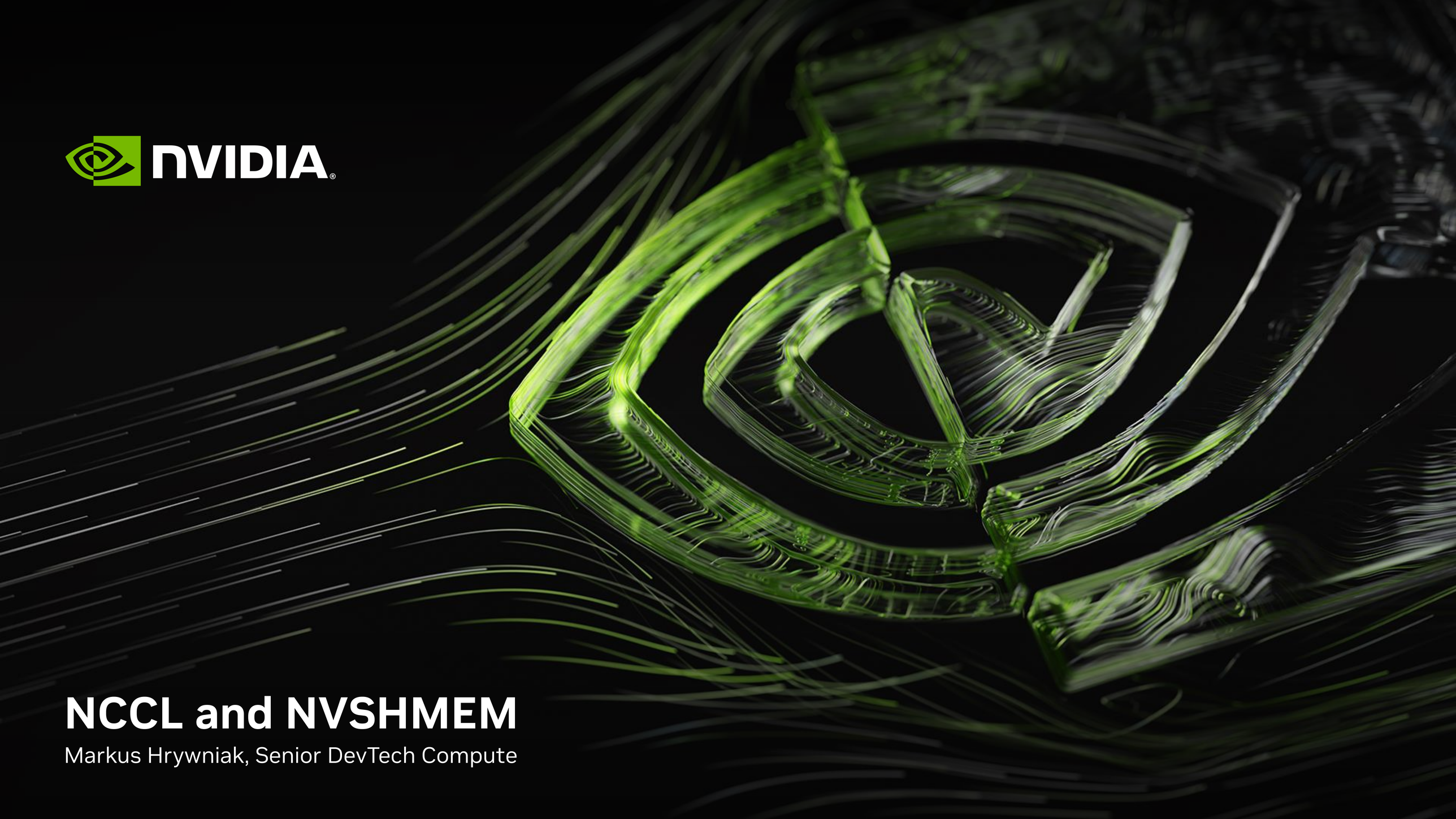


NCCL and NVSHMEM

Markus Hrywniak, Senior DevTech Compute

Advanced Communication Libraries

Why NCCL and NVSHMEM

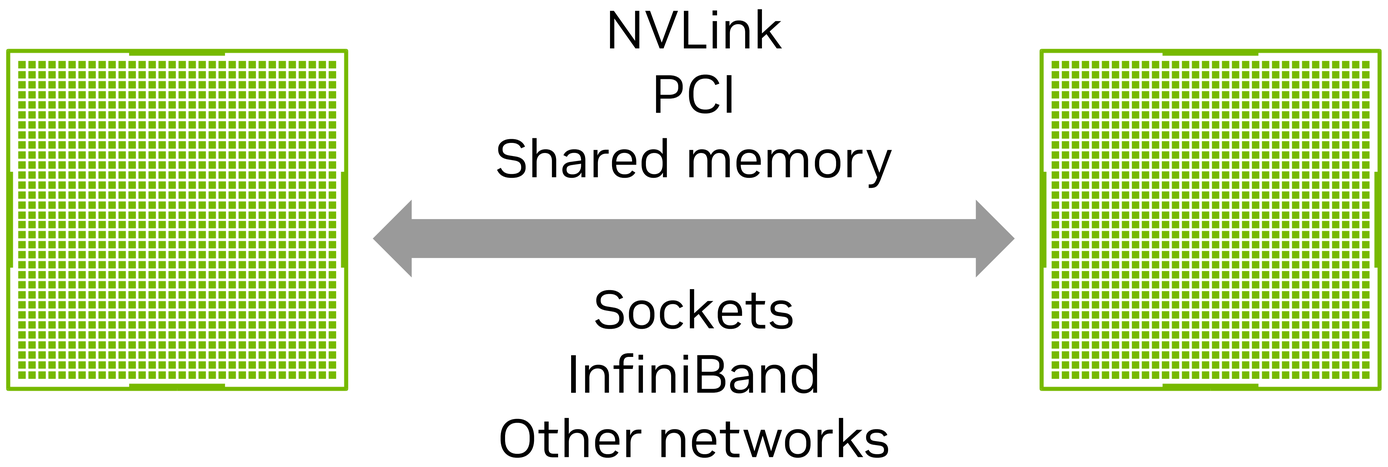
- NCCL - GPU collectives, point-to-point, stream-aware
  - Operations initiated asynchronously on host, into CUDA stream (same as kernel launch)
- NVSHMEM - OpenSHMEM for NVIDIA GPUs
  - From host - superficially similar to NCCL, but *not* message-passing - PGAS!
  - Also: communicating directly from the device, from inside a kernel
- Pure MPI: No stream-awareness/direct accelerator support - more synchronization
- Learn about usage of NCCL and NVSHMEM, and build a good mental model
  - Debug performance - my code, system configuration, use case?
- Material adapted from many sources (colleagues, GTC talks, SC/ISC tutorial, ...)
  - References have a lot more depth and detail

- Some questions!
  - Used libraries?
  - PGAS?

NCCL : NVIDIA Collective Communication Library

Pronounced "Nickel"

Communication library running on GPUs, for GPU buffers.



Binaries : https://developer.nvidia.com/nccl and in NGC containers
Source code : https://github.com/nvidia/nccl
Perf tests : https://github.com/nvidia/nccl-tests

NCCL

Initialization with MPI interop

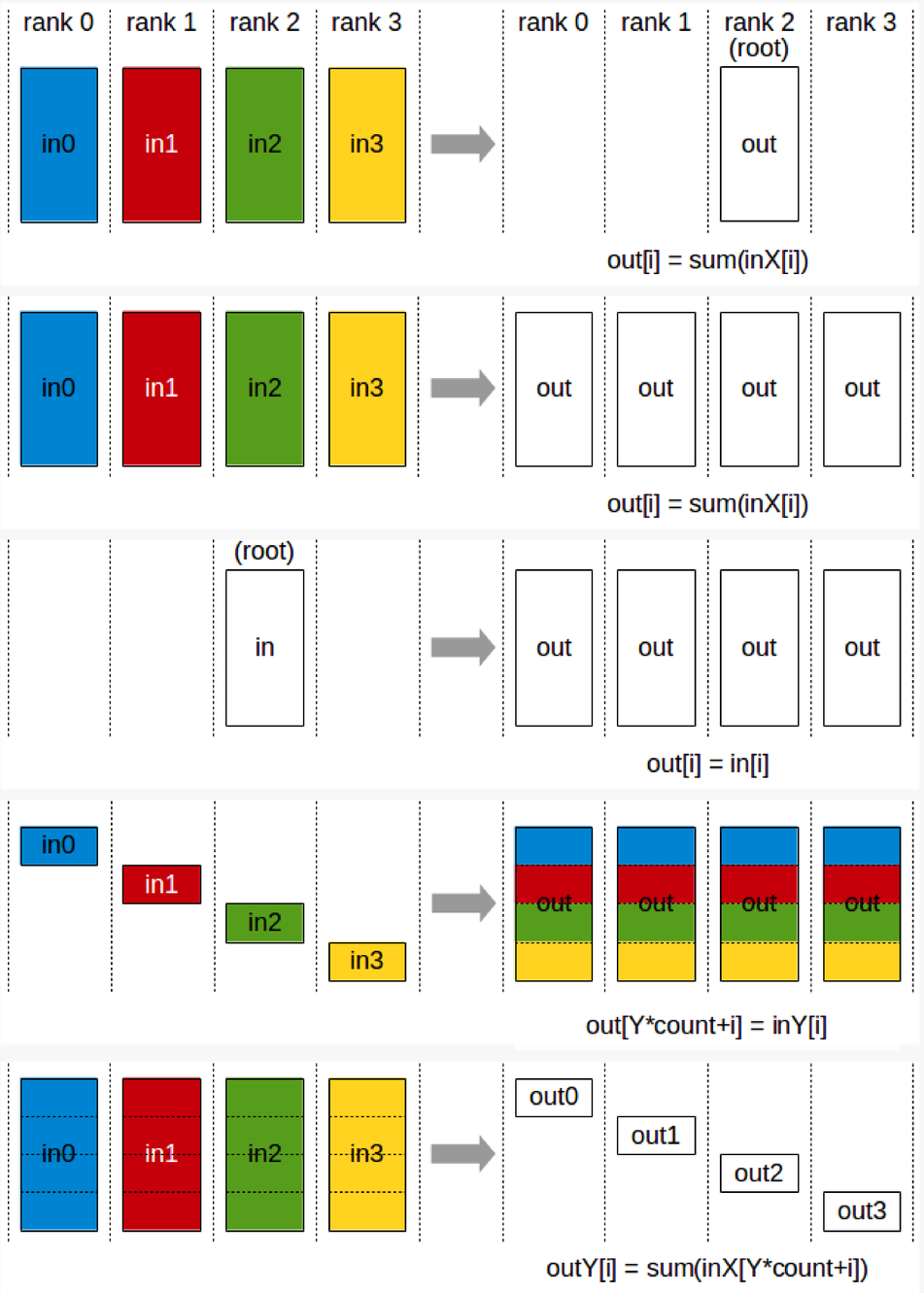
```
#include <nccl.h>
int main(int argc, char *argv[]) {
    ...
    ncclUniqueId nccl_uid;
    if (rank == 0) ncclGetUniqueId(&nccl_uid);
    MPI_Bcast(&nccl_uid, sizeof(ncclUniqueId), MPI_BYTE, 0, MPI_COMM_WORLD);
    ncclComm_t nccl_comm;
    ncclCommInitRank(&nccl_comm, size, nccl_uid, rank);
    ...
    ncclCommDestroy(nccl_comm);
    ...
    return 0;
}
```

NCCL has a slim API

Seven* Communication functions

```
// Collective communication
ncclReduce
ncclAllReduce
ncclBroadcast
ncclAllGather
ncclReduceScatter

// Point to point communication
ncclSend
ncclRecv
```

More details at https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html



NCCL Aggregated operations

Merge multiple operations on the same CUDA device

- Pay the launch overhead only once (more operations per second)
- Use multiple NVLinks simultaneously (more bandwidth)
- Use `ncclGroupStart()` / `ncclGroupEnd()` around the NCCL operations we want to aggregate :

```
ncclGroupStart();
for (int op=0; op<nops; op++) {
  ncclAllReduce(
    layers[op].localGradients,
    layers[op].globalGradients,
    layers[op].gradientSize,
    ncclFloat, ncclSum, ncclComm, ncclStream);
}
ncclGroupEnd();
// All operations are only guaranteed to be posted on the stream after ncclGroupEnd
cudaStreamSynchronize(ncclStream);
```

NCCL Background - Topology Detection

Mapping algorithms to the hardware

Topology detection

Build graph with all GPUs, NICs, CPUs, PCI switches, NVLink, NVSwitch.

Topology injection for VMs.

Graph search

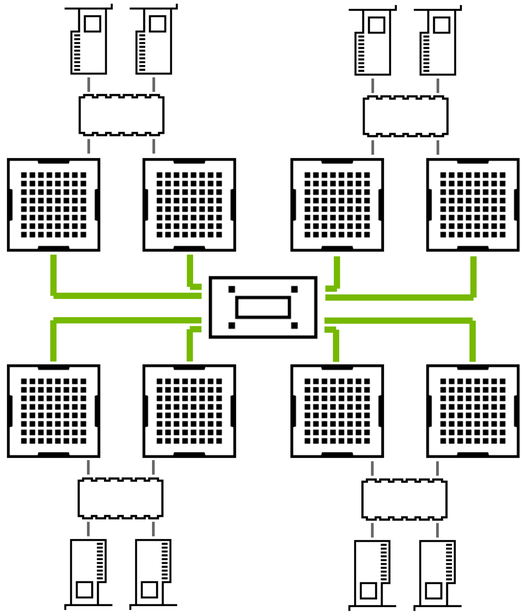
Find optimal set of paths within the node for rings, trees and chains.

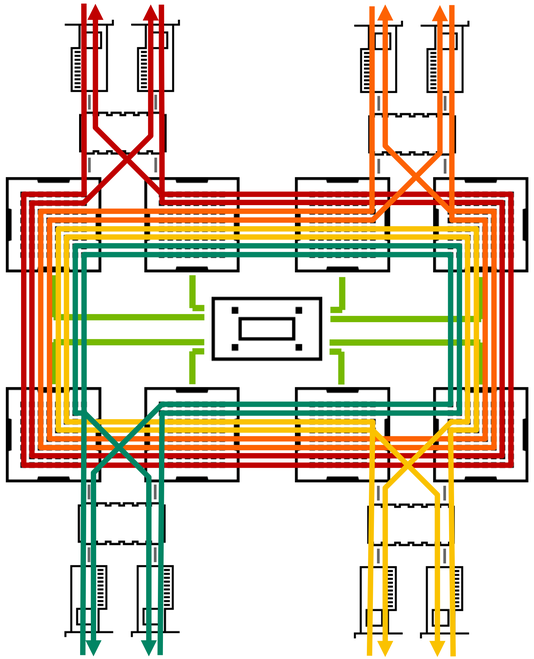
Performance model for each algorithm, tuning.

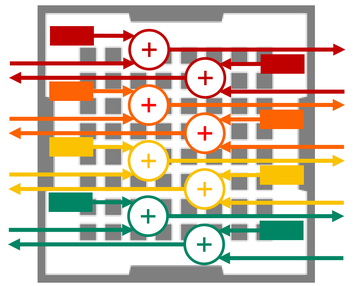
CUDA Kernels

Optimized reductions and copies for a minimal SM usage.

CPU threads for network communication.



