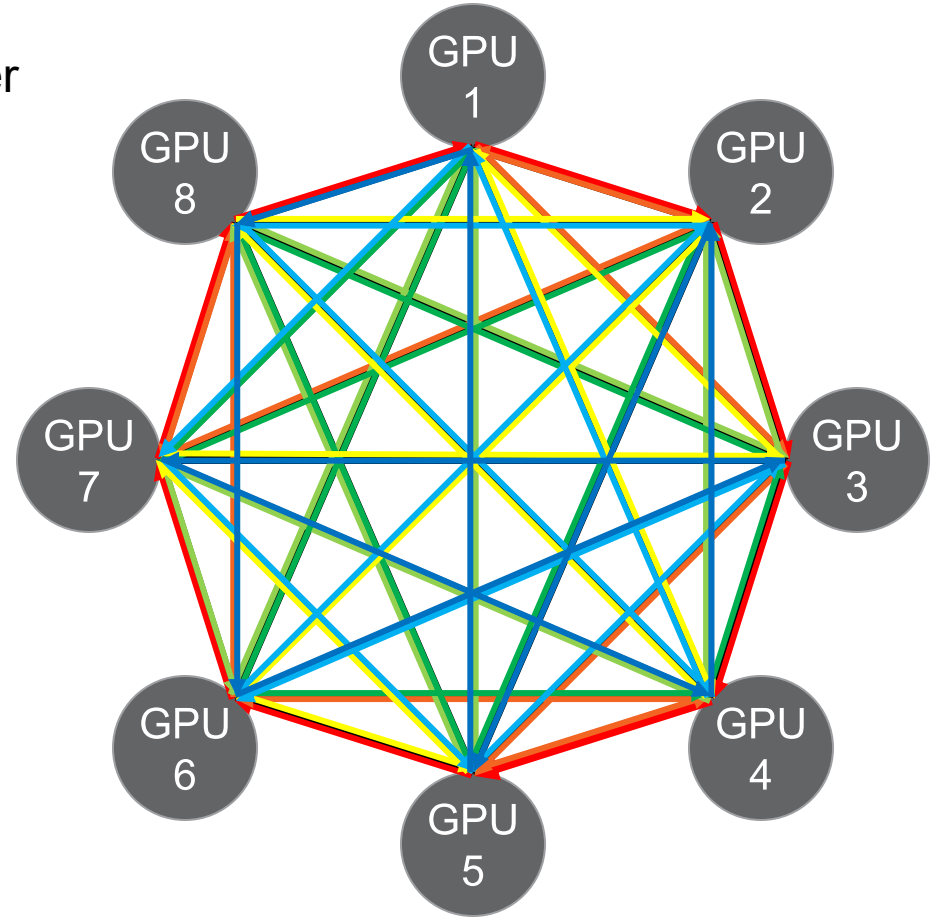
- A single MI300X node consists of 8 GPUs fully connected by XGMI Gen 3 links
- XGMI Gen 3 theoretical bandwidth is 64GB/s, however the measured bandwidth is about 46GB/s per direction
- Each GPU has a total bandwidth per direction of ~7x46GB/s = 322 GB/s



Ring Building on Fully-connected point to point Topology

Finding uni-directional rings that do not collide is much harder

Ring	Ordering								
A	1	2	3	4	5	6	7	8	1
B	1	3	5	4	6	8	7	2	1
C	1	4	8	2	6	5	7	3	1
D	1	5	8	3	2	4	7	6	1
E	1	6	4	3	8	5	2	7	1
F	1	7	5	3	6	2	8	4	1
G	1	8	6	3	7	4	2	5	



- Using 7 rings, each GPU can get roughly 7 x 46GB/s = 322 GB/s of bandwidth per direction

Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

RCCL Build Dependencies

- ROCm installed
- libstdc++-12-dev
  - This version avoids errors in cmath
- CMake >= 3.25
  - Ubuntu 22.04 comes with CMake 3.22 (if installed via apt)

RCCL is publicly available at: https://github.com/ROCm/rocm-systems/tree/develop/projects/rccl

For more detailed information:
https://amd.atlassian.net/wiki/spaces/MLSE/pages/744188783/Getting+Started+How+to+Install+and+Test+RCCL

Building RCCL (Simple)

```
> git clone https://github.com/ROCm/rocm-systems.git
> cd projects/rccl
> ./install.sh

(Library gets built in ./build/release)
```

Building RCCL (Faster)

```
> git clone https://github.com/ROCm/rocm-systems.git
> cd projects/rccl
> ./install.sh -l

(Library gets built in ./build/release)

Will only support local (detected) GPU architecture
```

Install script options

```
./install.sh --help
RCCL build & installation helper script
 Options:
       --address-sanitizer     Build with address sanitizer enabled
    -c|--enable-code-coverage  Enable code coverage
    -d|--dependencies          Install RCCL dependencies
       --debug                 Build debug library
       --enable_backtrace      Build with custom backtrace support
       --disable-colltrace     Build without collective trace
       --disable-msccl-kernel  Build without MSCCL kernels
       --disable-mscclpp       Build without MSCCL++ support
       --enable-mscclpp-clip   Build MSCCL++ with clip wrapper on bfloat16 and half addition routines
       --disable-roctx         Build without ROCTX logging
    -f|--fast                  Quick-build RCCL (local gpu arch only, no backtrace, and collective trace support)
    -h|--help                  Prints this help message
    -i|--install               Install RCCL library (see --prefix argument below)
    -j|--jobs                  Specify how many parallel compilation jobs to run (224 by default)
    -l|--local_gpu_only        Only compile for local GPU architecture
       --amdgpu_targets        Only compile for specified GPU architecture(s). For multiple targets, separate by ';' (builds for all supported GPU architectures by default)
       --no_clean              Don't delete files if they already exist
       --npkit-enable          Compile with npkit enabled
       --log-trace             Build with log trace enabled (i.e. NCCL_DEBUG=TRACE)
       --openmp-test-enable    Enable OpenMP in rccl unit tests
    -p|--package_build         Build RCCL package
       --prefix                Specify custom directory to install RCCL to (default: `/opt/rocm`)
       --run_tests_all         Run all rccl unit tests (must be built already)
    -r|--run_tests_quick       Run small subset of rccl unit tests (must be built already)
       --static                Build RCCL as a static library instead of shared library
    -t|--tests_build           Build rccl unit tests, but do not run
       --time-trace            Plot the build time of RCCL (requires `ninja-build` package installed on the system)
       --verbose               Show compile commands
       --force-reduce-pipeline Force reduce_copy sw pipeline to be used for every reduce-based collectives and datatypes
```

Use ./install.sh –help to see build options

RCCL Test (Single node) Build Dependencies

- ROCm installed
- CMake >= 3.16
- RCCL
  - can be either available under `/opt/rocm/lib/librccl.so`
  - (or) point to your local build

RCCL tests are publicly available at: https://github.com/ROCm/rocm-systems/tree/develop/projects/rccl-tests

Building rccl-tests

```
> git clone https://github.com/ROCm/rocm-systems.git
> cd projects/rccl-tests
> ./install.sh

Various executables are built in ./build/ corresponding to various collectives:
  all_gather_perf
  all_reduce_perf
  alltoall_perf
  alltoallv_perf
  broadcast_perf
  gather_perf
  reduce_perf
  reduce_scatter_perf
  scatter_perf
sendrecv_perf
```

Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

Running rccl-tests (Single Node)

```
> all_reduce_perf -g 8 -b 1M -e 1G -f 2

(This will execute AllReduce across 8 GPUs with the data size beginning from 1MB and
ending at 1GB going up by a multiplicative factor of 2)
```

```
> broadcast_perf -g 8 -r 4 -b 1M -e 32M -i 2097152

(This will execute Broadcast across 8 GPUs from root GPU 4 with the data size
beginning from 1MB and ending at 32M going up by an additive increment of 2MB )
```

```
> reduce_scatter_perf -g 8 -b 1M -e 32M -f 2 -d bfloat16

(This will execute ReduceScatter across 8 GPUs with the datatype bfloat16, and data
size beginning from 1MB and ending at 32M going up by a multiplicative factor of 2)
```

LD_LIBRARY_PATH

```
> LD_LIBRARY_PATH=/my/rccl/build:$LD_LIBRARY_PATH ./gather_perf ...
```

Rccl-test command-line options

```
These command line arguments are shared across all test executables:
Commonly used arguments:

-t: # of threads to use          (default 1)
-g: # of GPUs to use per thread (default 1)
-b: Starting data size to test  (default 32M)
-e: Ending data size to test    (default 32M)
-i: Additive increment          (default 1M)
-f: Multiplicative factor       (default 1)
-w: # of warmup iterations      (default 5)
-n: # of iterations             (default 20)
-N: # of run-cycles             (default 1, 0 = infinite)
-G: # of graph launches         (default 0)
-r: root rank index             (default 0)
-o: Reduction operator          (sum/prod/min/max/avg/mulsum/all)
-d: Datatype                    (int8/uint8/int32/uint32/int64/uint64/half/float/
                                 double/bfloat16/fp8_e4m3/fp8_e5m2)
```

Sample MI300X results

```
./all_reduce_perf -g 8 -b 1M -e 1G -f 2
# nThread 1 nGpus 8 minBytes 1048576 maxBytes 1073741824 step: 2(factor) warmup iters: 5 iters: 20 agg iters: 1 validation: 1 graph: 0
#
rccl-tests: Version develop:41b383a
# Using devices
#  Rank  0 Group  0 Pid 614425 on dell300x-pla-t11-03 device  0 [0000:1b:00] AMD Instinct MI300X
#  Rank  1 Group  0 Pid 614425 on dell300x-pla-t11-03 device  1 [0000:3d:00] AMD Instinct MI300X
#  Rank  2 Group  0 Pid 614425 on dell300x-pla-t11-03 device  2 [0000:4e:00] AMD Instinct MI300X
#  Rank  3 Group  0 Pid 614425 on dell300x-pla-t11-03 device  3 [0000:5f:00] AMD Instinct MI300X
#  Rank  4 Group  0 Pid 614425 on dell300x-pla-t11-03 device  4 [0000:9d:00] AMD Instinct MI300X
#  Rank  5 Group  0 Pid 614425 on dell300x-pla-t11-03 device  5 [0000:bd:00] AMD Instinct MI300X
#  Rank  6 Group  0 Pid 614425 on dell300x-pla-t11-03 device  6 [0000:cd:00] AMD Instinct MI300X
#  Rank  7 Group  0 Pid 614425 on dell300x-pla-t11-03 device  7 [0000:dd:00] AMD Instinct MI300X
#
#                                                     out-of-place                       in-place
#       size         count      type   redop    root     time   algbw   busbw #wrong     time   algbw   busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)            (us)  (GB/s)  (GB/s)
     1048576        262144     float     sum      -1    66.48   15.77   27.60      0    67.59   15.51   27.15      0
     2097152        524288     float     sum      -1    69.18   30.31   53.05      0    69.07   30.36   53.13      0
     4194304       1048576     float     sum      -1    70.01   59.91  104.85      0    68.89   60.89  106.55      0
     8388608       2097152     float     sum      -1    88.18   95.13  166.48      0    90.89   92.30  161.52      0
    16777216       4194304     float     sum      -1    153.0  109.65  191.90      0    159.1  105.46  184.56      0
    33554432       8388608     float     sum      -1    243.9  137.56  240.72      0    254.9  131.63  230.35      0
    67108864      16777216     float     sum      -1    423.2  158.58  277.51      0    432.8  155.04  271.32      0
   134217728      33554432     float     sum      -1    788.2  170.29  298.02      0    795.5  168.72  295.26      0
   268435456      67108864     float     sum      -1   1532.0  175.22  306.63      0   1538.2  174.51  305.39      0
   536870912     134217728     float     sum      -1   3022.0  177.65  310.89      0   3034.0  176.95  309.67      0
  1073741824     268435456     float     sum      -1   5968.5  179.90  314.83      0   5978.2  179.61  314.32      0
# Errors with asterisks indicate errors that have exceeded the maximum threshold.
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 206.895
#
```

AlgBw vs BusBw

Bandwidth is calculated as $\frac{Data\ size}{Time}$.

However, there are different ways to count data size

- Algorithmic Bandwidth uses the “input” data size
- Bus Bandwidth uses the “actual amount of (non-local) data transferred”
  - Bus bandwidth allows for comparisons across different collectives can be compared against hardware expectations

AlgBw vs BusBw

Bus Bandwidth and Algorithm Bandwidth are just different by a multiplicative factor (usually based on the number of participating GPUs)

For example, for 8 GPU AllReduce, for every 1GB of input data, the RingReduce algorithm actually sends $\frac{2(8-1)}{8} = 1.75$ GB of data

```
  1073741824      268435456     float      sum      -1    5968.5  179.90  314.83
```

179.90 x 1.75 = 314.83

For more info: https://github.com/ROCm/rccl-tests/blob/develop/doc/PERFORMANCE.md

Running rccl-tests (Best practices)

For maximum performance, we advise using MPI to launch rccl-tests with 1 MPI process per GPU.

```
> mpirun -np 8 ./all_reduce_perf -g 1 -b 1M -e 1G -f 2

(This will execute AllReduce across 8 GPUs (1 process per GPU) with the data size
beginning from 1MB and ending at 1GB going up by a multiplicative factor of 2)
```

```
> mpirun -np 8 ./broadcast_perf -g 1 -r 4 -b 1M -e 32M -i 2097152

(This will execute Broadcast across 8 GPUs (1 process per GPU) from root GPU 4 with
the data size beginning from 1MB and ending at 32M going up by an additive increment
of 2MB )
```

Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

RCCL Protocol

- RCCL currently supports 3 different protocols modifiable by the env var NCCL_PROTO

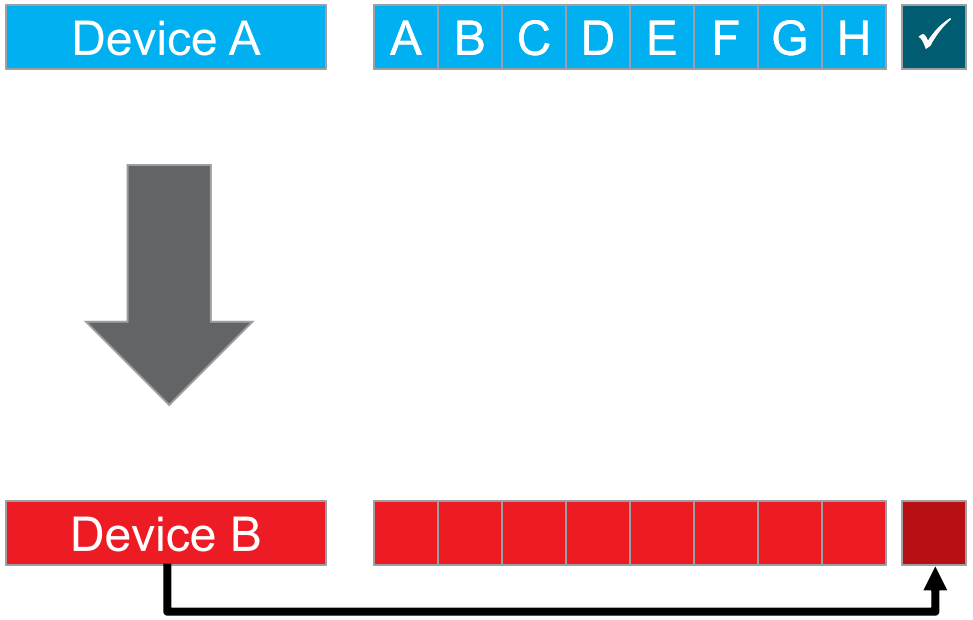
NCCL_PROTO	Data Efficiency	Notes
Simple	< 100%	Payload followed by flag
LL	50%	4B payload + 4B flag
LL128	93.75%	60B payload + 4B flag (multi-node only)

  - This is a debug variable and NOT recommended to be used in production environments
    - This is a global override, and performance will suffer for data ranges the protocol is not suitable for

  - NCCL_PROTO can be specified as a comma-separated-list

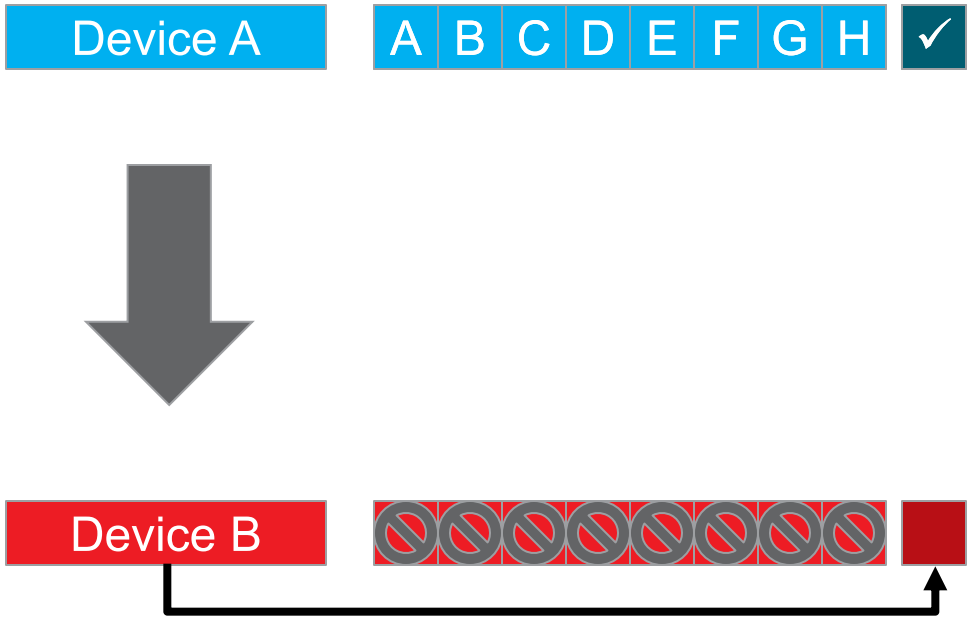
Simple Protocol:

Device A sends payload followed by a flag to signal completion



Device B polls on the flag to know when payload is ready

Simple Protocol:



Need to guarantee that the flag will be seen ONLY after the entire payload has arrived

Simple Protocol: Pitfall 1: Compile-time re-ordering

Written Code
Send A
Send C
Send G
Send E
Send H
Send D
Send B
Send F
Send Flag

Emitted Code

The order of instructions (that aren't dependent on one another) emitted by the compiler is NOT guaranteed to be the same order as how the code was written*

This allows for some compile-time performance optimizations such as fusing 2 DWORDx2 into 1 DWORDx4.

*(For languages with weakly-ordered memory models, such as C++11, CUDA, HIP, ARM, Java)

Simple Protocol: Pitfall 1: Compile-time re-ordering

Written Code
Send A
Send C
Send G
Send E
Send H
Send D
Send B
Send F
Memory Fence
Send Flag

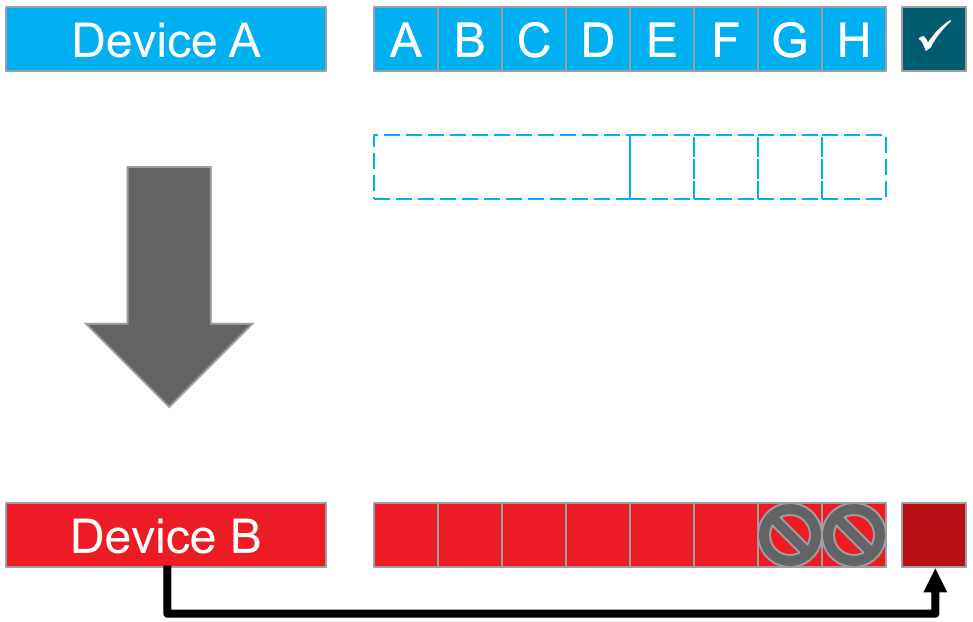
Emitted Code

Requires explicit memory fence instructions to prevent re-ordering

In CUDA/HIP memory fence instructions are provided via the `__threadfence()` family of intrinsics

https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_cpp_language_extensions.html#memory-fence-instructions

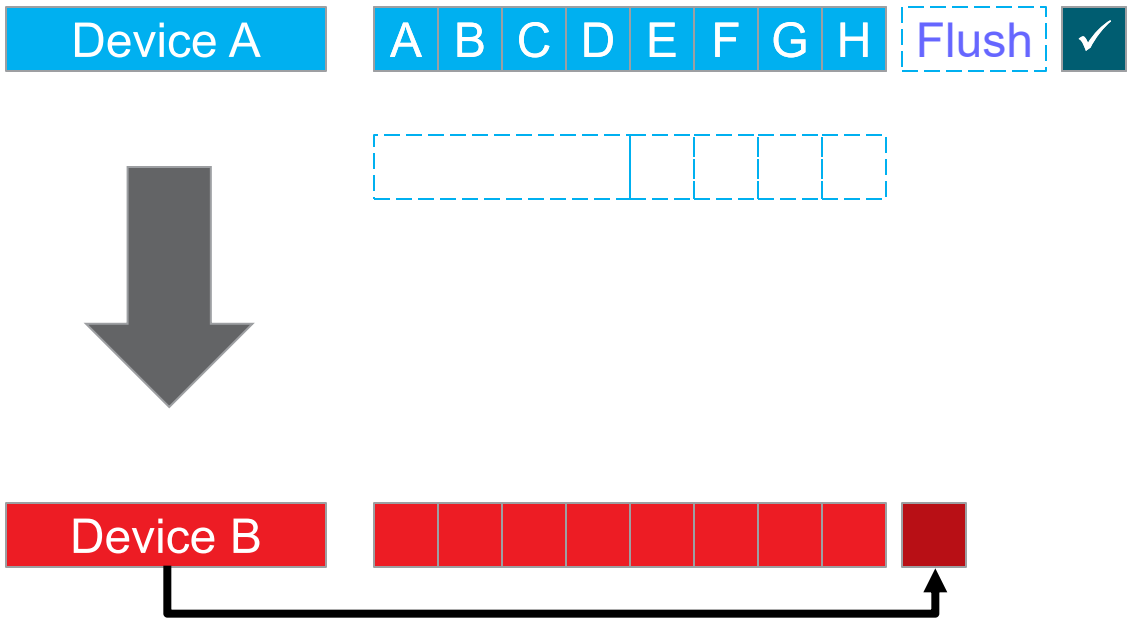
Simple Protocol: Pitfall 2: Run-time re-ordering



It is also possible that the underlying hardware re-orders memory operations during run-time

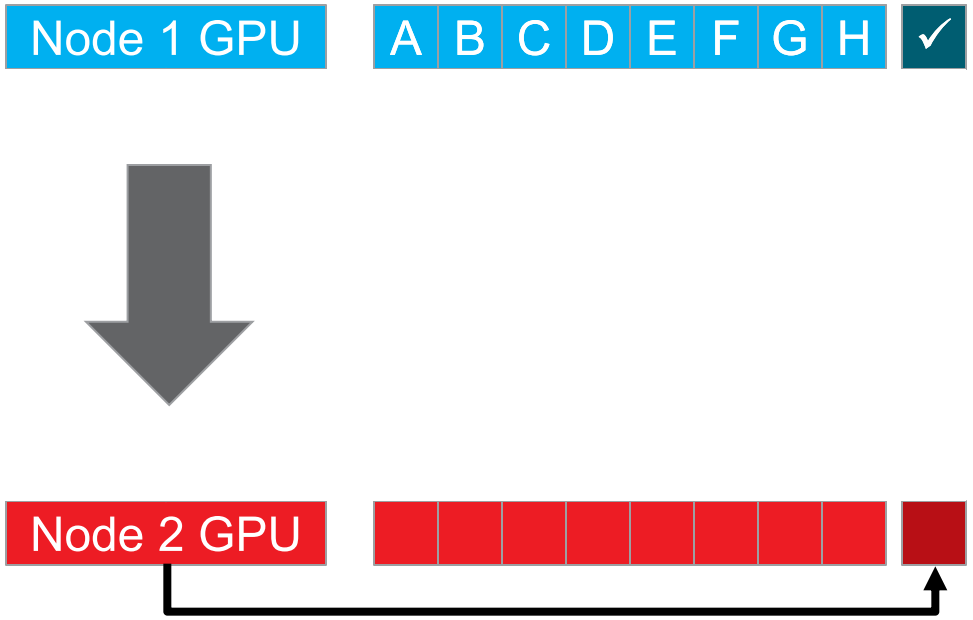
For example, being buffered in a cache or PCIe switch or along the data fabric

Simple Protocol: Pitfall 2: Run-time re-ordering



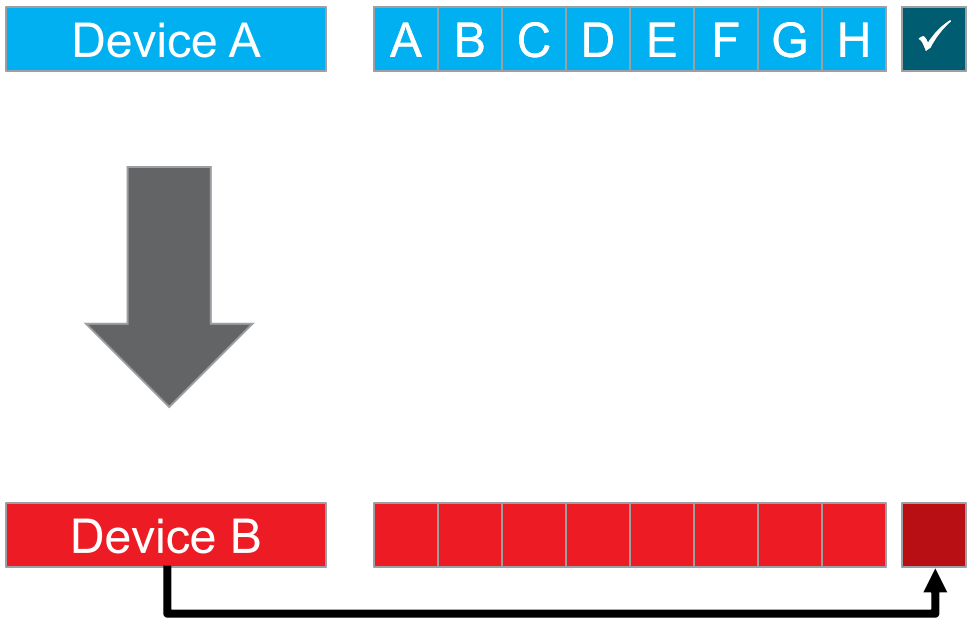
This may require explicit cache flushes / non-temporal stores / use of fine-grained memory to solve (and dependent on which two hardware devices are involved)

Simple Protocol: Pitfall 3: Multi-device/Multi-node coherency



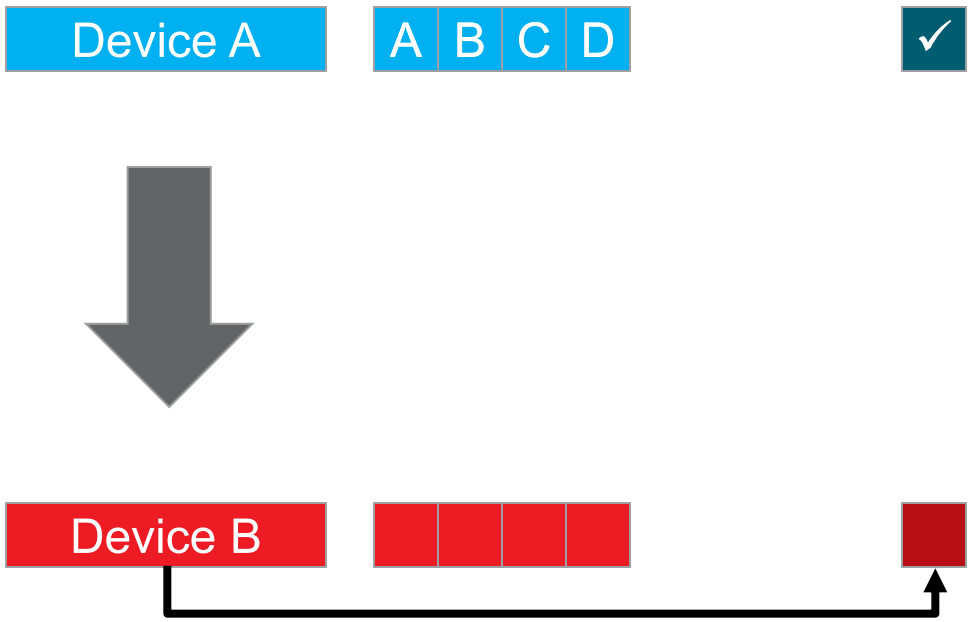
Guaranteeing that the flag is sent after the data gets even more complicated when data is sent across nodes, requiring even more coordination between intermediate devices such as network interface cards (NICs)

Low Latency (LL) Protocol:



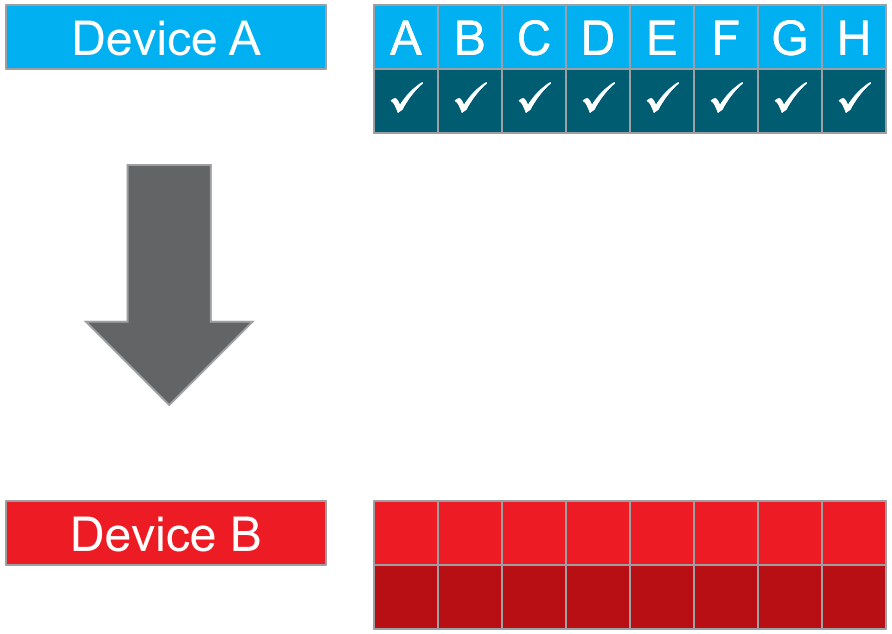
Device B may end up waiting quite a bit of time even though partial results are ready

Low Latency (LL) Protocol:



One solution would be to reduce the size of the payload at the cost of reduced efficiency

Low Latency (LL) Protocol:

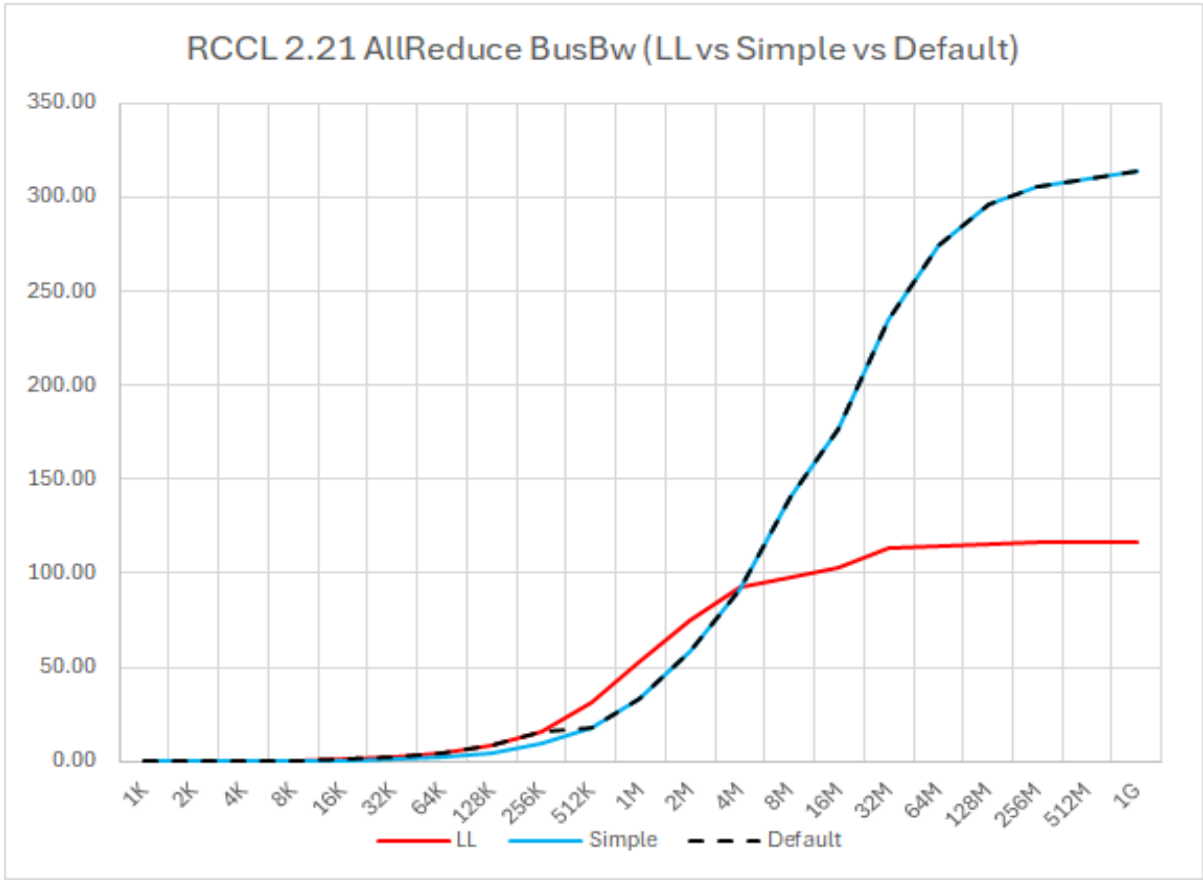


The low latency (LL) protocol takes this idea to the extreme and fits the payload and flag into the largest indivisible transaction size

For example, on MI300X this is 64 bytes, which can be 60 bytes of payload and a 4 byte flag

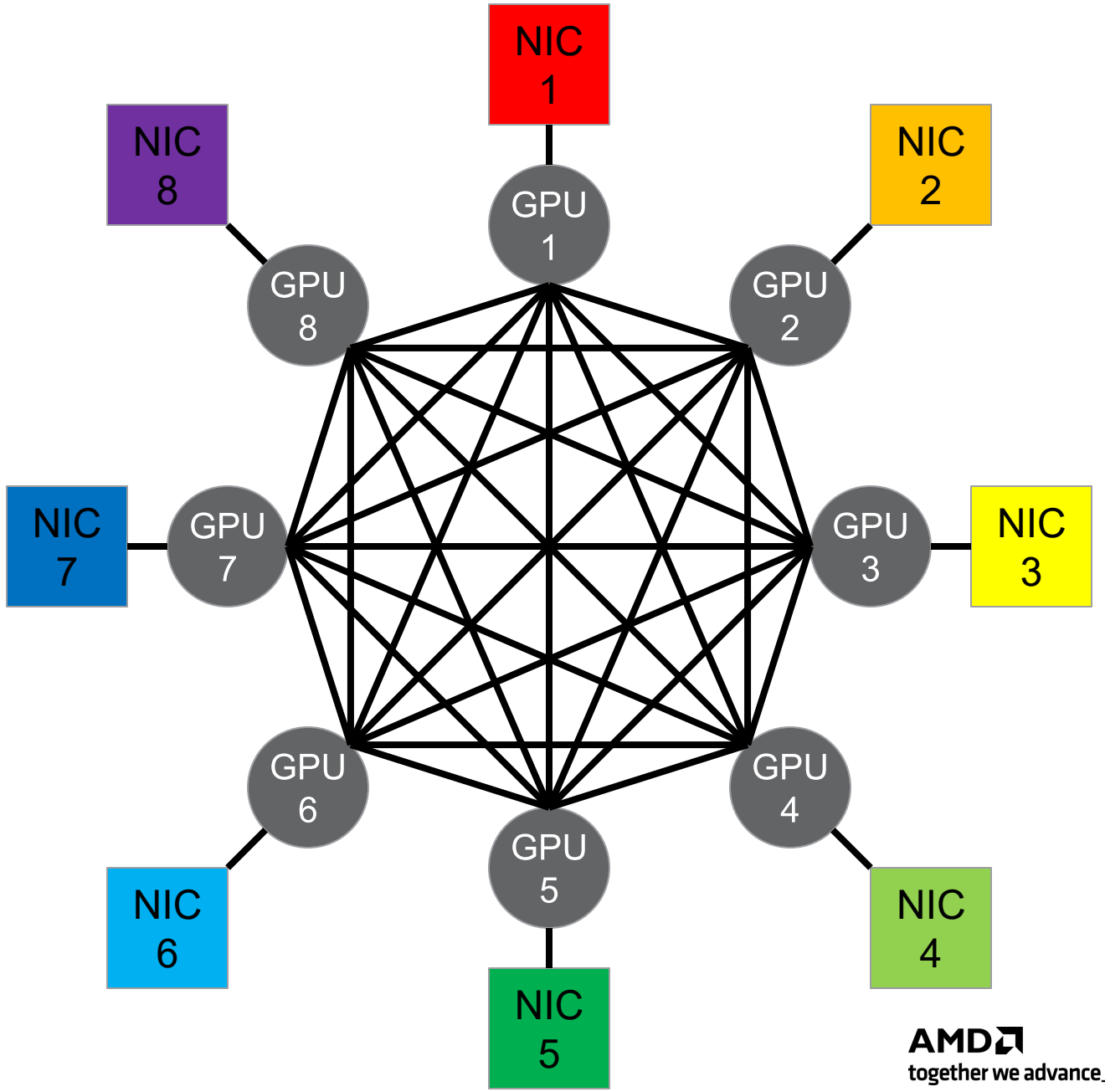
RCCL Protocol

RCCL 2.21	MI300X AllReduce (fp32) BusBw (GB/s)		
Datasize	LL	Simple	Default
1K	0.08	0.04	0.08
2K	0.17	0.09	0.17
4K	0.32	0.18	0.34
8K	0.65	0.34	0.64
16K	1.05	0.65	1.05
32K	2.09	1.28	2.08
64K	4.12	2.50	4.15
128K	8.19	4.81	8.17
256K	16.19	9.34	16.27
512K	31.51	17.94	17.98
1M	53.28	33.33	33.30
2M	75.35	58.60	58.45
4M	92.59	91.82	91.84
8M	97.42	140.03	140.00
16M	103.46	176.43	176.48
32M	113.91	234.79	235.02
64M	114.79	274.11	274.18
128M	115.84	296.33	296.34
256M	116.22	305.40	305.72
512M	116.19	309.72	310.11
1G	116.33	314.38	314.37



MI300X Single Node + NICs

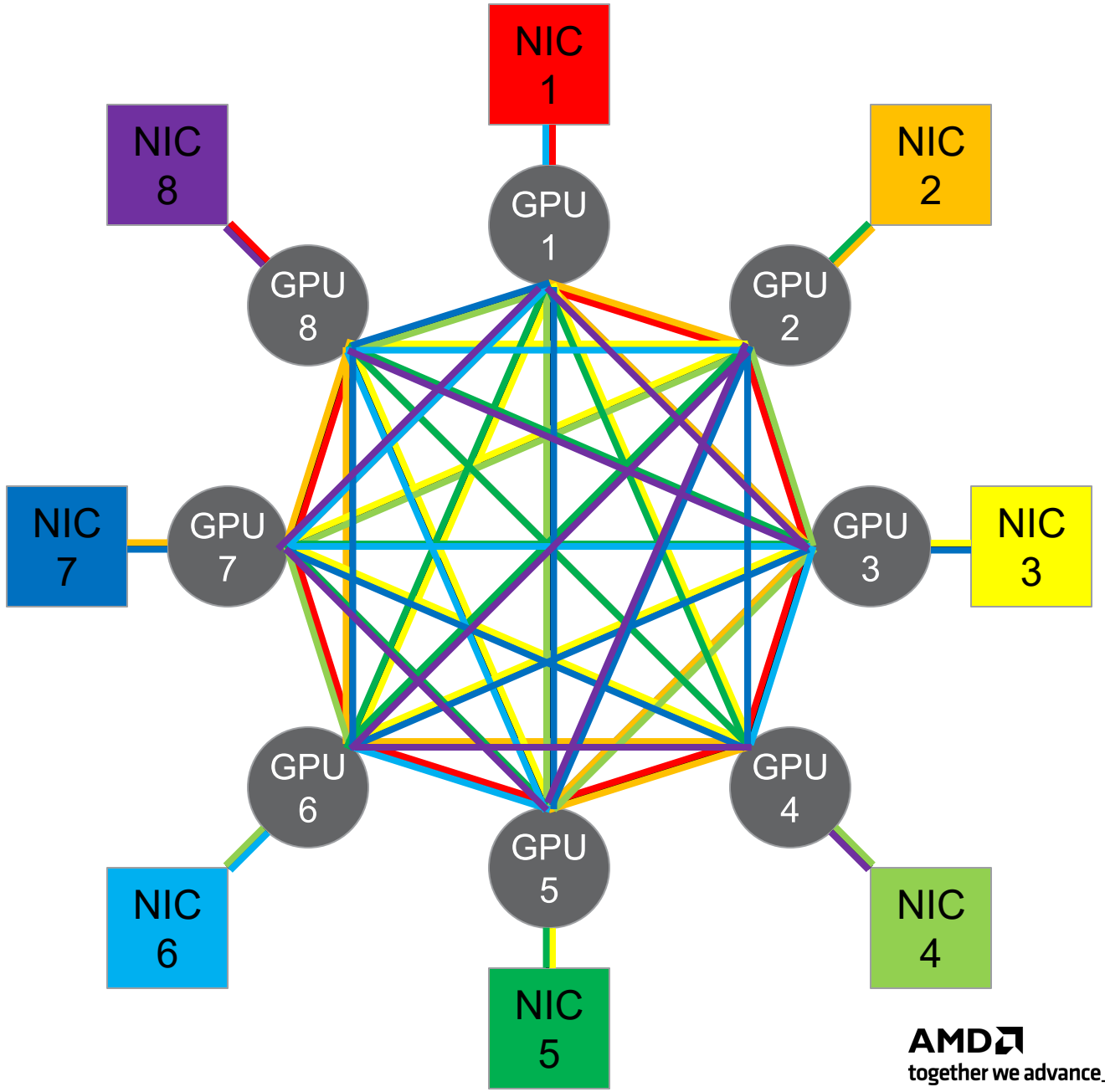
- Each GPU on MI300X is connected to a network interface card (NIC)
- Each NIC is rated at 400Gbits/s which roughly gets about 45GB/s per direction



MI300X Single Node + NICs

- To maximize NIC and XGMI bandwidth, 8 unidirectional lines can be established which utilize every link exactly once

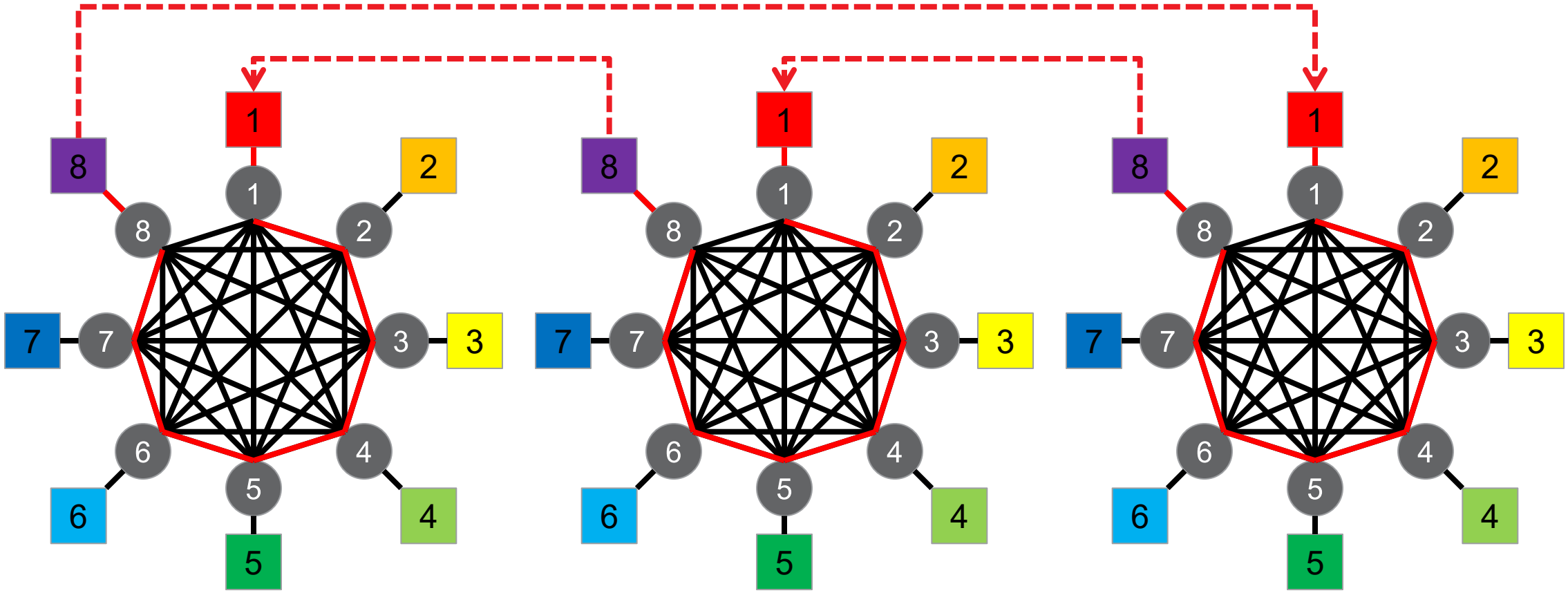
Line	Ordering							
A	1	2	3	4	5	6	7	8
B	2	1	3	5	4	6	8	7
C	3	6	1	4	7	2	8	5
D	4	8	1	5	3	2	7	6
E	5	7	3	8	4	1	6	2
F	6	5	8	2	4	3	7	1
G	7	4	2	5	1	8	6	3
H	8	3	1	7	5	2	6	4



MI300X Single Node + NICs

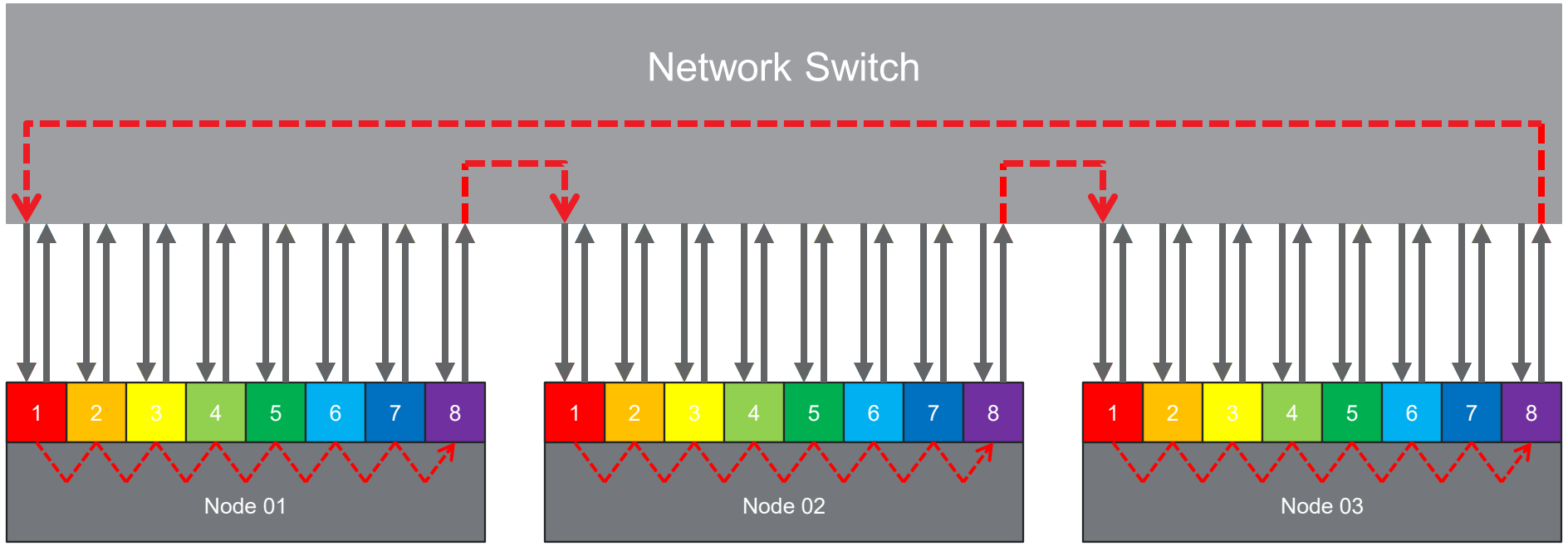
Line	Ordering							
A	1	2	3	4	5	6	7	8

- Each of the 8 lines can be used form rings across multiple nodes:



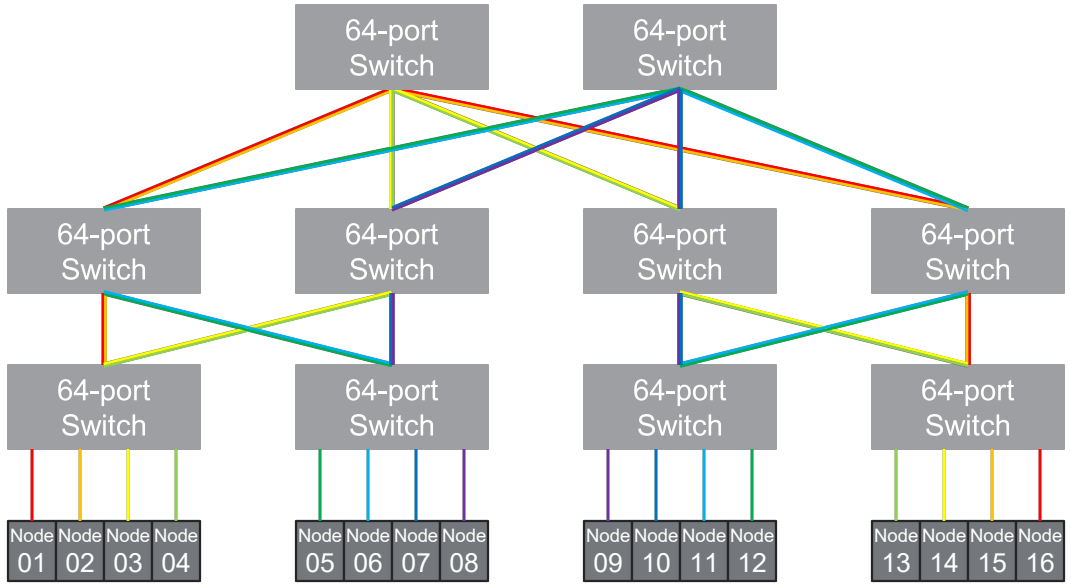
Network Topologies

Line	Ordering							
A	1	2	3	4	5	6	7	8



- In general, nodes are not connected directly to one another (static network), but connected via a network switch
- Non-blocking switches can support full bidirectional bandwidth on all ports

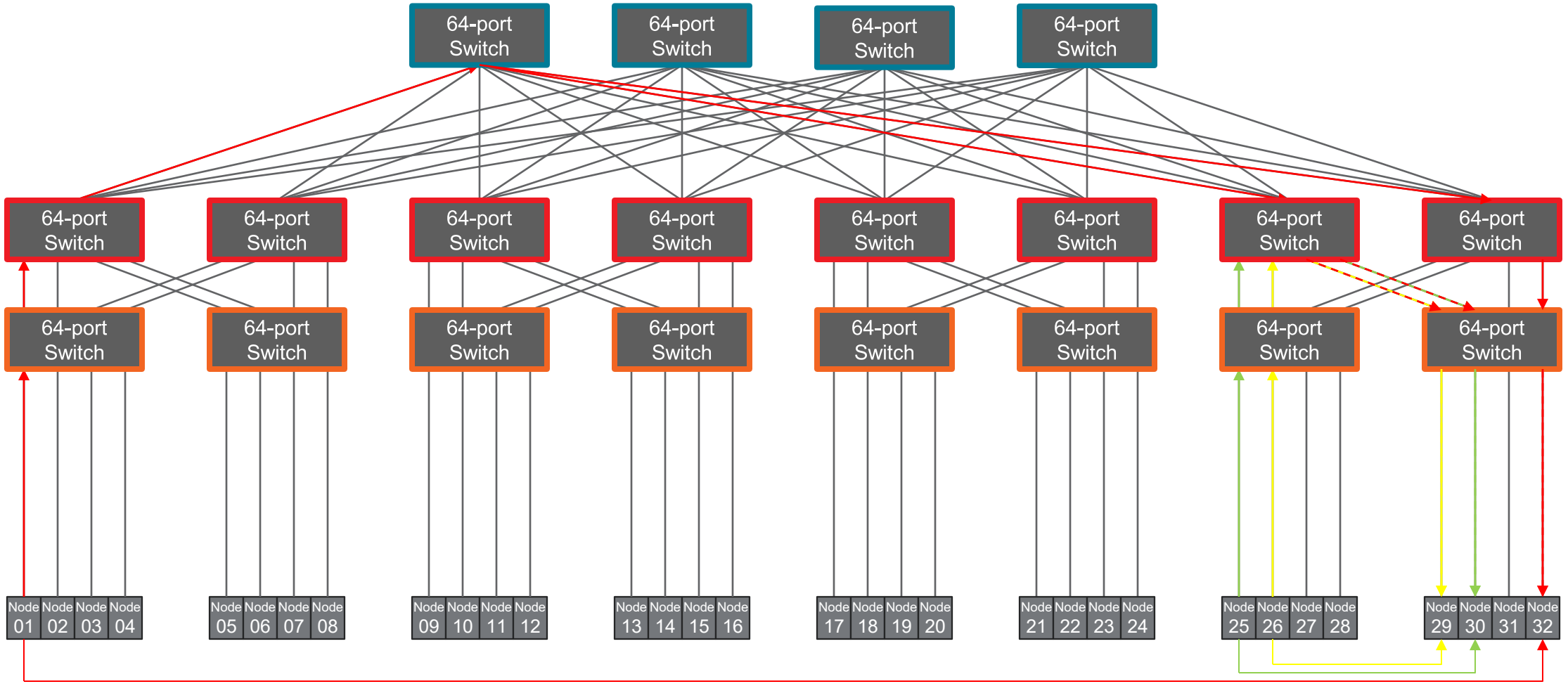
Network Topologies



Generally, additional levels of network switches are required to attain a non-blocking network topology

This is an example of a 16-node non-blocking network topology called a fat tree (edges near root of the tree are “fatter”)

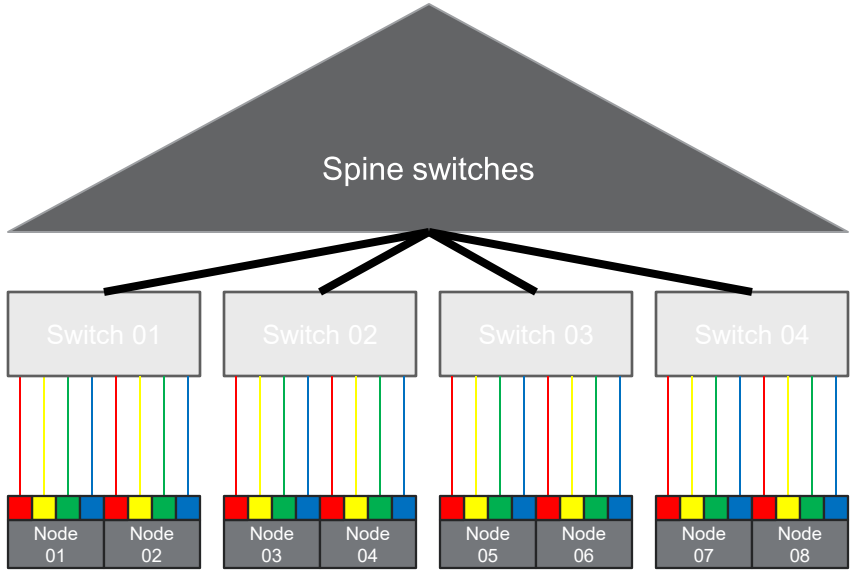
Network Routing



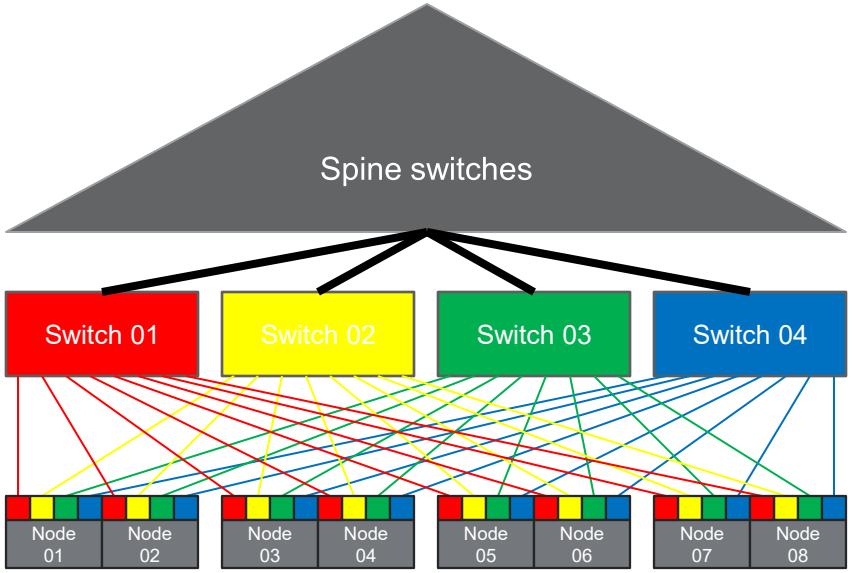
With multi-level switches, routing choices may not be optimal – switches are generally unaware of software usage patterns / remote switch workloads

Rail-Optimized Topologies

Classic fabric



Rail-optimized

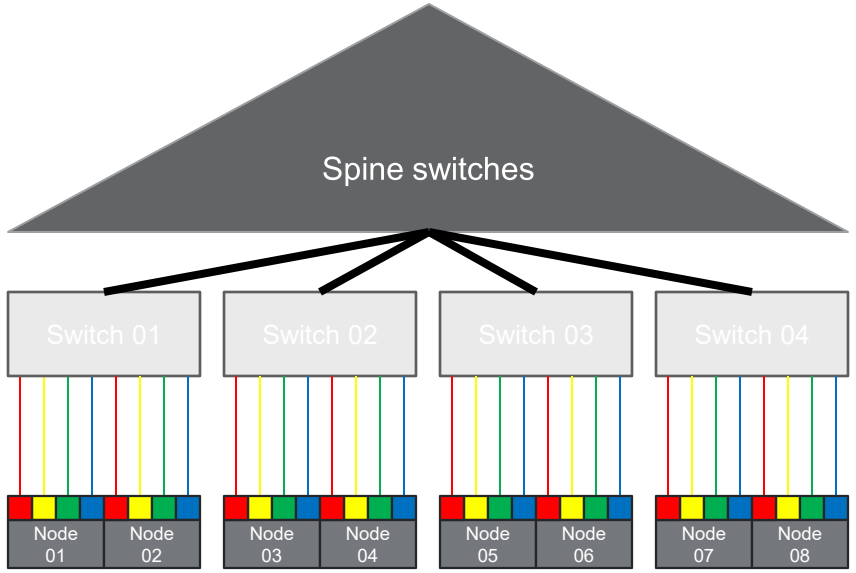


Classically, all NICs from a node are connected to the same switch

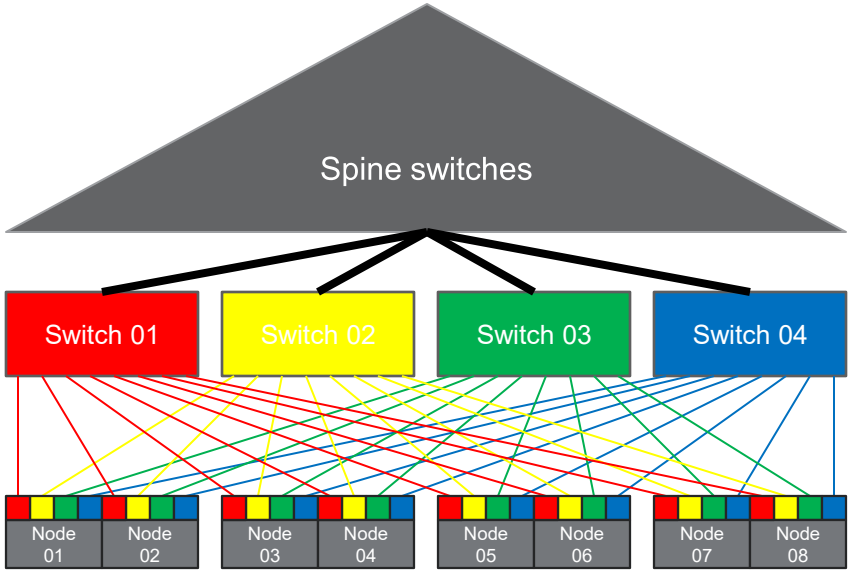
In a rail-optimized topology, the first NICs from each node all connect to the first switch, the second NICs to the second switch, and so on

Rail-Optimized Topologies

Classic fabric



Rail-optimized

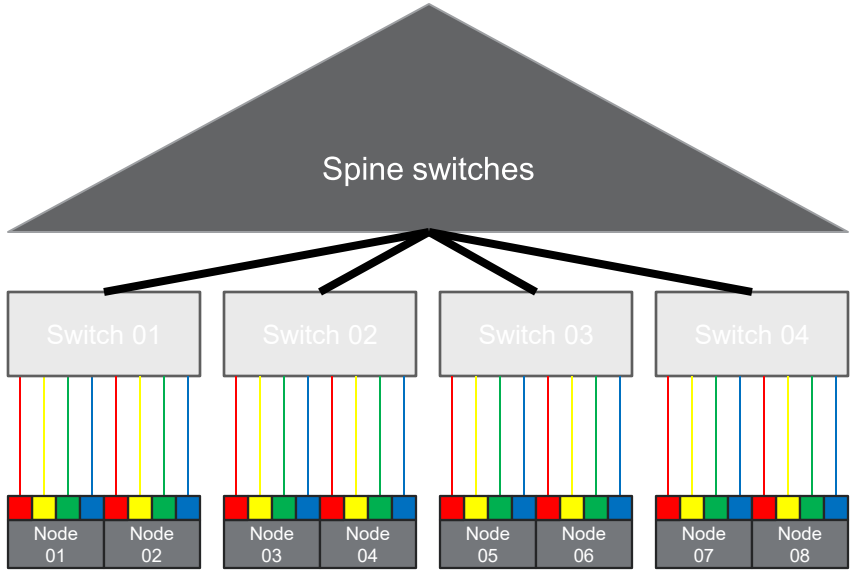


Note that these changes only occur between the nodes and first tier of switches

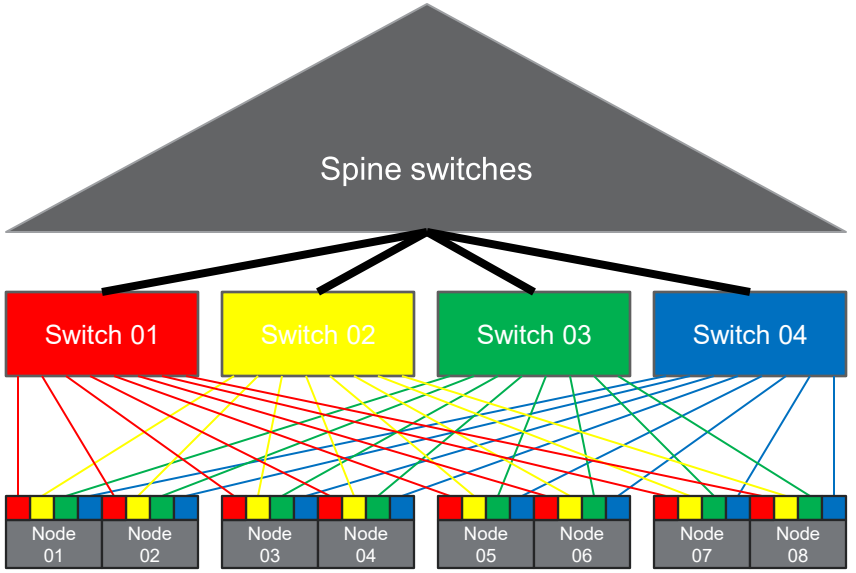
This potentially permits software more control over routing instead of “hoping for the best”

Rail-Optimized Topologies

Classic fabric



Rail-optimized

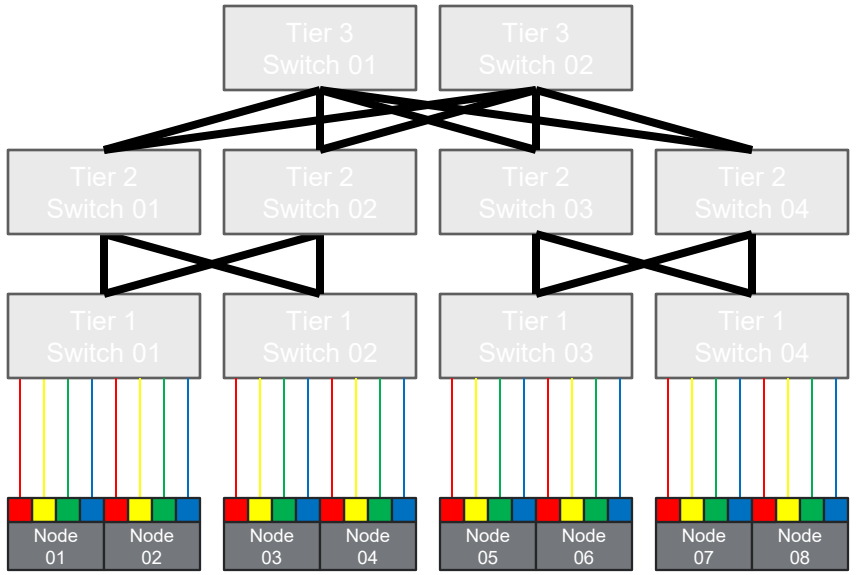


Rail-optimized topologies may also focus traffic away from spine, enabling rings that exist purely within the first tier of switches.

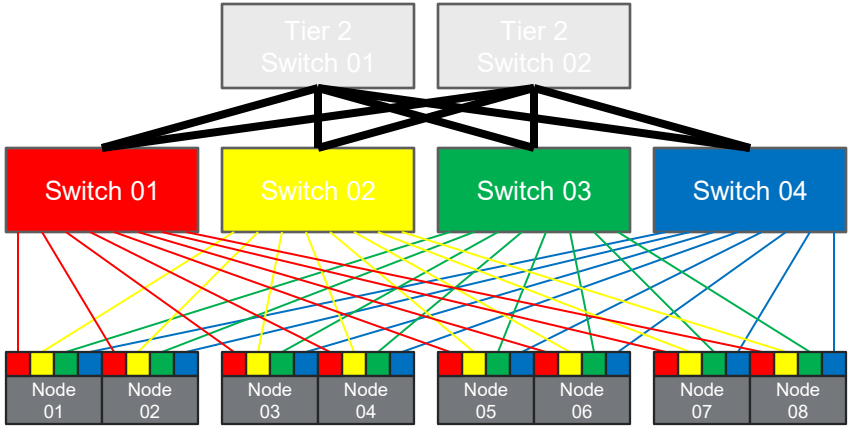
Some usage patterns will perform worse with rail-optimized topologies – e.g. traffic between two NICs on the same node now gets routed through the spine

Rail-Optimized Topologies

Classic fabric



Rail-optimized



For nodes with high-speed internal interconnects (XGMI/NVSwitch), rail-optimized topologies may also choose to decrease overall switch costs by utilizing fewer switches in the spine

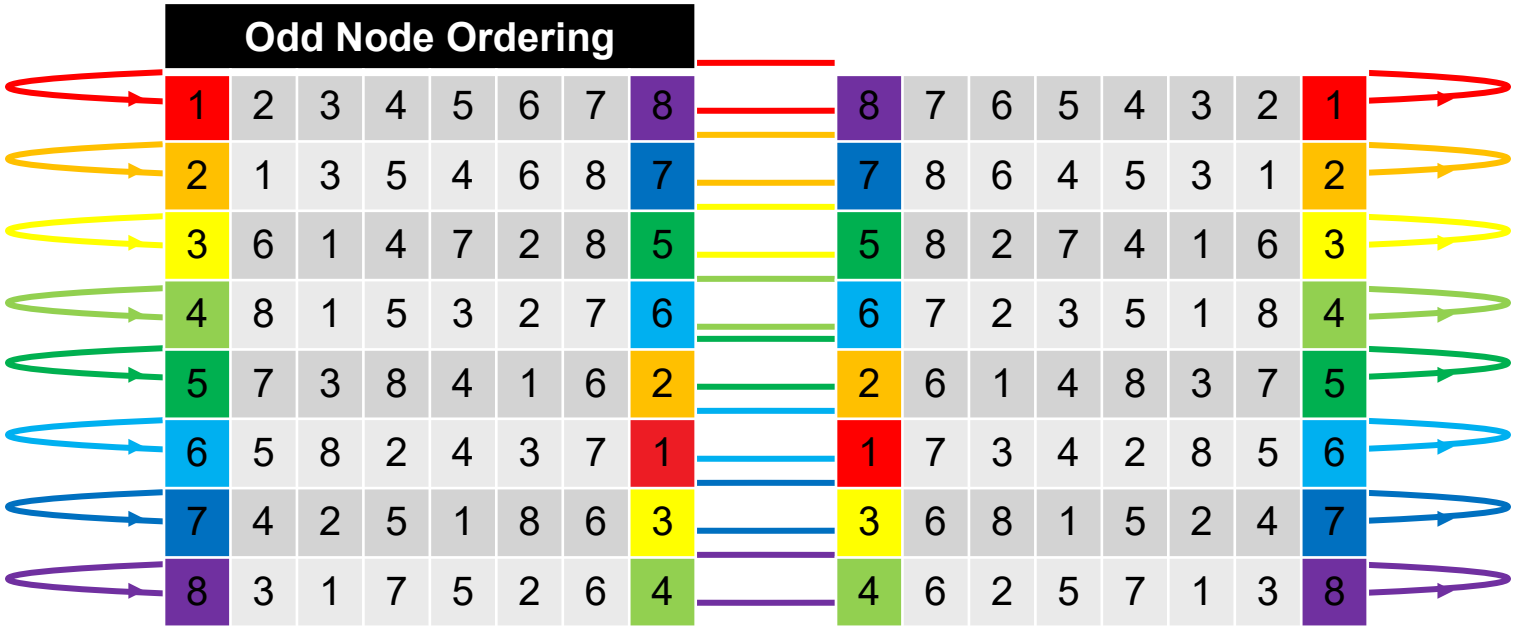
Default RCCL Algorithms

NCCL_ALGO = Ring/Tree

Collectives	Algorithms	Scale up	Scale out
Allreduce	Ring, Tree	Yes	Yes
Allgather	Ring	Yes	Yes
Reduce-Scatter	Ring	Yes	Yes
AlltoAll	Direct	Yes	Yes
Broadcast	Ring	Yes	Yes
Reduce	Ring	Yes	Yes

Ring Collectives (Rail-Optimized) [Current implementation]

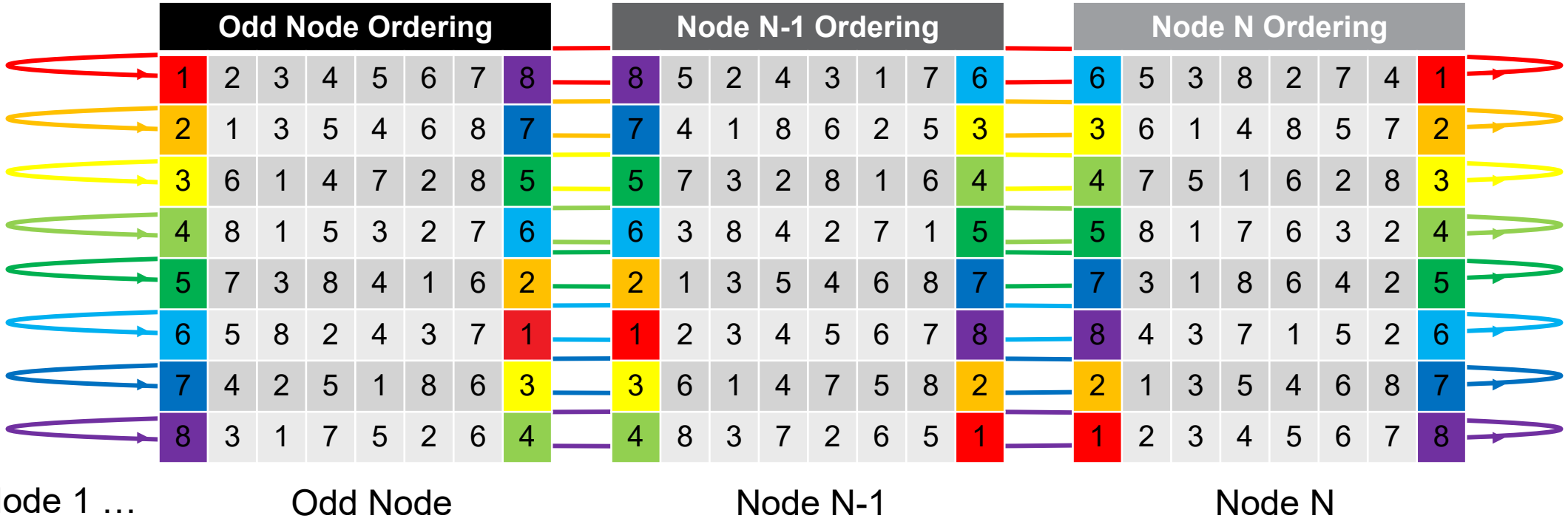
When total number of nodes is even:



In the current implementation, nodes define their own lines differently, based on if they are odd or even.
By traversing the line forwards, then backwards, NIC traffic no longer “jump” rails.