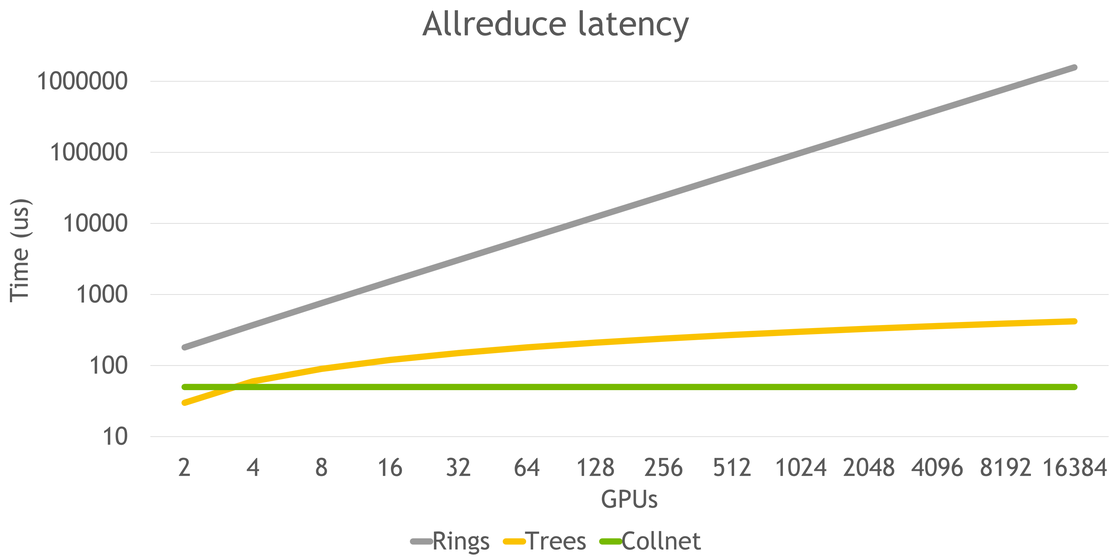


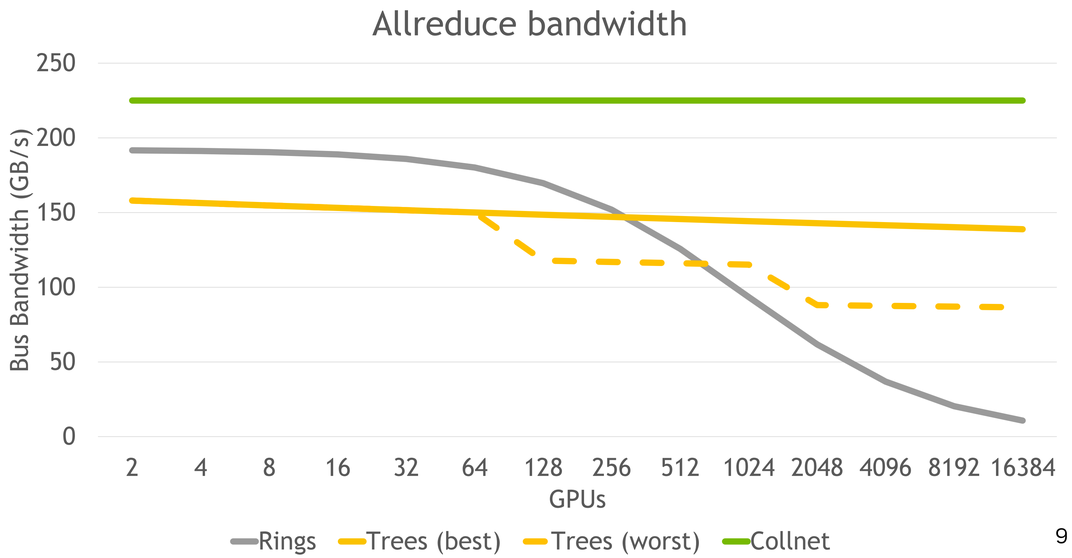


Algorithms Summary

Pros and cons

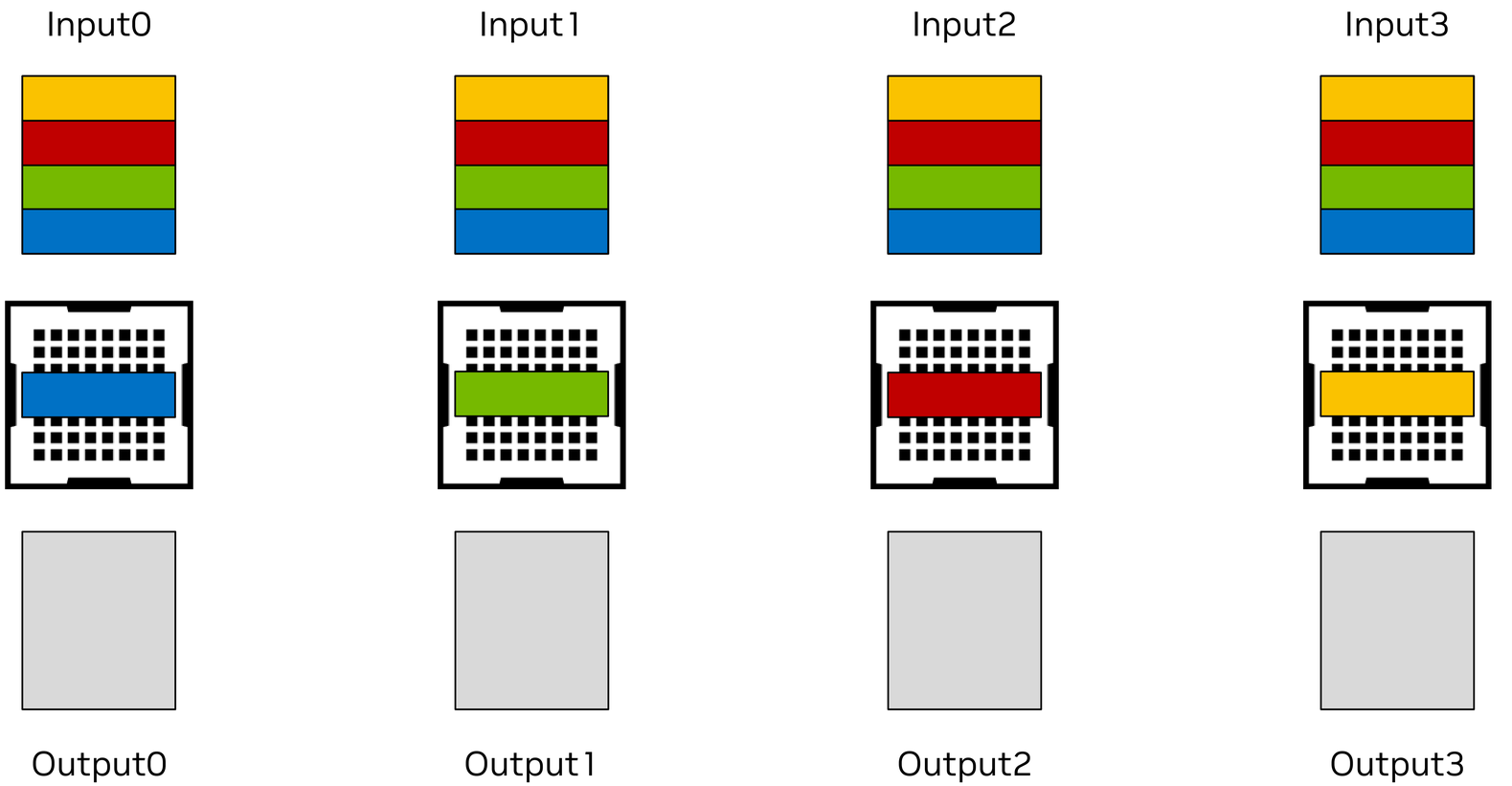
Algorithm	Latency	Bandwidth	Computing	Network Pattern
Rings	Linear	Perfect	Uniform	Few flows
Trees	Log	Close to perfect	Imbalanced	Many flows
Collnet	Constant	Close to 2x (may be limited by NVLink)	Almost uniform	Minimal flows





Ring Algorithm

For allreduce

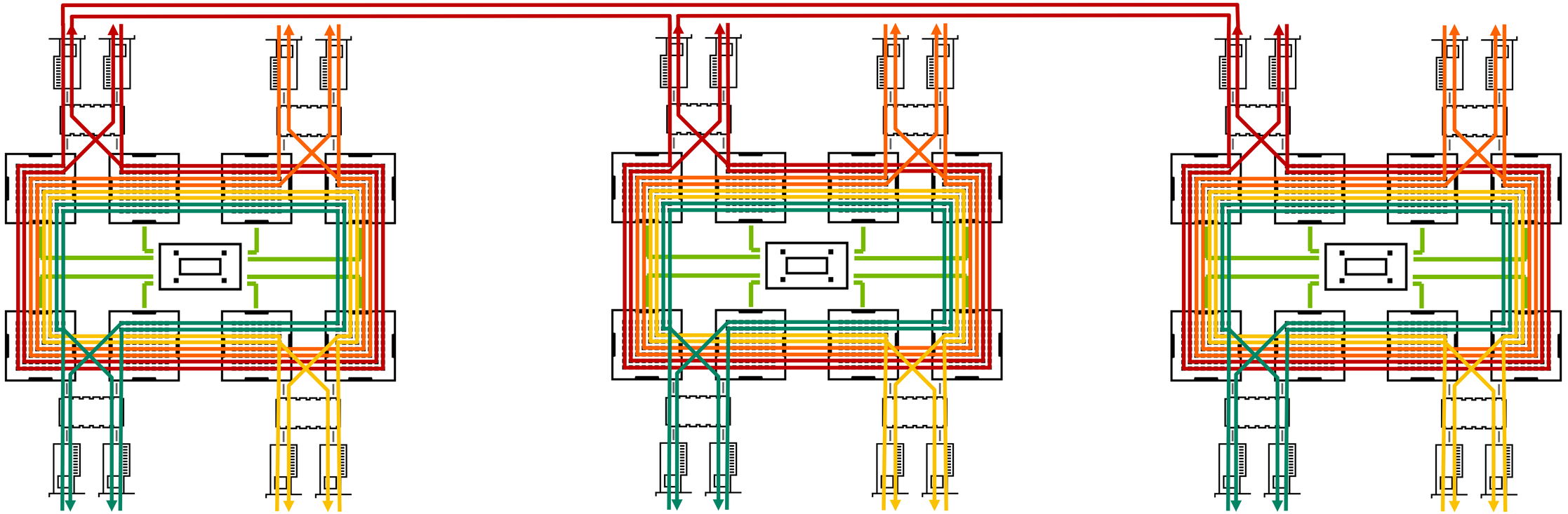


Simple algorithm, works on all topologies, balanced computation.

Latency increases linearly with the number of GPUs; quickly degrades at scale.

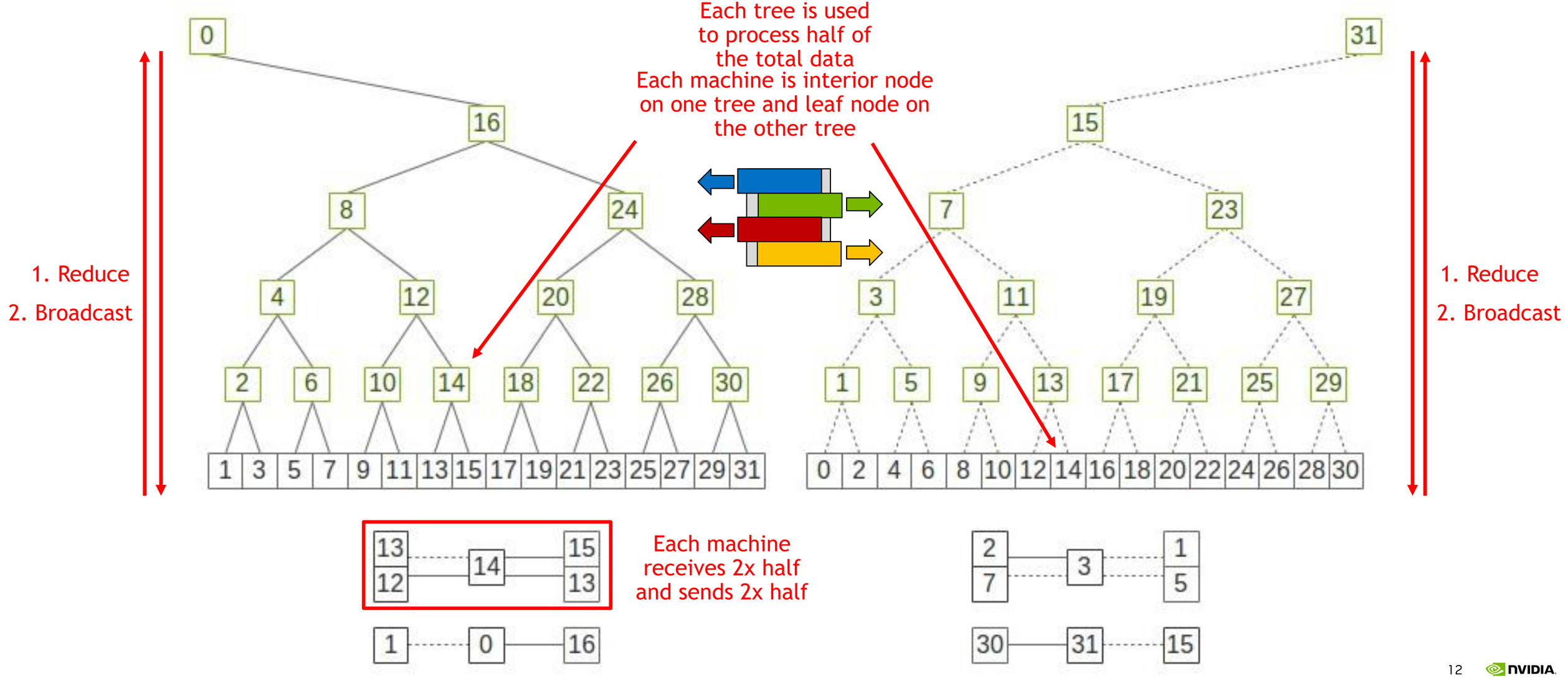
Ring algorithm

Multi-rings on DGX



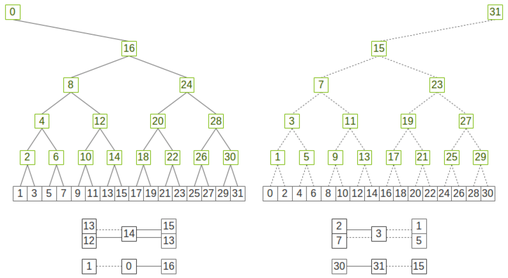
TREE Algorithm

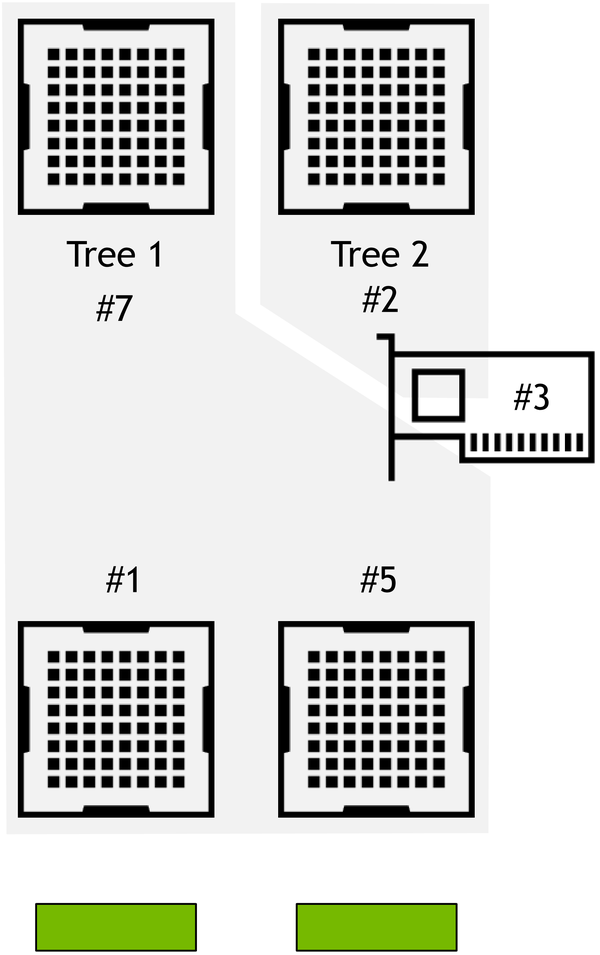
For allreduce

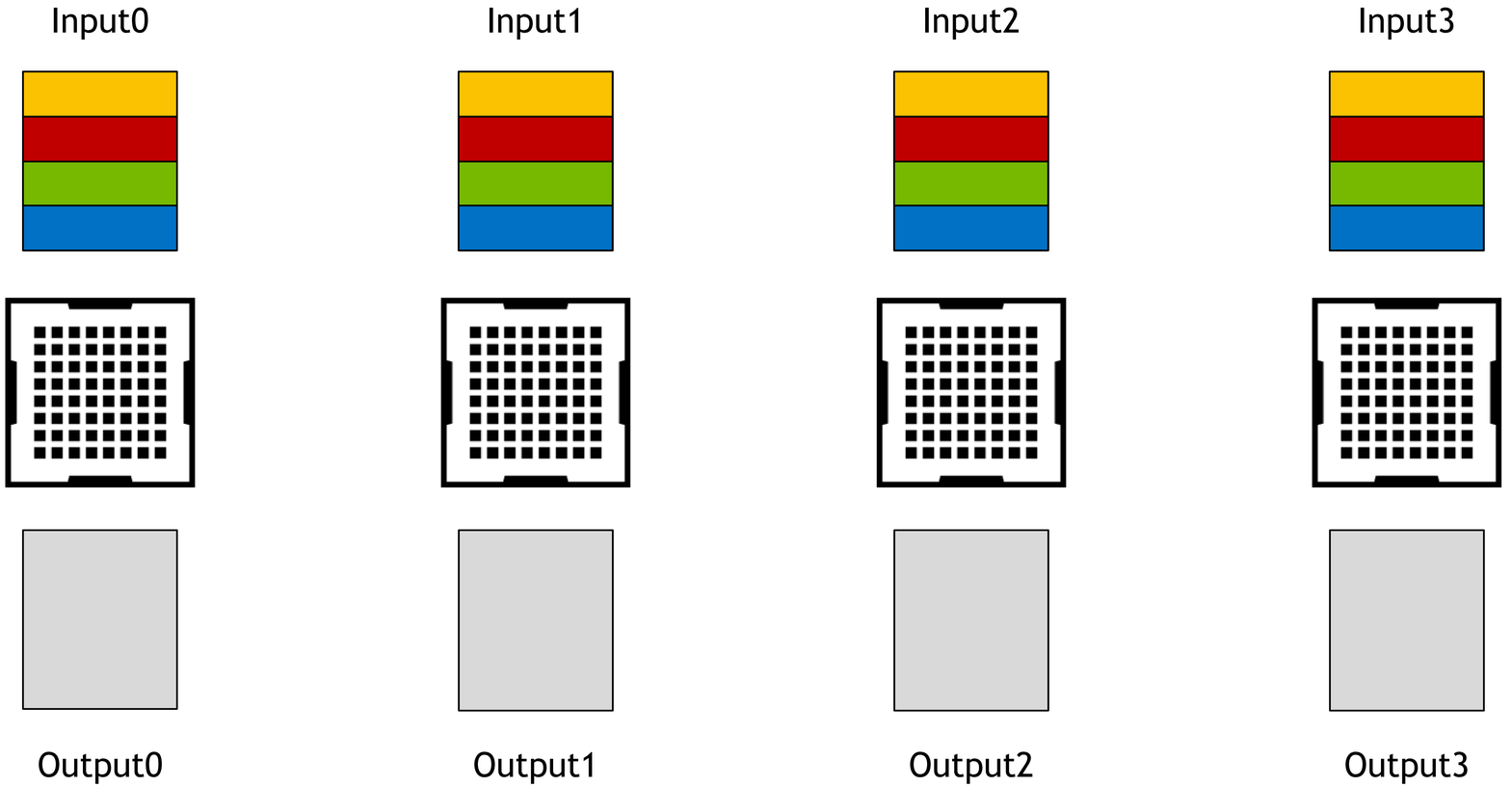


TREE Algorithm

For allreduce

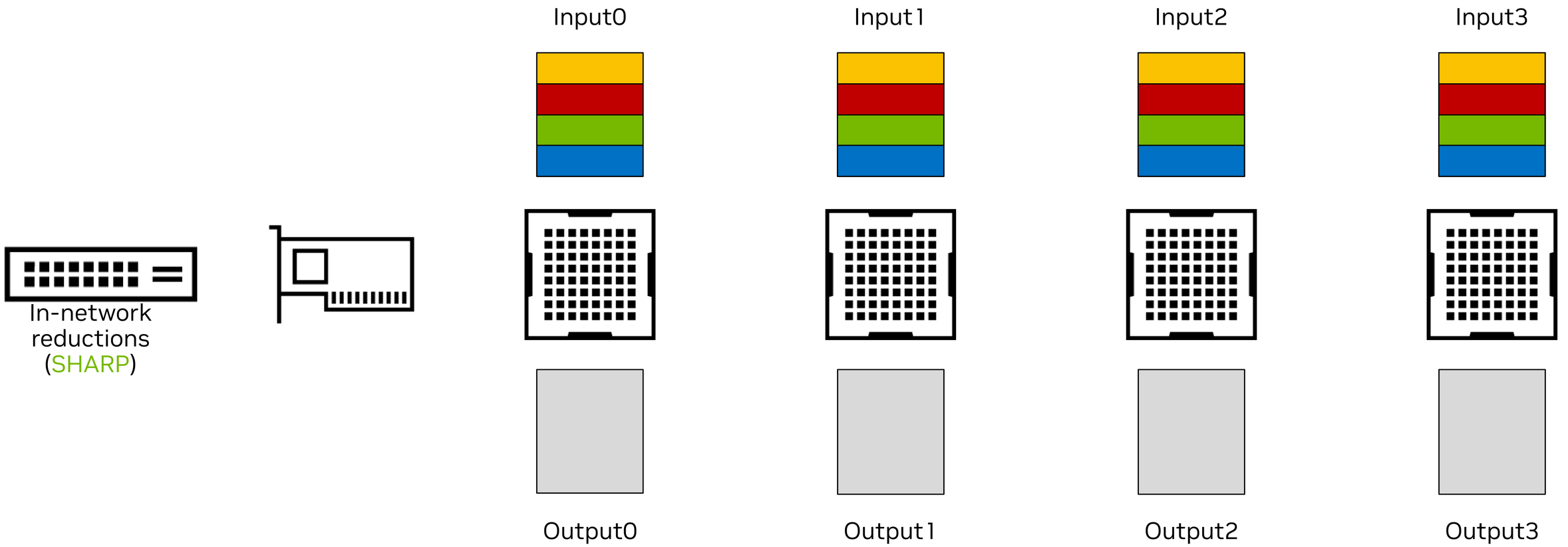






Collnet Algorithm

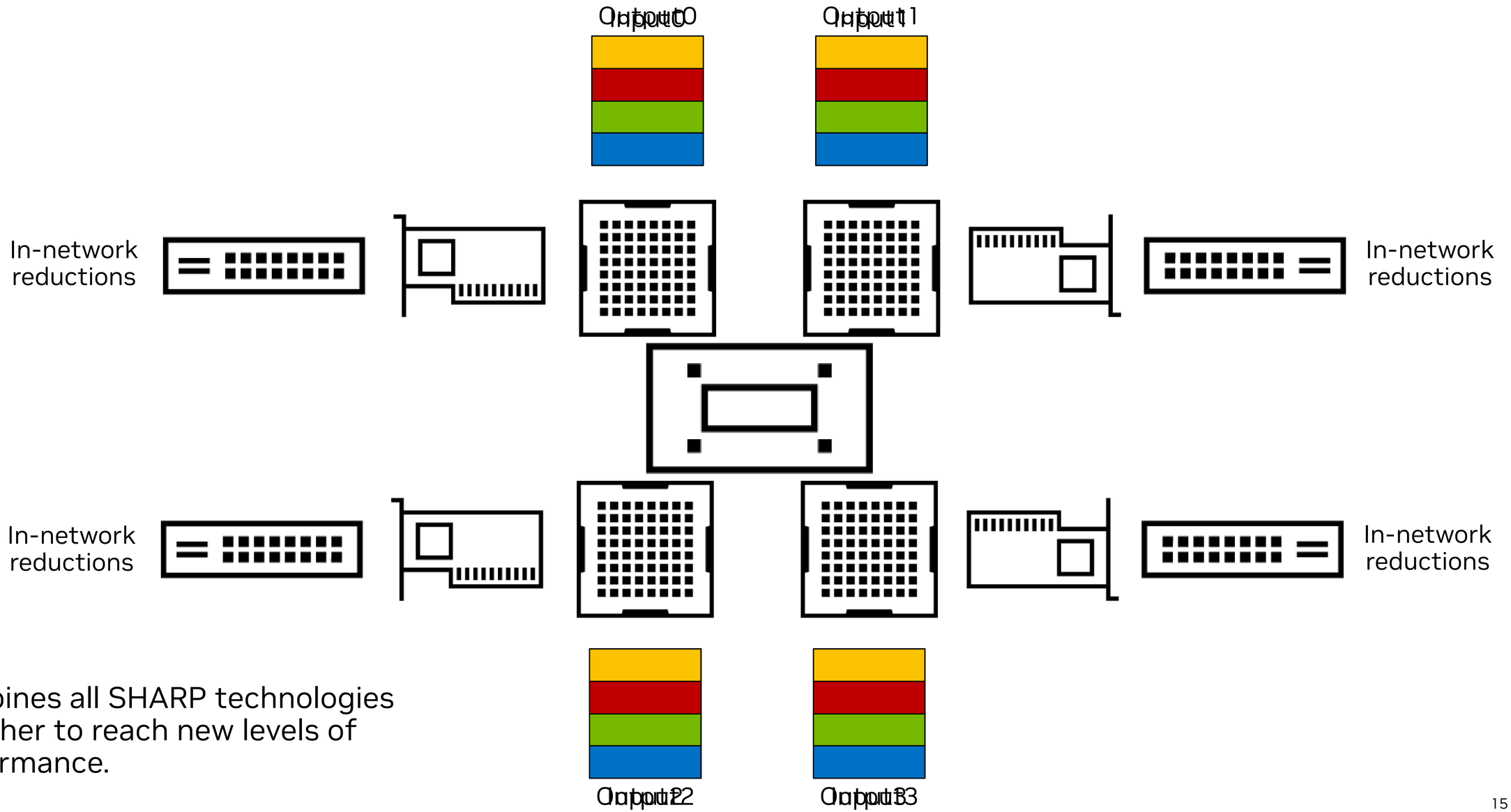
Chain version



Traffic on the network is largely reduced, improving peak bandwidth if the network was the bottleneck.
At-scale latency is nearly constant so performance is stable at scale.

H100 SHARP

NVLink, NVSwitch and IB



Combines all SHARP technologies together to reach new levels of performance.

NCCL PerfTests

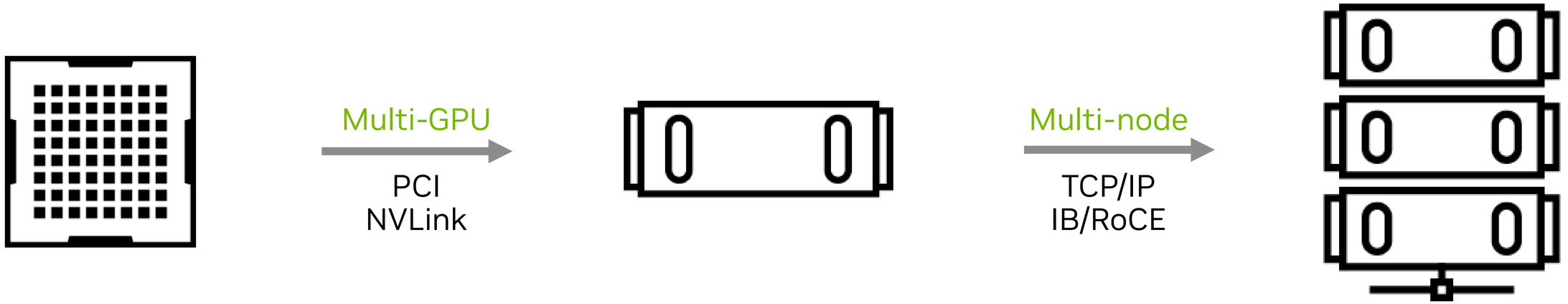
https://github.com/NVIDIA/nccl-tests

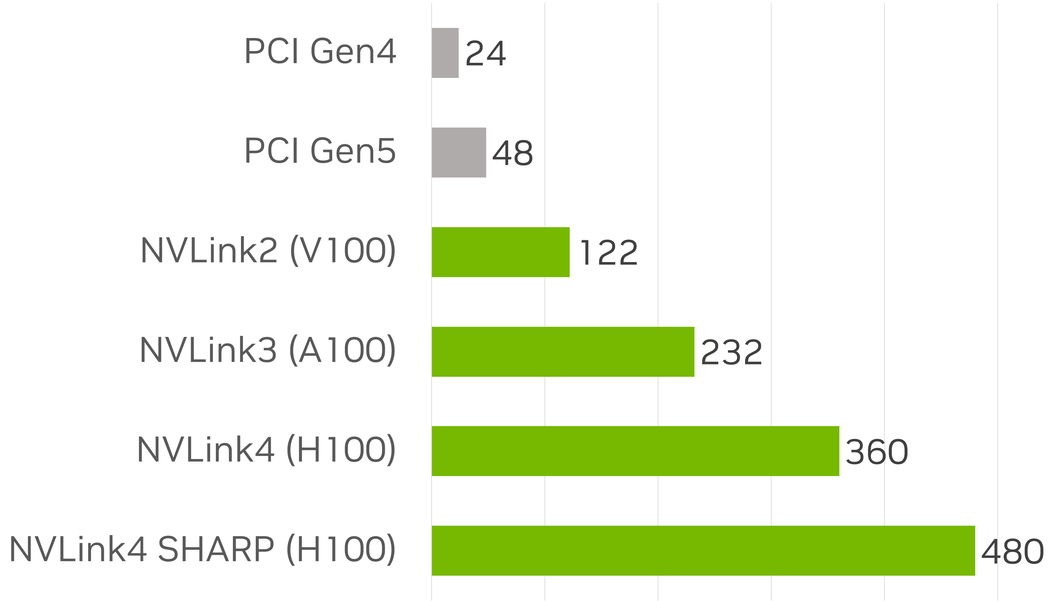
- How to measure performance of Collectives, e.g. Allreduce?
- Run on 8 GPUs (`-g 8`), scanning from 8 Bytes to 128 MBytes :

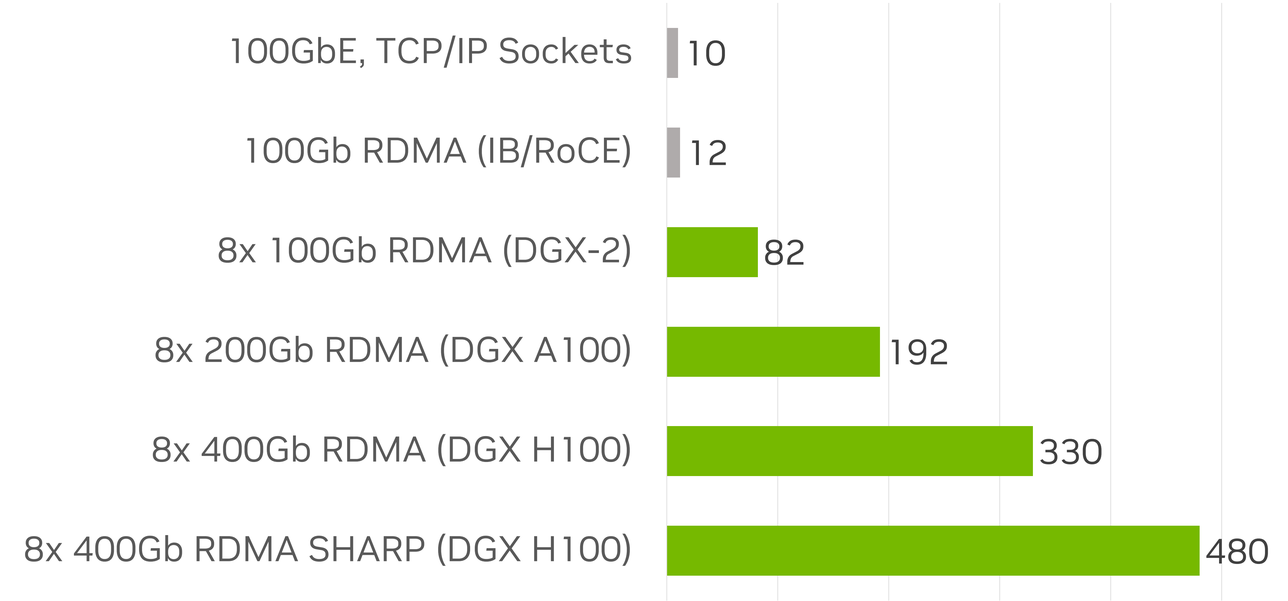
```
$ ./build/all_reduce_perf -b 8 -e 128M -f 2 -g 8
```

- Algorithm bandwith, "algbw"
  - Commonly used: `algbw = Size / Time`
- How to interpret or compare between different systems, interconnects, implementations?
  - Flawed for Collectives - fine for point-to-point where it can be measured against hardware limits (line rate)
- Bus bandwith, "busbw"
  - Correction factor for "algbw", depends on Collective operation and number of ranks n
  - For Allreduce, `busbw = algbw * 2*(n-1)/n`
  - For point-to-point, `algbw = busbw`
- See https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md for full details

Collective Communication Bandwidth





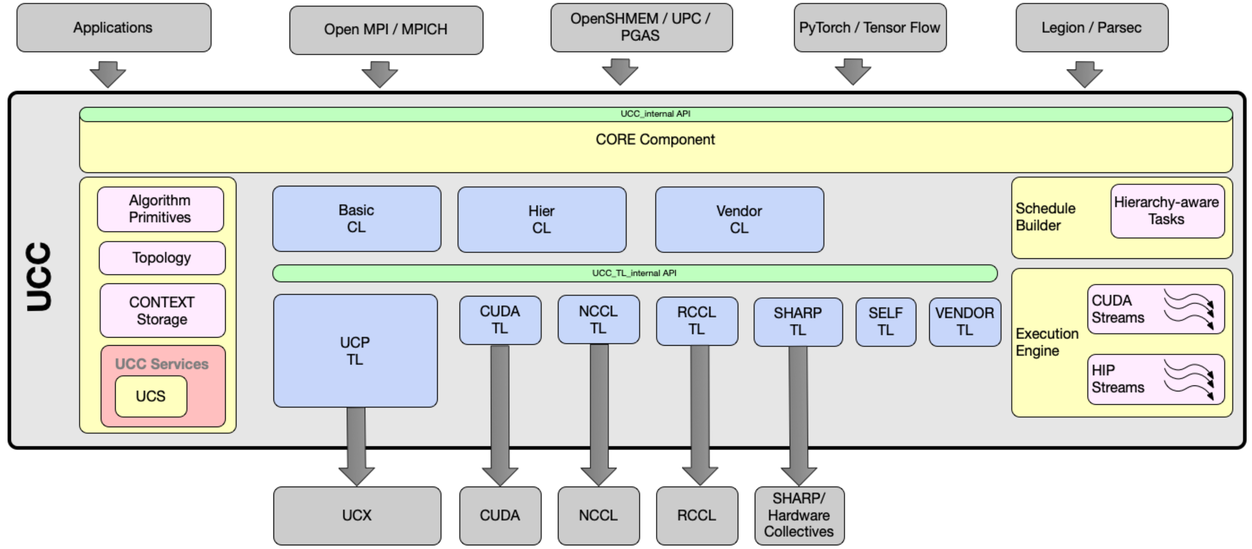


NCCL Tests Allreduce Bus Bandwidth in GB/s

Using NCCL below MPI

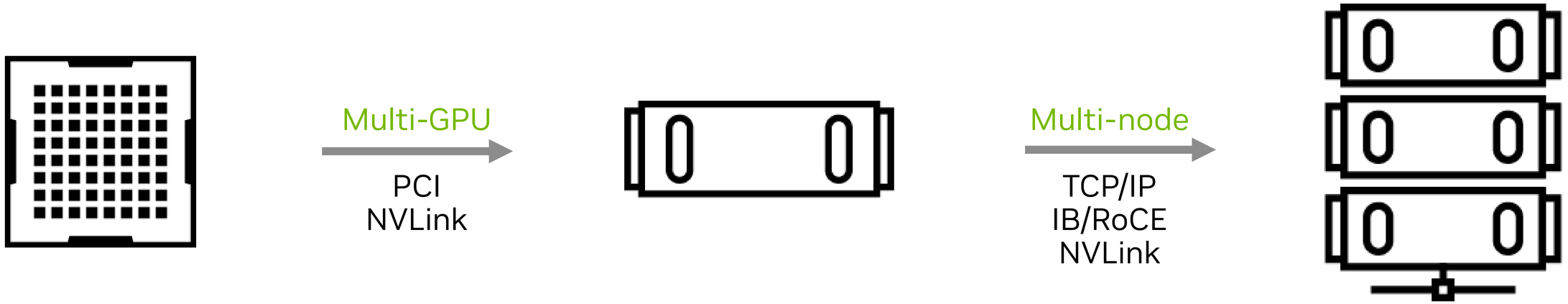
Unified Collective Communication (UCC)