Ring Collectives (Rail-Optimized) [Current implementation]

When total number of nodes is odd:



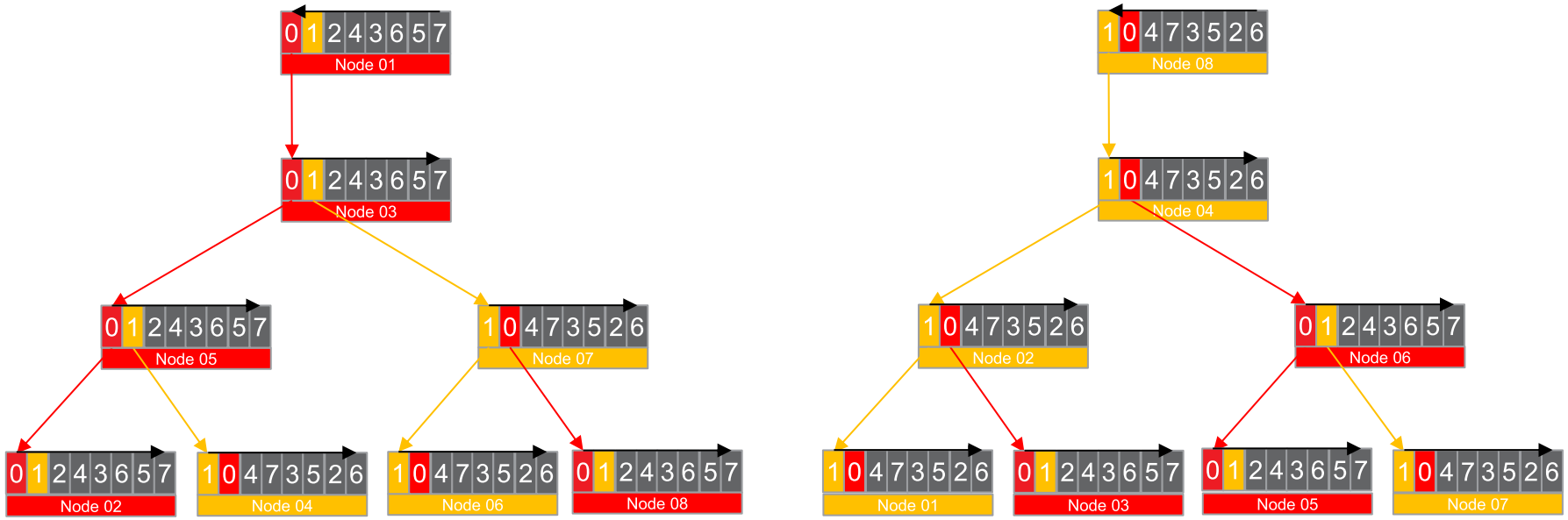
If there are an odd number of nodes, the last two nodes use the above set of lines.
Together, they hit every internal XGMI link once, and restrict NIC traffic to only within rails

Tree Collectives

- Ring-based AllReduce collectives are bandwidth-optimal (each rank sends/receives the least possible amount, however more steps are required as the number of ranks scale up
  - For example, for 8 ranks, in order to perform a 1GB AllReduce, each rank actually sends/receives 1.75GB.
- Tree-based AllReduce sends more data, however requires fewer steps

Total Ranks	Data Multiplier			Steps		
	Ring	Tree	Diff	Ring	Tree	Diff
8	1.75	2.00	114%	14	6	43%
16	1.88	2.00	107%	30	8	27%
32	1.94	2.00	103%	62	10	16%
64	1.97	2.00	102%	126	12	10%
128	1.98	2.00	101%	254	14	6%
256	1.99	2.00	100%	510	16	3%

MI300X Multinode Trees (Rail-optimized) (Current)



Each subtree keeps the same left subtree, but alternates on right subtrees

Tuning Models

RCCL tuning is a table-driven analytical cost model

- It is not a runtime auto-tuner
- Pre-measured, empirical constants per GPU architecture
- Chooses entry with lowest cost
  - Based on architecture / topology / collective / message size
  - Outputs algorithm / protocol / channel count

Every GPU arch maps to a tuning model

```
int rcclGetTuningIndexForArch(const char* gfxarch) {
  static const ... tuningIndexMap = {
    {"gfx906", 0}, {"gfx908", 0}, {"gfx90a", 0},
    {"gfx942", 5},  // MI300
    {"gfx950", 6},  // MI350
    {"gfx1200", 7}, {"gfx1201", 7},
  };
  ...
  return 0;  // default
}
```

What a model holds

`hwLat` per-hop latency (XGMI / PCIe / net)

`bwRatio` % of raw bandwidth each combo hits

`tree / ringCorrectionFactor` size-bucketed fudge curves

`llProtoRanges` when to force LL / LL128

`channelThresholds` channel count by size

Topology-Aware Tuning with Rome Models

What they are

- **A fingerprint of a real server**
  GPU/CPU/NIC counts, PCI IDs, NUMA affinity, XGMI connection matrix, GDR levels.
- **Hand-picked routings**
  Optimal ring & tree GPU orderings (ringBase / treeBase / treeRail).
- **Matched at init**
  RCCL fingerprints the live machine, then scans for an exact match (4P2H, 16P1H, 4H4P, ...).

Why it matters for tuning

On a match, the model's **`options`** string feeds tuning directly — most importantly **`tuning=N`**, which overrides the arch default and picks a topology-specific cost table.

```
.options = "tuning=5,ll128Enabled=1,
  treeDefined=1,baseBw=161.4"

tuning=N        → which cost-model table
treeDefined     → force Tree algorithm
ll128Enabled    → allow LL128 proto
baseBw          → baseline bandwidth
pivotA2AEnabled→ special all-to-all
```

No match → RCCL falls back to generic graph search + the arch-based tuning index.

External Tuner Plugin

- **RCCL fills a cost table**
  `One predicted time per algorithm × protocol.`
- **Your plugin gets it**
  `getCollInfo(collType, nBytes, nNodes, costTable, nChannels)`
- **You override**
  `Set a cell to 0.0 to force it; set *nChannels; leave rest.`
- **RCCL picks the min**
  `Your zeros win — then normal launch continues.`

How to enable

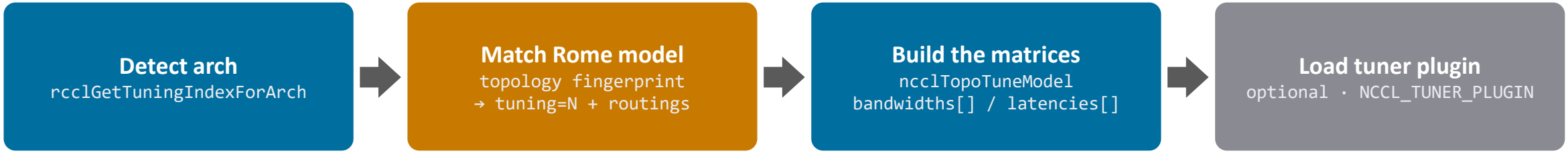
```
export NCCL_TUNER_PLUGIN=libmytuner.so
```

Versioned interface v2 – v5.

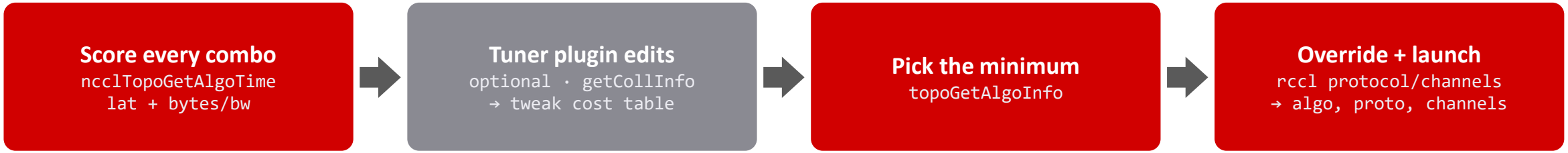
Reference plugins live under ext-tuner/ (config-file + math-model demos).

End-to-end Tuning Flow

AT INIT (ncclCommInitRank) · once per communicator



PER COLLECTIVE · every launch



core logic in src/graph/tuning.cc, runtime selection in src/enqueue.cc

Environment Options for Tuning

Standard NCCL (honored by RCCL)

`NCCL_ALGO / NCCL_PROTO`
Enable/disable algorithms or protocols

`NCCL_MIN_NCHANNELS / NCCL_MAX_NCHANNELS`
Bound the channel count

`NCCL_NTHREADS`
Threads per block / thread tuning

`NCCL_DEBUG=INFO + NCCL_DEBUG_SUBSYS=TUNING`
Dump the lat/bw matrix at init

RCCL-specific

`RCCL_OVERRIDE_ALGO / RCCL_OVERRIDE_PROTO`
Force a choice after cost selection

`RCCL_CHANNEL_TUNING_ENABLE`
Toggle channel overrides (default 1)

`RCCL_LL128_FORCE_ENABLE`
Force the LL128 protocol

`RCCL_DIRECT_*_THRESHOLD`
Direct algorithm message size thresholds

Outline

- Introduction to collectives
- Introduction to RCCL
- Building RCCL and RCCL Tests
- Running RCCL Test
- RCCL Design and Implementation: Deep Dive
- New RCCL Features

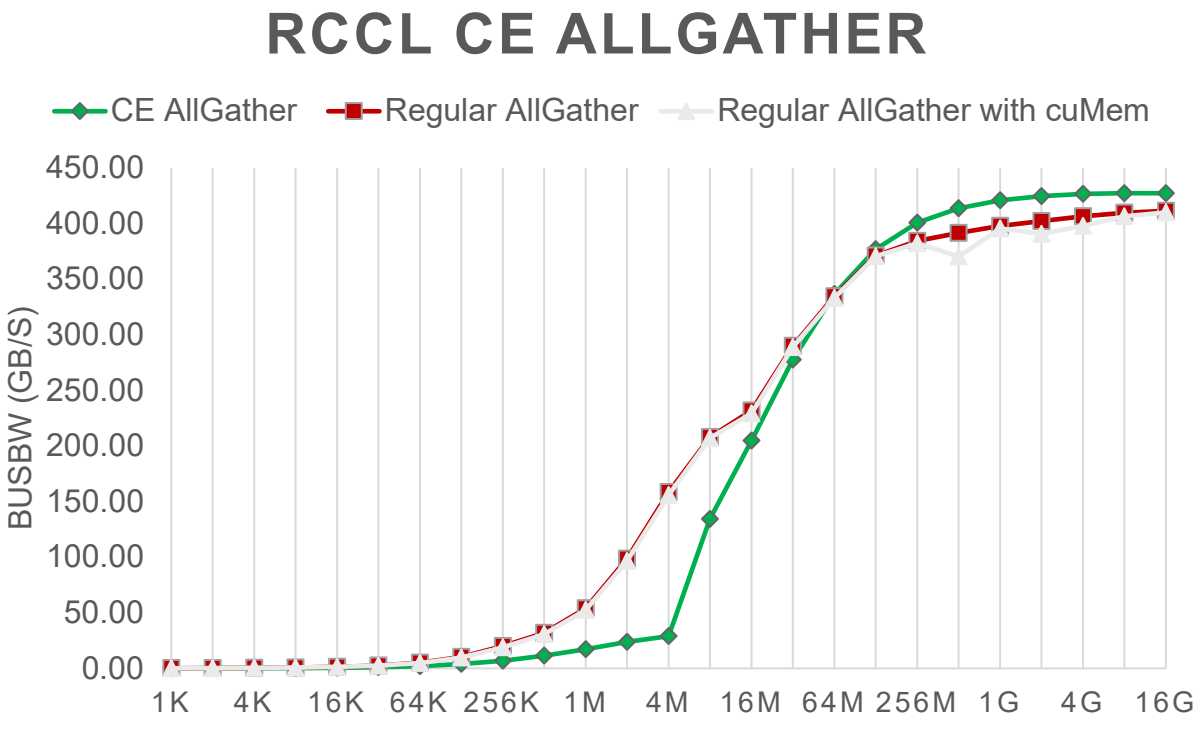
RCCL New Features

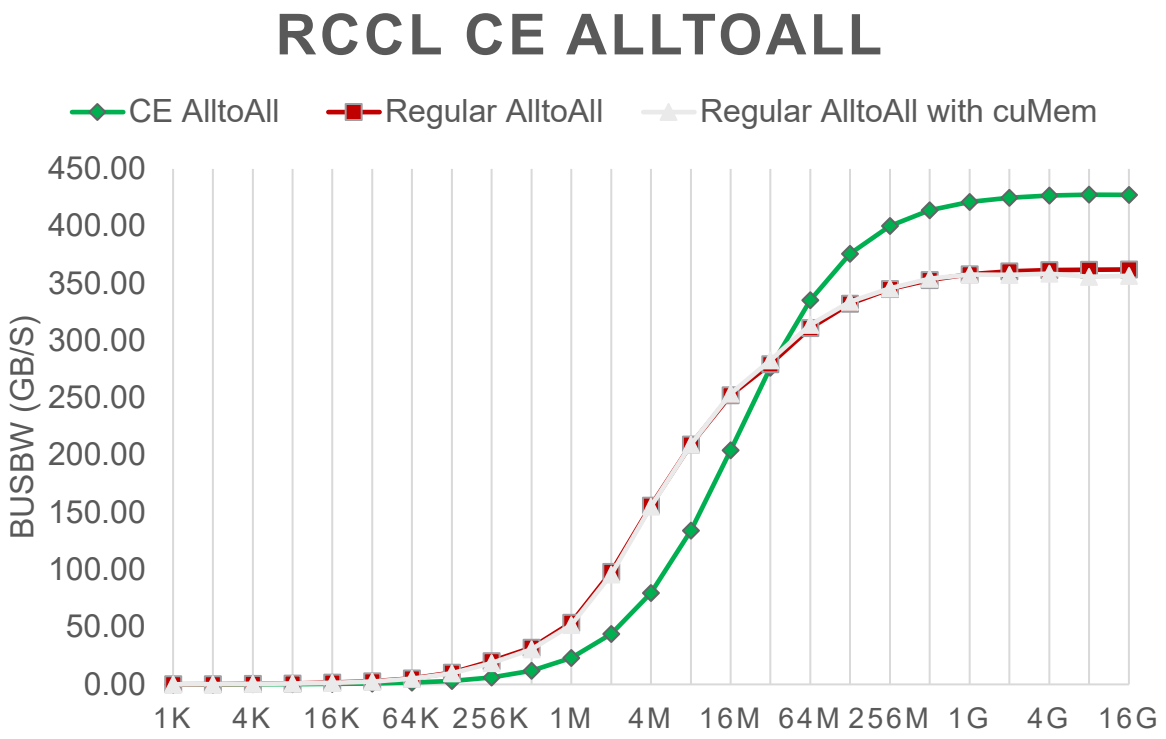
- SDMA (Copy Engine) Collectives to overlap with Compute
- Low Latency Collectives for Single and Multiple Nodes
- Device-Initiated Communication
- Load Balancing to Reduce Congestion

Use SDMA (Copy Engines) to Move Data across GPUs

- Added collectives that use SDMA for data-transfer only collectives
  - allgather, alltoall, gather and scatter
  - Support for alltoallv in progress
- SDMA engines perform data transfers so that compute resources are fully available to GEMMs and other compute operations
  - Allows better overlap communication-computation
- Currently enabled within the scaleup domain when using symmetric memory window registration and NCCL_CTA_POLICY_ZERO communicator flag
- Functional at ROCm 7.12 and backported to ROCm 7.0.2.2

CE collectives in RCCL achieve peak bandwidth



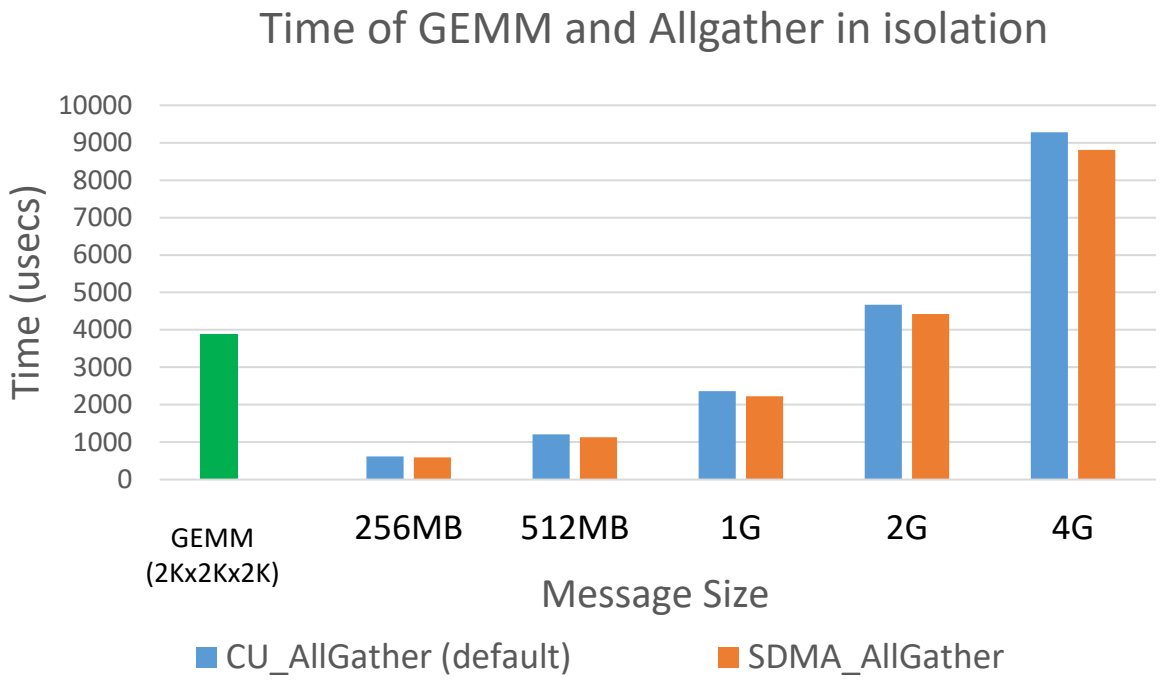


CE collectives achieve a peak bandwidth (428 GBps) on MI350; higher than CU-based RCCL collectives

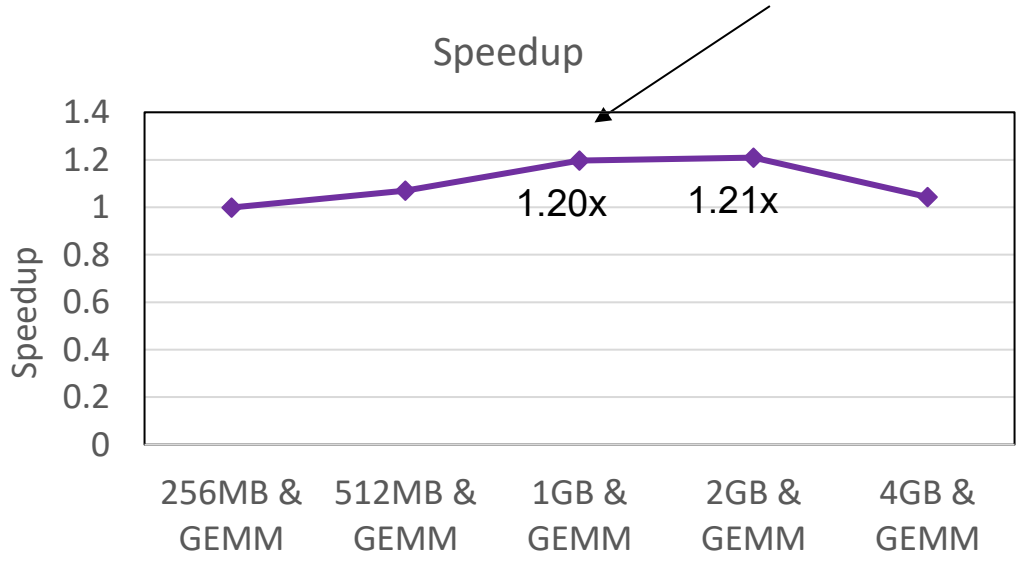
```
mpirun -np 8 --allow-run-as-root --bind-to numa --mca pml ucx --mca btl ^openib -x PATH=${PATH} -x LD_LIBRARY_PATH=rccl/build/release:${LD_LIBRARY_PATH}
-x NCCL_IGNORE_CPU_AFFINITY=1 -x HSA_NO_SCRATCH_RECLAIM=1 -x NCCL_DEBUG=INFO -x NCCL_DEBUG_SUBSYSTEM=TUNING,INIT,COLL -x
NCCL_CUMEM_ENABLE=1 rccl-tests/build/alltoall_perf -b 1024 -e 16G -f 2 -g 1 -N 2 -n 20 -c 1 -R 2 -x 2
```

SDMA Collectives perform better when run alongside GEMM

- Performance is up-to 1.21x faster when GEMM runs concurrently with an SDMA-based Allgather versus a CU-based Allgather



Speedup of GEMM+Allgather with SDMA



Test configurations

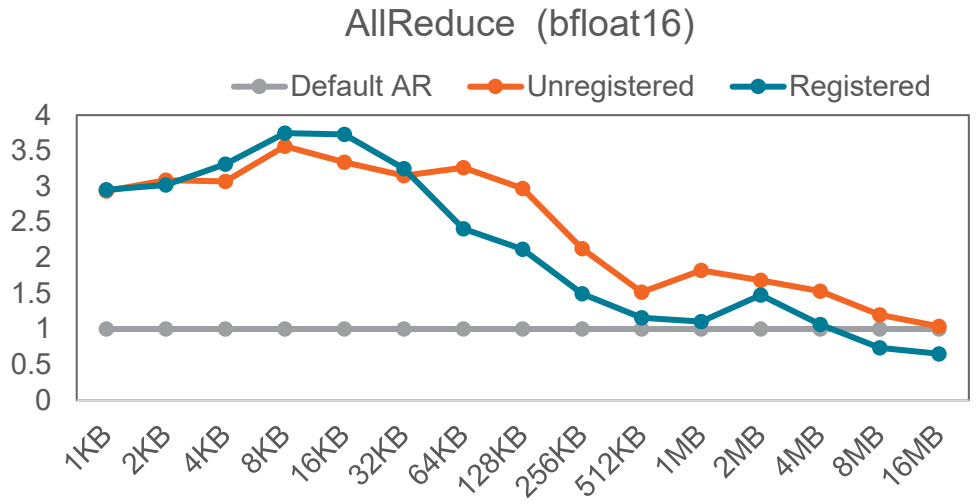
- 8GPUs in AMD Instinct™ MI355 single node
- GEMM: naïve M=K=N=2048 (~3.9ms)
- Allgather and GEMM on separate HIP streams

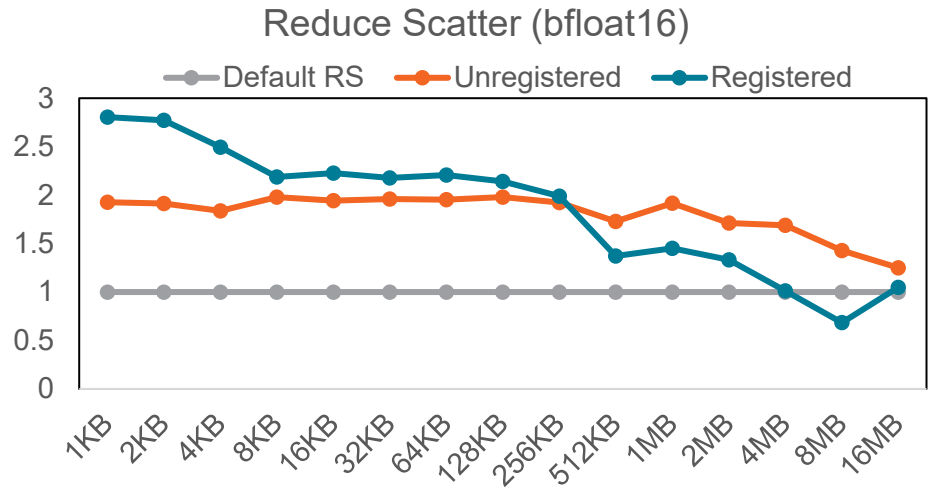
Low Latency for Small/Medium Message Collectives (Single Node)

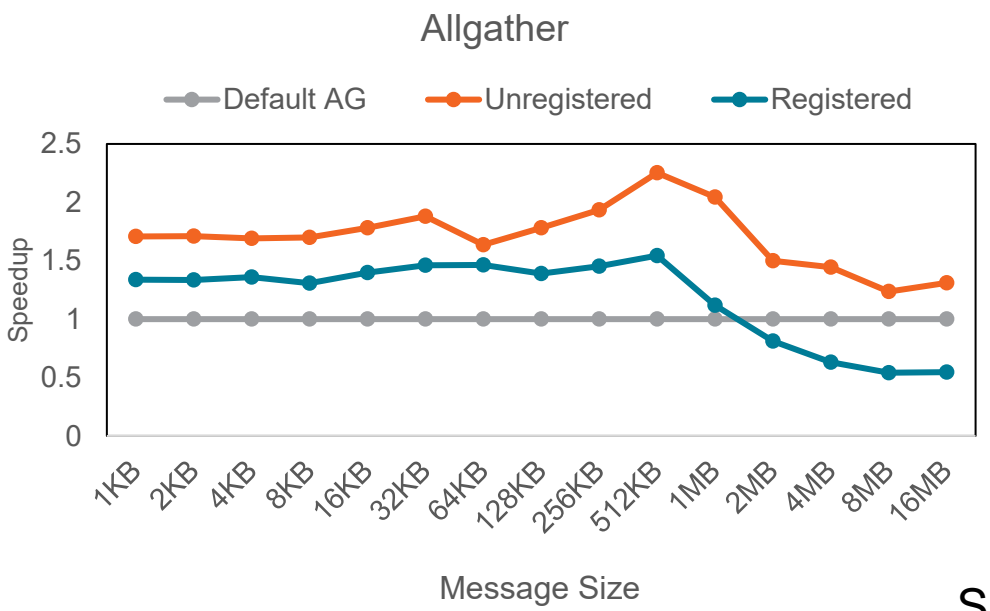
- Ring algorithm incurs high latency for small/medium messages
- Added one and two shot algorithms to significantly reduce latency
- Provided solutions for registered and un-registered buffers

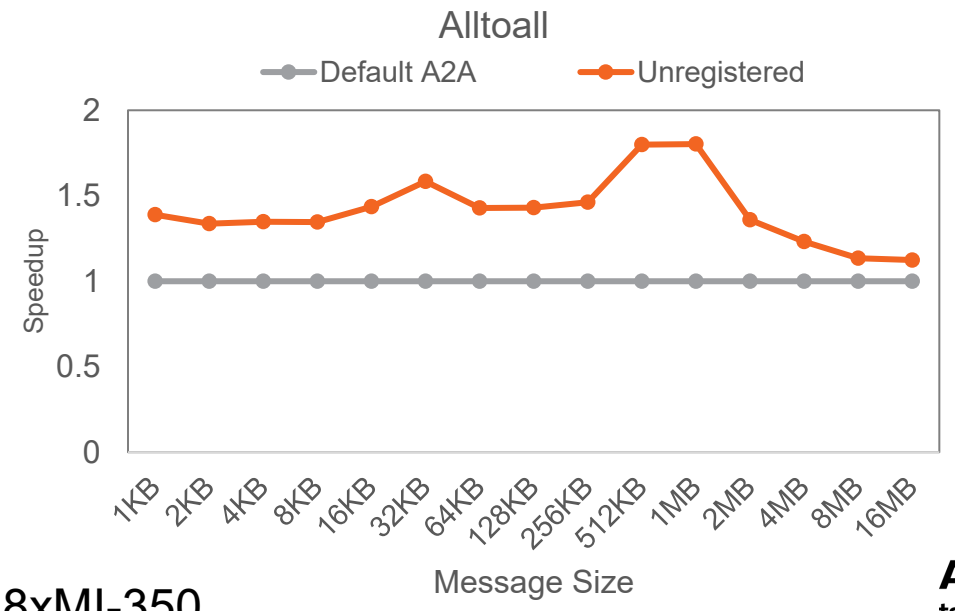
- **Why did we add support for unregistered buffers?**
  - User buffer registration requires changes in the application
  - Support for unregistered buffers provides low latency without app changes

New Algos Faster for small/medium messages on Single Node









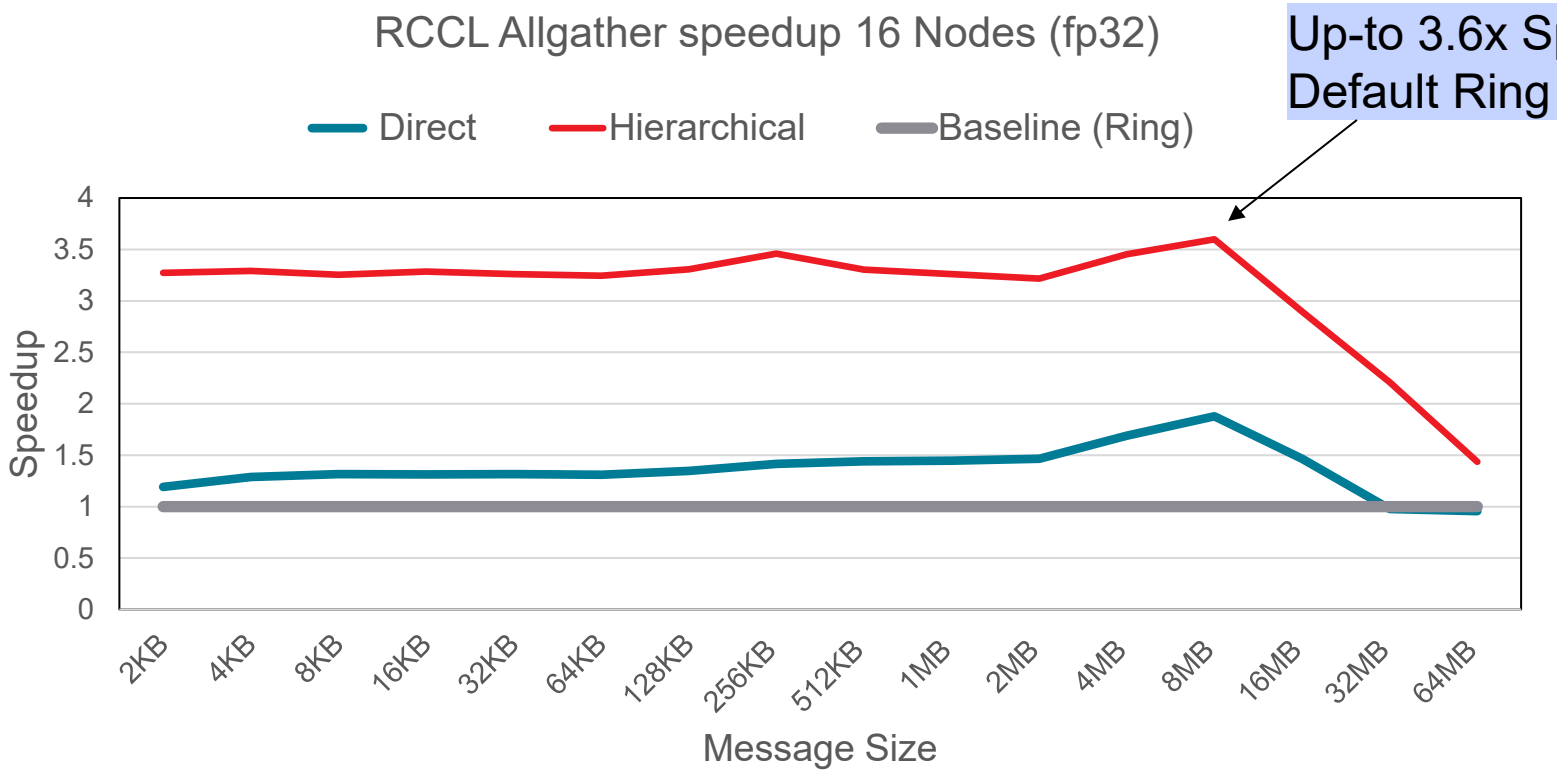
Single node 8xMI-350

Low Latency with Hierarchical Collectives (multi-node)

- Current solutions are not optimal
  - Ring algorithm incurs high latency for small/medium messages
  - Direct Algorithms suffer from concurrency contention
  - PAT is only enabled for single GPU per node
- Enable Hierarchical algorithms that are topology-aware

- Example: Allgather
  - Decompose the flat communicator into topology-aware phases:
  - Inter-comm: groups GPUs with the same local_rank across nodes
    - Perform inter-node Allgather using PAT
  - Intra-comm: groups GPUs within the same node
    - Perform intra-node Allgather using direct/Ring/One shot/Two Shots

Faster Allgather with Hierarchical Algorithm (multi-node)



Up-to 3.6x Speedup versus Default Ring Algorithm

Hierarchical RS is coming

Hierarchical Allgather delivers significant speedups with respect to default Ring algorithm on 16 nodes (8 GPUs on each MI-350 node)

Advanced Collective Algorithms in RCCL

RCCL_DDA_ENABLE = 1
RCCL_HIERARCHICAL_ALLGATHER

Collectives	Algorithms	Scale up	Scale out
Allreduce	Direct Data Access (DDA)	Yes	No
Allgather	Direct Data Access (DDA)	Yes	No
Allgather	Direct, Hierarchical	Yes	Yes
Reduce-Scatter	Direct Data Access	Yes	No
Reduce-Scatter	Direct	Yes	Yes

GPU-initiated Communication

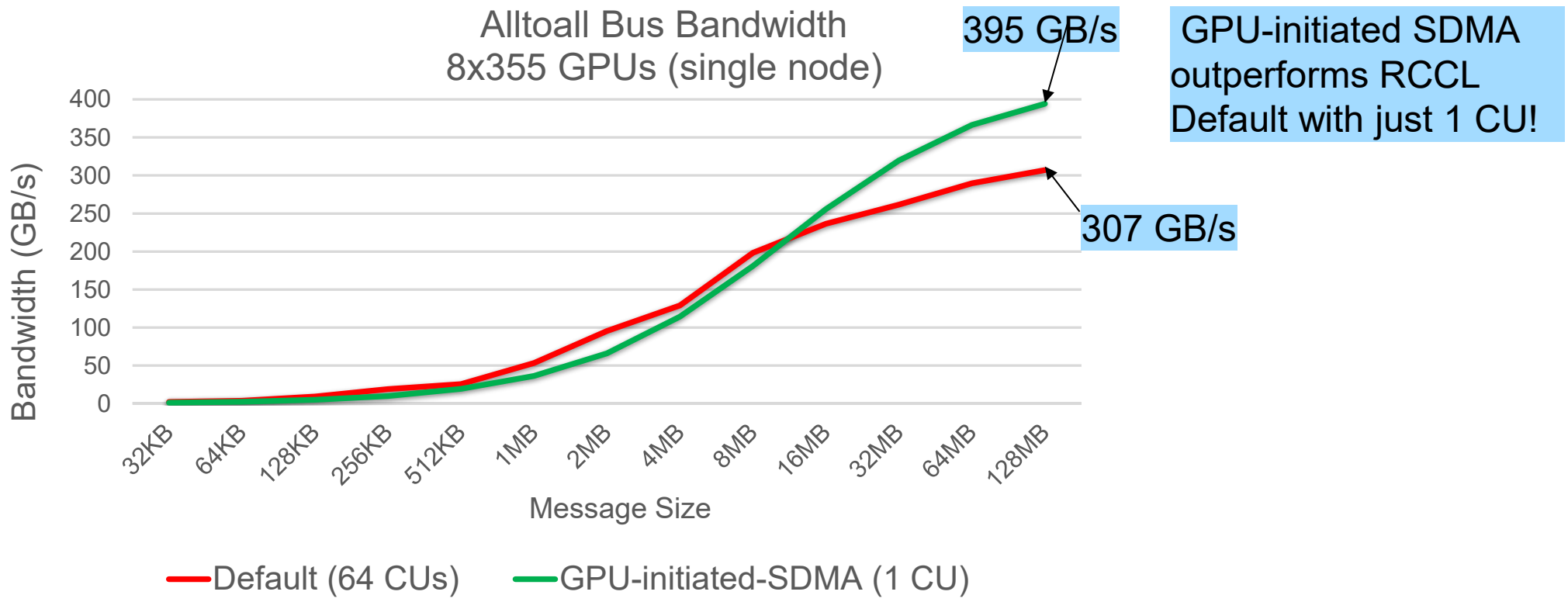
- Enabled
  - GPU-initiated Load-Store Accessible (LSA) APIs for scaleup
  - GPU-Initiated Networking (GIN) APIs for scaleout

- For scaleup, we also support GPU-initiated using SDMA (using single CU)

- For GIN, the GPUDirect Async Kernel-Initiated supports three NICs
  - AMD Pensando™ Pollara AI NIC
  - Broadcom ThorX
  - NVIDIA® ConnectX®-7

Next slides will show some performance for these features. The code is currently under review and should be available soon.