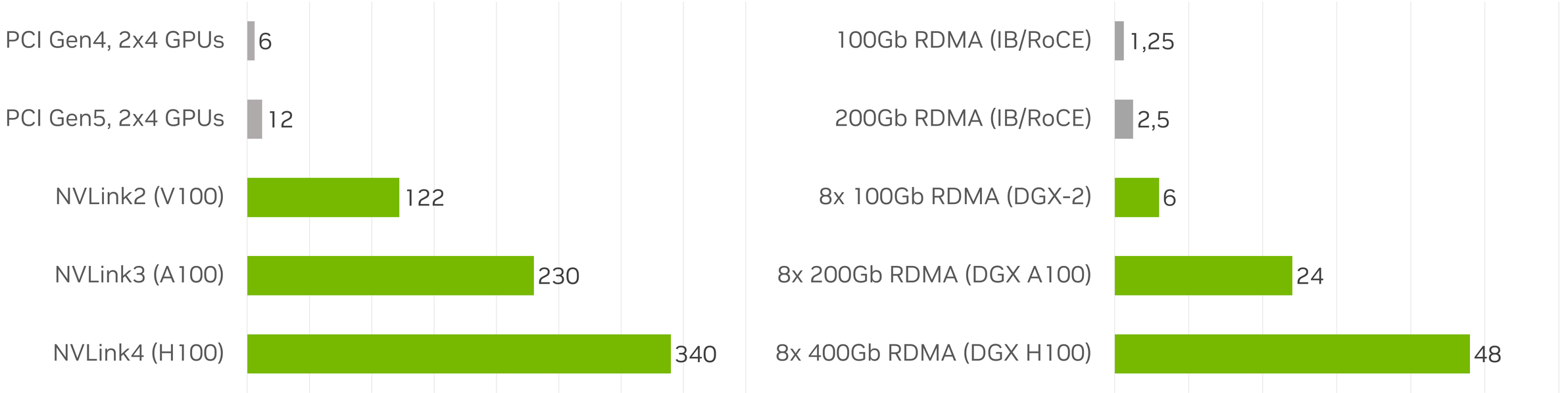
- As UCX chooses transport, let UCC choose collective backend
- Can use NCCL, SHARP, … under the hood
- Additional compile-time dependency
  - OpenMPI + UCX + UCC
  - Ask your sysadmin :-)
- Since user code remains unchanged: No explicit stream-awareness



- https://github.com/openucx/ucc#open-mpi-and-ucc-collectives
- https://github.com/openucx/ucc
- https://github.com/openucx/ucc/blob/master/docs/user_guide.md

Point-to-point Communication Bandwidth





NCCL Tests Alltoall Bus Bandwidth in GB/s

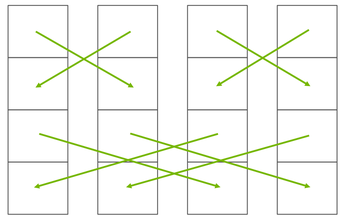
Point-to-point communication

ncclSend/ncclRecv

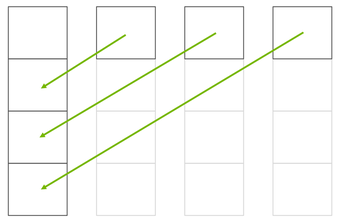
```
ncclResult_t ncclSend(buff, count, type, comm, stream);

ncclResult_t ncclRecv(buff, count, type, comm, stream);
```

Combined with ncclGroupStart/ncclGroupEnd to form sendrecv, gather[v], scatter[v], alltoall[v,w], neighbor collectives, …

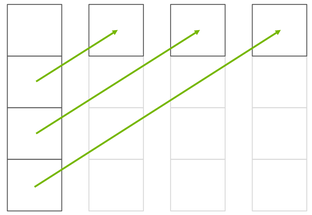
sendrecv



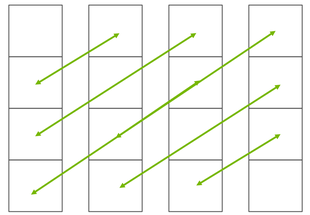
gather



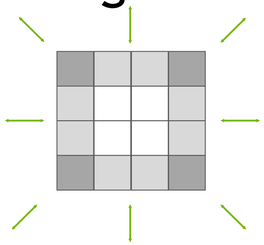
scatter



alltoall



neighbor



Using NCCL for "Sendrecv"

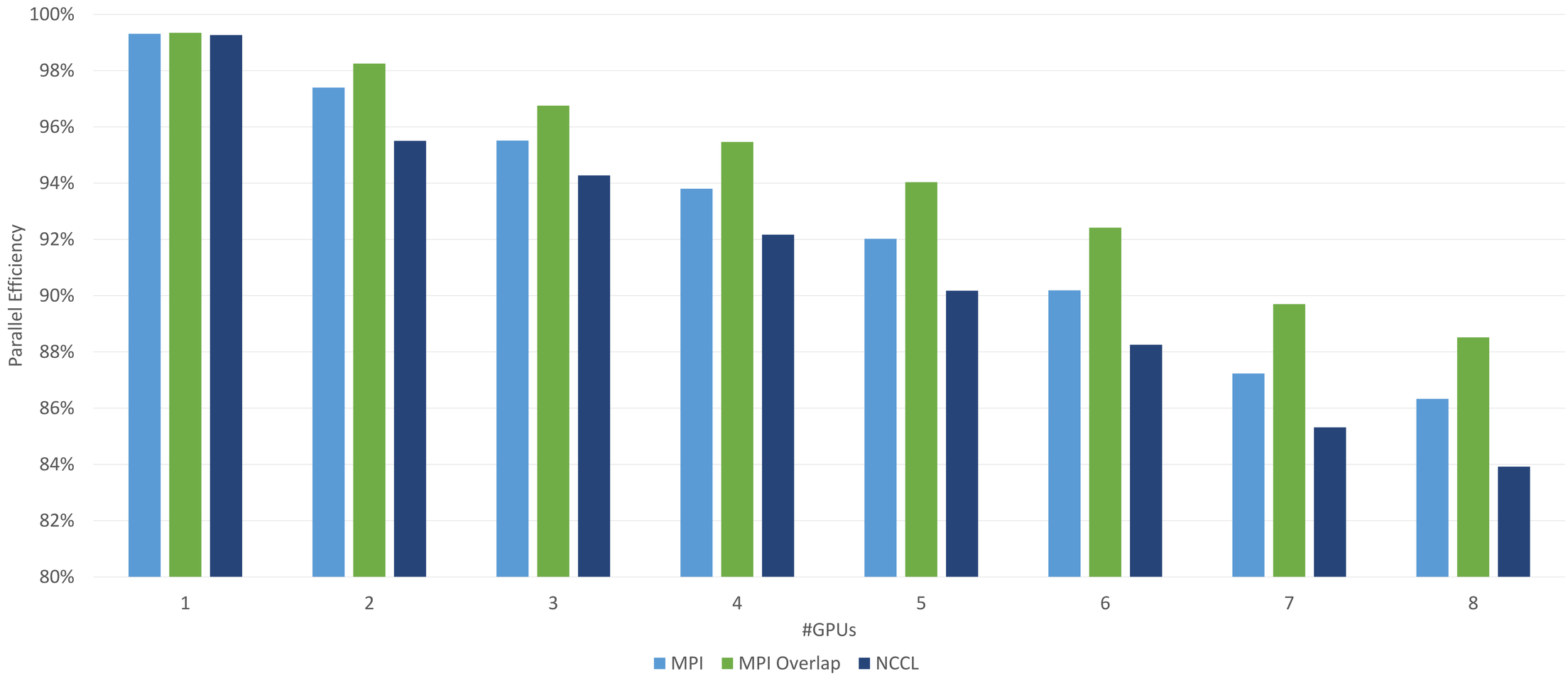
Jacobi mini-app: Boundary exchange

```
#ifdef USE_DOUBLE
#define NCCL_REAL_TYPE ncclDouble
#else
#define NCCL_REAL_TYPE ncclFloat
#endif
…
ncclGroupStart();
  ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, stream);
  ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, stream);
  ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, stream);
  ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, stream);
ncclGroupEnd();
```

Message ordering is important if messages are **not** unique, e.g. here if `top == bottom` (#GPU <= 2)

Multi GPU Jacobi Parallel Efficiency - NCCL

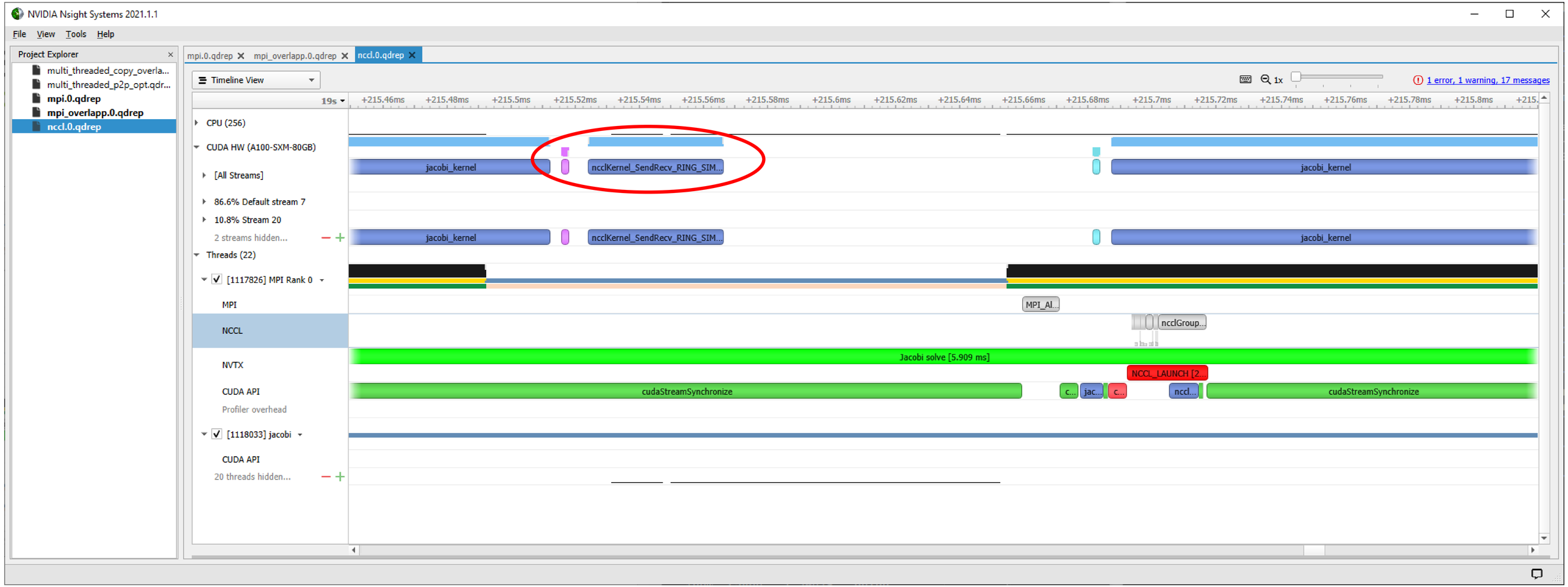
DGX A100 – 20480 x 20480, 1000 iterations



Benchmarksetup: DGX A100, Ubuntu 20.04.1 LTS, GCC 9.3.0, CUDA Driver 460.58, CUDA 11.2, UCX 1.11.0, OpenMPI 4.1.1rc1, NCCL 2.8.4, NVSHMEM 2.1, GPUs@1410Mhz AC, Reported Runtime is the minimum of 5 repetitions

Multi GPU Jacobi Nsight Systems Timeline - NCCL

NCCL 8 NVIDIA A100 80GB on DGX A100



Hands-on: Using NCCL for Inter-GPU Communication

Location: 02-NCCL_NVSHMEM/01-NCCL, follow instructions from tasks/Instructions.ipynb

- Use NCCL instead of MPI to implement a multi-GPU Jacobi solver. Work on TODOs in jacobi.cpp:
  - Include NCCL headers
  - Create a NCCL unique ID, and initialize it
  - Create a NCCL communicator and initilize it
  - Replace the MPI_Sendrecv calls with ncclRecv and ncclSend calls for the warmup stage
  - Replace MPI for the periodic boundary conditions with NCCL
  - Fix output message to indicate NCCL rather than MPI
  - Destroy NCCL communicator
- Compile with
  - make
- Submit your compiled application to the batch system with
  - make run
- Study the performance
  - make profile
- The environment variable NP can change the number of processes

NCCL

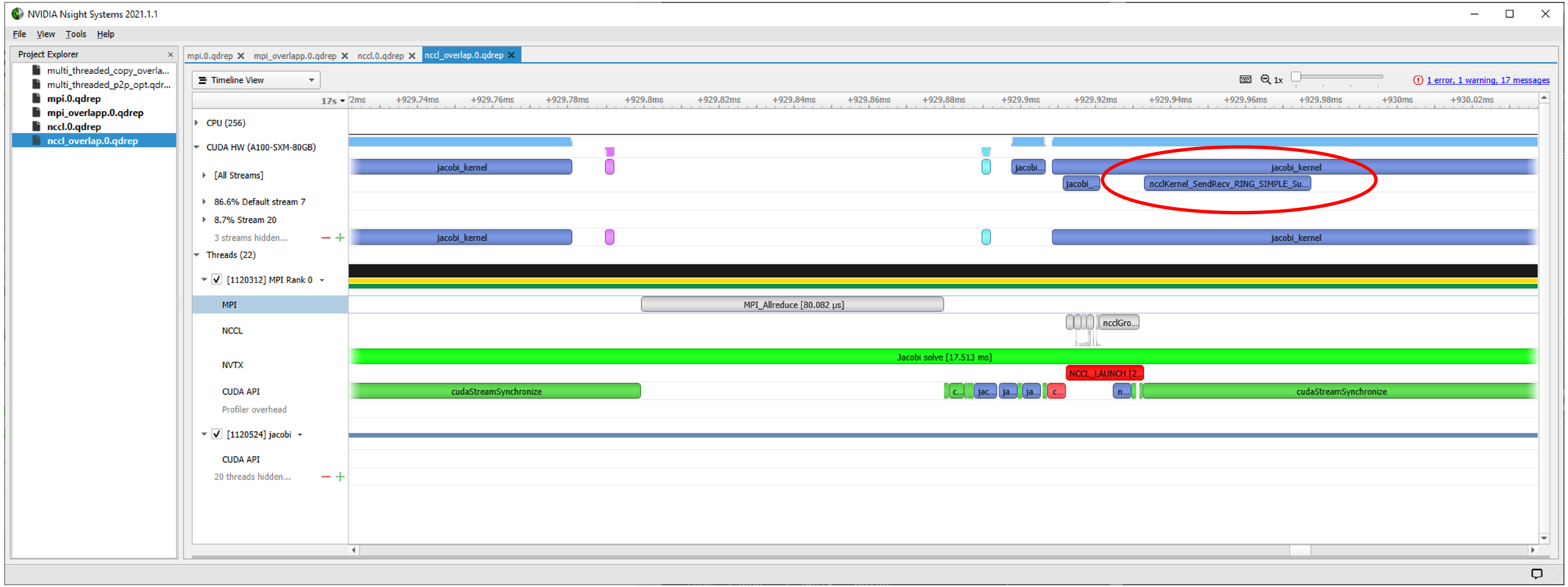
Communication Computation Overlap

- **comm_str** with high priority, for boundary layer and exchange
- **comm_str** with low/normal priority, for bulk compute kernel only

```
launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start,   (iy_start+1), nx, comm_str );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1),       iy_end, nx, comm_str );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, comp_str );
…
ncclGroupStart();
  ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, comm_str);
  ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, comm_str);
  ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, comm_str);
  ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, comm_str);
ncclGroupEnd();
```

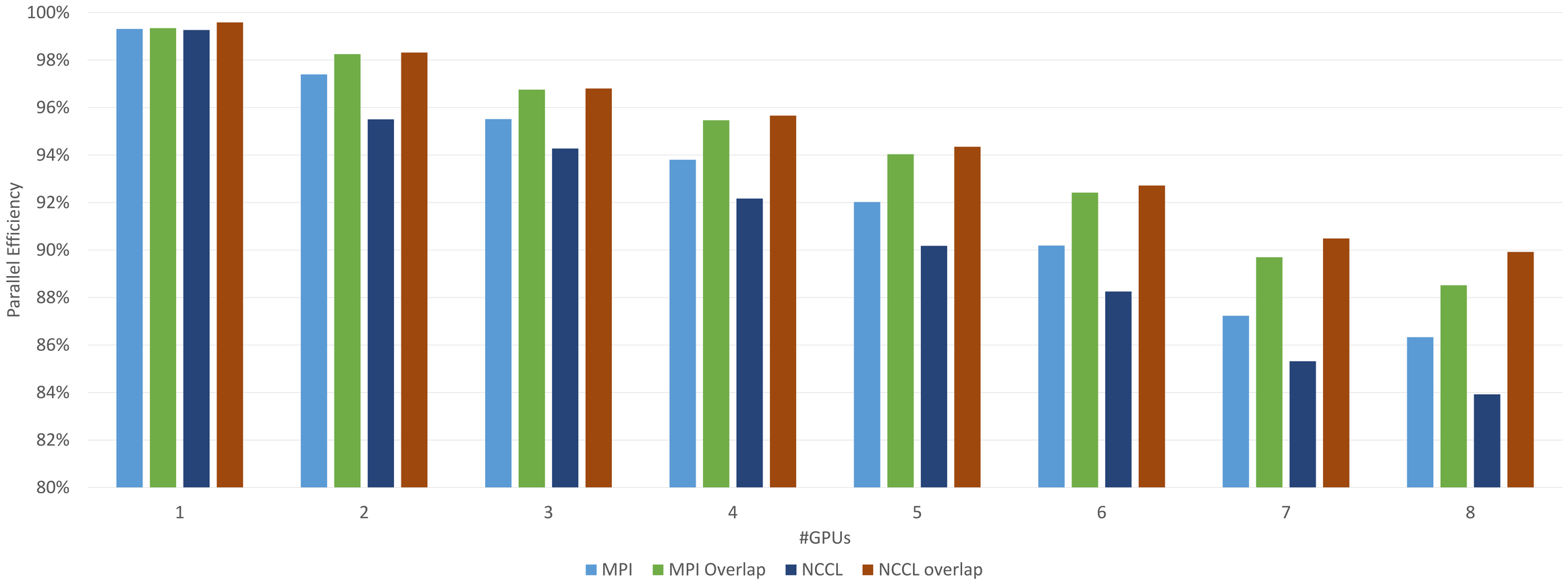
Multi GPU Jacobi Nsight Systems Timeline