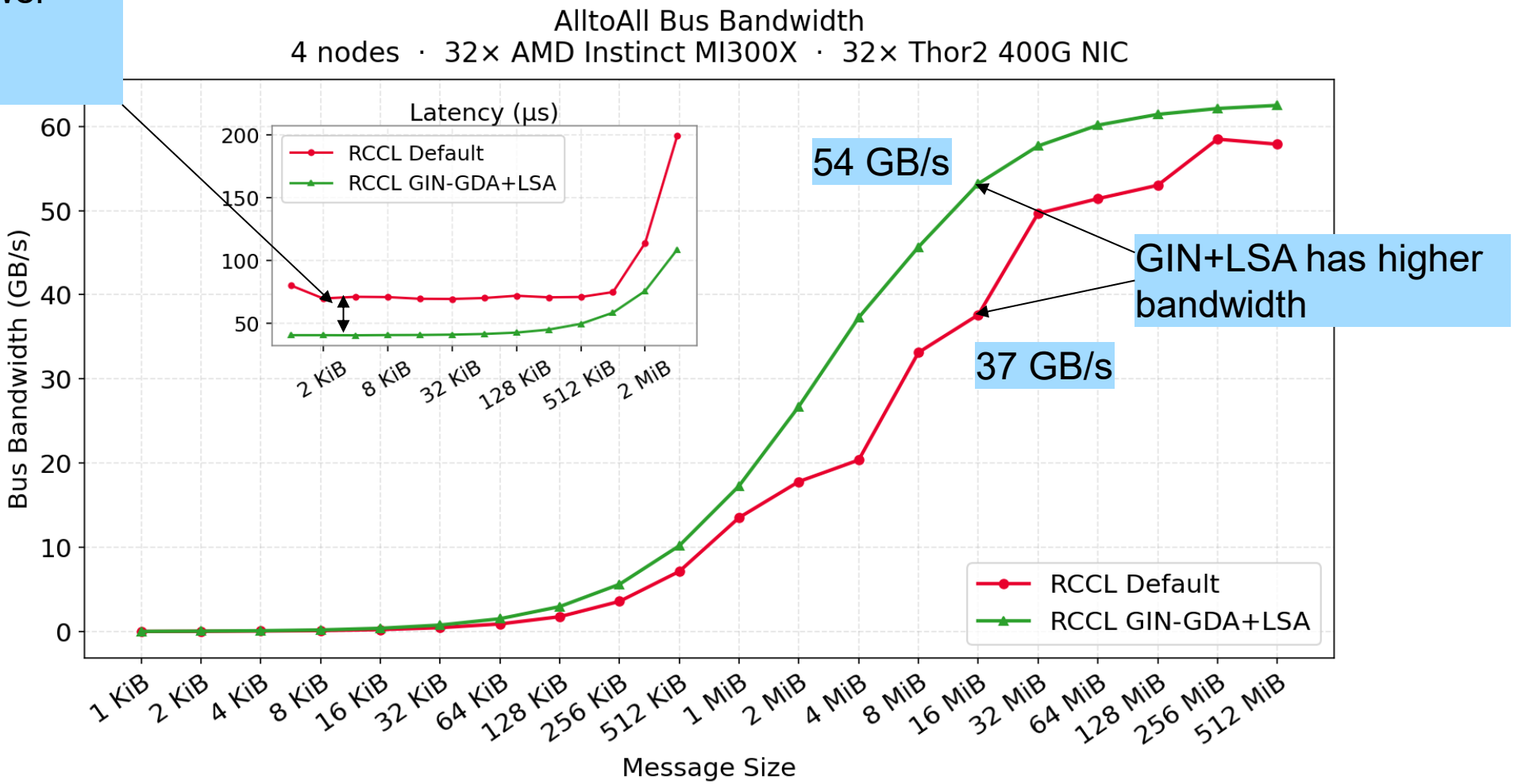
Alltoall with GPU-initiated-DMA outperforms RCCL Default



Device-Initiated SDMA supports data movement across GPUs, while only using one Compute Unit

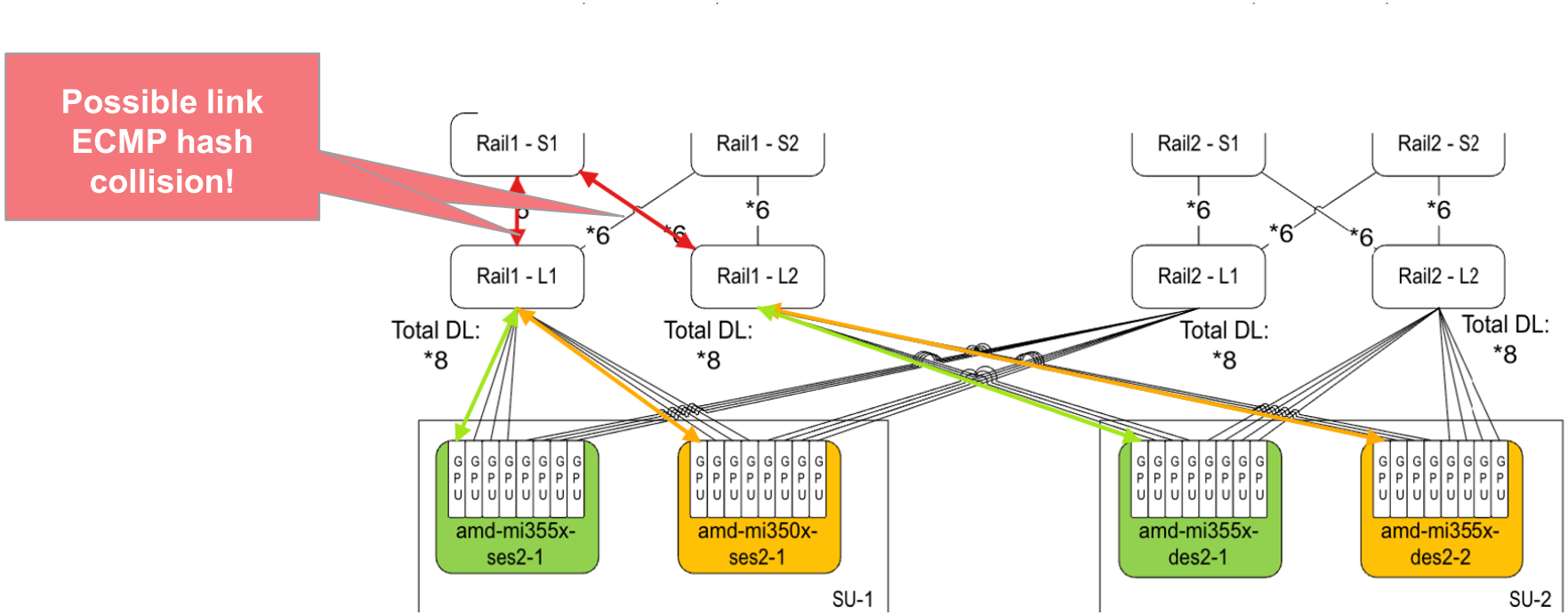
Alltoall with GIN+LSA outperforms RCCL Default

GIN+LSA has lower latency for small messages



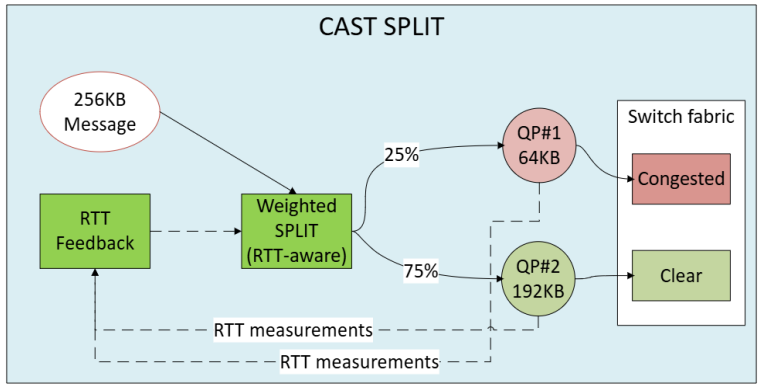
Congestion: ECMP is not enough to avoid/reduce congestion

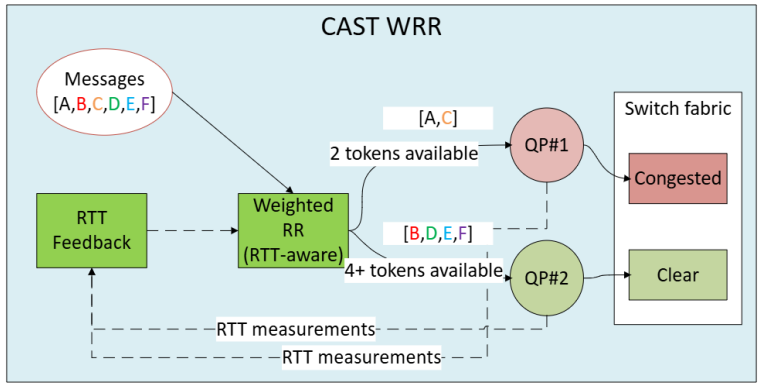
- RoCE v2 uses IP/UDP headers for routing, and ECMP (Equal Cost Multi-Path) distributes flows across available paths by header at each router.
- ECMP hash collisions can create severe bottlenecks along a path, particularly when "elephant flows" end up sharing the same links.

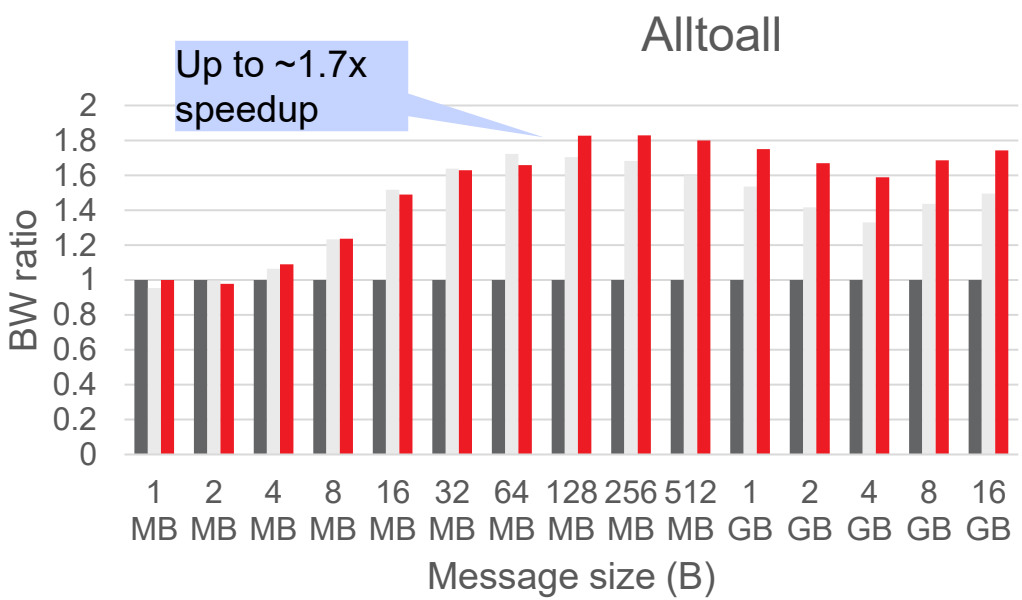


Reduce congestion through dynamic load balancing

- Added CAST (Congestion Aware Sprayed Traffic) to RCCL
  - RCCL already supported QP load balancing by spraying data across multiple QPs (Round-robin, Split)
- CAST uses Round-Trip Time (RTT) information to dynamically distribute the load across the QPs between two endpoints (weighted split (CAST SPLIT) for large messages or weighted distribution (CAST WRR) for small messages







CAST helps decrease traffic collision on spine switch

NCCL_NET=IB-CAST
NCCL_IB_QP_SCHED_ENABLE=1

2 nodes 8xMI-355 with Pensando/Pollara NICs

15 min break

Part 3: rocSHMEM features and tutorial

Modern HPC and data centers are already GPU-centric

Why SHMEM?

System reality

GPUs now supply most throughput and memory bandwidth.

Application reality

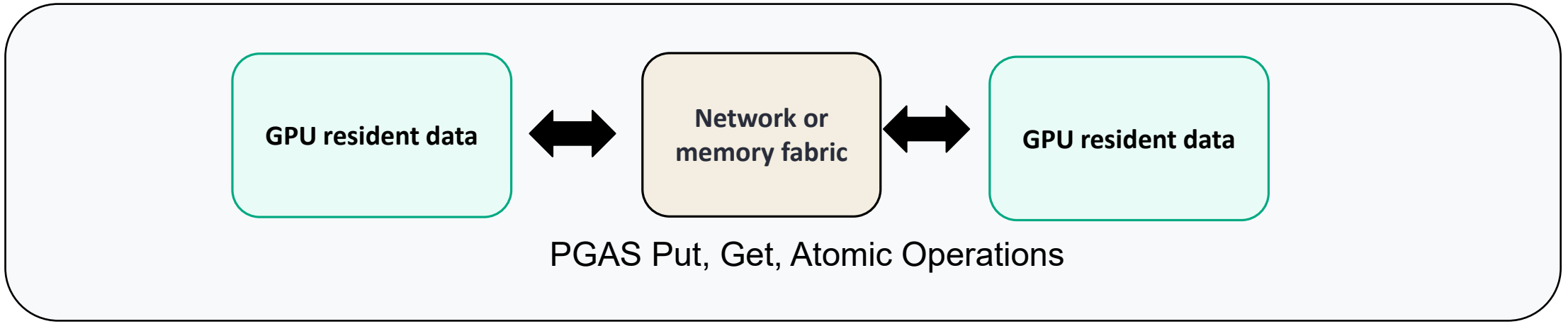
Sparse, halo, FFT, AI, and analytics phases all need fine-grain exchange.

Residency reality

Performance and energy both favor keeping data on device.

Programming-model reality

Communication must compose with kernels instead of forcing data back to the host.



The talk is about how rocSHMEM enables this GPU-resident communication possible

SHMEM: Why It Makes Sense for GPUs

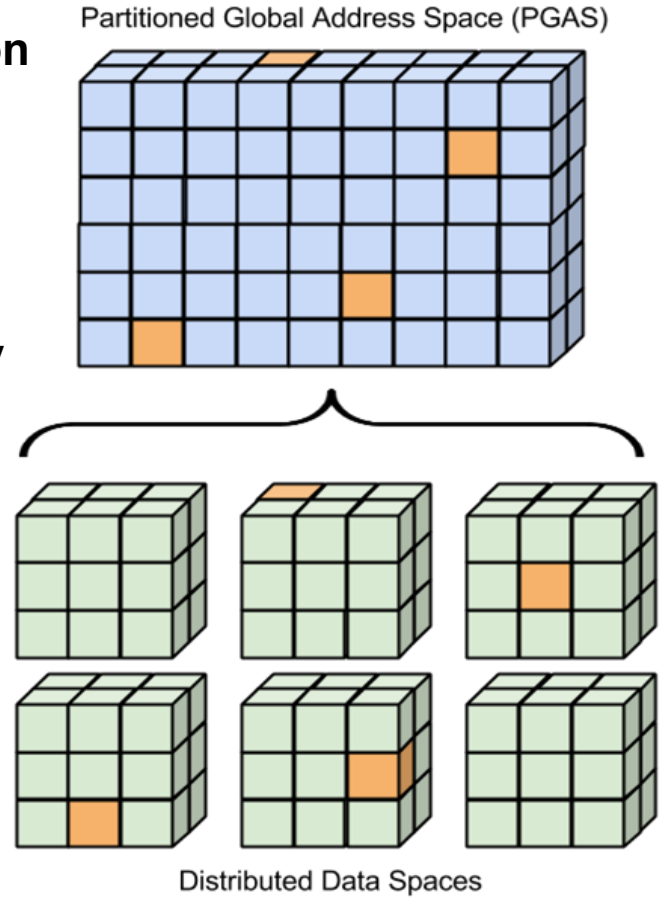
PGAS Model (SPMD) library specification
Same program, different data
Processing Elements (PEs)

Memory Model
Symmetric allocations (same layout)
Fast remote access (pointers)
No coherence - user manages consistency

Communication (One-sided)
Put / Get / Atomics
Put-with-signal

Synchronization
Quiet, Fence
Wait/Test, Barrier

Collectives
Reduce, Broadcast, …



Open SHMEM http://www.openshmem.org

What is rocSHMEM

- rocSHMEM: extension of the OpenSHMEM specification for AMD Instinct™ GPUs
- **Support for GPU initiated communication operations**

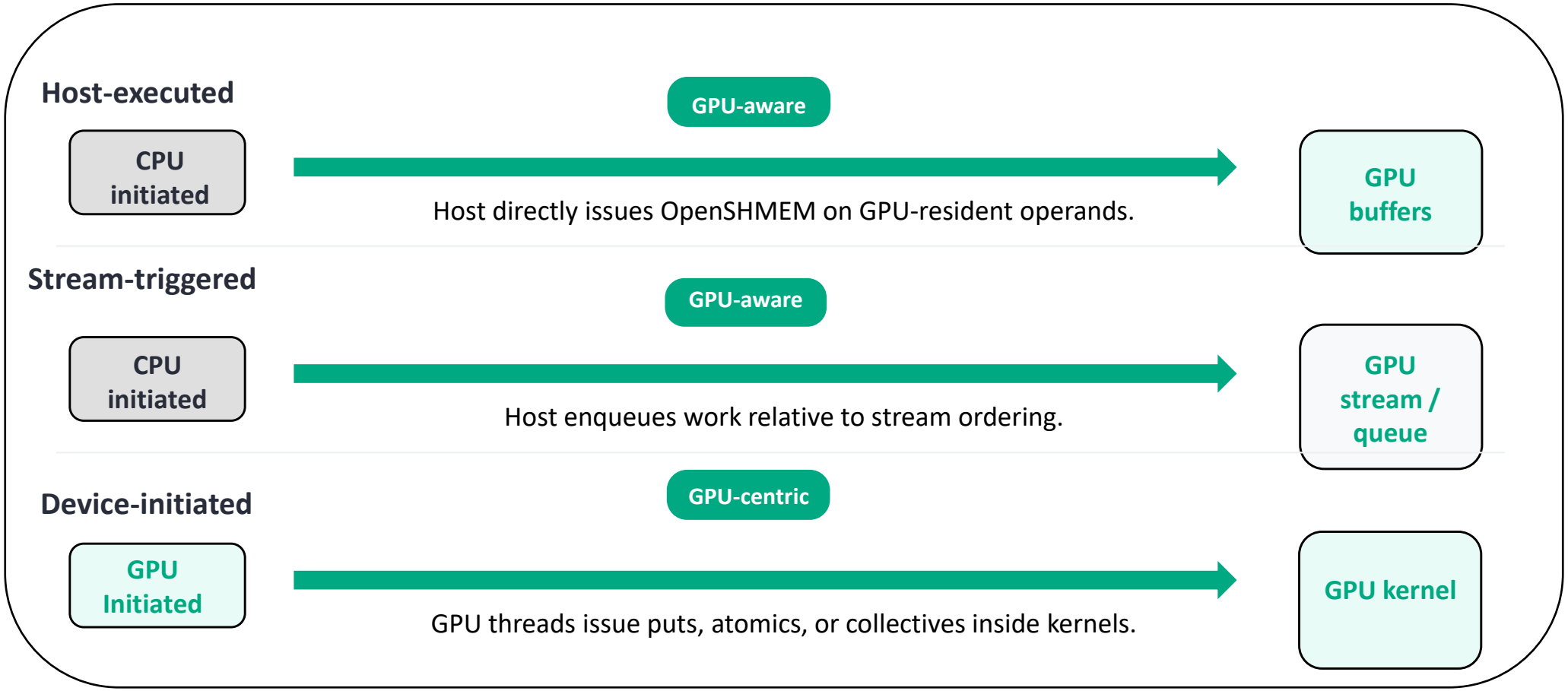
- Target applications
  - HPC: FFT, BLAS, etc.
  - LLM: MoE, DeepEP, SGLang, etc.
  - Languages and frameworks: Triton, Composable Kernels, etc.

- Started as a project at AMD Research
  - see e.g. *K. Hamidouche, and M. LeBeane. "GPU initiated OpenSHMEM: correct and efficient intra-kernel networking for dgpus." Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2020.*
  - Designed independently from other device-initiated SHMEM libraries
- Joint effort between the AMD GPU communication libraries team and AMD Research to productize rocSHMEM
  - Single node version part of AMD ROCm™ 6.4 as beta-preview (scale-up)
  - Support for multi-node scale-out added in ROCm 7.0 (without GDA)
  - GDA communication added to accelerate scale out in ROCm 7.1
  - Optimization and API coverage improvements, Python and Triton support

ROCm 6.4	ROCm 7.0	ROCm 7.1	ROCm 7.2	ROCm 7.14
Q2 2025	Q3 2025	Q4 2025	Q1 2026	Q2 2026
rocSHMEM - IPC	rocSHMEM - RO	rocSHMEM - GDA	rocSHMEM - GDA	rocSHMEM - API
GPU ⟺ GPU SHMEM for intranode communication	Uses a host-side thread to trigger GPU ⟺ GPU one-sided MPI operations	Uses Direct Verbs for GPU ⟺ GPU data transfers bypassing CPU	GDA on AMD Pollara NIC autodetection on startup Collectives on GDA Python bindings	Added on-stream APIs VMM symmetric heap buffer registration, Tile API SDMA

GPU Centric vs GPU aware

- rocSHMEM supports all three operational modes:
- **Device initiated, Host initiated, Host initiated on stream**



Quick list of features

C++ API, C API, Python API

Device, Host, on-stream API

RMA communication

- put, put_wave, put_wg
- get, get_wave, get_wg

RMA communication, non-blocking

- put_nbi, put_nbi_wave, put_nbi_wg
- get_nbi, get_nbi_wave, get_nbi_wg

Atomic operations

- atomic_add, fetch_add, inc, fetch_inc
- atomic_and, or, xor, fetch_and, fetch_or, fetch_xor
- atomic_set, fetch, swap, compare_swap

Ordering operations

- fence, quiet, quiet_pe

Signal Operations

- put_signal, put_signal_nbi, signal_fetch

Synchronization routines

- wait, wait_until, wait_until_all, wait_until_some, wait_until_*_vector, signal_wait_until
- test, test_all, test_any, test_some, test_*_vector

Collective communication

- barrier, sync, alltoall, *alltoallv*, broadcast, fcollect, scatter_reduce

New rocSHMEM features in RoCM 7.14

- Added **IPC tile-granular RMA** operations
  - rocshmem_tile_put/get (also wave/wg)
  - tile_allgather, tile_broadcast (wave/wg)
- **Python bindings**
  - Added Python bindings of memory-management APIs
  - Added Python bindings coverage for team APIs
- **SDMA support**
  - Added support for GPU initiated operations using the SDMA engines
- Added a new **on-stream APIs**
  - rocshmem_reduce_on_stream
- Host-API IPC w/o MPI support
  - Added **IPC host-initiated** RMA operations backend **without MPI requirement**
- Added new memory management APIs:
  - **rocshmem_align**, rocshmem_calloc
  - rocshmem_buffer_unregister_all
  - **rocshmem_buffer_register/unregister** for GDA backend
- Team splitting API
  - Added rocshmem_team_split_2D
  - Support for team creation using non-contiguous parent teams in the IPC backend
- Added ASAN build support

rocSHMEM APIs

- Extension to OpenSHMEM to enable **device initiated communication**
- APIs are **not identical to NVSHMEM**, but many commonalities
- GPU initiated communication operations per thread

```
void nvshmem_putmem()  ──────►  void rocshmem_putmem()
void nvshmem_getmem()  ──────►  void rocshmem_getmem()
```

- **Thread-group based interfaces** use AMD nomenclature

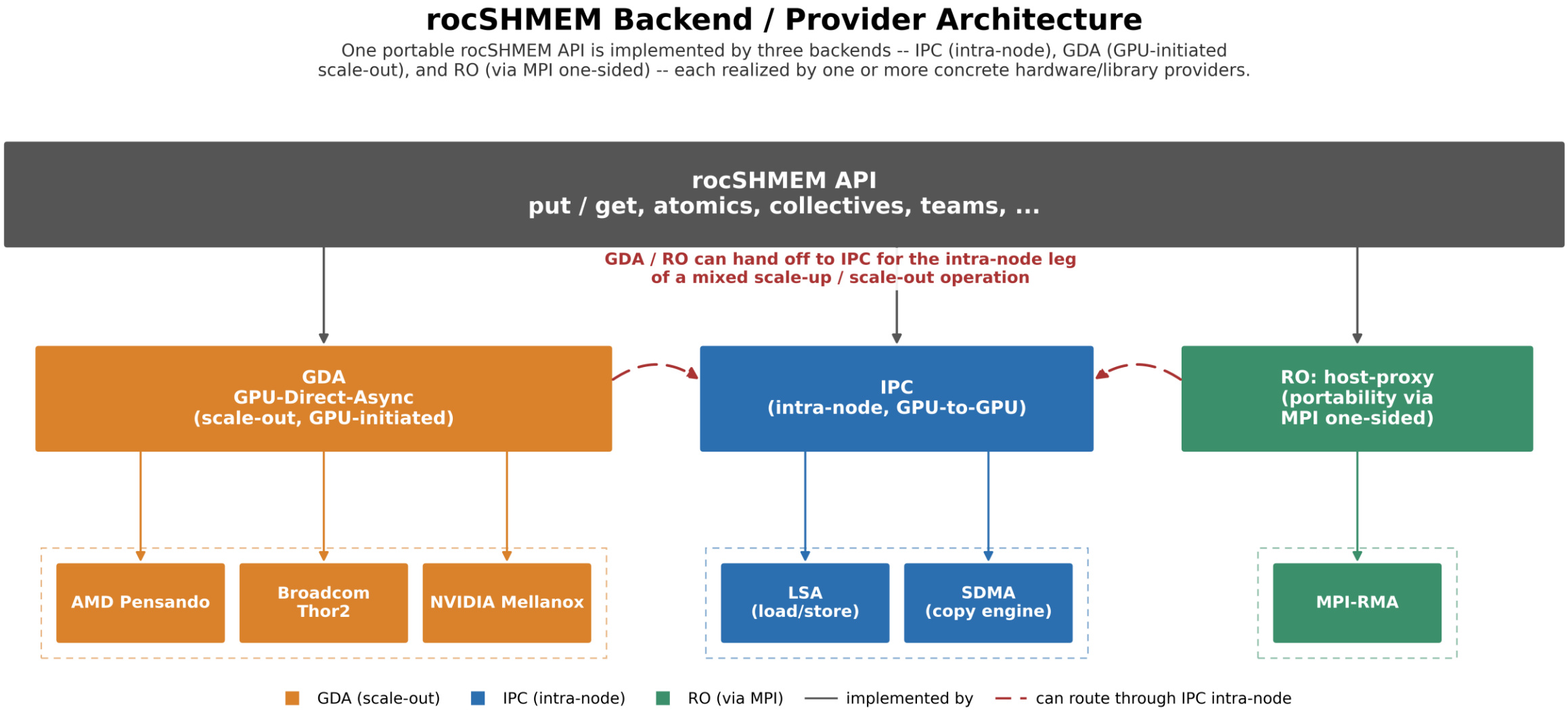
```
void nvshmem_putmem_block()  ──────►  void rocshmem_putmem_wg()
void nvshmem_getmem_block()  ──────►  void rocshmem_getmem_wg()
void nvshmem_putmem_warp()   ──────►  void rocshmem_putmem_wave()
void nvshmem_getmem_warp()   ──────►  void rocshmem_getmem_wave()
```

- Collective operations take an additional ctx argument in rocSHMEM

```
int nvshmemx_alltoall_int_block  (shmem_team_t team, int *dest,
                                  const int *source, size_t nelems)

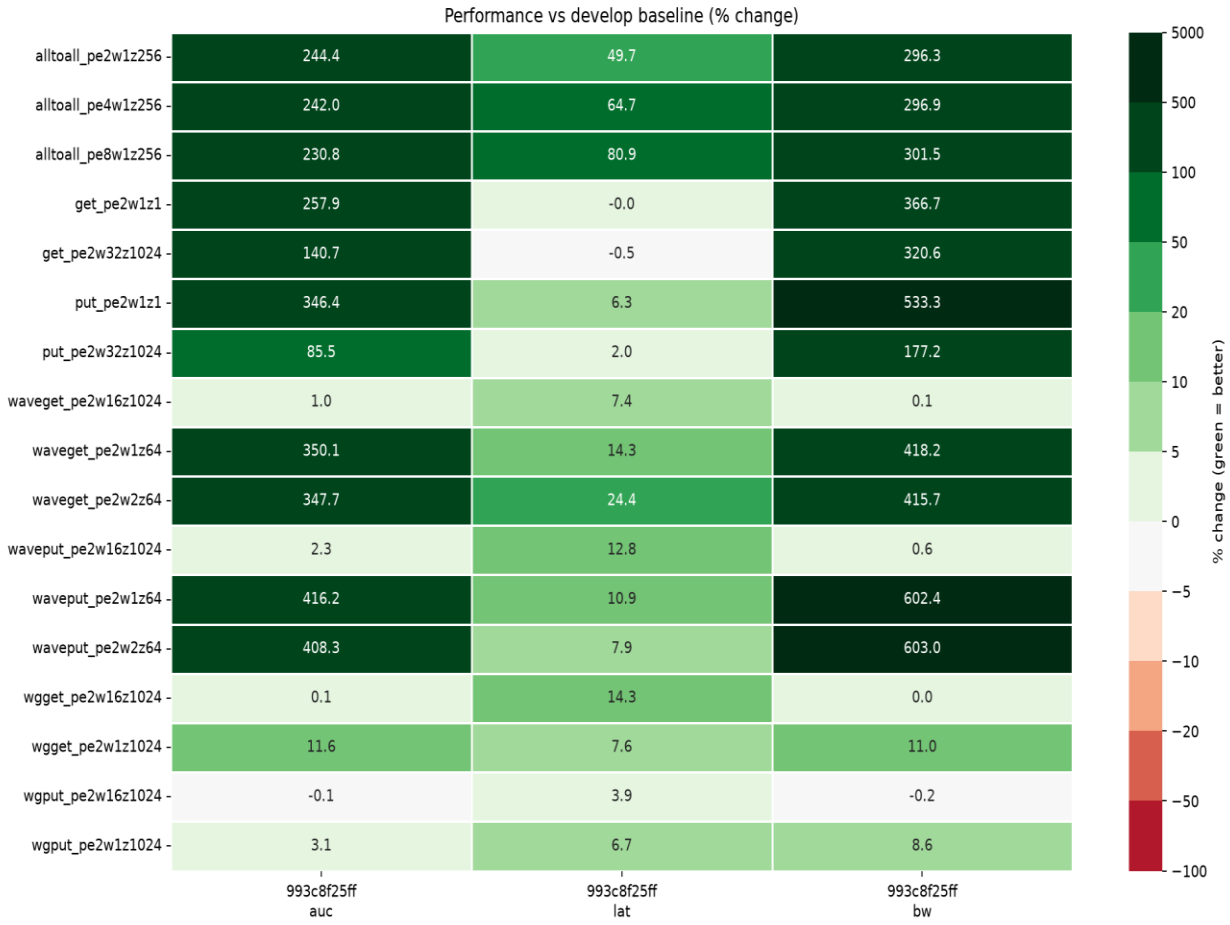
void rocshmem_ctx_int_alltoall_wg(rocshmem_ctx_t ctx, rocshmem_team_t team,
                                  int *dest, const int *source, int nelems)
```

Supported transports

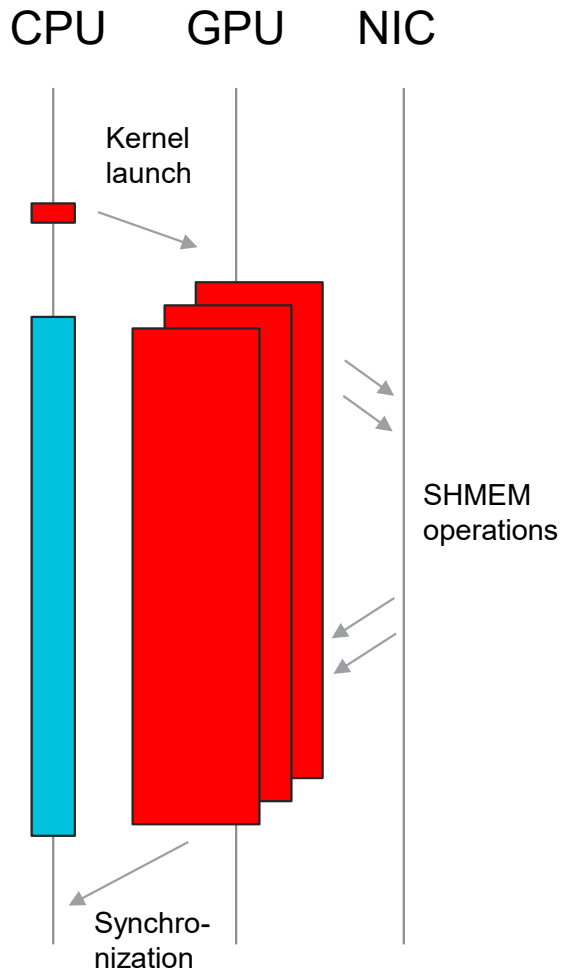


Why defer IPC (scale-up) to rocSHMEM?

- Multiple IPC optimization deployed
  - Use 128b loads and stores, flat load-store when appropriate
  - Explicit **control of cache policy** during load-stores
    - Temporal, non-temporal, workgroup,agent,system-scope control
  - Smart unroll prepares a VGPR buffer with loads before issuing **coalesced bulk stores** for the whole buffer
  - Unroll by 16, with **buffered instructions**
    - Relaxed bound checking
    - Compiler issues load-store pipeline optimal code
  - **Explicit waitcnt** when using cache bypassing assembly (correctness)
- Use **intimate knowledge of architecture and assembly** to optimize
- Vastly outperforms a simple “parallel memcpy” code



GPU Direct Async (GDA) backend (scale-out)



- Refers to the technology to initiate network (RDMA) communication from GPU kernels
- **Host is not required** for control during device initiated communication
  - *Host is still required for setup and teardown of NIC resources*

- GDA designed to improve Message Rate in **fused-kernel** scenarios
  - **Fine-grain communication/computation overlap**
  - **No synchronization overhead with host**
  - 1 thread Latency and bandwidth competitive with host-proxy approaches
  - Message Rate and full-line bandwidth improves with multi-WG communication

  - **GDA** communication completely **hidden inside rocSHMEM** routines (no spillover in application code, as has been done in prior work on MoE)

GPU-Direct Async (GDA) Implementation Challenges

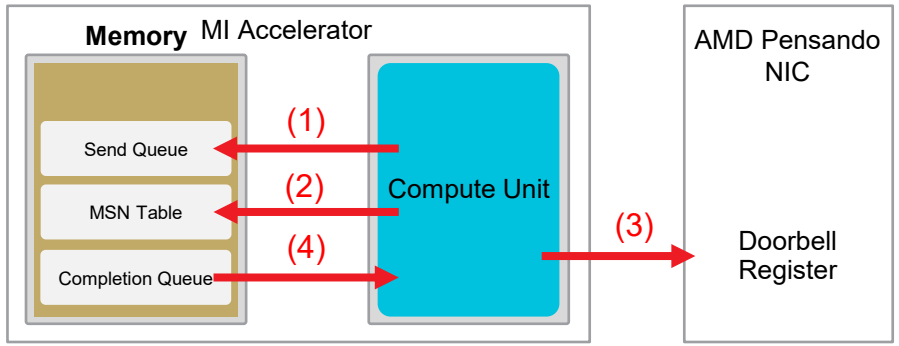
- InfiniBand Verbs is a standardized low-level API for programming RDMA interfaces (from host)
- Direct Verbs is vendor specific APIs which allow non-standard functionality
  - E.g. allows for allocation of Queues in GPU Memory

- Preparation of WQEs and reading CQ on device is not standardized, requires ad-hoc code
  - Prepares structures in NIC specific layout
  - Writes to NIC registers in a vendor-specific way

- rocSHMEM isolate end-user from the complexity

InfiniBand Verbs	BNXT Direct Verbs
ibv_create_qp(…); /* Move QP to Init */ ibv_modify_qp(…);	bnxt_re_dv_qp_mem_alloc(…); hipMalloc(…); bnxt_re_dv_umem_reg(…); bnxt_re_dv_create_qp(…); bnxt_re_dv_get_default_db_region(…); /* Move QP to Init */ bnxt_re_dv_modify_qp(…);

Example workflow of preparing a network put from a GPU to an RDMA NIC



1. Write WQE to SQ
2. Write MSN Table entry
3. Write to Doorbell Register
4. Poll CQ until valid completion

Obtaining rocSHMEM

Documentation
https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/install.html

Obtaining rocSHMEM

Documentation
https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/install.html

OS packages - ***sudo apt install amdrocm-rocshmem-dev***

Python

- Rocshmem4py Python API
- Available as a wheel
- env ROCM_PATH="$rocm_dir" ROCSHMEM_HOME="$installdir" pip install -e "$ROCSHMEM_SRC/python"

Build from sources

Docker

Obtaining rocSHMEM

Documentation https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/install.html

OS packages - ***sudo apt install amdrocm-rocshmem-dev***

Python – pip wheel rocshmem4py

Build from sources

- https://github.com/ROCm/rocm-systems/tree/develop/projects/rocshmem
- CMake based package, build as you would build a normal CMake package
- Multiple build recipes in scripts/build_configs (human readable, essentially just presets for CMake)
- `` `mkdir build && cd build && ../scripts/build_configs/all_backends` `` builds all backends and all NIC providers (all available for runtime selection)
- gda_ionic, gda_bnxt, gda_mlx5: build only for the specific NIC provider
  - gda_ionic –DUSE_IPC=ON to retain ability to do mixed scale-up IPC and GDA scale-out
- ipc_single: build for scale-up only support (no NIC providers)
  - ipc_single –DUSE_SDMA=ON to enable SDMA support

Docker

Obtaining rocSHMEM

Documentation https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/install.html

OS packages - ***sudo apt install amdrocm-rocshmem-dev***

Python

Build from sources

Docker

- Documentation: https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/docker.html
- Dockerfile available in docker/ directory (in the sources)
- cd docker && docker build --tag $USER/rocshmem -f Dockerfile.ubuntu .
- docker run -it --rm --shm-size 64G --network host --device /dev/dri --device /dev/kfd --device /dev/infiniband --ipc host --group-add video --cap-add SYS_PTRACE --security-opt seccomp=unconfined --privileged $USER/rocshmem tools/rocshmem_info
- For GDA the docker image will need compatible NIC drivers installed in the image

The rocshmem_info tool

- Used to perform introspection on how rocSHMEM was compiled (what features and archs it support)
- Used to mimic runtime hardware probe (Avail NICs, NIC-GPU topological mapping)

```
################################################################################
#                                rocSHMEM Info                                 #
################################################################################
# Version                    : 3.6.0                                           #
# Vendor String              : rocSHMEM_v3.6.0_GDA(IONIC/BNXT/MLX5)+RO+IPC     #
# Git Hash                   : 6337a565ec7381c41715d271b6b22d0e28188b7f        #
# Build Type                 : Release                                         #
# Install Prefix             : /home/abouteil/rocshmem                         #
# Compiled Arch(s)           : gfx950                                          #
# System Arch                : gfx950:sramecc+:xnack-                          #
# System Arch is supported   : Yes                                             #
# ROCm                       : 7.2.1                                           #
# External MPI               : Open MPI 5.1.0                                  #
#-------------------------------------------------------------------------------#
#                            Build Configuration                                #
#-------------------------------------------------------------------------------#
# PROFILE                    : OFF                                             #
# BUILD_DEBUG_TRACE_HOST     : ON                                              #
# BUILD_DEBUG_TRACE_DEVICE   : OFF                                             #
# BUILD_DEBUG_DEVICE         : ON                                              #
# USE_RO                     : ON                                              #
# USE_IPC                    : ON                                              #
# USE_GDA                    : ON                                              #
# USE_SDMA                   : ON                                              #
# USE_THREADS                : OFF                                             #
# USE_SHARED_CTX             : OFF                                             #
# USE_WF_COAL                : OFF                                             #
# USE_HEAP_DEVICE_FINEGRAIN  : ON                                              #
# USE_HEAP_DEVICE_UNCACHED   : OFF                                             #
# USE_HEAP_DEVICE_COARSEGRAIN : OFF                                            #
# USE_HEAP_DEVICE_VMM_POSIX  : OFF                                             #
# USE_HEAP_DEVICE_VMM_FABRIC : OFF                                             #
# HAVE_AMDSMI_GPU_FABRIC_INFO : OFF                                            #
# USE_FUNC_CALL              : OFF                                             #
# USE_SINGLE_NODE            : OFF                                             #
# USE_HDP_FLUSH              : OFF                                             #
# USE_HDP_FLUSH_HOST_SIDE    : OFF                                             #
# GDA_IONIC                  : ON                                              #
# GDA_BNXT                   : ON                                              #
# GDA_MLX5                   : ON                                              #
# HAVE_EXTERNAL_MPI          : ON                                              #
# HAVE_DEVICE_MALLOC_UNCACHED : ON                                             #
################################################################################
```

The rocshmem_info tool

- Used to perform introspection on how rocSHMEM was compiled (what features and archs it support)
- Used to mimic runtime hardware probe
  - Accepted environment variables
  - Avail NICs
  - NIC-GPU preferred mapping
  - NUMA topology

```
################################################################################
#                    rocSHMEM Environment Variables                            #
################################################################################
################################################################################

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

Detected Topology:
==================
  2 configured CPU NUMA node(s) [2 total]
  8 GPU device(s)
  16 Supported NIC device(s)
 NIC | Device Name | Active | PCIe Bus ID   | NUMA | Closest GPU(s) | GID Index | GID Descriptor
-----+-------------+--------+---------------+------+----------------+-----------+-------------------
 0   | rocep6s0f0  | Yes    | 0000:06:00.0 | 0    | 0              | 3         | RoCEv2 IPv4-mapped IPv6
 1   | rocep6s0f1  | Yes    | 0000:06:00.1 | 0    |                | 3         | RoCEv2 IPv4-mapped IPv6
 2   | rocep22s0f0 | Yes    | 0000:16:00.0 | 0    | 1              | 3         | RoCEv2 IPv4-mapped IPv6
 3   | rocep22s0f1 | Yes    | 0000:16:00.1 | 0    |                | 3         | RoCEv2 IPv4-mapped IPv6
 4   | rocep102s0f0 | Yes    | 0000:66:00.0 | 0    | 2              | 3         | RoCEv2 IPv4-mapped IPv6
 5   | rocep102s0f1 | Yes    | 0000:66:00.1 | 0    |                | 3         | RoCEv2 IPv4-mapped IPv6
 6   | rocep118s0f0 | Yes    | 0000:76:00.0 | 0    | 3              | 3         | RoCEv2 IPv4-mapped IPv6
 7   | rocep118s0f1 | Yes    | 0000:76:00.1 | 0    |                | 3         | RoCEv2 IPv4-mapped IPv6
 8   | rocep134s0f0 | Yes    | 0000:86:00.0 | 1    | 4              | 3         | RoCEv2 IPv4-mapped IPv6
 9   | rocep134s0f1 | Yes    | 0000:86:00.1 | 1    |                | 3         | RoCEv2 IPv4-mapped IPv6
 10  | rocep150s0f0 | Yes    | 0000:96:00.0 | 1    | 5              | 3         | RoCEv2 IPv4-mapped IPv6
 11  | rocep150s0f1 | Yes    | 0000:96:00.1 | 1    |                | 3         | RoCEv2 IPv4-mapped IPv6
 12  | rocep230s0f0 | Yes    | 0000:e6:00.0 | 1    | 6              | 3         | RoCEv2 IPv4-mapped IPv6
 13  | rocep230s0f1 | Yes    | 0000:e6:00.1 | 1    |                | 3         | RoCEv2 IPv4-mapped IPv6
 14  | rocep246s0f0 | Yes    | 0000:f6:00.0 | 1    | 7              | 3         | RoCEv2 IPv4-mapped IPv6
 15  | rocep246s0f1 | Yes    | 0000:f6:00.1 | 1    |                | 3         | RoCEv2 IPv4-mapped IPv6


            |NUMA 00|NUMA 01| #Cpus | Closest GPU(s)
------------+-------+-------+-------+---------------
NUMA 00 (00)|    10 |    32 |   128 | 0 1 2 3
NUMA 01 (01)|    32 |    10 |   128 | 4 5 6 7

################################################################################
```

Lets create a rocSHMEM program!

Minimal rocSHMEM program

```
38 #include <rocshmem/rocshmem.hpp>
39
40 #include "util.h"
41
42 using namespace rocshmem;
43
44 int main (int argc, char **argv)
45 {
46     CHECK_HIP(hipSetDevice(get_launcher_local_rank()));
47
48     rocshmem_init();
49
50
51     rocshmem_finalize();
52     return 0;
53 }
```

What device will we use in this PE

Init after selecting the device, finalize before leaving

Minimal rocSHMEM program

What device will we use in this PE

```
38 #include <rocshmem/rocshmem.hpp>
39
40 #include "util.h"
41
42 using namespace rocshmem;
43
44 int main (int argc, char **argv)
45 {
46     CHECK_HIP(hipSetDevice(get_launcher_local_rank()));
47
48     rocshmem_init();
49
50
51     rocshmem_finalize();
52     return 0;
53 }
```

```
33 #ifndef CHECK_HIP
34 #define CHECK_HIP(condition) {
35     hipError_t error = condition;
36     if(error != hipSuccess){
37         fprintf(stderr, "HIP error: %d line: %s:%d\n",
38                 error, __FILE__,  __LINE__);
39         exit(error);
40     }
41 }
42 #endif
```

Init after selecting the device, finalize before leaving

Minimal rocSHMEM program

```
38 #include <rocshmem/rocshmem.hpp>
39
40 #include "util.h"
41
42 using namespace rocshmem;
43
44 int main (int argc, char **argv)
45 {
46     CHECK_HIP(hipSetDevice(get_launcher_local_rank()));
47
48     rocshmem_init();
49
50
51     rocshmem_finalize();
52     return 0;
53 }
```

```
58 [[maybe_unused]] static int get_launcher_local_rank() {
59     // Check launcher-specific env vars in priority order
60     for (const char* var : {"OMPI_COMM_WORLD_LOCAL_RANK",  // Open MPI / prterun
61                             "SLURM_LOCALID",               // SLURM
62                             "FLUX_TASK_LOCAL_ID",          // Flux
63                             "PMIX_LOCAL_RANK",             // PMIx / MPICH
64                             "PBS_O_TASKNUM",               // PBS/Torque
65                             "LOCAL_RANK"}) {               // torchrun / generic
66         const char* v = getenv(var);
67         if (v) return atoi(v);
68     }
69     return -1;
70 }
```

What device will we use in this PE

Init after selecting the device, finalize before leaving

Building and running a rocSHMEM program

Building your rocSHMEM program

Direct build

```
hipcc -c -fgpu-rdc -x hip rocshmem_ex1.cc \
  --offload-arch=<target>:<xnack>                \
  -I/opt/rocm/include -I$ROCSHMEM_INSTALL_DIR/include
```

Direct Link

```
hipcc -fgpu-rdc --hip-link rocshmem_ex1.o -o rocshmem_ex1 \
  --offload-arch=<target>:<xnack>                          \
  $ROCSHMEM_INSTALL_DIR/lib/librocshmem.a                  \
  $OPENMPI_UCX_INSTALL_DIR/lib/libmpi.so                   \
  -L/opt/rocm/lib -lamdhip64 -lhsa-runtime64
```

Use the architecture as reported by
`rocshmem_info | grep gfx`

```
# Compiled Arch(s) : gfx90a:xnack-,gfx942,gfx950
# System Arch      : gfx950:sramecc+:xnack-
```

System arch (present on running system) indicates gfx950 class (e.g., MI355X)

Matching compiled arch is gfx950, we should use **--offload-arch=gfx950** in our client code.

Building your rocSHMEM program: CMake

```
24 cmake_minimum_required(VERSION 3.16.3 FATAL_ERROR)
25
26 project(rocshmem_examples VERSION 1.0.0 LANGUAGES CXX)
27
28 find_package(MPI)
29 find_package(hip REQUIRED)
30 if (NOT TARGET roc::rocshmem)
31   find_package(rocshmem REQUIRED)
32 endif()
33
34 set(EXAMPLE_SOURCES
35   rocshmem_ex1.cc
36 )
37
38 foreach(SOURCE_FILE IN LISTS EXAMPLE_SOURCES)
39   get_filename_component(EXECUTABLE_NAME ${SOURCE_FILE} NAME_WE)
40
41   add_executable(${EXECUTABLE_NAME} ${SOURCE_FILE})
42
43   target_link_libraries(
44     ${EXECUTABLE_NAME}
45     PRIVATE
46       roc::rocshmem
47       $<TARGET_NAME_IF_EXISTS:MPI::MPI_CXX>
48   )
49 endforeach()
```

rocSHMEM available with find_package

- Creates target **roc::rocshmem**
- Required dependencies will be added to the target automatically
- MPI can be used by rocSHMEM but is not a hard dependency
  - If MPI not found, features dependent on MPI will be disabled
  - Library **functional without MPI, both for on-node and off-node** operation
  - Hence your program may use MPI as a convenient launch mechanism, or not use MPI at all, your choice.

Running a rocSHMEM enabled program

SLR – Simple Local Runtime

- Used to deploy IPC based applications
- Fork-exec based launcher, 1 PE spawned per local GPU

MPI based launcher

- Used to deploy scale-out applications
- With GDA, MPI used only as a “Launcher”
- With GDA, MPI not required to be initialized

Slurm

- Used to deploy scale-out applications
- With GDA, MPI not required to be initialized

Launching a rocSHMEM functional test on 16 workgroup, 256 thread/wg, 2 PEs

- ```
  ROCSHMEM_SLR_NP=2
  ROCSHMEM_MAX_NUM_CONTEXTS=16
  ROCSHMEM_HEAP_SIZE=6442450944
  tests/functional_tests/rocshmem_functional_tests -a
  waveputnbi -w 16 -z 256
  ```

- ```
  mpiexec -n 2
  -x ROCSHMEM_MAX_NUM_CONTEXTS=16
  -x ROCSHMEM_HEAP_SIZE=6442450944
  tests/functional_tests/rocshmem_functional_tests -a
  waveputnbi -w 16 -z 256
  ```

- ```
  srun -n 2
  -e ROCSHMEM_MAX_NUM_CONTEXTS=16
  -e ROCSHMEM_HEAP_SIZE=6442450944
  tests/functional_tests/rocshmem_functional_tests -a
  waveputnbi -w 16 -z 256
  ```
  ```

Important environment variables

- **Full list of environment available:** https://rocm.docs.amd.com/projects/rocSHMEM/en/latest/env_variables.html
- **ROCSHMEM_DEBUG_LEVEL**=error,warn,info,trace: Make rocshmem verbose (some trace level require build options)
- **ROCSHMEM_HEAP_SIZE**: size of the symmetric heap
- **ROCSHMEM_MAX_NUM_CONTEXTS**: maximum number of contexts (also controls indirectly the number of QPs per PE)
- **ROCSHMEM_DISABLE_MIXED_IPC**: prevent GDA/RO to IPC mixing (always use scale-out network, even on-node)
- **ROCSHMEM_HBA_LIST**: list of RDMA interfaces to use (can also be used to exclude ifaces that are not active).
- **ROCSHMEM_SDMA_ENABLED**: if compiled-in, enable SDMA copy-engine IPC
- **ROCSHMEM_BOOTSTRAP_SOCKET_IFNAME**: interface to use to setup rocSHMEM's internal bootstrap

GPU initiated communication in the rocSHMEM program

rocSHMEM program: what happens on host, what happens on device

On host

- **Initialization**
- **Allocate symmetric memory**
- **Team and context management** (optional)
- Host-initiated communication (optional)
- On-stream communication (optional)

On device

- Acquire a context (optional)
- **Device-initiated communication**
- Return the context (if one acquired)

More host-based initialization and setup

```
54    CHECK_HIP(hipSetDevice(get_launcher_local_rank()));
55
56    rocshmem_init();
57
58    [[maybe_unused]] int my_pe = rocshmem_my_pe();
59    [[maybe_unused]] int npes =  rocshmem_n_pes();
60
61    int *src = (int *)rocshmem_malloc(nelem * sizeof(int));
62    int *dst = (int *)rocshmem_malloc(nelem * sizeof(int));
63    if (NULL == src || NULL == dst) {
64        std::cout << "Error allocating memory from symmetric heap" << std::endl;
65        std::cout << "source: " << src << ", dest: " << dst << ", size: "
66            << sizeof(int) * nelem << std::endl;
67        rocshmem_global_exit(1);
68    }
69
70    for (int i=0; i<nelem; i++) {
71        src[i] = 0;
72        dst[i] = 1;
73    }
74    CHECK_HIP(hipDeviceSynchronize());
75
76    //....
77
78    rocshmem_free(src);
79    rocshmem_free(dst);
80    rocshmem_finalize();
```

Identify PE rank and number of PEs
(also callable from device)

Allocate buffer from the symmetric memory

Error handling

Initialize the symmetric memory buffers
(assumes fine-grain symmetric heap, use hipMemcpy if not)

Return buffer to symmetric memory pool

Allocating symmetric memory

Implicit Symmetric heap allocated on startup

Control size with env **ROCSHMEM_HEAP_SIZE**

Managing user-usable memory

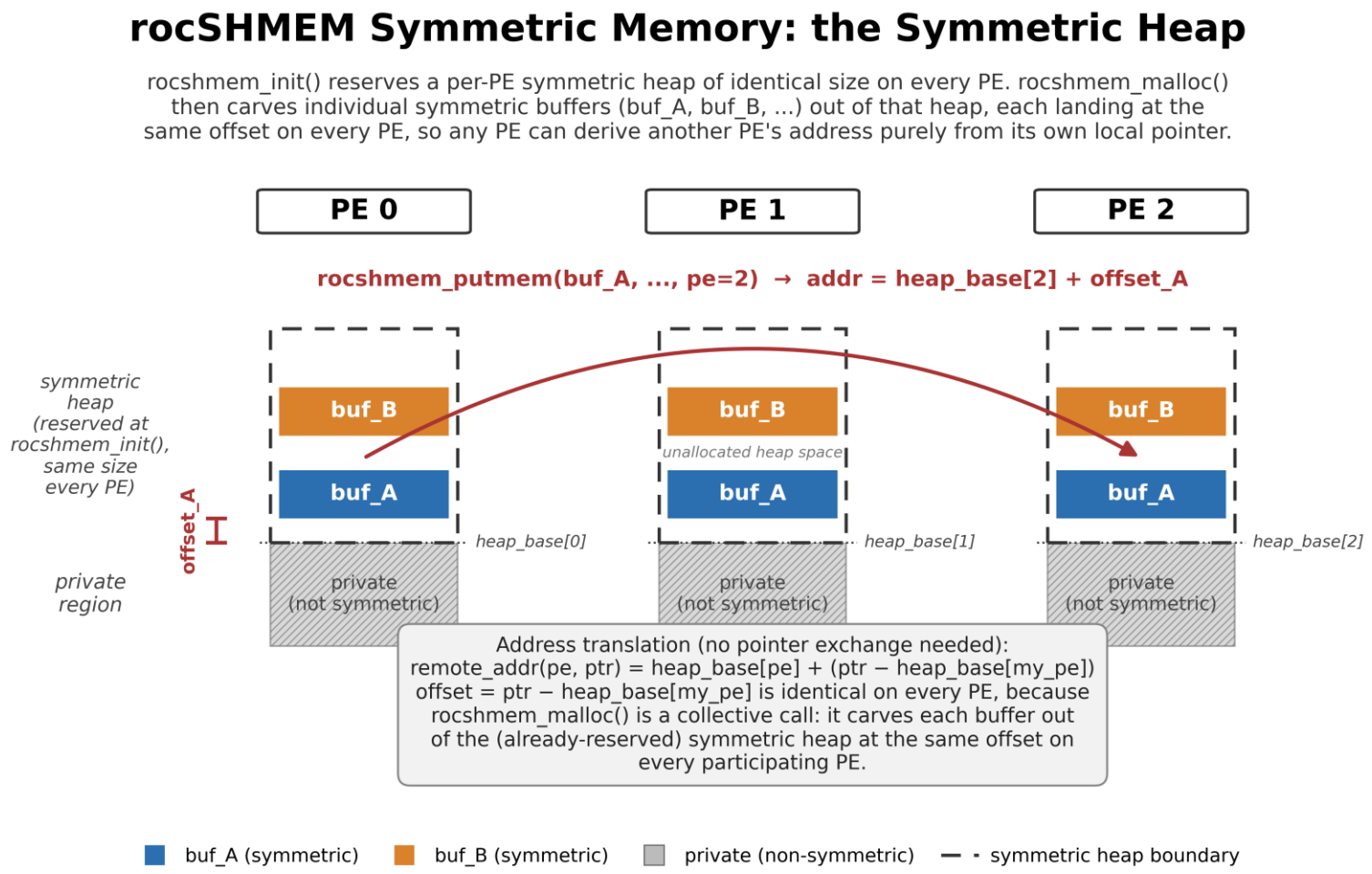
- rocshmem_malloc, rocshmem_calloc
- rocshmem_align
- rocshmem_free

Attaching user-allocated buffers to the symmetric heap

- rocshmem_buffer_register/unregister *(non collective)*
- rocshmem_buffer_register_symmetric/unregister_symmetric

Memory **allocation and management is collective and host-only**

No defragmentation, recommendation: do long-standing allocations, place individual structures and objects at fixed positions yourself



Kernel initiated communication

```
44 __global__ void simple_getmem_test(int *src, int *dst, size_t nelems)
45 {
46   if (blockIdx.x == 0){
47     if (threadId == 0) {
48       int my_pe = rocshmem_my_pe();
49       int peer =  my_pe ? 0 : 1;
50       rocshmem_getmem(dst, src, nelems * sizeof(int), peer);
51     }
52
53     __syncthreads();
54     for (int i = threadId; i < nelems; i += blockDim.x) {
55       hadamard[i] += src[i] * dst[i];
56     }
57   }
58 }
```

rocshmem_my_pe callable from device code

Simple blocking get, when the procedure returns, local dst is a copy of peer's src buffer

Only threadId 0 guaranteed to have visibility of updates to dst!

Syncthreads required before other workgroup/threads access dst

Launching the communicating kernel

```
86      for (int i=0; i<nelem; i++) {
87          src[i] = 0;
88          dst[i] = 1;
89      }
90      CHECK_HIP(hipDeviceSynchronize());
91      rocshmem_sync_all();
92
93      int threadsPerBlock=256;
94      simple_getmem_test<<<dim3(1), dim3(threadsPerBlock), 0, 0>>>(src, dst, nelem);
95      CHECK_HIP(hipDeviceSynchronize());
96      rocshmem_sync_all();
97
98      bool pass = true;
99      for (int i=0; i<nelem; i++) {
100         if (dst[i] != 0) {
101             pass = false;
102 #if VERBOSE
103             printf("[%d] Error in element %d expected 0 got %d\n", my_pe, i, dst[i]);
104 #endif
105         }
106     }
107     printf("Test %s \t %s\n", argv[0], pass ? "[PASS]" : "[FAIL]");
```

Make sure src visible on device before launching the kernels

Communicating kernel launched like any other kernel

Kernel boundary ensures that dst is ready for host consumption

Using host initiated rocshmem to synchronize all PEs: prevents src from being modified while getmem still reading it

HIP memory model and synchronization still apply!

Host-device updates for communicated to/from buffers must be user enforced, as per usual

get, get_wave, get_wg

```
void shmem_TYPENAME_get(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_ctx_TYPENAME_get(shmem_ctx_t ctx, TYPE *dest, const TYPE *source, size_t nelems,
    int pe);
```

where *TYPE* is one of the standard RMA types and has a corresponding *TYPENAME* specified by Table 5.

```
void shmem_getSIZE(void *dest, const void *source, size_t  nelems, int pe);
void shmem_ctx_getSIZE(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems, int
    pe);
```

where *SIZE* is one of *8, 16, 32, 64, 128*.

```
void shmem_getmem(void *dest, const void *source, size_t nelems, int pe);
void shmem_ctx_getmem(shmem_ctx_t ctx, void *dest, const void *source, size_t nelems, int
    pe);
```

- rocshmem_get (and variants) copy nelems TYPE contiguous elements from symmetric heap at pe, into local **symmetric heap** locally
  - The requirement for symmetric heap for the local buffer is specific to GPU code (not required in CPU OpenSHMEM)
- rocshmem also support templated form **rocshmem_get(<TYPE> *dest…)** converts automatically to **rocshmem_TYPENAME_get**
- get_wave, get_wg
  - Same semantic, but the call is collaborative on the wave/wg
  - **Visibility** is guaranteed on the **calling scope** when the **procedure return**

TYPE	*TYPENAME*
float	float
double	double
long double	longdouble
char	char
signed char	schar
short	short
int	int
long	long
long long	longlong
unsigned char	uchar
unsigned short	ushort
unsigned int	uint
unsigned long	ulong
unsigned long long	ulonglong
int8_t	int8
int16_t	int16
int32_t	int32
int64_t	int64
uint8_t	uint8
uint16_t	uint16
uint32_t	uint32
uint64_t	uint64
size_t	size
ptrdiff_t	ptrdiff

Table 5: Standard RMA Types and Names

Wave and WG primitives, motivation and example

```
44 __global__ void simple_getmem_test(int *src, int *dst, size_t nelems)
45 {
46   if (blockIdx.x == 0){
47     if (threadId == 0) {
48       int my_pe = rocshmem_my_pe();
49       int peer =  my_pe ? 0 : 1;
50       rocshmem_getmem(dst, src, nelems * sizeof(int), peer);
51     }
52
53     __syncthreads();
54     for (int i = threadId; i < nelems; i += blockDim.x) {
55       hadamard[i] += src[i] * dst[i];
56     }
57   }
58 }
```

Only one thread active to advance the getmem (matters for IPC bandwidth)

Visibility not enforced on the scope of interest

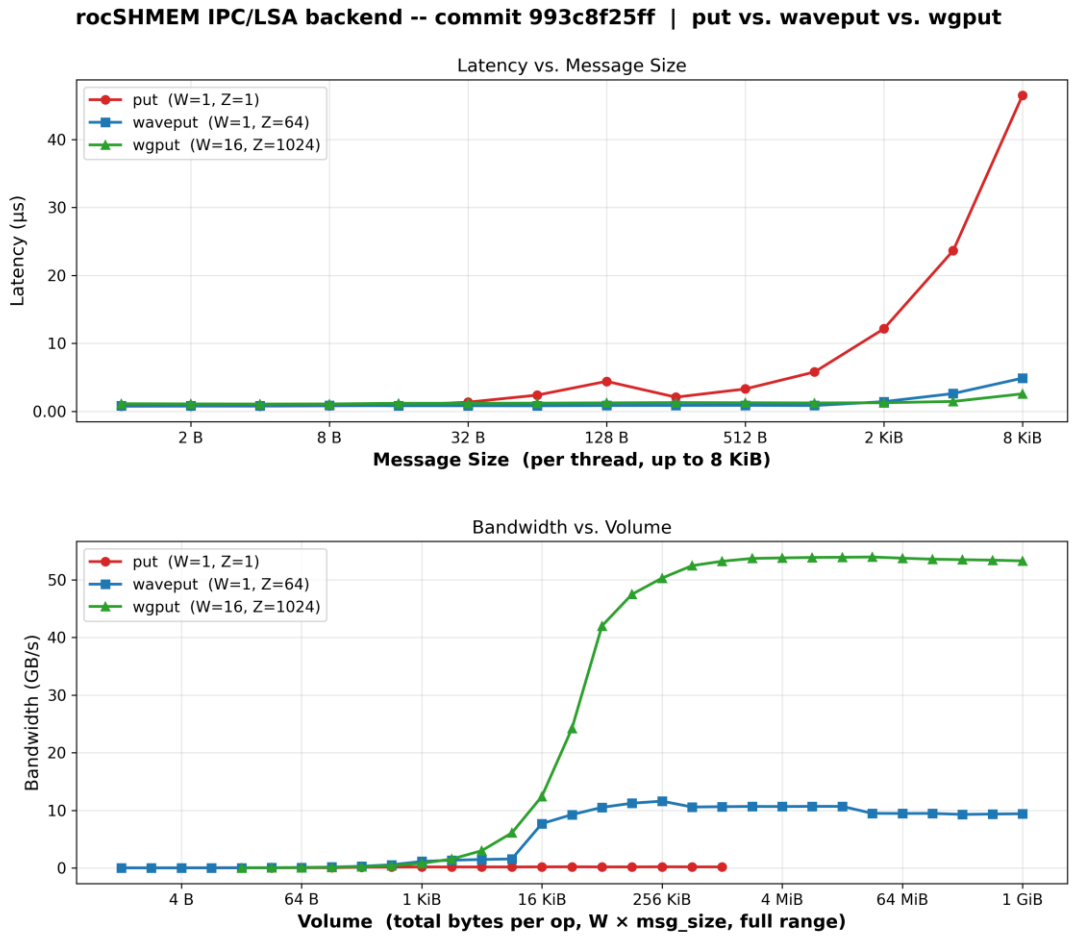
```
60 __global__ void simple_getmemwg_test(int *src, int *dst, int *hadamard, size_t
61   nelem)
62 {
63   if (blockIdx.x == 0) {
64     int my_pe = rocshmem_my_pe();
65     int peer  = my_pe ? 0 : 1;
66
67     rocshmem_getmem_wg(dst, src, nelem * sizeof(int), peer);
68
69     for (int i = threadIdx.x; i < nelem; i += blockDim.x)
70       hadamard[i] += src[i] * dst[i];
71   }
72 }
```

All the threads in the workgroup call into the getmem_wg

Visibility enforced: at the end of the getmem the whole workgroup can use the buffer

Motivation: IPC bandwidth with different active threads

- Comparing the performance of single-threaded rocshmem_put, vs rocshmem_put_wave, vs rocshmem_put_wg
- The wave and wg variants give the opportunity for the communication routine to use enough CUs to perform the communication
- Can reach saturation when enough threads participate



WG primitives, multiple workgroups example

```
74 /* launched with <<<num_blocks, block_size>>>                          */
75 /* caller ensures nelems_per_wg = total_nelems / num_blocks (no rem)  */
76 __global__ void multiwg_getmemwg_test(int *src, int *dst, int *hadamard,
77                                       size_t nelems_per_wg, int peer)
78 {
79   int wg_id = blockIdx.x * nelems_per_wg;
80
81   rocshmem_getmem_wg(&dst[wg_id], &src[wg_id], nelems_per_wg * sizeof(int), peer);
82
83   for (int i = threadIdx.x; i < nelems_per_wg; i += blockDim.x)
84     hadamard[wg_id + i] += src[wg_id + i] * dst[wg_id + i];
85 }
```

- We split the buffer into per-wg slices
- Each slice can communicate and compute independently
- Uses all compute units for both IPC communication and compute

Wave variant use case: wave specialized code

```
88  __global__ void double_buf_wave_get(int *buf[2], int *hadamard,
89                                      size_t nelems, int peer, int iters)
90  {
91    int wf_id = threadIdx.x / warpSize;
92    int lane  = threadIdx.x % warpSize;
93
94    for (int it = 0; it < iters; it++) {
95      int fetch   = it & 1;
96      int compute = 1 - fetch;
97
98      if (wf_id == 0 && it < iters - 1) {
99        rocshmem_getmem_wave(buf[fetch], buf[fetch], nelems * sizeof(int), peer);
100     }
101     else if(wf_id != 0 && it > 0) {
102       for (int i = lane + (wf_id - 1) * warpSize;
103                i < nelems;
104            i += (blockDim.x - warpSize))
105       hadamard[i] += buf[compute][i] * buf[compute][i];
106     }
107     __syncthreads();
108   }
109 }
```

Compute wavefront id based on wavefront size (varies by architecture, 32/64)

Wave specialization (the SIMD unit)
Wave 0 communicates, other waves compute

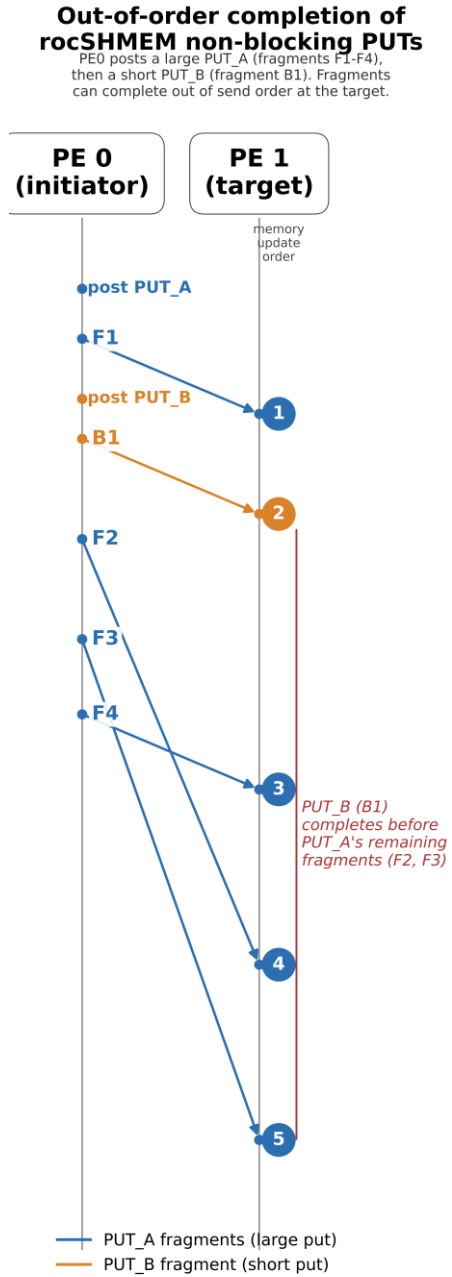
Wave level communication: optimize bandwidth on IPC, reduce latency on GDA

Make data get from wf0 visible to the whole workgroup
Synchronize buffer swapping

Put, put_wave, put_wg

```
215 __device__ ATTR_NO_INLINE void rocshmem_ctx_putmem(rocshmem_ctx_t ctx,
216                                                    void *dest,
217                                                    const void *source,
218                                                    size_t nelems, int pe);
219
220 __device__ ATTR_NO_INLINE void rocshmem_putmem(void *dest, const void *source,
221                                                size_t nelems, int pe);
```

- The routines return after the data has been **copied out of the source array on the local PE.**
- The **delivery of data words** into the data object on the destination PE may **occur in any order**.
- Furthermore, two **successive put** routines may deliver data **out of order**



Single element put/get

- Latency optimized variants
- **Support static/local variable local buffer**
  - **rocSHMEM cannot use static/local variables in regular put/get**
- Can use all of the basic types (integrals and floating point).