NCCL Overlap 8 NVIDIA A100 80GB on DGX A100



Multi GPU Jacobi Parallel Efficiency

DGX A100 – 20480 x 20480, 1000 iterations

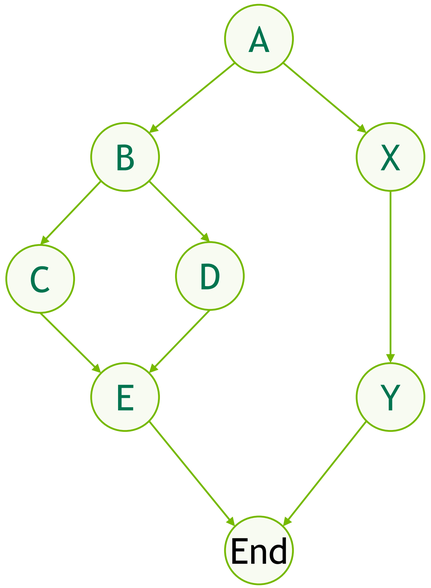


Benchmarksetup: DGX A100, Ubuntu 20.04.1 LTS, GCC 9.3.0, CUDA Driver 460.58, CUDA 11.2, UCX 1.11.0, OpenMPI 4.1.1rc1, NCCL 2.8.4, NVSHMEM 2.1, GPUs@1410Mhz AC, Reported Runtime is the minimum of 5 repetitions

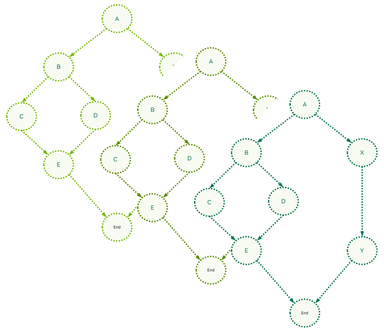
CUDA Graphs: Three-Stage Execution Model

Minimizes Execution Overheads – Pre-Initialize As Much As Possible

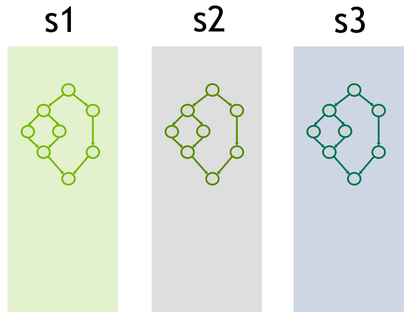
Define



Instantiate



Execute



Single Graph “Template”

Created in host code

or built up from libraries

Multiple “Executable Graphs”

Snapshot of templates

Sets up & initializes GPU execution structures (create once, run many times)

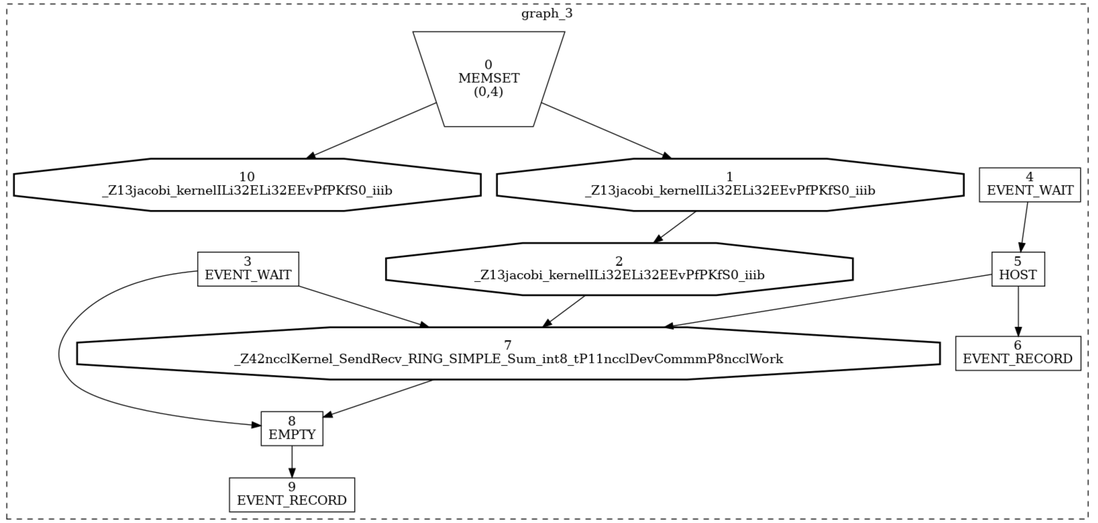
Executable Graphs Running in CUDA Streams

Concurrency in graph **is not** limited by stream

Using CUDA Graphs with NCCL

Graphs Can Be Generated Once Then Launched Repeatedly

```
while (l2_norm > tol && iter < iter_max) {

    cudaGraphLaunch(graph_calc_norm_exec[iter%2],
                    compute_stream);
    cudaStreamSynchronize(compute_stream);
    MPI_Allreduce(l2_norm_h, &l2_norm, 1,
                  MPI_REAL_TYPE, MPI_SUM,
                  MPI_COMM_WORLD);
    l2_norm = std::sqrt(l2_norm);

    if (!csv && 0 == rank && (iter % 100) == 0) {
        printf("%5d, %0.6f\n", iter, l2_norm);
    }
}
```



Generated with

```
cudaGraphDebugDotPrint(graphs[calculate_norm][0],
                       "jacobi_graph.dot",0)
```

and

```
dot -Tpng jacobi_graph.dot -o jacobi_grap.png
```

More details in hands-on exercise and corresponding lecture at
https://github.com/FZJ-JSC/tutorial-multi-gpu/tree/main/10-H_CUDA_Graphs_and_Device-initiated_Communication_with_NVSHMEM

Full NCCL API

```
// Initialization, finalize, abort and fault tolerance
ncclResult_t ncclGetUniqueId(ncclUniqueId* uniqueId);                                                         // Since 2.0
ncclResult_t ncclCommInitRank(ncclComm_t* comm, int nranks, ncclUniqueId commId, int rank);                   // Since 2.0
ncclResult_t ncclCommInitRankConfig(ncclComm_t* comm, int nranks, ncclUniqueId commId, int rank, ncclConfig_t* config); // Since 2.14
ncclResult_t ncclCommFinalize(ncclComm_t comm);                                                               // Since 2.14
ncclResult_t ncclCommDestroy(ncclComm_t comm);                                                                // Since 2.0
ncclResult_t ncclCommAbort(ncclComm_t comm);                                                                  // Since 2.4
ncclResult_t ncclCommGetAsyncError(ncclComm_t comm, ncclResult_t *asyncError);                                // Since 2.4
ncclResult_t ncclCommSplit(ncclComm_t comm, int color, int key, ncclComm_t *newcomm, ncclConfig_t* config);   // Planned for 2.18

// Collective communication - since 2.0
ncclResult_t ncclReduce       (const void* sbuff, void* rbuff, size_t cnt,  ncclDataType_t dtype, ncclRedOp_t op, int root, ncclComm_t comm, cudaStream_t s);
ncclResult_t ncclBroadcast    (const void* sbuff, void* rbuff, size_t cnt,  ncclDataType_t dtype,                 int root, ncclComm_t comm, cudaStream_t s);
ncclResult_t ncclAllReduce    (const void* sbuff, void* rbuff, size_t cnt,  ncclDataType_t dtype, ncclRedOp_t op,           ncclComm_t comm, cudaStream_t s);
ncclResult_t ncclReduceScatter(const void* sbuff, void* rbuff, size_t rcnt, ncclDataType_t dtype, ncclRedOp_t op,           ncclComm_t comm, cudaStream_t s);
ncclResult_t ncclAllGather    (const void* sbuff, void* rbuff, size_t scnt, ncclDataType_t dtype,                           ncclComm_t comm, cudaStream_t s);

// Point to point communication - since 2.7
ncclResult_t ncclSend         (const void* sbuff,              size_t cnt,  ncclDataType_t dtype,                 int peer, ncclComm_t comm, cudaStream_t s);
ncclResult_t ncclRecv         (                   void* rbuff, size_t cnt,  ncclDataType_t dtype,                 int peer, ncclComm_t comm, cudaStream_t s);

// Fuse operations together - since 2.0
ncclResult_t ncclGroupStart();
ncclResult_t ncclGroupEnd();

// Custom scaling factor - since 2.11
ncclResult_t ncclRedOpCreatePreMulSum(ncclRedOp_t *op, void *scalar, ncclDataType_t datatype, ncclScalarResidence_t residence, ncclComm_t comm);
ncclResult_t ncclRedOpDestroy(ncclRedOp_t op, ncclComm_t comm);

// Misc
ncclResult_t ncclGetVersion(int *version);                                                                    // Since 2.0
const char*  ncclGetLastError(ncclComm_t comm);                                                               // Since 2.13
ncclResult_t ncclCommCount(const ncclComm_t comm, int* count);                                                // Since 2.0
ncclResult_t ncclCommCuDevice(const ncclComm_t comm, int* device);                                            // Since 2.0
ncclResult_t ncclCommUserRank(const ncclComm_t comm, int* rank);                                              // Since 2.0
```

CPU-INITIATED COMMUICATION

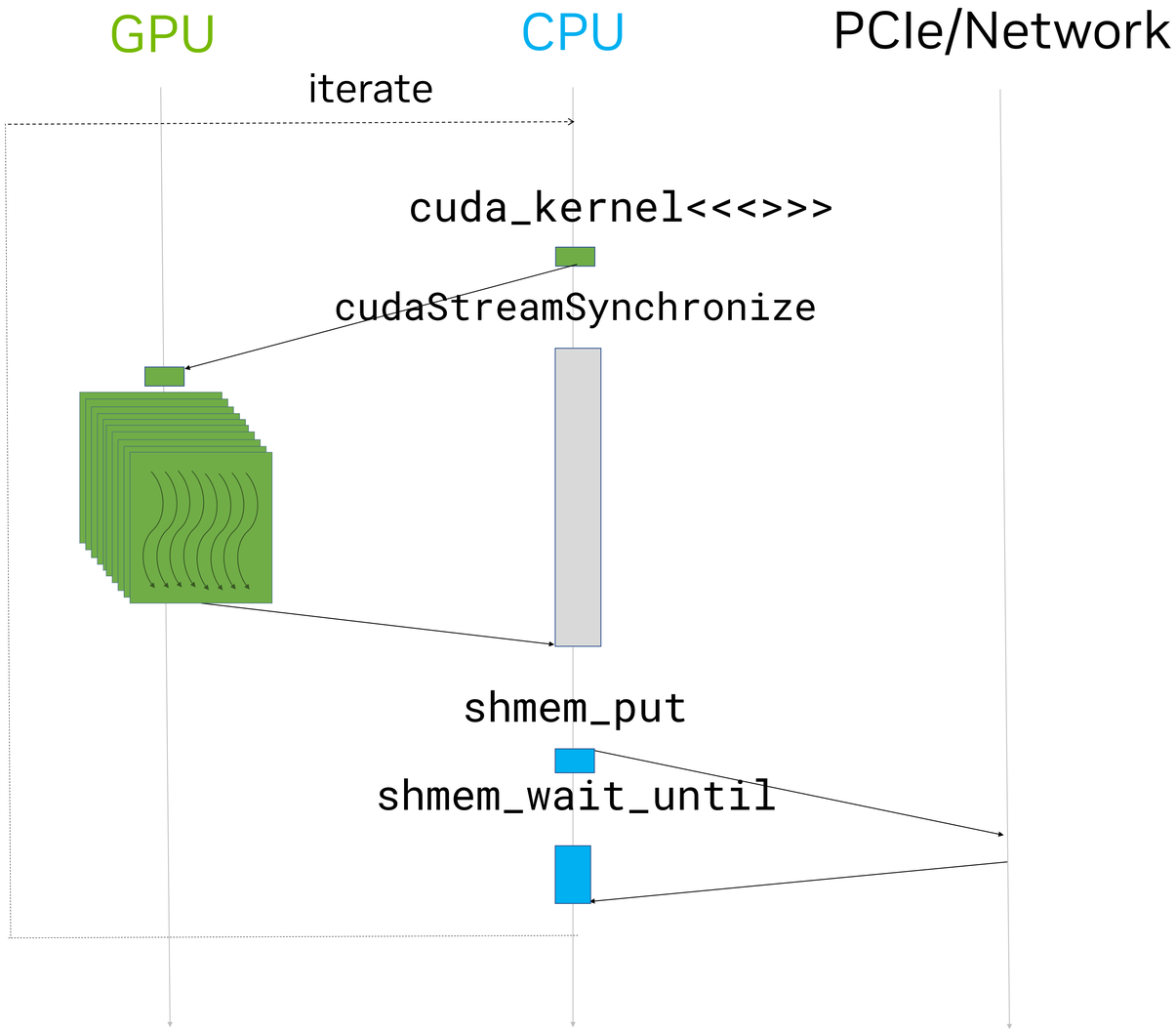
- Compute on GPU
- Communication from CPU

Synchronization at boundaries

Commonly used model, but –

- Offload latencies in critical path
- Communication is not overlapped

Hiding increases code complexity,
Not hiding limits strong scaling



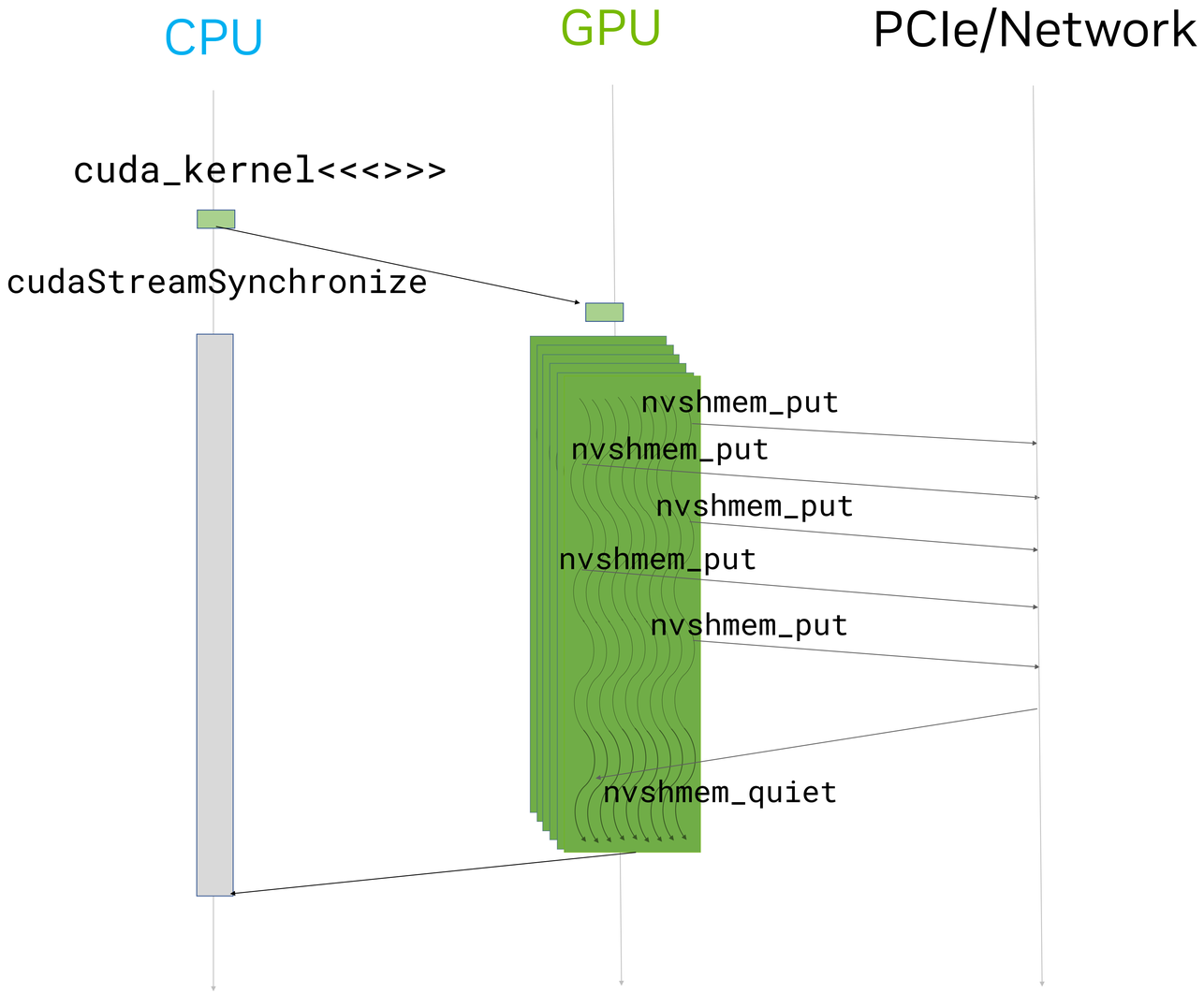
GPU-INITIATED COMMUNICATION

- Compute on GPU
- Communication from GPU

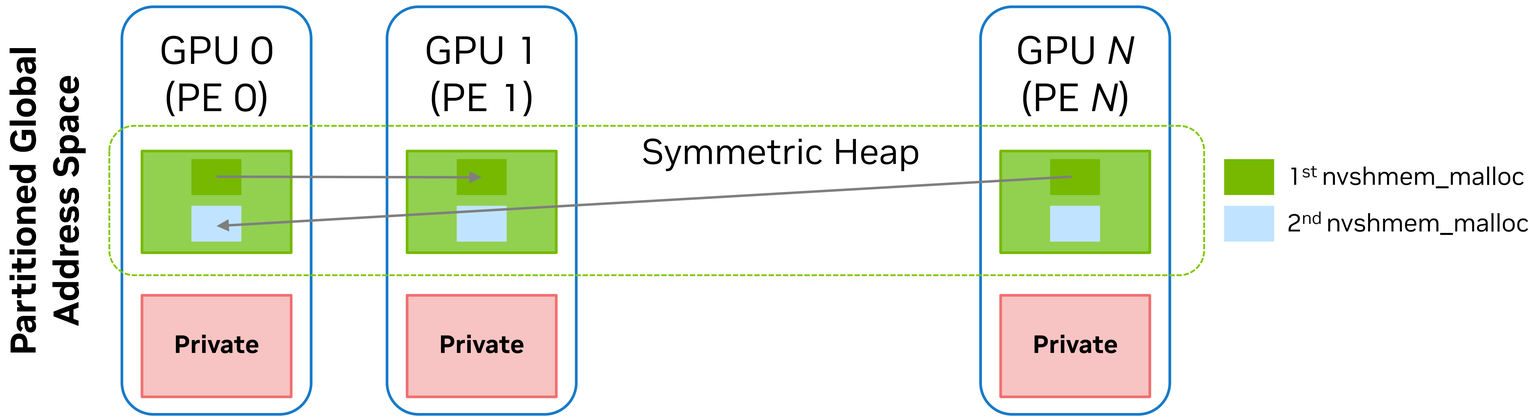
Benefits -

- Eliminates offload latencies
- Compute and communication overlap
- Latencies hidden by threading
- Easier to express algorithms with inline communication