```
111 __global__ void rocshmem_p_put_example(int *sym_buf, int peer)
112 {
113   /* rocshmem_put: symmetric buffer as source — correct */
114   rocshmem_put(&sym_buf[0], &sym_buf[0], 1, peer);
115
116   /* rocshmem_put: local var as source — NOT on symmetric heap, incorrect */
117   // rocshmem_put(&sym_buf[1], &local_val, 1, peer);
118
119   /* rocshmem_p: source is scalar by value — local var is fine */
120   int local_val = 42;
121   rocshmem_p(&sym_buf[2], local_val, peer);
122
123 }
```

Non-blocking communication

Non-blocking communication marked with _nbi suffix
E.g. *rocshmem_put_nbi_wave*

No 'request', non-blocking operations are queued, and completed later (in bulk) with a **quiet call.**

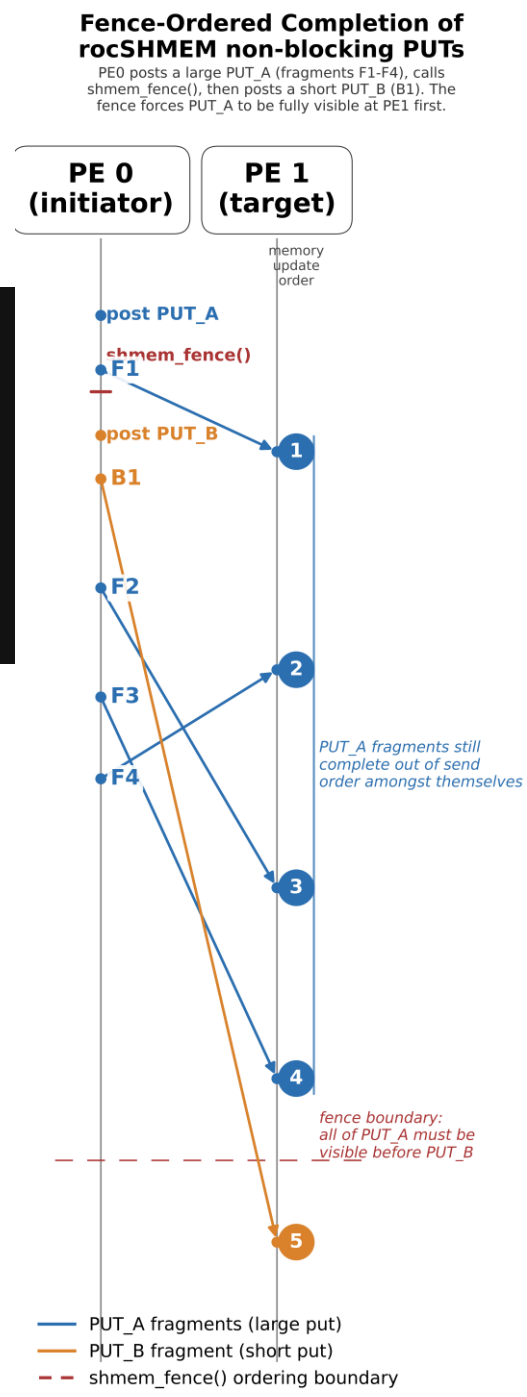
- **Non-blocking get_nbi vs get**
  - When the **Blocking** rocshmem_get procedure returns, the destination buffer is **ready to be consumed**
  - When the rocshmem_get_nbi procedure returns, the operation is in flight
  - **Non-blocking: destination** buffer **not ready to be consumed**
  - Call to quiet required to guarantee the destination buffer is ready to be consumed

- **Non-blocking put_nbi vs put**
  - When the **Blocking** rocshmem_put procedure returns, the destination buffer is ready to be reused**, the destination visibility is not enforced!**
  - When the rocshmem_put_nbi procedure returns, the operation is in flight
  - **Non-blocking: source** buffer **not ready to be reused**
  - Call to quiet required to guarantee the **destination buffer is ready** to be consumed **and local buffer ready to be reused**

Fence ordering

```
125 __global__ void ordered_puts(int *sym_buf, int peer)
126 {
127   if (rocshmem_my_pe() == 0) {
128     rocshmem_putmem(sym_buf, sym_buf, nelems * sizeof(int), peer);
129     rocshmem_fence();
130     rocshmem_p(&sym_buf[nelems], 1, peer);
131   }
132 }
```

- Fence enforces visibility ordering at the target
  - All of sym_buff[0..nelems-1] will be visible before sym_buf[nelems] is updated
- Says nothing about completion from the origin perspective
  - Not sure if non-blocking put completed locally (source reuse)
  - Not sure if preceding blocking put visible at target yet (target visibility)



Quiet ordering

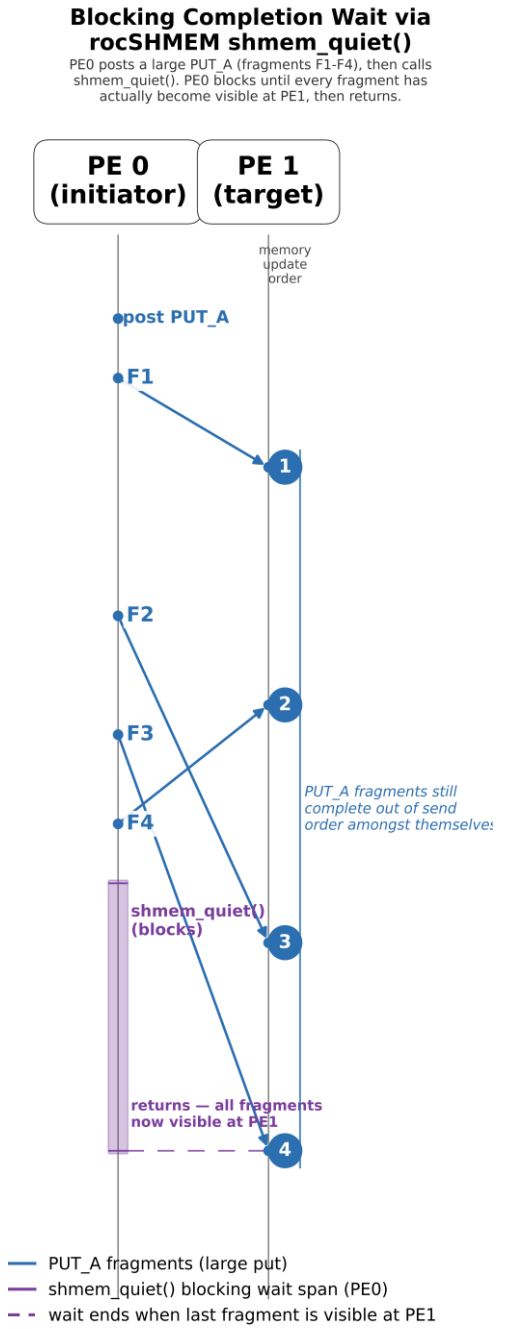
Quiet enforces both **completion at the origin** and **visibility at the target**

Use quiet to complete non-blocking operations

Use quiet to force a network flush

More expensive than fence!

Use with context to quiet only a specific QP/domain, reduce thread contention



Signals

- put_signal, put_signal_nbi
  - Combines put and signal
  - Signal visible AFTER put visible
  - Orders only THIS SPECIFIC put
  - Possible signal operations SIGNAL_ADD and SIGNAL_SET only
- signal_wait_until
  - At the target, enabled reading the signal in a safe way (appropriate cache-bypassing flags etc are set on the load)

```
char *my_data_s = data_s_buf + size * wg_id;
char *my_data_r = data_r_buf + size * wg_id;
uint64_t *my_sig = &sig_addr[wg_id];
uint64_t expected = static_cast<uint64_t>(i + 1);
if (pe == 0) {
  rocshmem_ctx_putmem_signal(ctx, my_data_r, my_data_s, size,
                             my_sig, 1, ROCSHMEM_SIGNAL_ADD, target);
  rocshmem_ulong_wait_until(
      reinterpret_cast<unsigned long *>(my_sig),
      ROCSHMEM_CMP_EQ, expected);
} else {
  rocshmem_ulong_wait_until(
      reinterpret_cast<unsigned long *>(my_sig),
      ROCSHMEM_CMP_EQ, expected);
  rocshmem_ctx_putmem_signal(ctx, my_data_r, my_data_s, size,
                             my_sig, 1, ROCSHMEM_SIGNAL_ADD, target);
}
```

- Example: pingpong with put_signal

Atomic operations

Atomic operations have **more supported types** and **operators** than signals

- **Integrals and floating-point** native types supported
- add, inc, fetch_add, fetch_inc
- and, or, xor, fetch_and, fetch_or, etc…
- fetch, set, swap, compare_swap

Does not fence prior operations (**relaxed by default**)

- Insert rocshmem_fence to order w.r.t. prior communication.
- Fetching atomics may cause ordering (semantically)

Reading from target must be done with either a fetching atomic, or wait_until/test_until

Collective operations

Rules for collective operations

Cooperative scope

- Workgroup or wavefront
- Few collectives (barrier, sync) have non-collaborative variants

Collective

- All Pes in the team must call
- Max 1 collective active per team, independent of scope!

No Implied Synchronization

- Different from MPI/RCCL!
- User responsibility to ensure that destination buffer is ready to receive before any PE calls locally

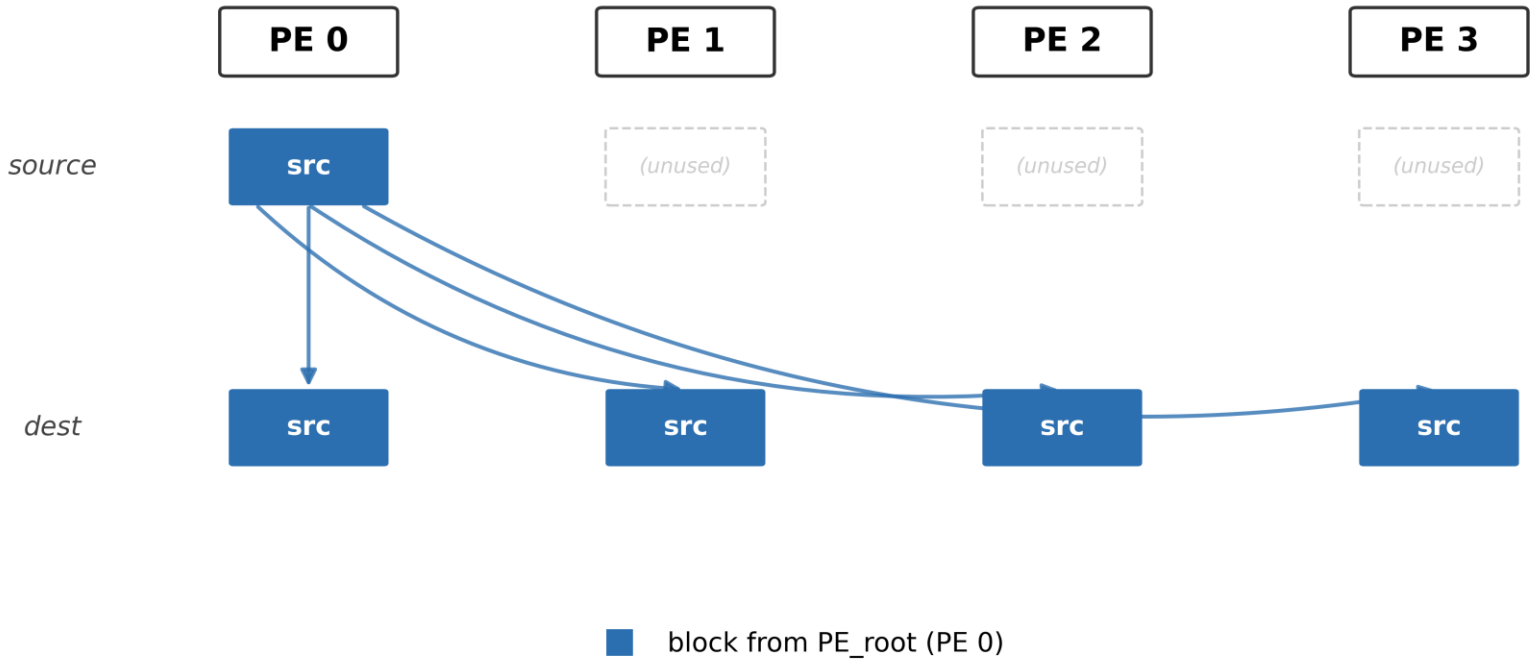
Broadcast

```
135 /* broadcast: PE 0 sends src to all PEs' dst */
136 __global__ void simple_broadcast(int *src, int *dst, int nelems)
137 {
138   if (blockIdx.x == 0)
139     rocshmem_ctx_int_broadcast_wg(ROCSHMEM_CTX_DEFAULT, ROCSHMEM_TEAM_WORLD,
140                                   dst, src, nelems, 0);
```

Collective on the TEAM: All team PE members must call.
No concurrent call at any PE on the same team!

shmem_broadcast: one source block dispatched to all PEs

PE_root (PE 0) supplies the source block; shmem_broadcast copies it into the dest array of every participating PE, including the root itself.



The content of src is unused on non-root PE Still need to be supplied (may be used for offset arithmetic)

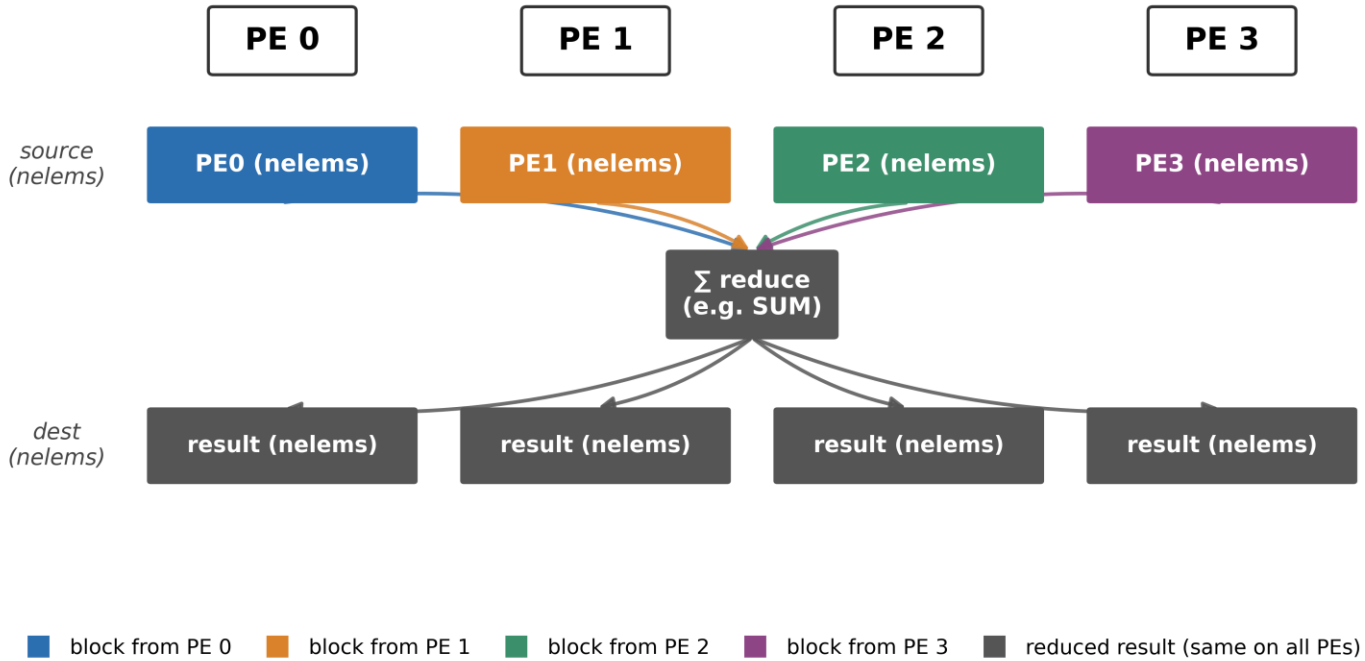
Reduce

```
159 /* reduce: allreduce sum, result on every PE */
160 __global__ void simple_reduce(int *src, int *dst, int nelems)
161 {
162   if (blockIdx.x == 0)
163     rocshmem_ctx_int_sum_reduce_wg(ROCSHMEM_CTX_DEFAULT, ROCSHMEM_TEAM_WORLD,
164                                    dst, src, nelems);
```

Collective on the TEAM: All team PE members must call.
No concurrent call at any PE on the same team!

shmem_*_reduce: all-reduce over one block per PE

Every PE contributes a full source block; all N blocks are combined element-wise (SUM/MAX/MIN/...). The single reduced result is delivered to every PE's dest array — not scattered or partitioned.



Available reduce operators

- rocshmem_ctx_TYPE_sum_reduce_wg/wave
- rocshmem_ctx_TYPE_min_reduce_wg/wave
- rocshmem_ctx_TYPE_max_reduce_wg/wave
- rocshmem_ctx_TYPE_prod_reduce_wg/wave
- rocshmem_ctx_TYPE_or_reduce_wg/wave
- rocshmem_ctx_TYPE_and_reduce_wg/wave
- rocshmem_ctx_TYPE_xor_reduce_wg/wave

Integral and floating-point types

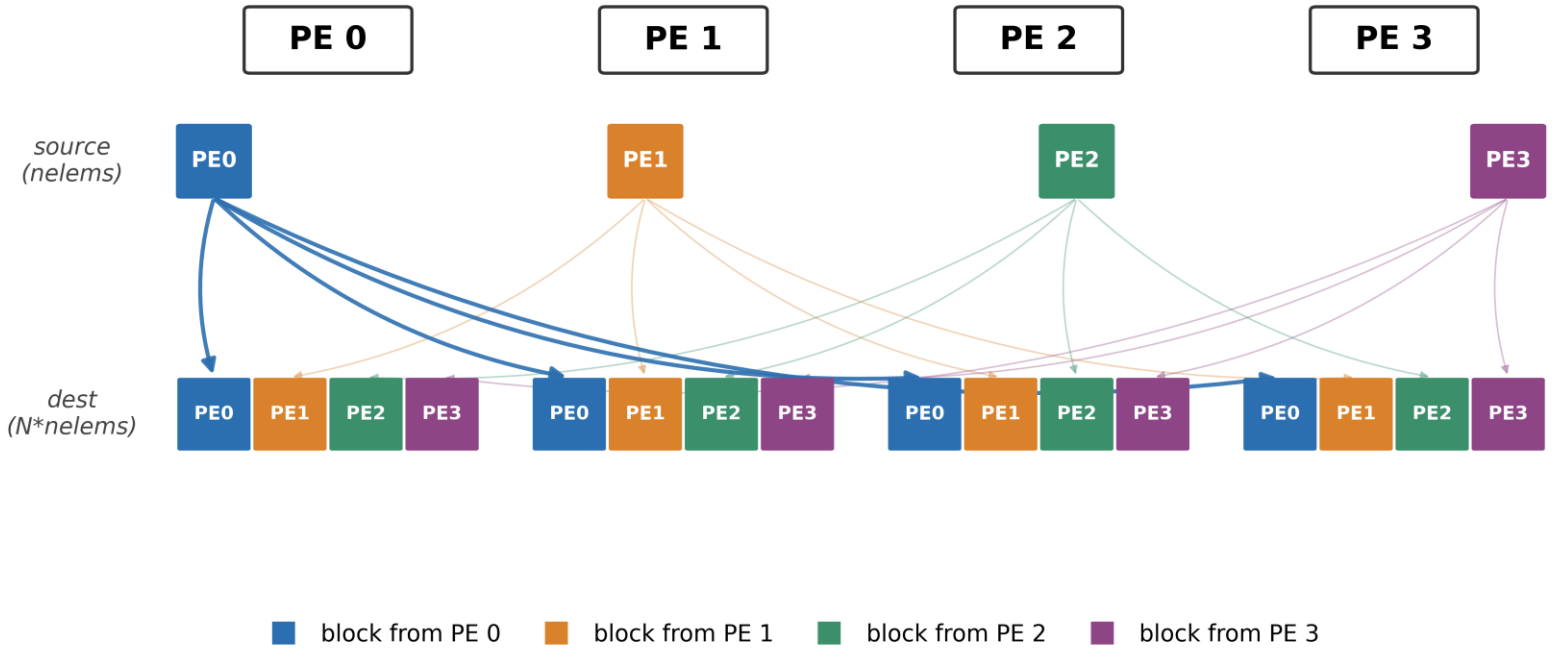
Fcollect

```
143 /* fcollect: each PE contributes nelems elements, dst has nelems * n_pes */
144 __global__ void simple_fcollect(int *src, int *dst, int nelems)
145 {
146   if (blockIdx.x == 0)
147     rocshmem_ctx_int_fcollect_wg(ROCSHMEM_CTX_DEFAULT, ROCSHMEM_TEAM_WORLD,
148                                  dst, src, nelems);
```

Collective on the TEAM: All team PE members must call.
No concurrent call at any PE on the same team!

shmem_fcollect: concatenate one block per PE into every dest array

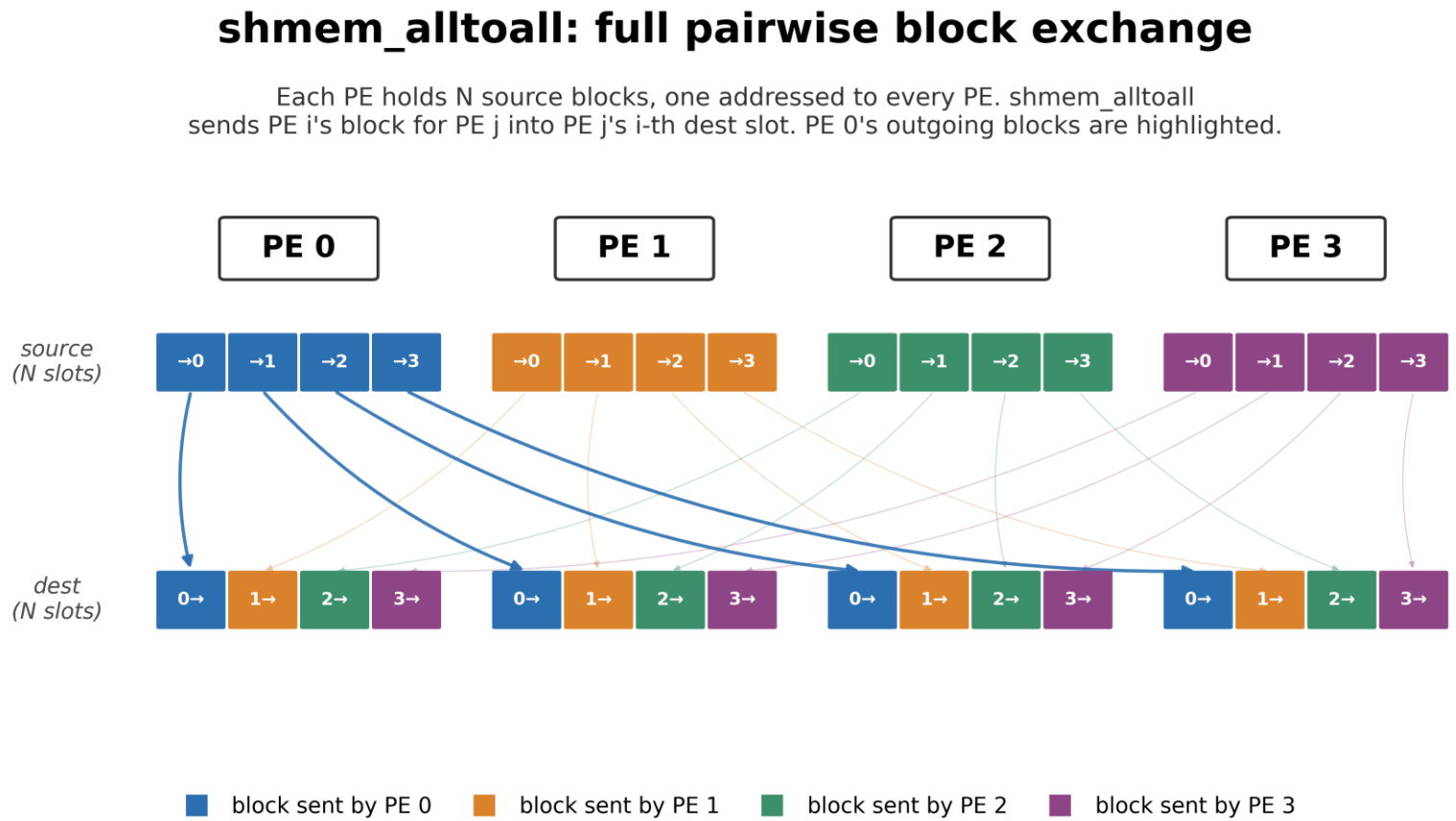
Each PE contributes one equal-size source block. shmem_fcollect concatenates them, ordered by PE number, into an identical dest array on every PE. PE 0's block is highlighted.



The fcollect communication pattern is often called “allgather” (e.g. in MPI)

Alltoall

```
151 /* alltoall: each PE sends nelems elements to every other PE */
152 __global__ void simple_alltoall(int *src, int *dst, int nelems)
153 {
154   if (blockIdx.x == 0)
155     rocshmem_ctx_int_alltoall_wg(ROCSHMEM_CTX_DEFAULT, ROCSHMEM_TEAM_WORLD,
156                                  dst, src, nelems);
```



Collective on the TEAM: All team PE members must call.
No concurrent call at any PE on the same team!

Teams and Contexts

Team operations

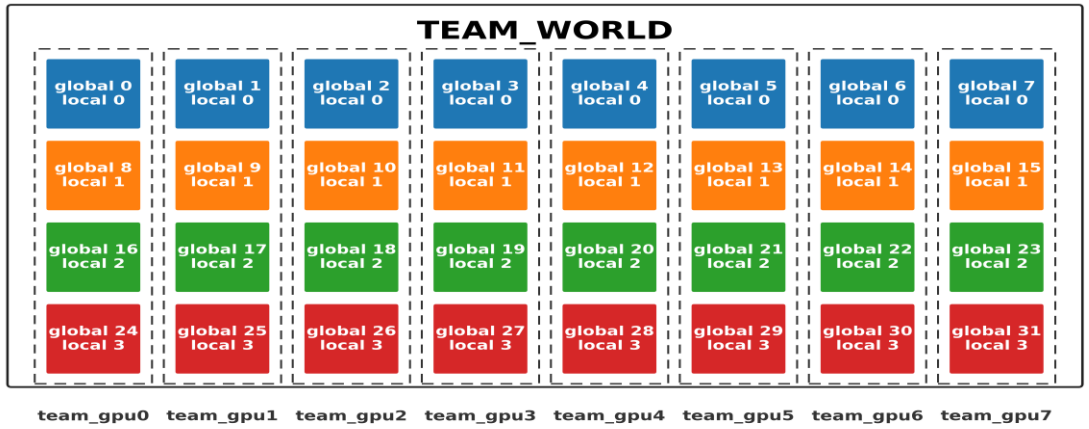
Host operations

- rocshmem_team_my_pe/n_pes
- rocshmem_team_translate_pe
- rocshmem_team_split_strided
- rocshmem_team_split_2d
- rocshmem_team_destroy

Device operations

- rocshmem_team_my_pe/n_pes
- rocshmem_team_translate_pe
- rocshmem_wg_team_create_ctx

```
169 // host side – called by all PEs collectively
170 __host__ void split_rails() {
171   int my_pe       = rocshmem_my_pe();
172   int n_pes       = rocshmem_n_pes();
173   int gpus_per_node = 8;
174
175   // HIP device number is my_pe % gpus_per_node
176   int hip_id      = my_pe % gpus_per_node;
177
178   // Group i contains PEs for HIP device i on each node.
179   rocshmem_team_t same_gpu_team;
180   rocshmem_team_split_strided(ROCSHMEM_TEAM_WORLD,
181                               hip_id,                /* PE_start */
182                               gpus_per_node,         /* stride   */
183                               n_pes / gpus_per_node,/* size     */
184                               nullptr, 0,
185                               &same_gpu_team);
186 }
```



Team management, concurrent collective operations

```
134  for (int team_i = 0; team_i < num_teams; team_i++) {
135    team_alltoall_world_dup[team_i] = ROCSHMEM_TEAM_INVALID;
136    rocshmem_team_split_strided(ROCSHMEM_TEAM_WORLD, 0, 1, n_pes, nullptr, 0,
137                                &team_alltoall_world_dup[team_i]);
138    if (team_alltoall_world_dup[team_i] == ROCSHMEM_TEAM_INVALID) {
139      std::cout << "Team " << team_i << " is invalid!" << std::endl;
140      abort();
141    }
142  }
```

Host code:

- Team creation and splitting
  - team_split / team_split2d, team_destroy
- Example: create one team per workgroup: a **dup of team_world**

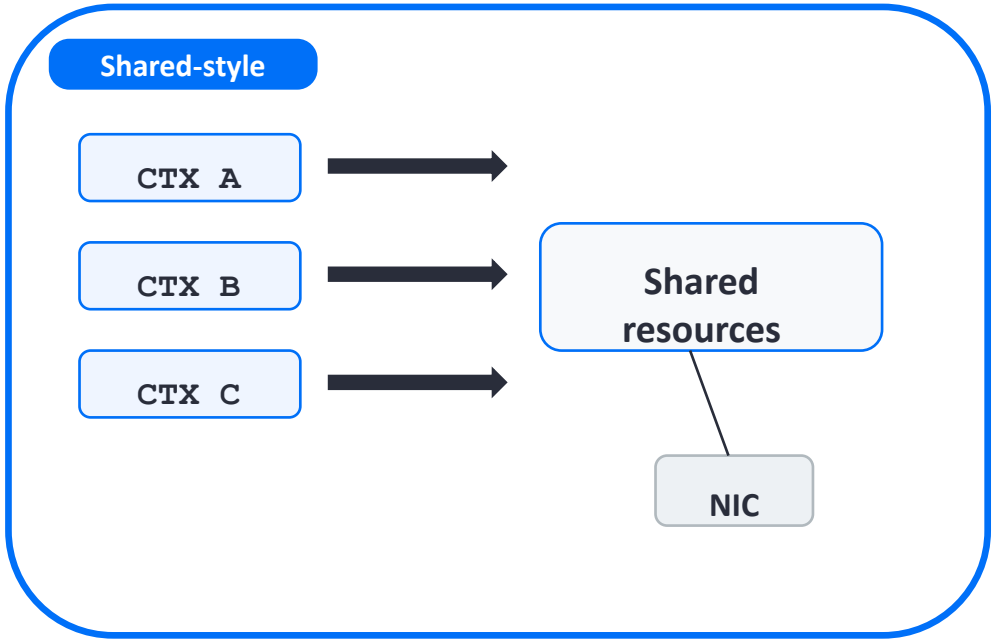
Device code:

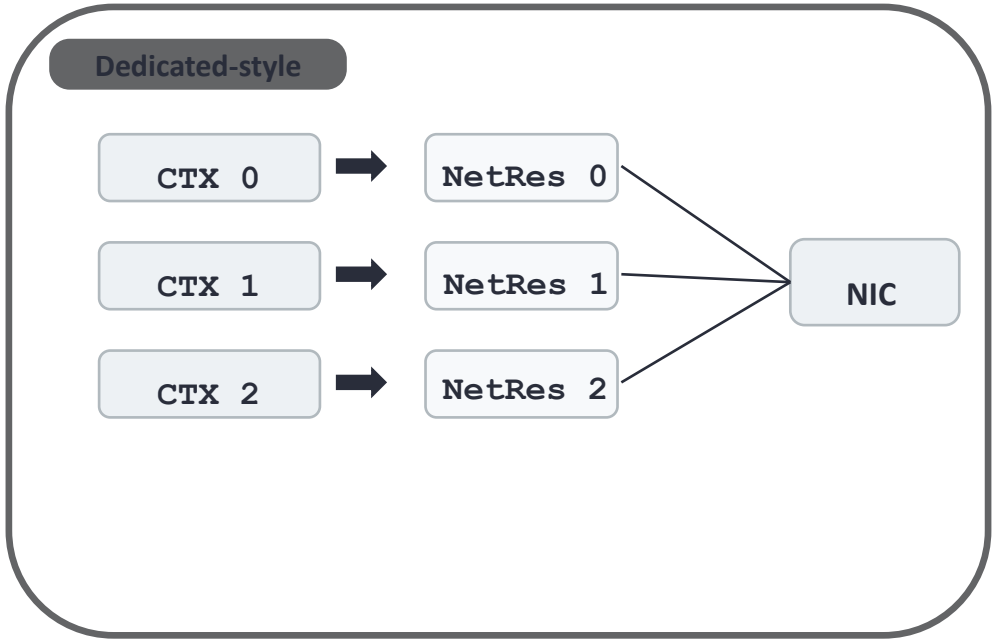
- get a **workgroup specific context** attached to the team
- issue **concurrent device alltoall** on the per-wg teams

```
64  __shared__ rocshmem_ctx_t ctx;
65  int wg_id = get_flat_grid_id();
66
67  rocshmem_wg_team_create_ctx(teams[wg_id], ctx_type, &ctx);
68
69  int n_pes = rocshmem_ctx_n_pes(ctx);
70
71  source_buf += wg_id * n_pes * num_elems;
72  dest_buf += wg_id * n_pes * num_elems;
73
74  rocshmem_ctx_int_alltoall_wg(ctx, teams[wg_id], dest_buf, source_buf, num_elems);
75
76  rocshmem_wg_ctx_destroy(&ctx);
77 }
```

Contexts, motivation

- Contexts provide an abstraction for hardware resource queues
- Help reduce contention when accessing hardware queues
- Provide user “intent” to the implementation about usage pattern: optimization opportunity





Hardware-context mapping is implementation dependent
Users can influence how the mapping is done to their preference

Context API

Obtain a workgroup specific context
- Reduced contention
- Dedicated QPs

Context-specific quiet reduces thread contention on QP

Batched non-blocking

```
__global__ void context_wave_put_nbi(char *source, char *dest, size_t size,
                                     int loop, int batch, ShmemContextType ctx_type)
{
  __shared__ rocshmem_ctx_t ctx;
  rocshmem_wg_ctx_create(ctx_type, &ctx);

  int wg_id  = get_flat_grid_id();
  int wf_id  = get_flat_block_id() / warpSize;
  int idx    = wf_id + wg_id * ((get_flat_block_size() - 1) / warpSize + 1);

  source += size * batch * idx;
  dest   += size * batch * idx;

  for (int i = 0; i < loop; i++) {
    size_t offset = (i % batch) * size;

    if (offset == 0)
      rocshmem_ctx_quiet(ctx);

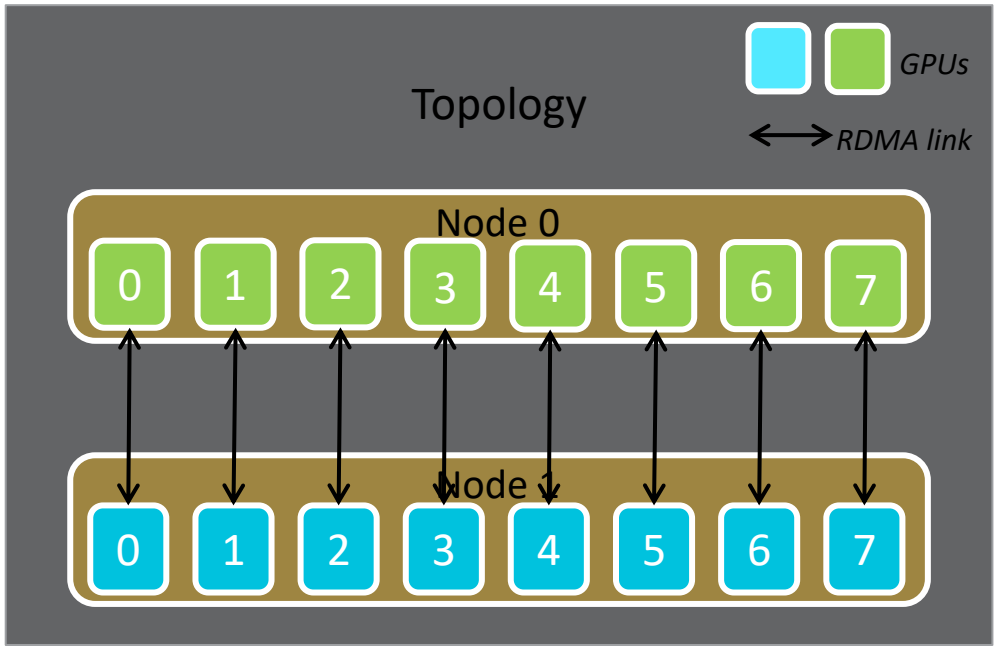
    rocshmem_ctx_putmem_nbi_wave(ctx, dest + offset, source + offset, size, 1);
  }

  rocshmem_ctx_quiet(ctx);
  rocshmem_wg_ctx_destroy(&ctx);
}
```

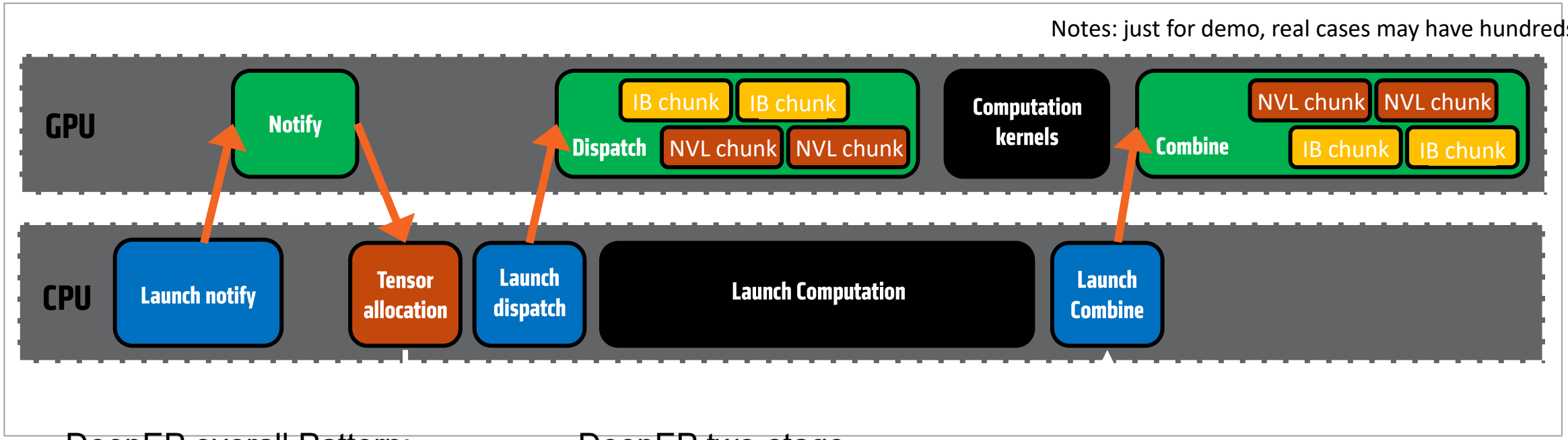
- Context optimized concurrent wf-scoped put from concurrent workgroups/threads

Real-life example: MoE kernel: DeepEP

- **DeepEP** is a high-performance communication framework optimized for **distributed inference** in ***Mixture-of-Experts (MoE)*** models.
- Core Features:
  - Even Expert Distribution:
    - Experts are evenly distributed across GPUs to ensure balanced load and scalability.
  - Routing Mechanism:
    - Each GPU:
      - Identifies the top-k expert GPUs for its local tokens
      - Sends tokens directly to those GPUs for processing
    - Topology-Aware Routing
  - GPU-Centric Communication:
    - Fine-grained, asynchronous, and CPU-free data exchange
    - Supports both intra-node and inter-node routing
  - High-Speed Data Transfer:
    - Utilizes GPU Direct RDMA via rocSHMEM
    - Employs Interprocess Communication (IPC) for local GPU coordination



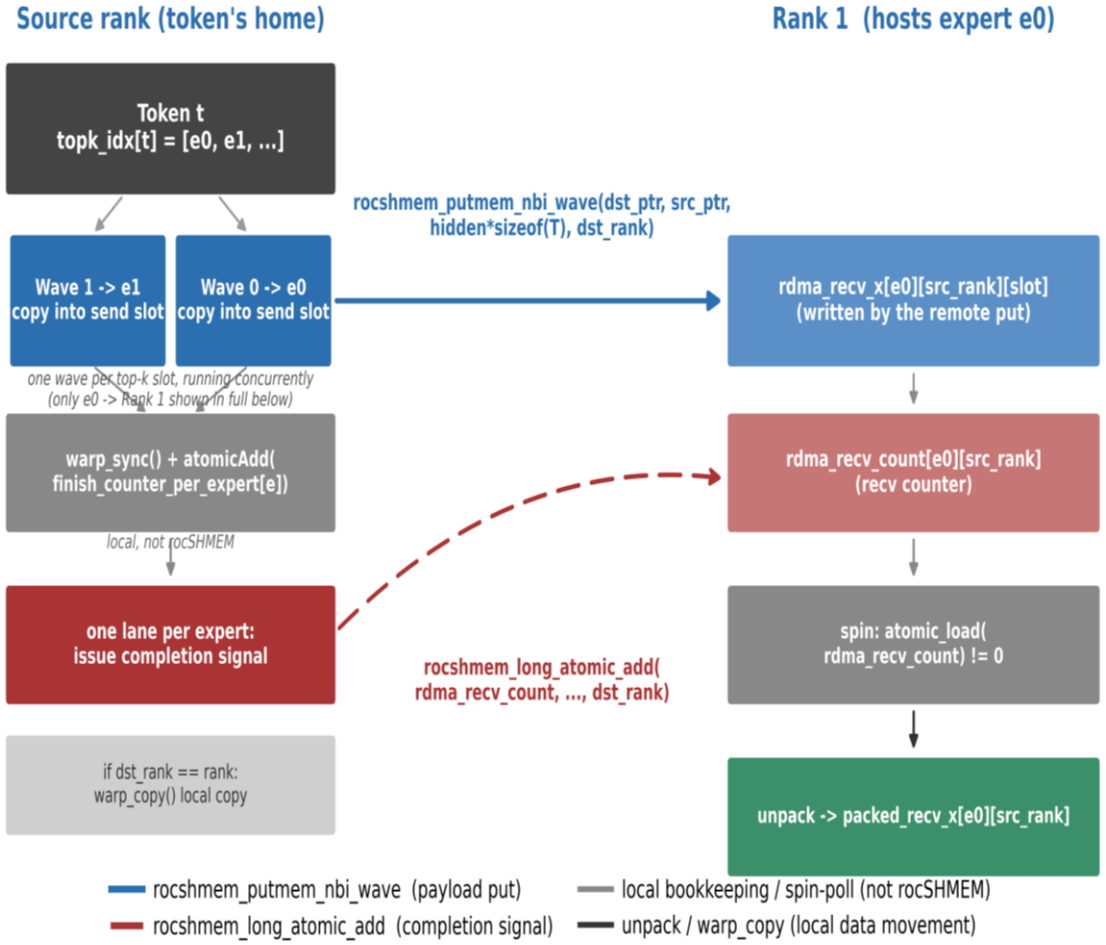
DeepEP Two-Stage Communication Pipeline



- DeepEP overall Pattern:
  - All-to-all, multi-consumer, multi-producer
  - Uses direct GPU-to-GPU communication
    - Supports intra-node and inter-node transfers
    - Token data is grouped and routed by expert placement
- DeepEP two-stage communication pipeline:
  1. Dispatch Phase:
     - Each GPU:
       - Selects top-k experts for its local tokens
       - Identifies destination GPUs for the selected experts
       - Sends token data to those destination GPUs
  2. Combine Phase
     - Each GPU:
       - Receives outputs for previously dispatched tokens
       - Merges partial results from multiple peers
       - Reconstructs final ordered output tensor

Examine code at https://github.com/ROCm/DeepEP

MoE Dispatch mockup code example




```
__global__ void moe_dispatch_mockup(void *rdma_recv_x, int64_t *rdma_recv_count,
    void *rdma_x, const void *x,
    const int64_t *topk_i
    int num_tokens, int h
    int num_topk, int num_
    int rank, int num_ran
{
  __shared__ rocshmem_ctx_t ctx;
  rocshmem_wg_ctx_create(ROCSHMEM_C                &ctx);
  //...
  for (int token_idx = blockIdx.x;                dx < num_tokens; token_idx += gridDim.x) {
    const float *x_ptr   = (const                *)x   + token_idx * hidden;
    float       *stg_ptr = (float                ma_x     + token_idx * hidden;

    /* ALL waves cooperate: copy token to symmetric staging buffer */
    for (int i = threadIdx.x; i < hidden; i += blockDim.x)
      stg_ptr[i] = x_ptr[i];
    __syncthreads();

    /* SPECIALIZATION 1: wave_id < num_
    * Issues NBI put to the destinatio
    * IPC-local path (dst_rank == rank
    * rdma_recv_x, no rocshmem call ne
    if (wave_id < num_topk) {
      int64_t dst_expert = topk_idx[token_i                _topk + wave_id];
      int     dst_rank   = (int)(dst_expe                m_local_experts);

      rocshmem_c                ve(ctx, dst_ptr, stg_ptr, token_bytes, dst_rank);
    }

    /* SPECIALIZATION 2: last wave — counts tokens per expert + cleans
    * next buffer (WG 0 only). Runs in parallel with puts above.
    * No IPC distinction needed: operates only on local counters. */
    if (wave_id == num_waves - 1) {
      /* count tokens routed to each responsible expert and
      * adjust atomic_finish_counter to reflect the complement */
    }
  }
```

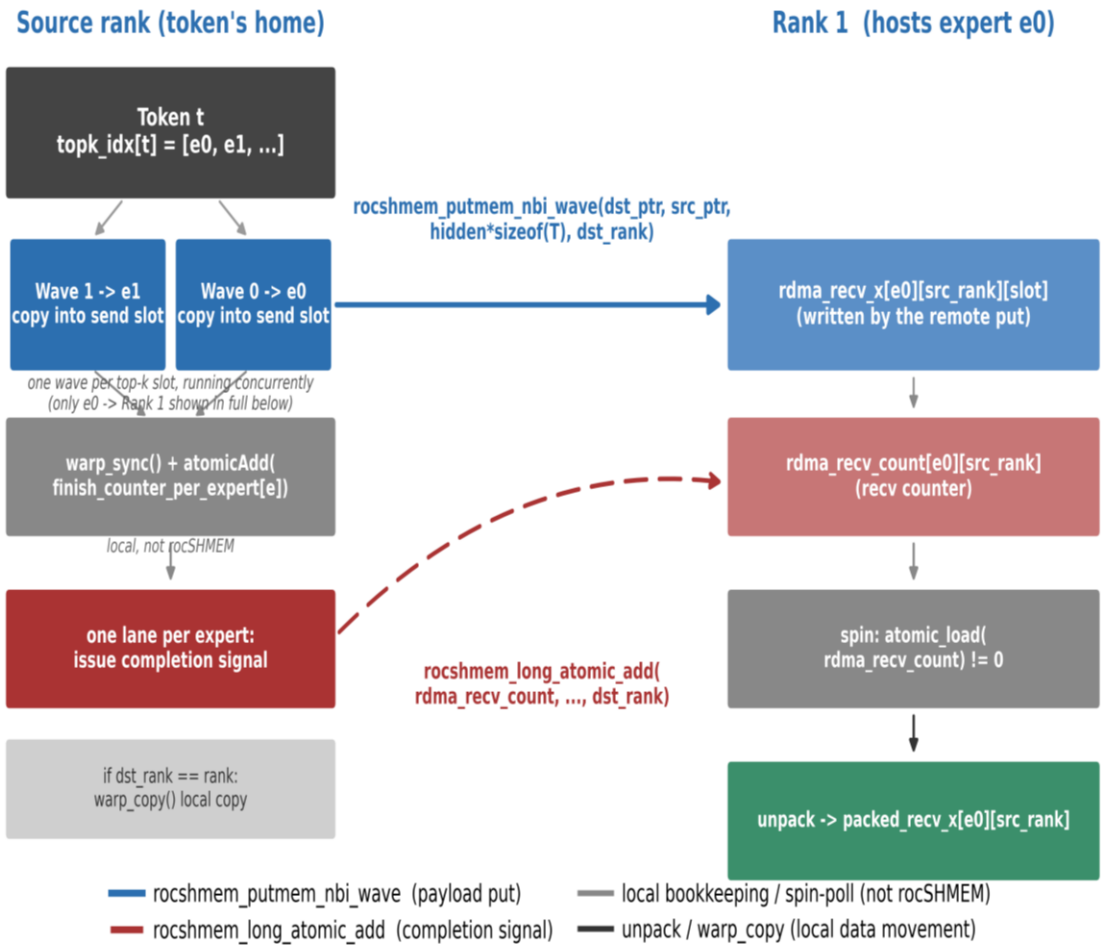
Copy pytorch tensors to symmetric memory stage-in buffer

Wave-Specialization: top-k put to experts in parallel

Compute-communication overlap, this wave performs next iteration cleanup

Examine real code at https://github.com/ROCm/DeepEP

MoE Dispatch mockup code example




```
259   /* Barrier: handoff between put-specialization and signal-specialization.
260   ¦* Signaling waves were not involved in puts — they need this guarantee. */
261   __syncthreads();
262
263   /* SPECIALIZATION 3: responsible_expert (wave_group_id) — one wave_group
264   ¦* per destination expert. Spins until all puts for that expert are
265   ¦* accounted for, then signals the receiver.
266   ¦* IPC-local path (dst_rank == rank): hip_atomic_store instead of
267   ¦* rocshmem_ctx_long_atomic_add. */
268   if (responsible_expert < num_experts && sub_wave_id == 0 && lane_id == 0) {
269   ¦  int dst_rank      = responsible_expert / num_local_experts;
270   ¦  int dst_local     = responsible_expert % num_local_experts;
271   ¦  rocshmem_ctx_fence_pe(ctx, dst_rank);
272   ¦  rocshmem_ctx_long_atomic_add(ctx,
273   ¦  ¦  ¦  rdma_recv_count + dst_local * num_ranks + rank,
274   ¦  ¦  ¦  -1L, dst_rank);
275   }
276
277   rocshmem_wg_ctx_destroy(&ctx);
278 }
```

Examine real code at https://github.com/ROCm/DeepEP

Using with Triton kernels

torchrun --nproc_per_node=2 pingpong_triton.py

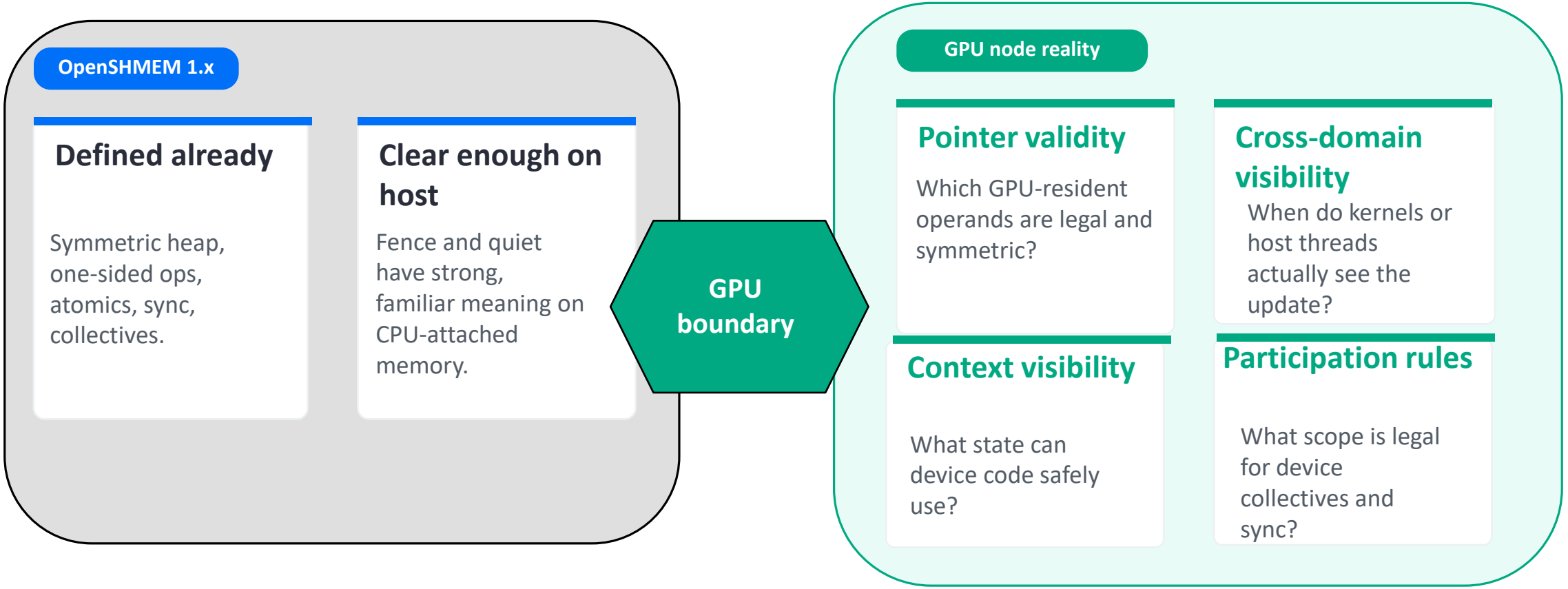
Ping pong with putsignal_wg

Selecting the rocSHMEM backed for pytorch symm_mem module

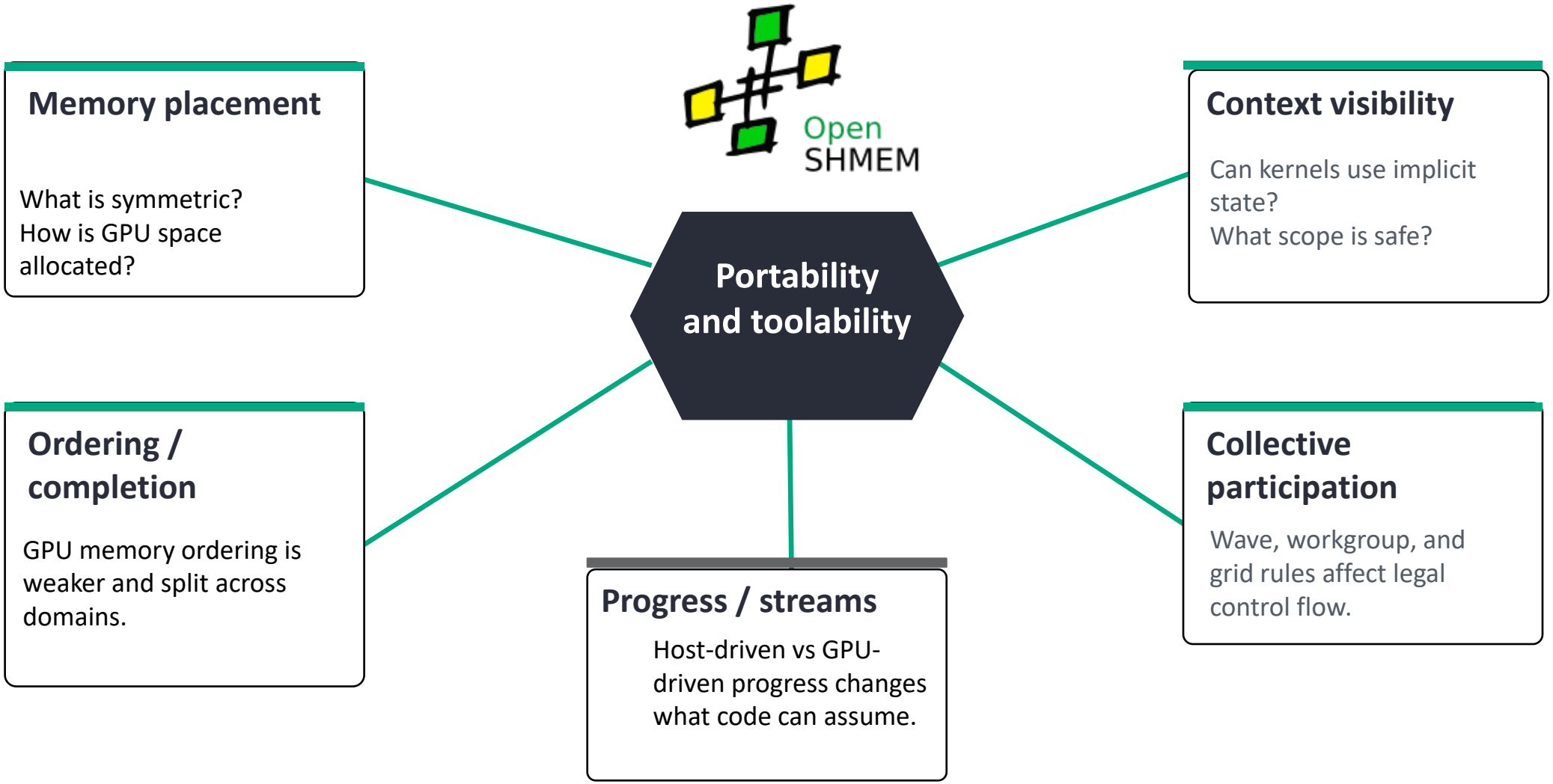
```python
import torch
import torch.distributed as dist
import torch.distributed._symmetric_memory as symm_mem
import torch.distributed._symmetric_memory._shmem_triton as shmem_triton
from torch._inductor.runtime.triton_compat import triton
import triton.language as tl

ROCSHMEM_SIGNAL_ADD = 1
ROCSHMEM_CMP_EQ     = 0

shmem_backend  = shmem_triton.get_shmem_backend_module()
requires_shmem = shmem_triton.requires_shmem

@requires_shmem
@triton.jit
def pingpong_kernel(dst, src, signal, peer, nbytes,
                    SIGNAL_ADD: tl.constexpr, CMP_EQ: tl.constexpr):
    if shmem_backend.my_pe() == 0:
        shmem_backend.putmem_signal_block(dst, src, nbytes, signal, 1, SIGNAL_ADD, peer)
        shmem_backend.signal_wait_until(signal, CMP_EQ, 1)
    else:
        shmem_backend.signal_wait_until(signal, CMP_EQ, 1)
        shmem_backend.putmem_signal_block(dst, src, nbytes, signal, 1, SIGNAL_ADD, peer)

dist.init_process_group("nccl")
rank  = dist.get_rank()
peer  = 1 - rank
torch.cuda.set_device(rank)
symm_mem.set_backend("ROCSHMEM")
group = dist.distributed_c10d._get_default_group().group_name

src     = symm_mem.empty(64, dtype=torch.int8, device=f"cuda:{rank}").fill_(rank + 1)
dst     = symm_mem.empty(64, dtype=torch.int8, device=f"cuda:{rank}").fill_(-1)
dst_hdl = symm_mem.rendezvous(dst, group=group)
symm_mem.rendezvous(src, group=group)
flag    = dst_hdl.get_signal_pad(rank, (1,), dtype=torch.int64).fill_(0)
dist.barrier()