Improving performance while making it easier to program



NVSHMEM



- **Implementation of OpenSHMEM, a Partitioned Global Address Space (PGAS) library**

- Symmetric objects are allocated collectively with the same size on every PE
  Symmetric memory: `nvshmem_malloc(…);` Private memory: `cudaMalloc(…);`

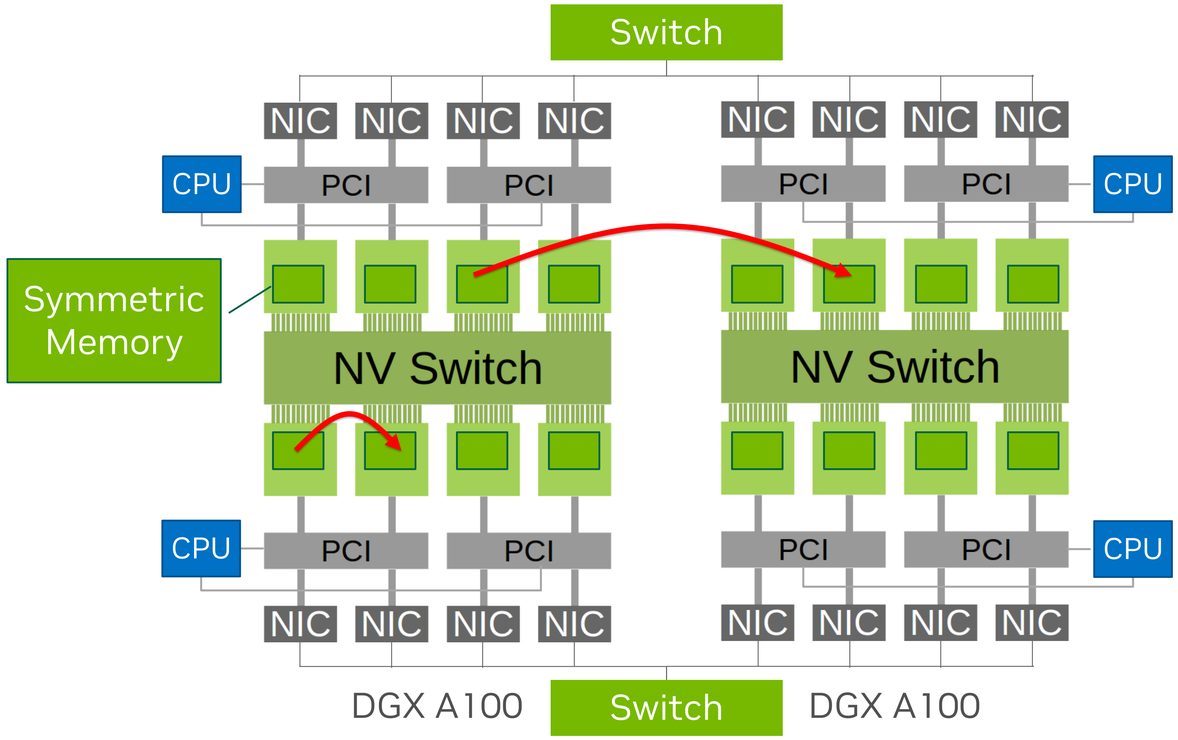
- CPU (blocking and stream-ordered) and CUDA Kernel interfaces
  Read: `nvshmem_get(…);` Write: `nvshmem_put(…);` Atomic: `nvshmem_atomic_add(…);`
  Flush writes: `nvshmem_quiet();` Order writes: `nvshmem_fence();`
  Synchronize: `nvshmem_barrier();` Poll: `nvshmem_wait_until(…);`

- Interoperable with MPI

NVSHMEM

OpenSHMEM, Adapted for Best Performance on NVIDIA GPU Clusters

- Memory of multiple GPUs in cluster
  - Aggregate into distributed global address space
- Data access via put, get, atomic APIs
- Collective communication APIs
- Communication integrated with CUDA execution model
  1. GPU kernel-initiated operations
  2. Operations on CUDA streams/graphs
  3. CPU initiated operations
- Interop with CPU OpenSHMEM or MPI for host memory communication

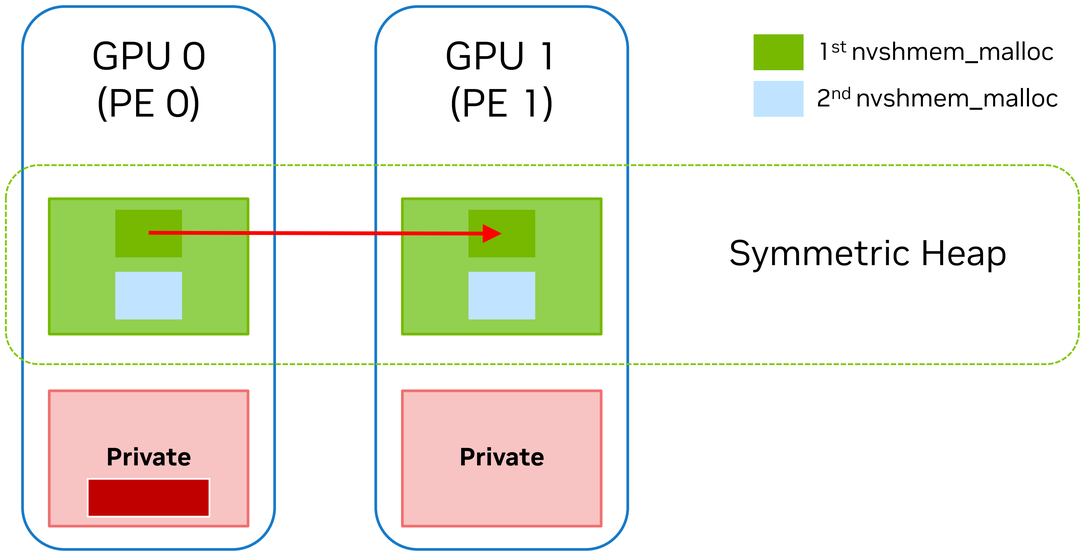


NVSHMEM - Skeleton

```
#include <nvshmem.h>
#include <nvshmemx.h>
int main(int argc, char *argv[]) {
    ...
    MPI_Comm mpi_comm;
    nvshmemx_init_attr_t attr;
    mpi_comm = MPI_COMM_WORLD;
    attr.mpi_comm = &mpi_comm;
    nvshmemx_init_attr (NVSHMEMX_INIT_WITH_MPI_COMM, &attr);
    int npes = nvshmem_n_pes();
    int mype = nvshmem_my_pe();
    ...
    return 0;
}
```

NVSHMEM Host API - Put

Copies *N* data elements of type *T* from symmetric heap objects *src* to *dest* on PE *pe* (can be different allocations)

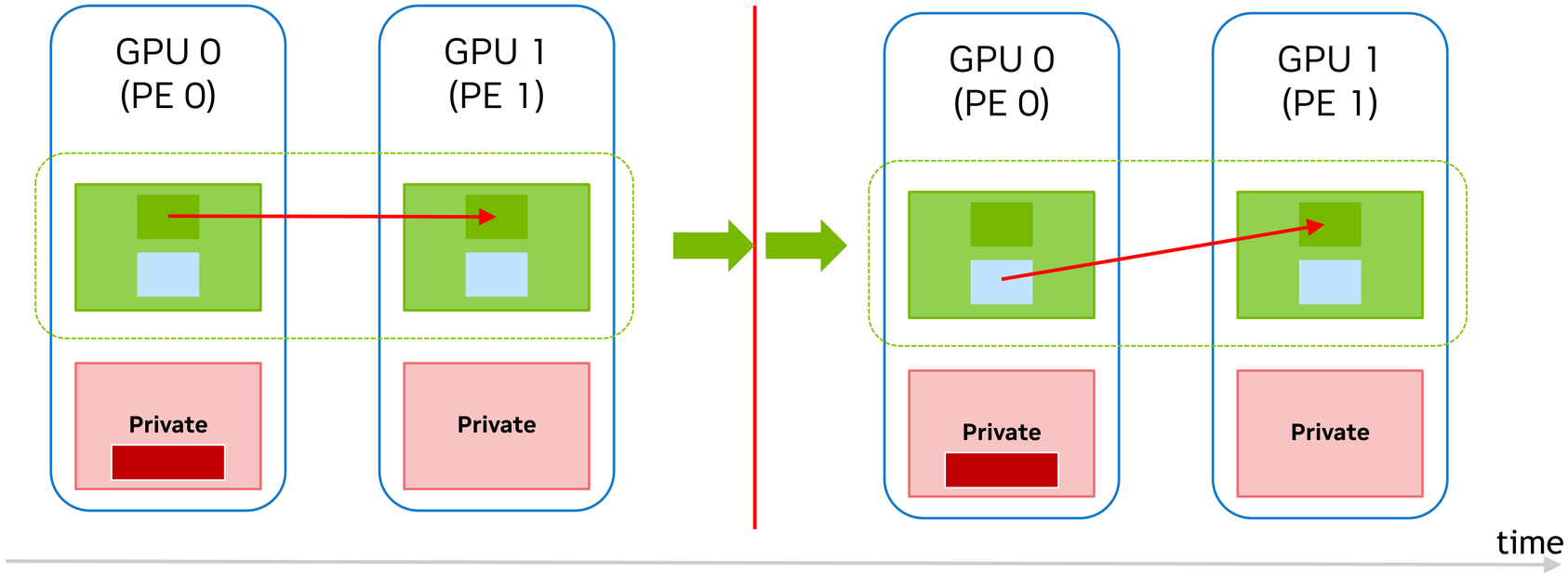


Extension from standard SHMEM

```
void nvshmem_<T>_put           (T* dest, const T* src, size_t N, int pe);
void nvshmemx_<T>_put_on_stream(T* dest, const T* src, size_t N, int pe, cudaStream_t stream);
```

NVSHMEM Host API - Barrier

Synchronize all PEs at once, ensures completion of communication/memory operations




```
void nvshmem_barrier                    ();
void nvshmemx_barrier_all_on_stream(cudaStream_t stream);
```

Jacobi with NVSHMEM

Host-side API

Size given to nvshmem_malloc must be same for all PEs. Else: **U**ndefined **B**ehavior!

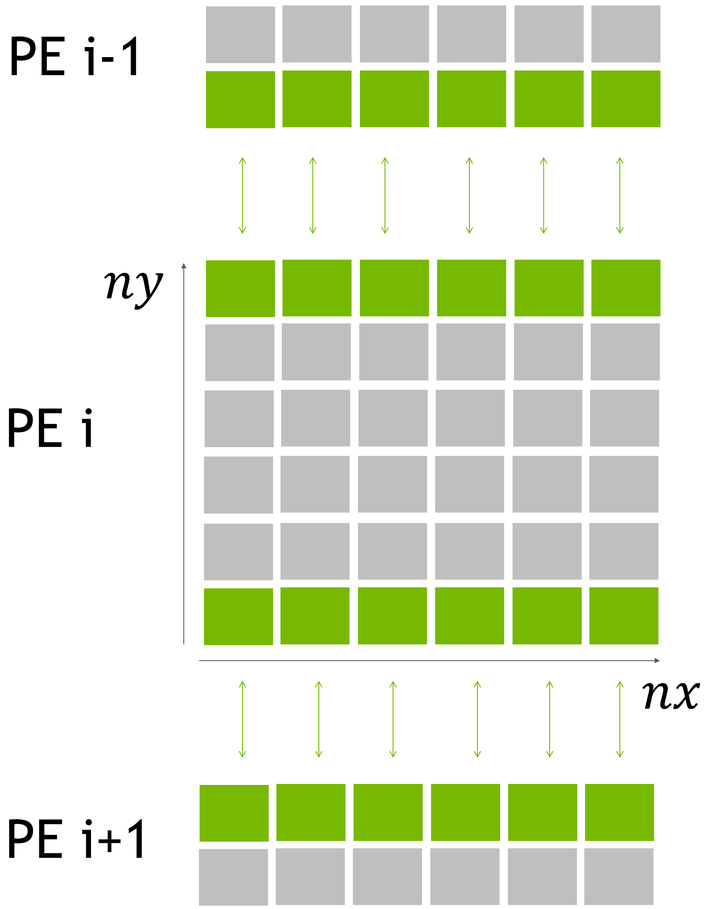
```
real* a     = (real*) nvshmem_malloc(nx * (chunk_size + 2) * sizeof(real));
real* a_new = (real*) nvshmem_malloc(nx * (chunk_size + 2) * sizeof(real));

launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start,     iy_start + 1, nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_end   - 1, iy_end      , nx, push_stream);
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start + 1, iy_end   - 1, nx, compute_stream);

nvshmemx_float_put_on_stream(&a_new[btm_halo], &a_new[my_last_row],  nx, btm, push_stream);

nvshmemx_float_put_on_stream(&a_new[top_halo], &a_new[my_first_row], nx, top, push_stream);

nvshmemx_barrier_all_on_stream(push_stream);
```

Hands-on: Host-initiated Communication with NVSHMEM

Location: 02-NCCL_NVSHMEM/02-NVSHMEM_Host, follow instructions from tasks/Instructions.ipynb

- Use NVSHMEM host API instead of MPI to implement a multi-GPU Jacobi solver. Work on TODOs in jacobi.cu:
  - Include NVSHMEM headers.
  - Initialize NVSHMEM using MPI_COMM_WORLD.
  - Allocate work arrays a and a_new from the NVSHMEM symmetric heap. Take care of passing in a consistent size!
  - Calculate halo/boundary row index of top and bottom neighbors.
  - Add necessary inter PE synchronization.
  - Replace MPI periodic boundary conditions with nvshmemx_float_put_on_stream to directly push values needed by top and bottom neighbors.
  - Deallocate memory from the NVSHMEM symetric heap.
  - Finalize NVSHMEM before existing the application
- Compile with
  - make
- Submit your compiled application to the batch system with
  - make run
- Study the performance
  - make profile
- The environment variable NP can change the number of processes

THREAD-LEVEL COMMUNICATION

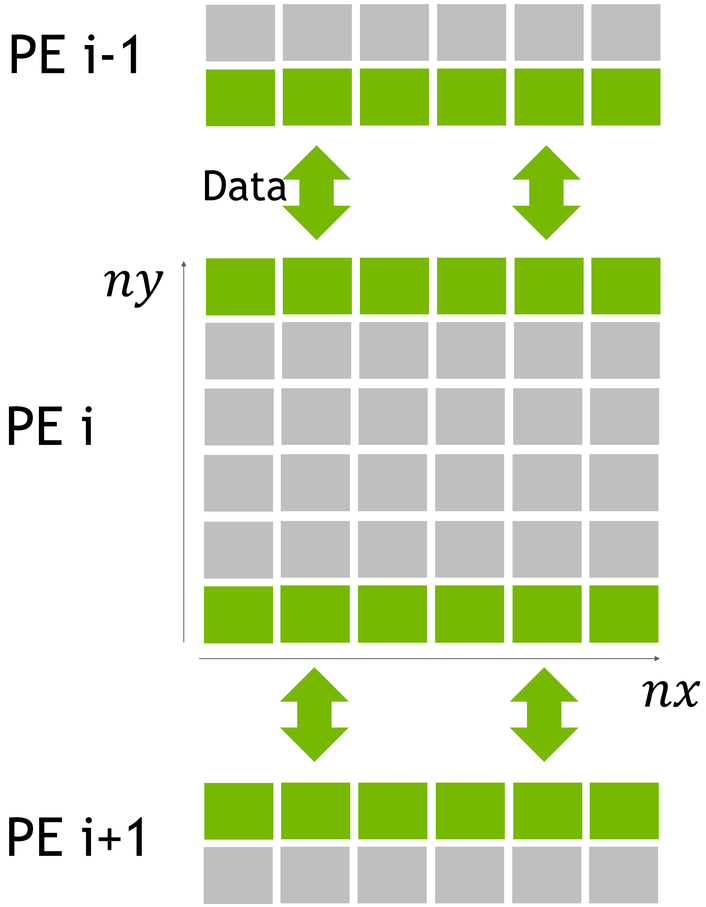


- Allows fine grained communication and overlap
- Efficient mapping to NVLink fabric on DGX systems

```
__global__ void stencil_single_step(float *u, float *v, …) {
   int ix = get_ix(blockIdx, blockDim, threadIdx);
   int iy = get_iy(blockIdx, blockDim, threadIdx);

   compute(u, v, ix, iy);
   // Thread-level data communication API
   if (iy == 1)
      nvshmem_float_p(u+(ny+1)*nx+ix, u[nx+ix], top_pe);
   if (iy == ny)
      nvshmem_float_p(u+ix, u[ny*nx+ix], bottom_pe);
}
```

```
for (int iter = 0; iter < N; iter++) {
   swap(u, v);
   stencil_single_step<<<..., stream>>>(u, v, …);
   nvshmemx_barrier_all_on_stream(stream);
}
```

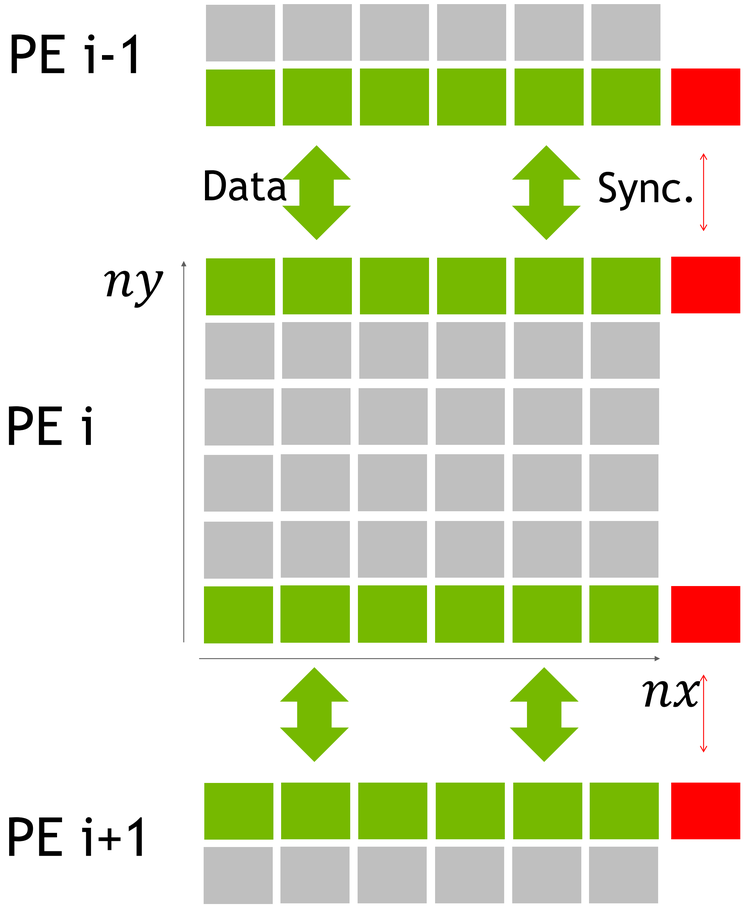
THREAD-GROUP COMMUNICATION



- ❑ NVSHMEM operations can be issued by all threads in a block/warp
- ❑ More efficient data transfers over networks like IB
- ❑ Still allows inter-warp/inter-block overlap

```
__global__ void stencil_single_step(float *u, float *v, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);

  compute(u, v, ix, iy);

  // Thread block-level communication API
  int boffset = get_block_offet(blockIdx,blockDim);
  if (blockIdx.y == 0)
    nvshmemx_float_put_nbi_block(u+(ny+1)*nx+boffset, u+nx+boffset, blockDim.x, top_pe);
  if (blockIdx.y == (blockDim.y-1))
    nvshmemx_float_put_nbi_block(u+boffset, u+ny*nx+boffset, blockDim.x, bottom_pe);
}

for (int iter = 0; iter < N; iter++) {
   swap(u, v);
   stencil_single_step<<<..., stream>>>(u, v, …);
   nvshmem_barrier_all_on_stream(stream);
}
```

IN-KERNEL SYNCHRONIZATION



Point-to-point synchronization across PEs within a kernel
Offload larger portion of application to the same CUDA kernel

```
__global__
void stencil_multi_step(float *u, float *v, int N, int *sync, …) {
  int ix = get_ix(blockIdx, blockDim, threadIdx);
  int iy = get_iy(blockIdx, blockDim, threadIdx);

  for (int iter = 0; iter < N; iter++) {
    swap(u, v);
    compute(u, v, ix, iy);
    // Thread block-level data exchange (assume even/odd iter buffering)
    int boffset = get_block_offet(blockIdx,blockDim);
    if (blockIdx.y == 0)
      nvshmemx_float_put_nbi_block(u + (ny+1)*nx+boffset, u+nx+boffset, blockDim.x, top_pe);
    if (blockIdx.y == (blockDim.y-1))
      nvshmemx_float_put_nbi_block(u + boffset, u+ny*nx+boffset,  blockDim.x, bottom_pe);
    if (blockIdx.y == 0 || blockIdx.y == (blockDim.y-1)) {
        __syncthreads();
        nvshmem_fence();
        if (threadIdx.x == 0 && threadIdx.y == 0) {
          nvshmem_atomic_inc(sync, top_pe);
          nvshmem_atomic_inc(sync, bottom_pe);
        }
    }
    nvshmem_wait_until(sync, NVSHMEM_CMP_GT, 2*iter*gridDim.x);
  }
}
```

Be aware of synchronization costs. Best strategy is application dependent!

NVSHMEM – Host Code

```
a = (float *) nvshmem_malloc(nx*(chunk_size+2)*sizeof(float));
a_new = (float *) nvshmem_malloc(nx*(chunk_size+2)*sizeof(float));
…
while ( l2_norm > tol && iter < iter_max ) {
    …
    jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(
        a_new, a, l2_norm_d, iy_start, iy_end, nx,
        top, iy_end_top, bottom, iy_start_bottom );
    nvshmemx_barrier_all_on_stream(compute_stream);
     …
}
nvshmem_barrier_all();
nvshmem_free(a);
nvshmem_free(a_new);
```

NVSHMEM - Kernel

```
__global__ void jacobi_kernel( ... ) {
    const int ix = bIdx.x*bDim.x+tIdx.x;
    const int iy = bIdx.y*bDim.y+tIdx.y + iy_start;
    float local_l2_norm = 0.0;
    if ( iy < iy_end && ix >= 1 && ix < (nx-1) ) {
        const float new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                                     + a[ (iy+1) * nx + ix ] + a[ (iy-1) * nx + ix ] );
        a_new[ iy * nx + ix ] = new_val;
        if ( iy_start == iy )
            nvshmem_float_p(a_new +    top_iy*nx + ix, new_val, top_pe);
        if ( iy_end == iy )
            nvshmem_float_p(a_new + bottom_iy*nx + ix, new_val, bottom_pe);
        float residue = new_val - a[ iy * nx + ix ];
    }
    atomicAdd( l2_norm, local_l2_norm );
}
```

Optimized for DGX A100/NVLink, other approach might be better for portable performance

Collective Kernel Launch

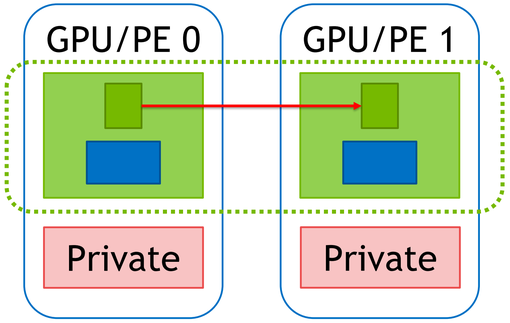
Ensures progress when using device-side inter-kernel synchronization

NVSHMEM Usage	CUDA Kernel launch
Device-Initiated Communication	Execution config syntax `<<<...>>>` or launch APIs
Device-Initiated Synchronization	`nvshmemx_collective_launch`

- CUDA's throughput computing model allows (encourages) grids much larger than a GPU can fit
- Inter-kernel synchronization requires producer and consumer threads to execute concurrently
- Collective launch guarantees co-residency using CUDA cooperative launch and requirement of 1PE/GPU

NVSHMEM API - Device side

Communicating from inside the kernel

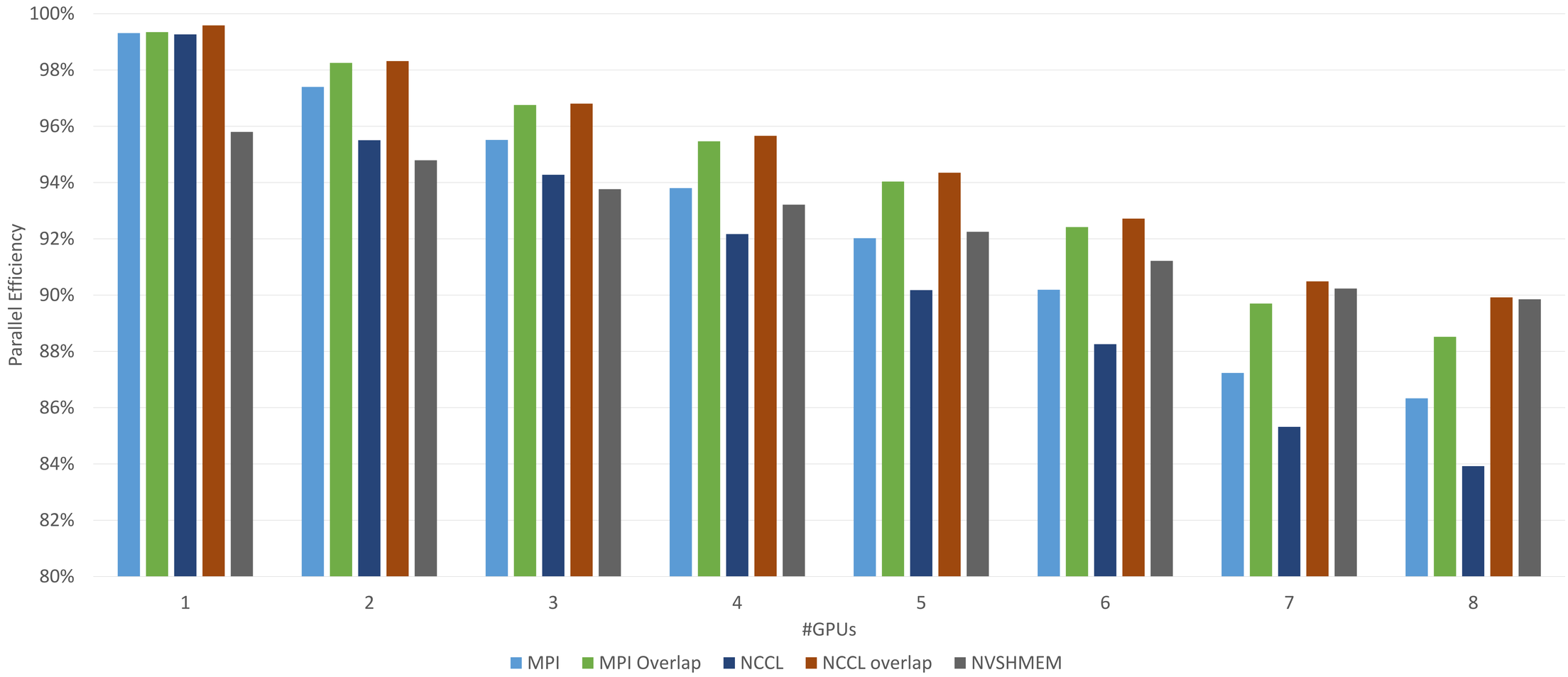


- Overview over different kinds of API calls
  - Type signatures omitted! See appendix for more
- Naming: types are encoded in function name (follows OpenSHMEM standard)

```
// single element put
__device__ void nvshmem_T_p(dest, value, pe)

// nonblocking block cooperative put
__device__ void nvshmemx_T_put_nbi_block(dest, source, nelems, pe)

// ordering and completion
__device__ void nvshmem_quiet()

// signal operation
__device__ inline void nvshmemx_signal_op(sig_addr, signal, sig_op, pe)

// atomic operation
__device__ void nvshmem_T_atomic_inc(dest, pe)

// wait operations
__device__ void nvshmem_T_wait_until_all(ivars, nelems, status, cmp, cmp_value)
__device__ void nvshmem_T_wait_until    (ivar,                   cmp, cmp_value)
```

Multi GPU Jacobi Parallel Efficiency

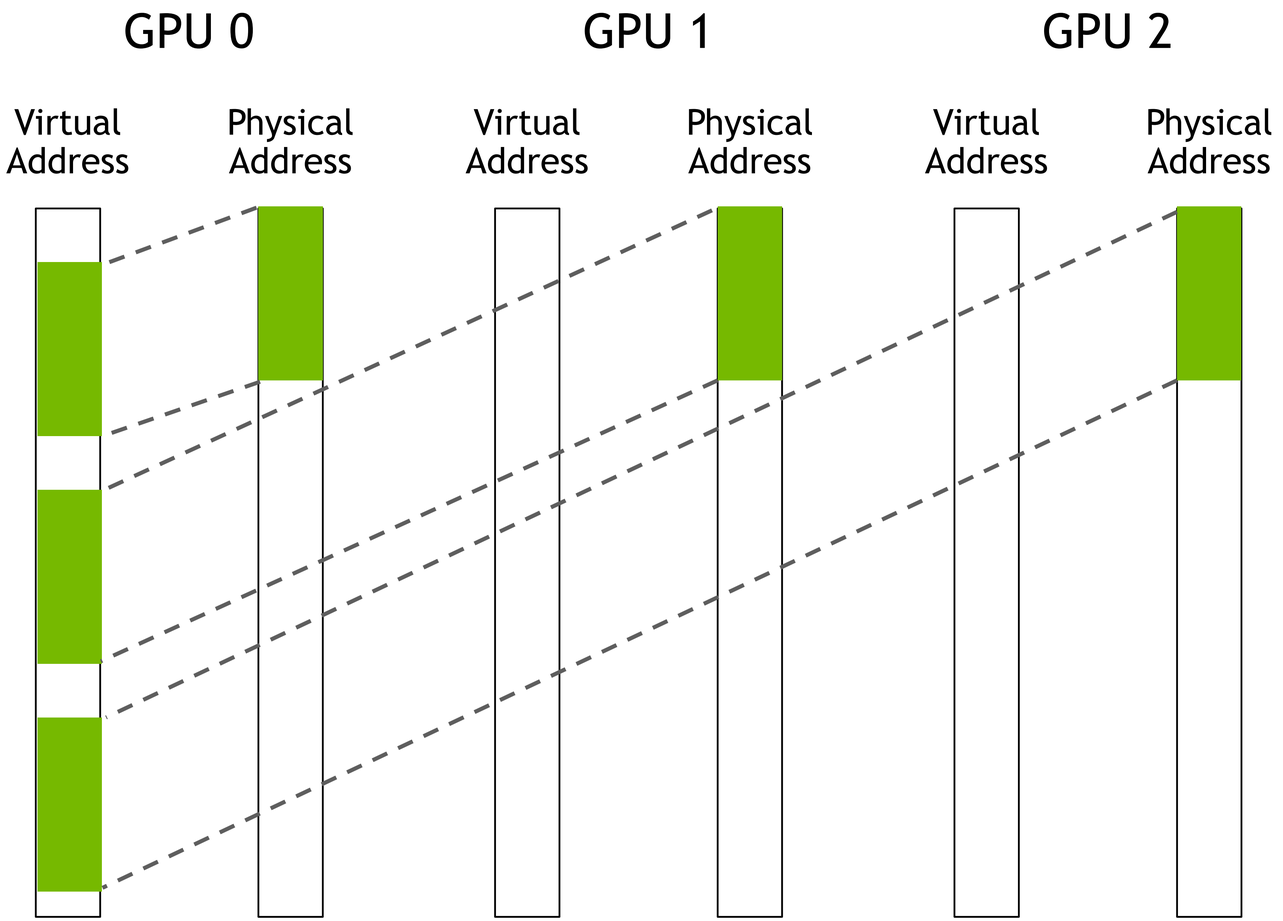
DGX A100 – 20480 x 20480, 1000 iterations



Benchmarksetup: DGX A100, Ubuntu 20.04.1 LTS, GCC 9.3.0, CUDA Driver 460.58, CUDA 11.2, UCX 1.11.0, OpenMPI 4.1.1rc1, NCCL 2.8.4, NVSHMEM 2.1, GPUs@1410Mhz AC, Reported Runtime is the minimum of 5 repetitions