pingpong_kernel[(1,1,1)](dst, src, flag, peer, 64,
                         SIGNAL_ADD=ROCSHMEM_SIGNAL_ADD, CMP_EQ=ROCSHMEM_CMP_EQ)
torch.cuda.synchronize()
assert (dst == peer + 1).all()
```

OpenSHMEM GPU Standardization effort

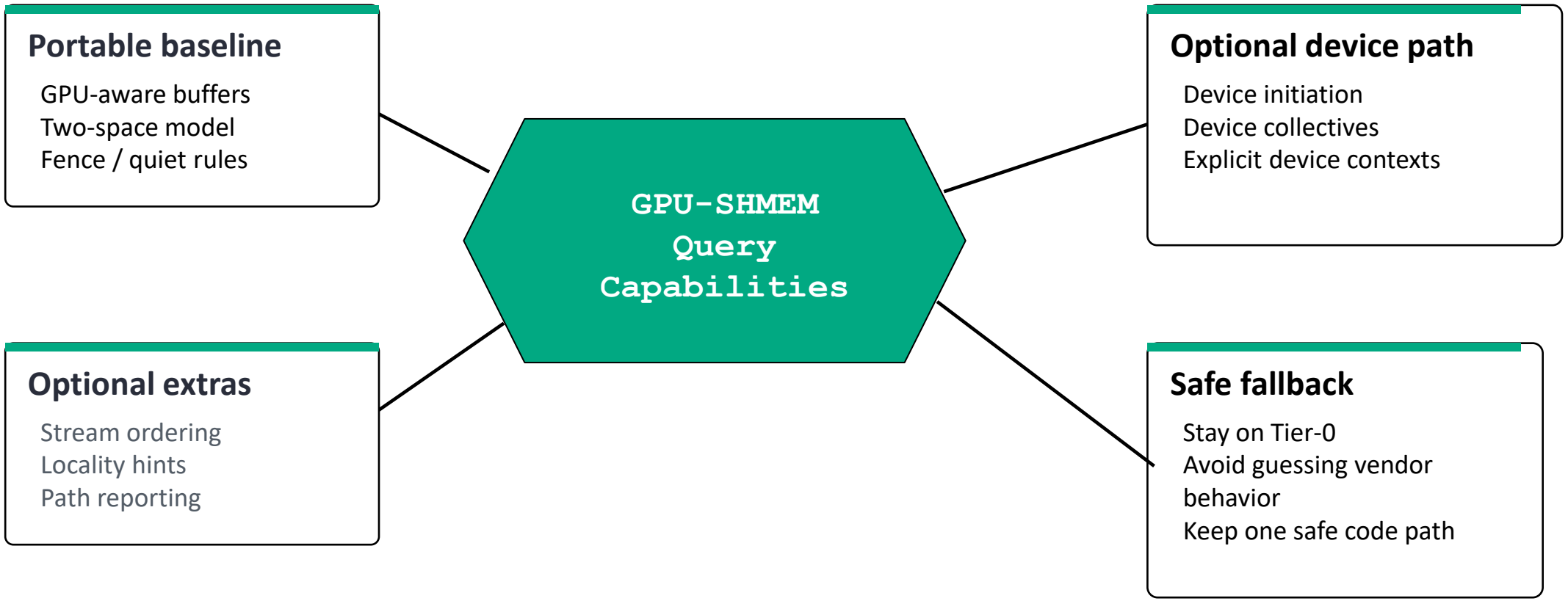
OpenSHMEM 1.6 works well on CPU-based systems but not on GPUs What can OpenSHMEM learn from rocSHMEM field experience



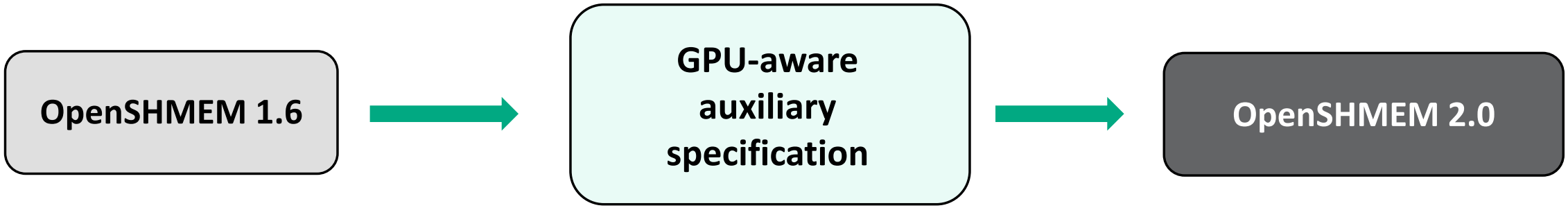
The real portability break is semantic divergence in memory, contexts, ordering, collectives, and progress.



Capability discovery lets applications target a safe portable baseline instead of reverse-engineering vendor behavior.



A small auxiliary spec is the lowest-risk bridge from OpenSHMEM 1.6 to a future 2.0.
Standardize adjacent to existing behavior for GPU enabled libraries like rocSHMEM



1 The need is real now.

Leadership systems are already GPU-centric, and OpenSHMEM 1.x stops at the boundary.

2 A portable baseline is feasible now.

Most Tier-0 behavior already exists in current implementations.

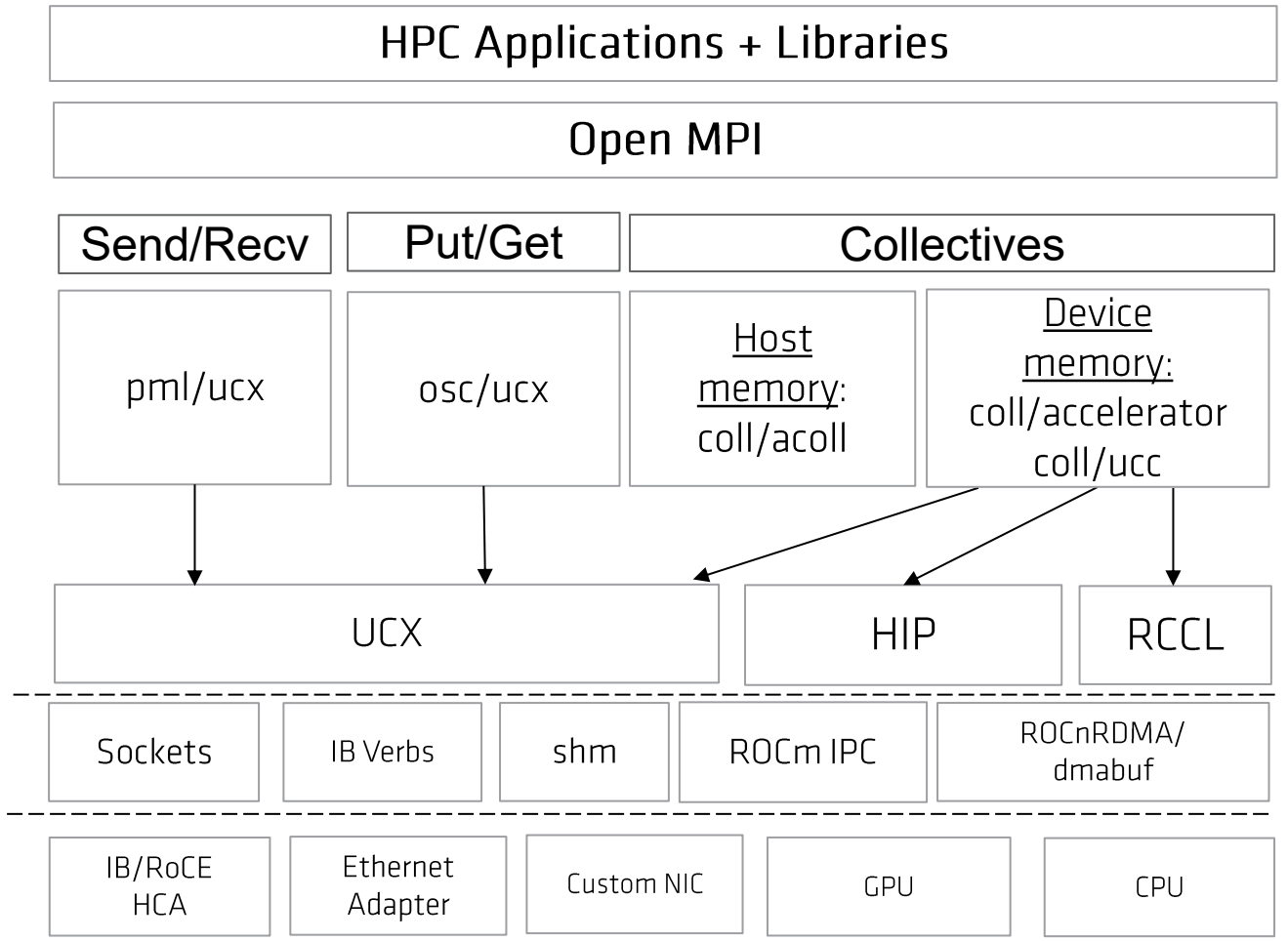
3 The roadmap preserves innovation later.

Optional device-centric features can mature without blocking convergence.

Part 4: HPC Communication Libraries – UCX, Open MPI

Recommended Open MPI software stack for AMD hardware

- Recommended software stack for InfiniBand/RoCE networks
- Most stable and best tested configuration



UCX

UCX overview

- **Open-source framework** for communication across low-latency network technologies
- Developed under the Unified Communication Framework Consortium

- **ROCm™ support in UCX**
  - **ROCM/COPY**: data transfer between host and device memory within a single process
  - **ROCM/IPC**: data transfer between device memories of different processes on the same node
  - GPUDirect RDMA for communication between processes on different nodes
    - Dma-buf support available for selected kernel/NIC combinations
  - Memory type detection of ROCm memory

Configuring UCX for AMD GPUs

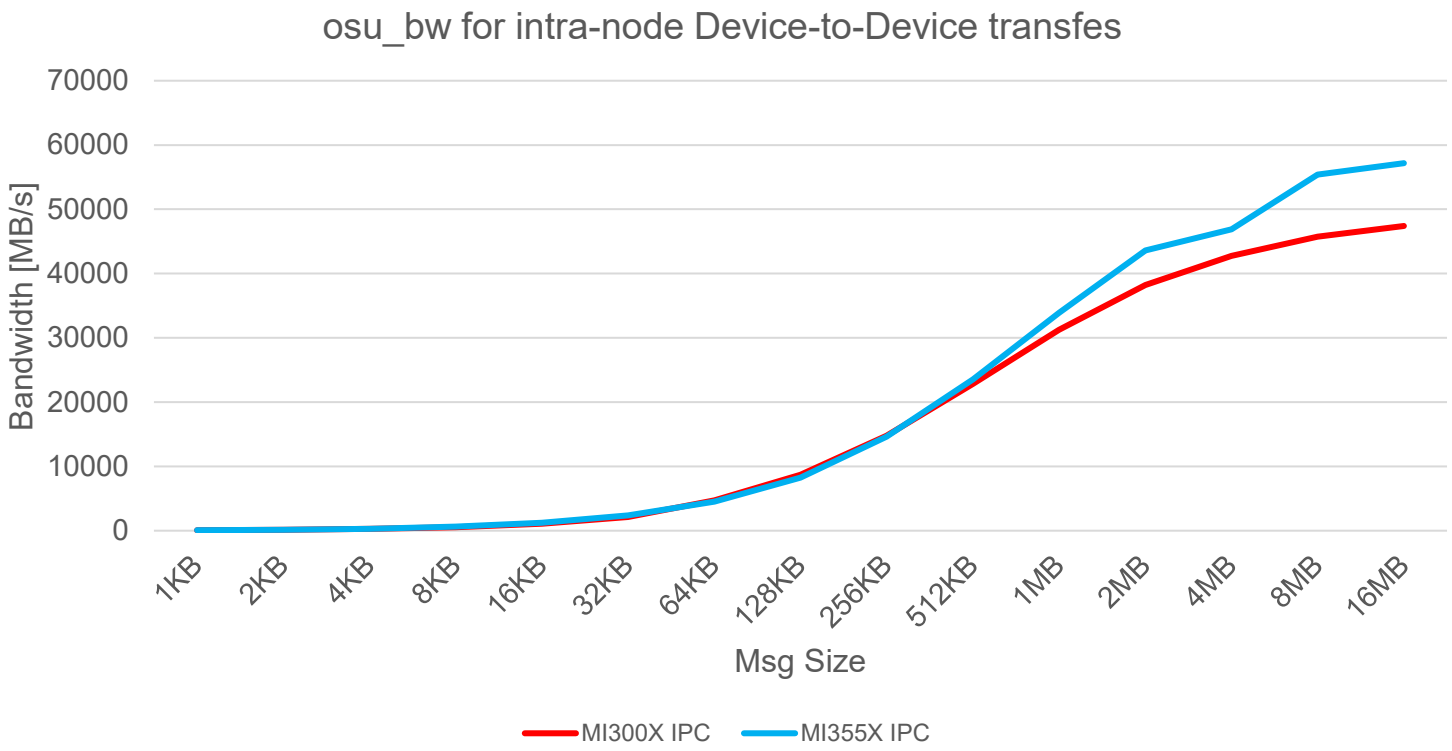
```
$ wget https://github.com/openucx/ucx/archive/refs/tags/v1.21.0.tar.gz
$ tar -xzf v1.21.0.tar.gz
$ cd ucx-1.21.0
$ mkdir build && cd build
$ ../configure --with-rocm=$ROCM_DIR --prefix=$UCX_INSTALL_DIR --enable-mt
               --disable-assertions
$ make && make install
```

AMD specific features

- **`UCX_ROCM_COPY_H2D_THRESH:`**
  - Switching point between using large BAR for Host-to-Device transfers vs. ROCm runtime functionality within a single process
  - Default value: 1 MBytes
- **`UCX_ROCM_COPY_D2H_THRESH:`**
  - Switching point between using large BAR for Device-to-Host transfers vs ROCm runtime functionality within a single process
  - Default value: 256 Bytes
- **`UCX_ROCM_IPC_MIN_ZCOPY`**
  - Min. message size for which to use the Device-to-Device transfer over InfinityFabricTM Links
  - Default value: 128 Bytes
- **`UCX_ROCM_COPY_DMABUF`**
  - Use kernel DMA-Buf mechanism for GPU to NIC communication instead of peermem kernel module
  - Requires Linux kernel support for two features (`CONFIG_DMABUF_MOVE_NOTIFY, CONFIG_PCI_P2PDMA`)
  - Distributions typically have the features enabled in newer distros (e.g. Ubuntu 24.04 or newer with Linux kernel 6.8.0)

UCX intra-node point-to-point performance

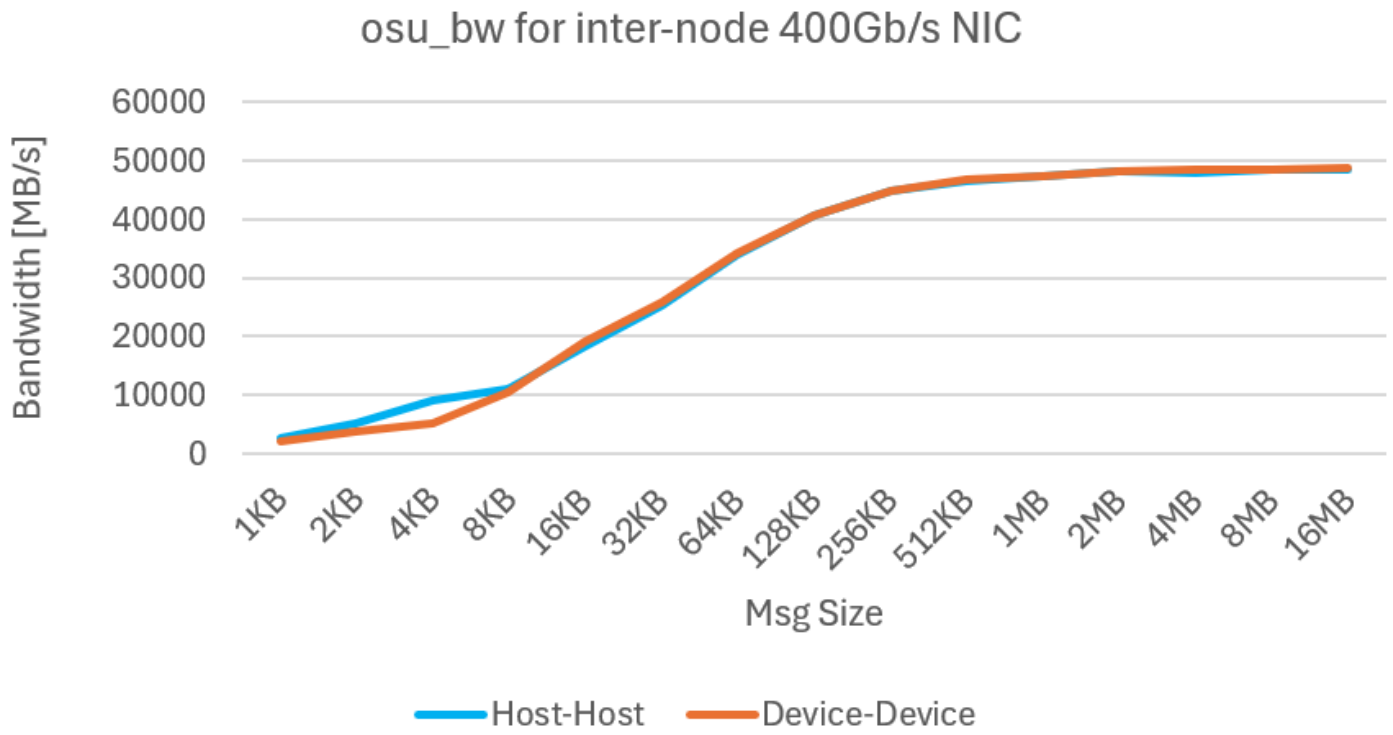
- UCX 1.21.0
- Open MPI 5.0.10
- osu benchmarks 7.5.1[1]



[1] OSU Benchmark Suite https://mvapich.cse.ohio-state.edu/benchmarks/ (BSD License).

UCX inter-node point-to-point performance

- UCX 1.21.0
- Open MPI 5.0.10
- osu benchmarks 7.5.1[1]
- Broadcom Thor2 400Gb/s NICs

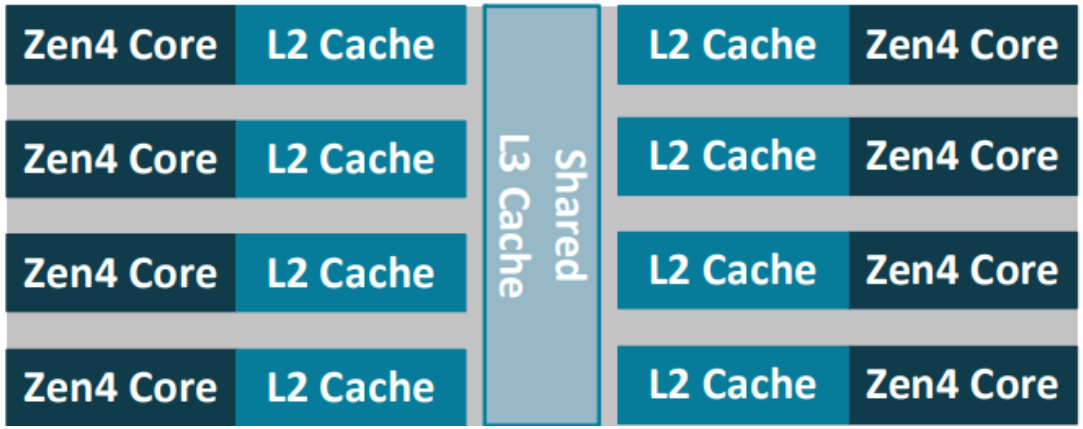


[1] OSU Benchmark Suite https://mvapich.cse.ohio-state.edu/benchmarks/ (BSD License).

Optimizations on the AMD “Zen 3/4/5” CPU architectures

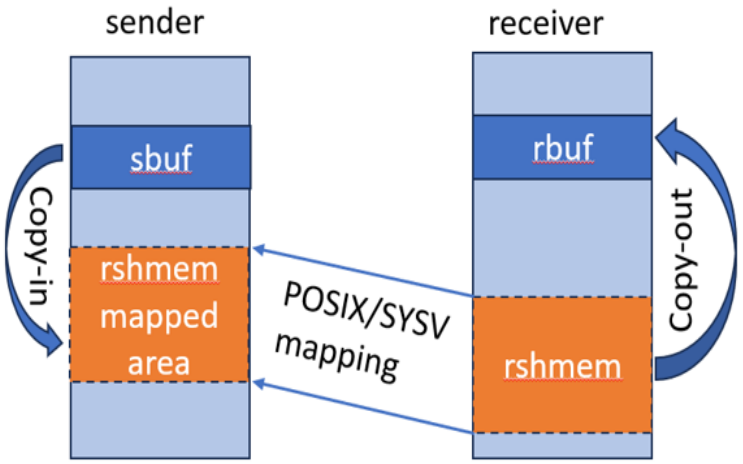
- Processors based on the AMD “Zen 3/4/5” architecture organize CPU cores into clusters of 8 or more that share a common L3cache
- Hybrid MPI applications often employ *'n-ranks* × *m-threads*' configuration

  n <= Number of unified L3caches in system
  m >= Number of cores that share a common L3cache
- Non-temporal buffer transfer improves data transfer performance for hybrid workloads
- Two shared-memory data transfer mechanisms available:
  - copy-in, copy-out
  - single copy (e.g., cma, xpmem, knem)
- To enable:
  - Configure UCX with `--enable-optimizations` option set
  - Set `UCX_NT_BUFFER_TRANSFER_MIN=0` at runtime

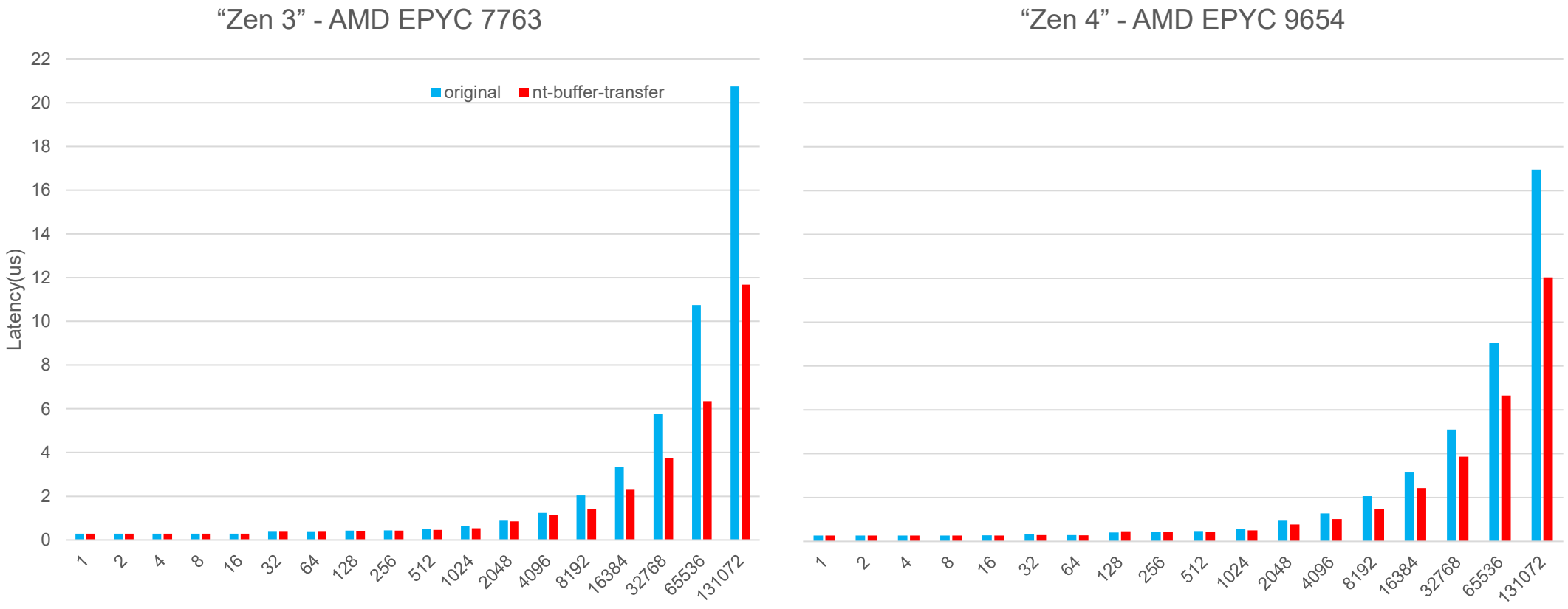


Improving UCX Copy-in, Copy-out performance

- Replaces glibc's `memcpy()` for both copy-in and copy-out
  - New copy-in routine:
    - Copy sender's buffer to receiver's shared memory
    - Use non-temporal store instead of normal store instructions while storing the data to shared memory destination
  - New copy-out routine:
    - Copy from shared memory to receiver's buffer
    - Use *'Loads with PREFETCHNTA'* instead of normal load instruction while loading from shared memory
- Reduces data transfer latency by circumventing cache-to-cache data transfers, which tend to be slower, when ranks/processes are situated in different Core Complex (CCX)
- Up to 40% performance improvement observed on "Zen 4" AMD EPYC 9654 when using non-temporal copies



CICO osu_latency[1] Benchmark Results (map-by l3cache)

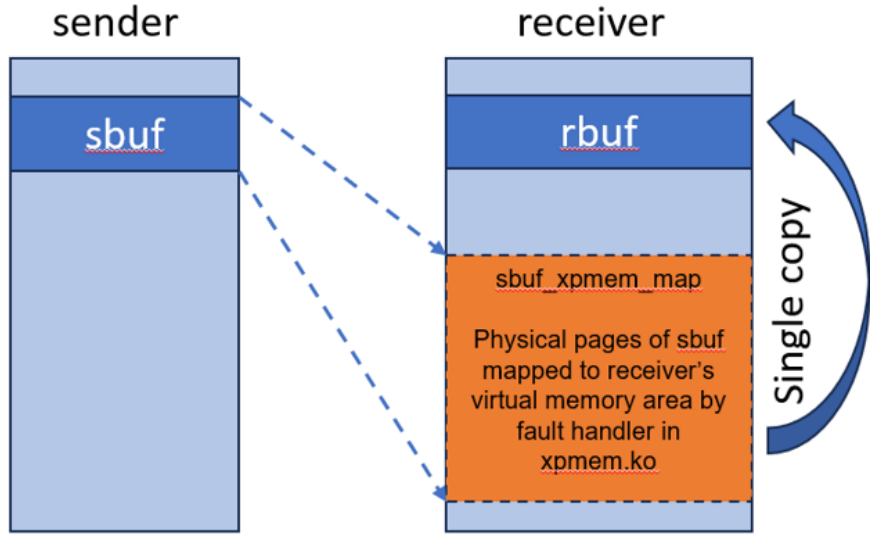


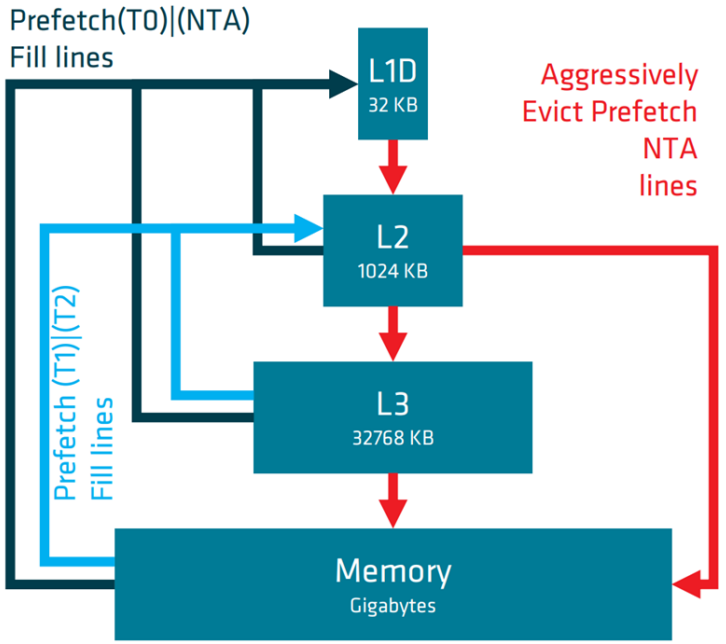
Gain: “Zen 3” up to 43 % @128 KB, ”Zen 4” upto 28 % @128 KB

[1] OSU Benchmark Suite https://mvapich.cse.ohio-state.edu/benchmarks/ (BSD License).

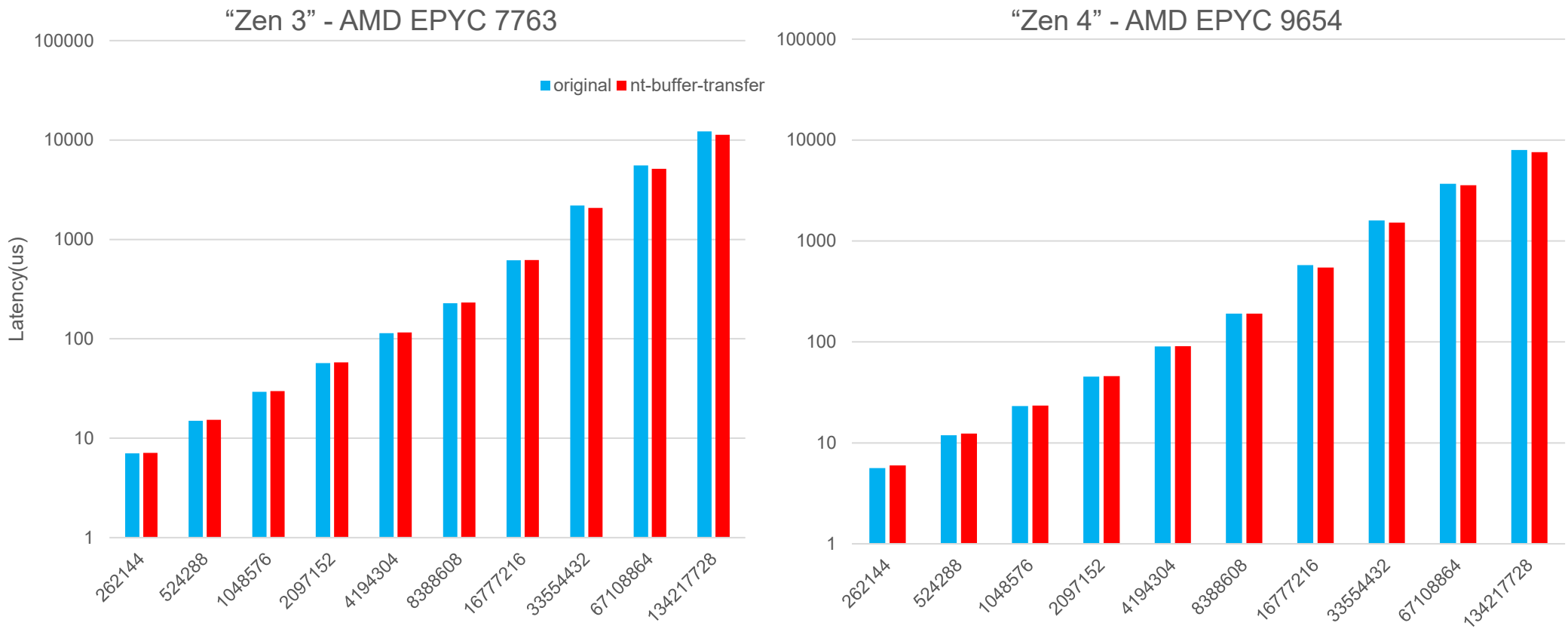
UCX single copy (xpmem) optimization

- Focusing on xpmem as it is the only single copy mechanism that uses `memcpy()` in userspace
- Replace the standard `memcpy()` operation with *'loads with PREFETCHNTA'*
- For very larger sizes (> L3 cache size) combine *'loads with PREFETCHNTA'* with nontemporal stores to final receiver buffer
- Reduces cache pollution and L3-to-L3 transfers
  - Up to 5% performance gain observed for short messages
  - 20-40% performance improvement observed for large messages
- Feature will be available in upcoming UCX 1.18 release
- User will have to explicitly ask for the feature initially





SCOPY osu_latency[1] Benchmark Results (map-by l3cache)

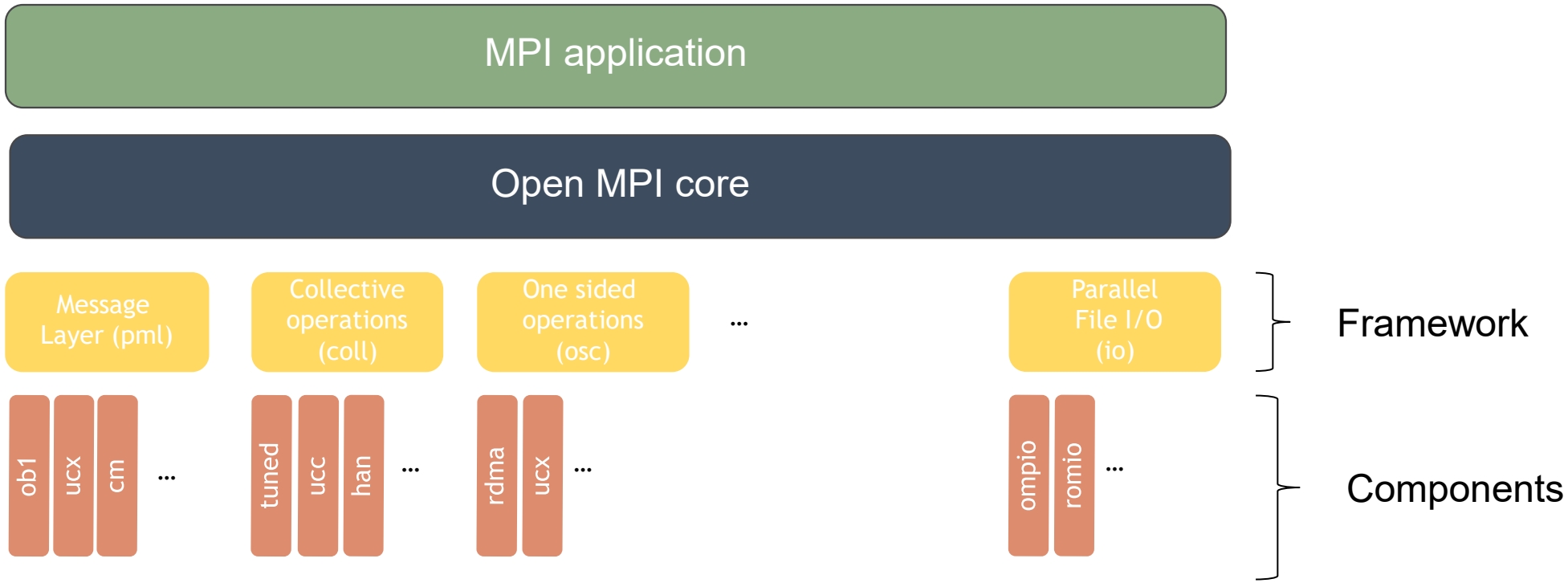


Gain: "Zen 3" 5-7 % for larger sizes, "Zen 4" 4-5 % for larger sizes

[1] OSU Benchmark Suite https://mvapich.cse.ohio-state.edu/benchmarks/ (BSD License).

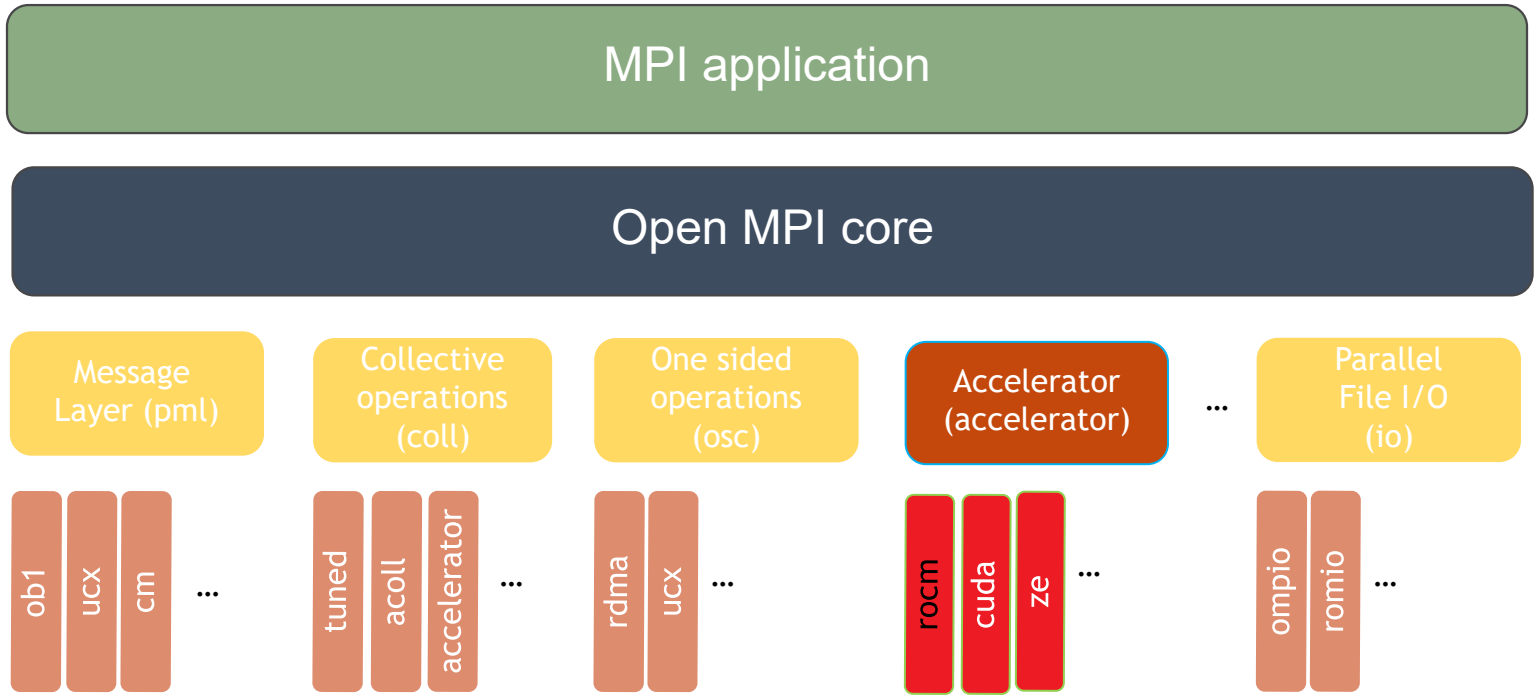
Open MPI

Open MPI Architecture



Accelerator framework

- New framework introducing an abstraction layer for GPU support in Open MPI
- Available starting Open MPI v.5.0



ROCm support in Open MPI

- ROCm support added in Open MPI starting v5.0
  - Configure logic to detect ROCm software stack
  - ROCm memory type detection
  - Data transfer to/from ROCm device memory
    - Beneficial for non-contiguous derived datatypes, file I/O, etc.
    - Enables usage of other components (e.g., libfabric, ob1, etc.)
  - Query availability of ROCm support in MPI library (`MPIX_QUERY_ROCM_SUPPORT`)
- ROCm device memory supported through **pml/ucx** and **osc/ucx**

Open MPI with UCX

Configuring und running Open MPI

- Configure, compile, and install UCX with ROCm support
- Configure, compile, and install UCC with ROCm (and optionally RCCL) support

```
./configure --with-rocm=<ROCM_DIR>
            --with-ucx=<UCX_DIR>
            --with-ucc=<UCC_DIR>
            --prefix=<OMPI_DIR>

mpirun   --mca pml ucx --mca osc ucx
         --mca coll_ucc_enable 1
         --mca coll_ucc_priority 100
         -np 64 ./my_mpi_app
```

Intra-node Send / Recv paths

Send buffer / Recv buffer	hipMalloc() hipMallocPitch() hipMalloc3D()	hipHostMalloc()	hipMallocManaged()	malloc()	malloc()+ hipHostRegister()
hipMalloc() hipMallocPitch() hipMalloc3D()	Infinity Fabric™	Bounce buffer + Host SM§	Bounce buffer + Host SM	Bounce buffer + Host SM	Bounce buffer + Host SM
hipHostMalloc()	Bounce buffer + Host SM§	Host SM	Host SM	Host SM	Host SM
hipMallocManaged()	Bounce buffer + Host SM	Host SM	Host SM	Host SM	Host SM
malloc()	Bounce buffer + Host SM	Host SM	Host SM	Host SM	Host SM
malloc()+ hipHostRegister()	Bounce buffer + Host SM	Host SM	Host SM	Host SM	Host SM

CPU to GPU NUMA locality

rocm-smi --showtopo

```
============================== Numa Nodes ==============================
GPU[0]          : (Topology) Numa Node: 3
GPU[0]          : (Topology) Numa Affinity: 3
GPU[1]          : (Topology) Numa Node: 3
GPU[1]          : (Topology) Numa Affinity: 3
GPU[2]          : (Topology) Numa Node: 1
GPU[2]          : (Topology) Numa Affinity: 1
GPU[3]          : (Topology) Numa Node: 1
GPU[3]          : (Topology) Numa Affinity: 1
GPU[4]          : (Topology) Numa Node: 0
GPU[4]          : (Topology) Numa Affinity: 0
GPU[5]          : (Topology) Numa Node: 0
GPU[5]          : (Topology) Numa Affinity: 0
GPU[6]          : (Topology) Numa Node: 2
GPU[6]          : (Topology) Numa Affinity: 2
GPU[7]          : (Topology) Numa Node: 2
GPU[7]          : (Topology) Numa Affinity: 2
========================== End of ROCm SMI Log ==========================
```

If MPI rank 0 is running on NUMA node 0 it should use GPU 4 or 5 for best performance!

CPU to GPU NUMA Locality (II)

```
bash~$ mpirun --report-binding --map-by numa --mca pml ucx -np 8
              ./osu_bcast -d rocm
[x1000c2s1b0n0:68075] Rank 0 bound to package[0][core:0-15]
[x1000c2s1b0n0:68075] Rank 1 bound to package[0][core:0-15]
[x1000c2s1b0n0:68075] Rank 2 bound to package[0][core:16-31]
[x1000c2s1b0n0:68075] Rank 3 bound to package[0][core:16-31]
[x1000c2s1b0n0:68075] Rank 5 bound to package[0][core:32-47]
[x1000c2s1b0n0:68075] Rank 4 bound to package[0][core:32-47]
[x1000c2s1b0n0:68075] Rank 6 bound to package[0][core:48-63]
[x1000c2s1b0n0:68075] Rank 7 bound to package[0][core:48-63]
```

To make sure that an MPI rank picks up a GPU that on the same NUMA node, adjust the order in which the GPUs are reported to the HIP application

```
bash~$ mpirun -x HIP_VISIBLE_DEVICES=4,5,2,3,6,7,0,1 --map-by numa
              --mca pml ucx -np 8 ./osu_bcast -d rocm
```

Force using a particular NIC and device

Many HPC applications prefer to explicitly set the GPU and the NIC that is being used by each MPI process

```
mpiexec --mca pml ucx -n 8 ./set_net_device.sh <my_executable>
                                               <application_args>
```

with set_net_device.sh being

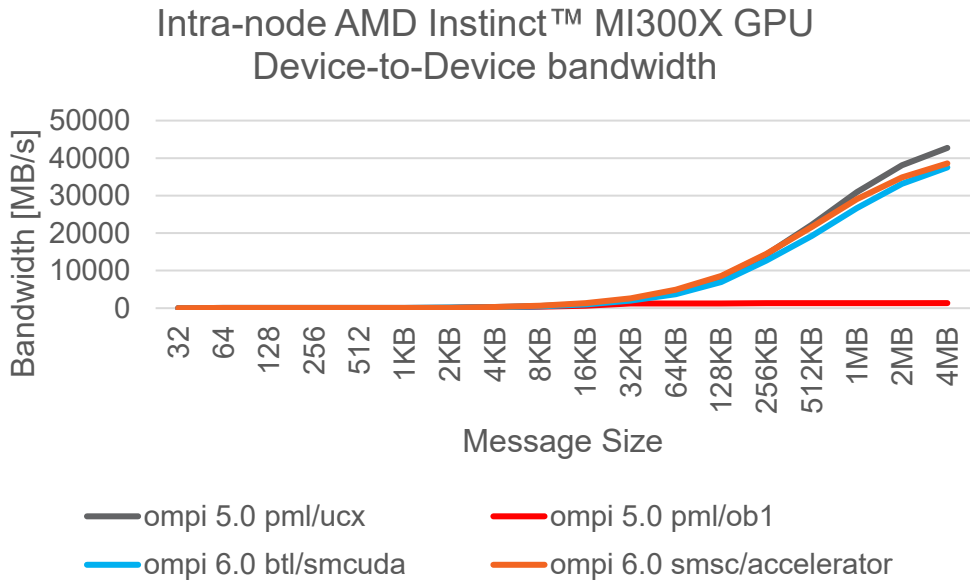
```
#!/bin/bash

if [[ -n ${OMPI_COMM_WORLD_LOCAL_RANK+x} ]]; then
        rank=${OMPI_COMM_WORLD_LOCAL_RANK}
fi

export UCX_NET_DEVICES=bnxt_re_bond$dev:1
export HIP_VISIBLE_DEVICES=$dev
“$@”
```

Standalone Open MPI in upcoming 6.0 release

Enhanced GPU support

- **Added support for Open MPI native intra-node device-to-device transfers**
  - **Single copy** accelerator component (***smsc/accelerator***)
  - Generalized the existing ***btl/smcuda*** component to work **with any accelerator** supported by Open MPI
  - Both components achieve similar performance to UCX for device-to-device transfers
  - Brings **performance parity** for system configuration that are not supported by UCX
  - Components have to be explicitly requested by user

```
mpirun --mca pml ob1 --mca smsc_accelerator_priority 80 -np 4 ./<executable>
```



Enhanced GPU support

- **Support for MPI 4.1 memory-allocation kinds**
  - User can restrict memory types supported for an `MPI_Comm`, `MPI_Win`, `MPI_File`, or `MPI_Session`
  - A few optimizations taking advantage of this feature already available in Open MPI 6.0
    - Some components can disqualify themselves during object creation based on the info object value
    - Buffer-pointer type-check bypass depending on the info object value in the parallel file I/O component
    - More to follow

```
MPI_Info info;
MPI_Info_create (&info);
char assert_key[] = "mpi_assert_memory_alloc_kinds";
char assert_value[] = "rocm:device";
MPI_Info_set (info, assert_key, assert_value);
MPI_Comm_dup_with_info (MPI_COMM_WORLD, info, &comm_dup);
```

Important options

- **mpi_accelerator_rocm_memcpyD2H_limit**
- **mpi_accelerator_rocm_memcpyH2D_limit**
  - control max. message until when to take advantage of large BAR feature

Collective Operations

Collective Components

- With GPU support enabled in Open MPI and UCX, most components can be used for most collective operations
  - e.g. an Alltoall implemented as a sequence of Isend/Irecv operations will work for both CPU and GPU buffers

- Challenges:
  - This assumption does not hold for components that use single-copy operations directly (xhc, acoll)
  - Reduction operations require special handling
  - No guarantees that a component will perform equally well on CPU and GPU buffers

Collective Components

- Highlighted Components:
  - **coll/accelerator** (Open MPI v6.0.x):
    - uses CPU bounce-buffers for GPU reduction operations
    - works well for short reduction operations
  - **coll/ucc** (Open MPI v.5.0.x and 6.0.x):
    - based on the UCC communication library (https://github.com/openucx/ucc) compiled with ROCm support
    - can make use of RCCL internally
    - best component if your application has large reduction operations
    - limited to platforms that support UCX
  - **coll/acoll** (Open MPI v6.0.x):
    - collective component tuned for AMD CPUs

Questions?

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18


AMD