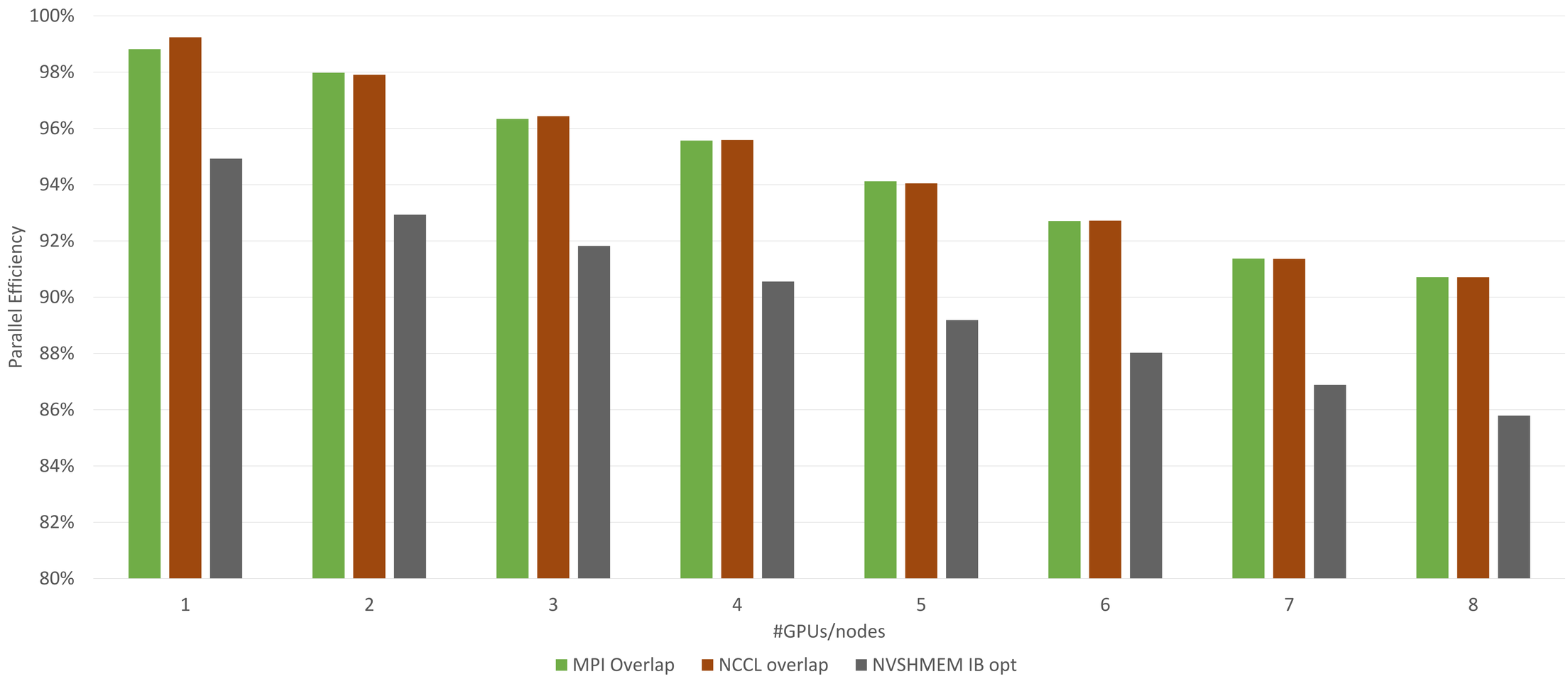
Optimized Intra-Node Communication

- Supported on NVLink and PCI-E
- Use CUDA IPC or `cuMem*` API to map symmetric memory of intra-node PEs into virtual address space
- `nvshmem_[put|get]` on device -> load/store
- `nvshmem_[put|get]_on_stream` -> `cudaMemcpyAsync`



Multi GPU Multi Node Jacobi Parallel Efficiency

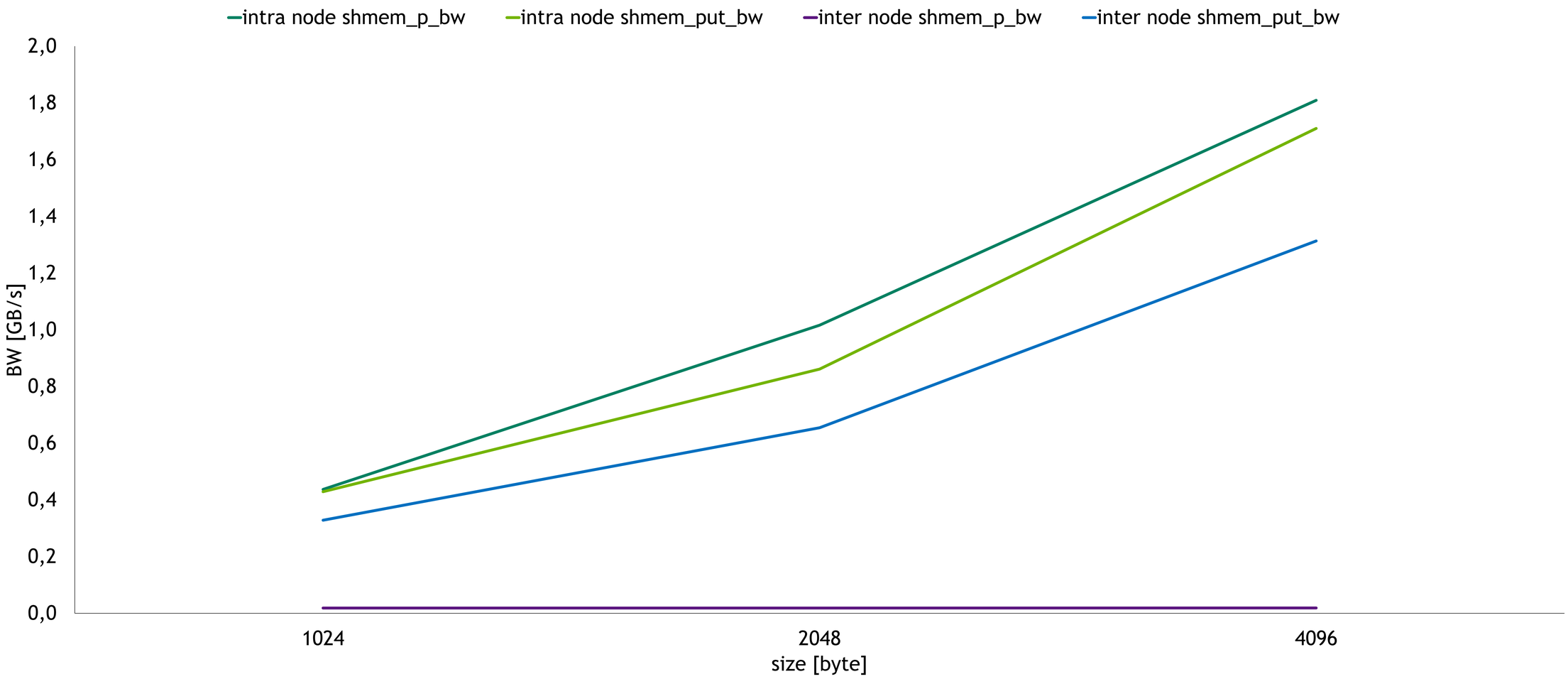
8 DGX A100 – 20480 x 20480, 1000 iterations



Benchmarksetup: DGX A100, Ubuntu 20.04.1 LTS, GCC 9.3.0, CUDA Driver 460.58, CUDA 11.2, UCX 1.11.0, OpenMPI 4.1.1rc1, NCCL 2.8.4, NVSHMEM 2.1, GPUs@1410Mhz AC, Reported Runtime is the minimum of 5 repetitions

NVSHMEM perftests

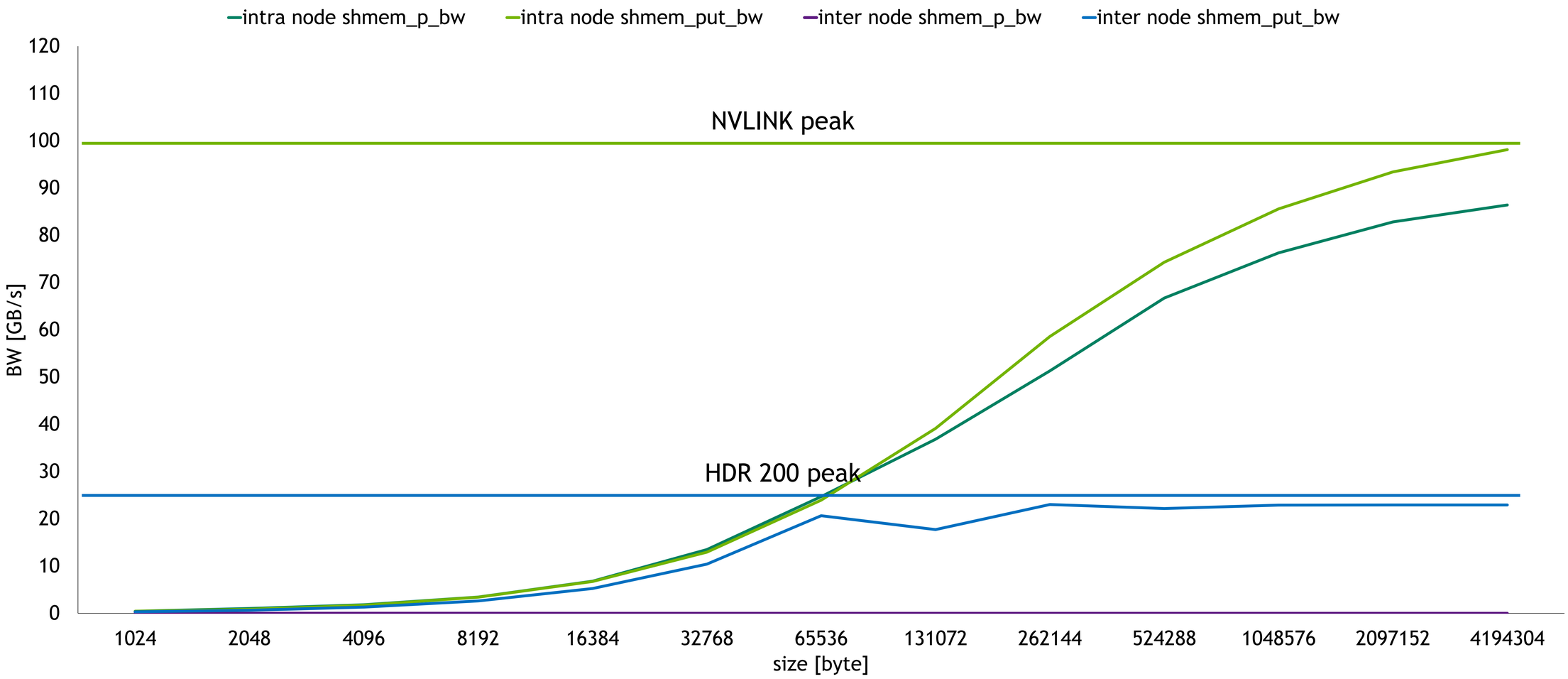
shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB



shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

NVSHMEM perftests

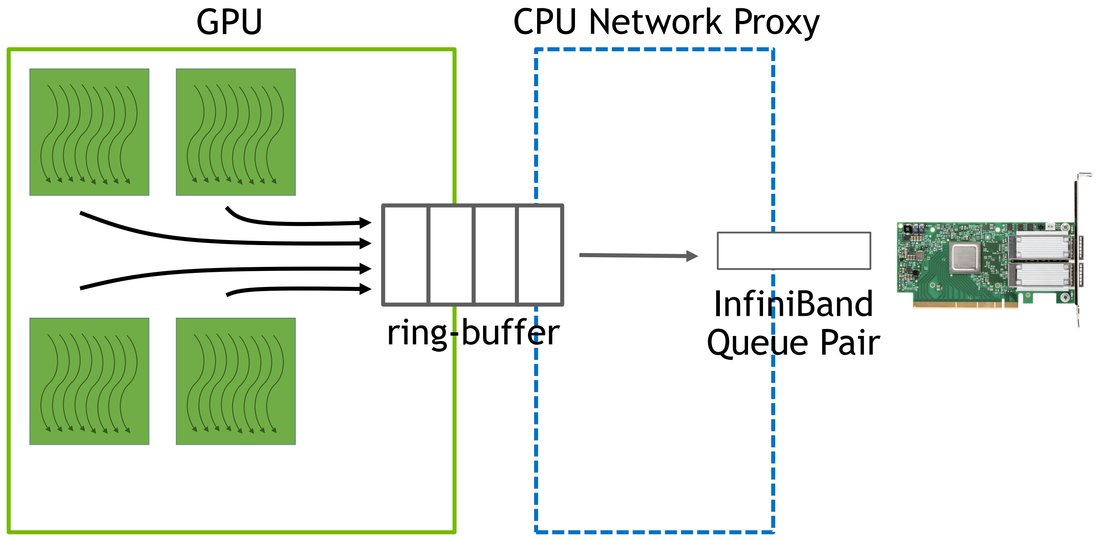
shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB

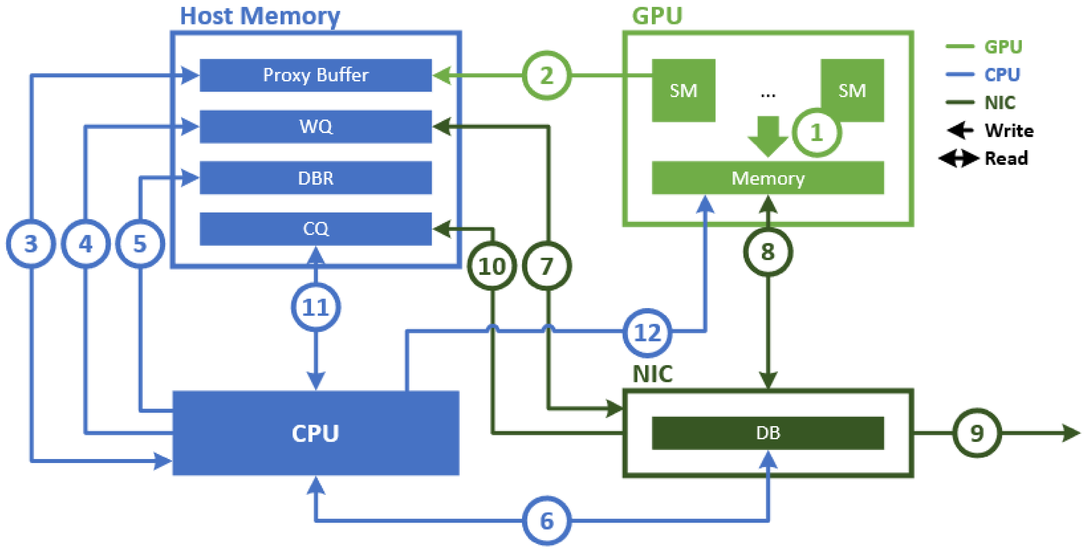


shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

Optimized Inter-Node Communication

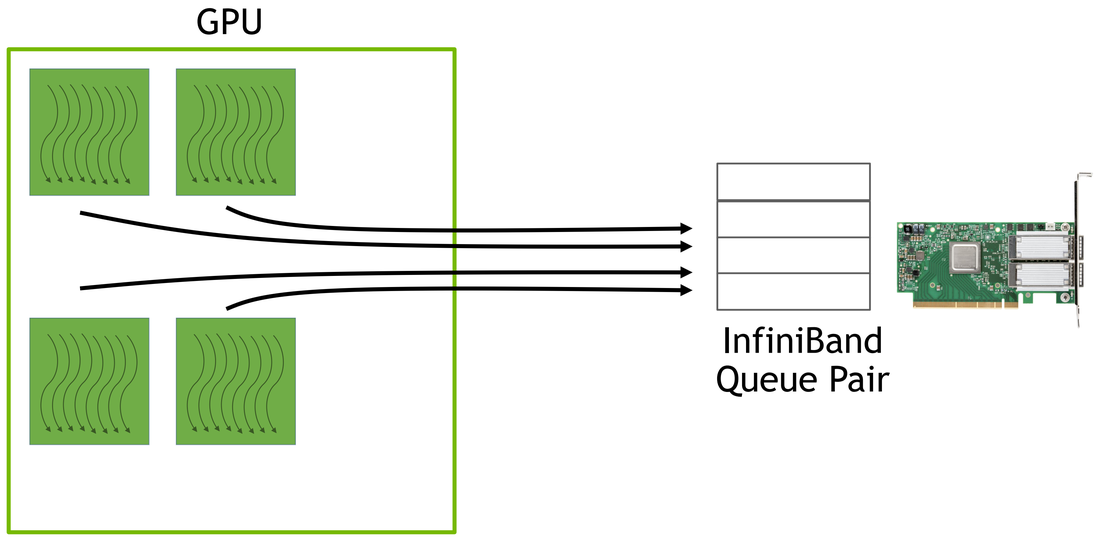
- NVSHMEM supports inter-node communication over InfiniBand, RoCE, and UCX (experimental)
- Using GPUDirect RDMA (data plane)
- Reverse offloads network transfers from GPU to the CPU (control plane)
- Ring buffer implementation avoids memory fences when interacting with CPU network proxy

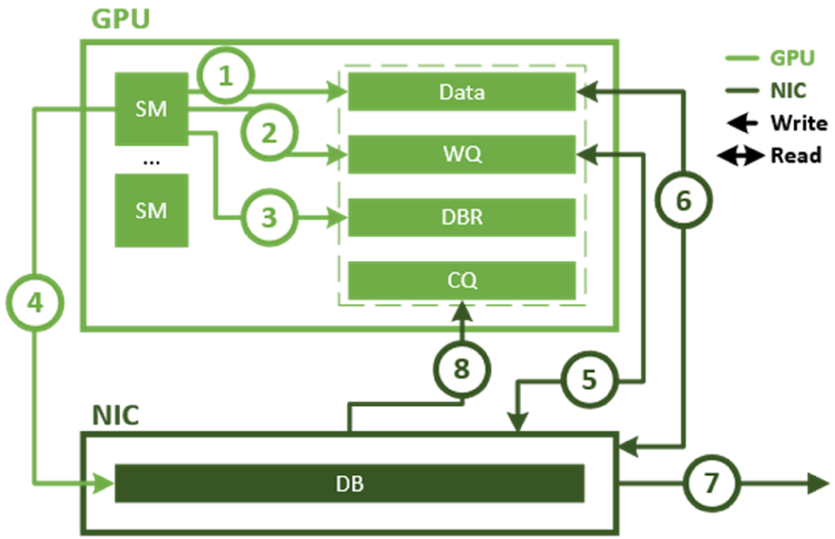




Optimized Inter-Node Communication Improved

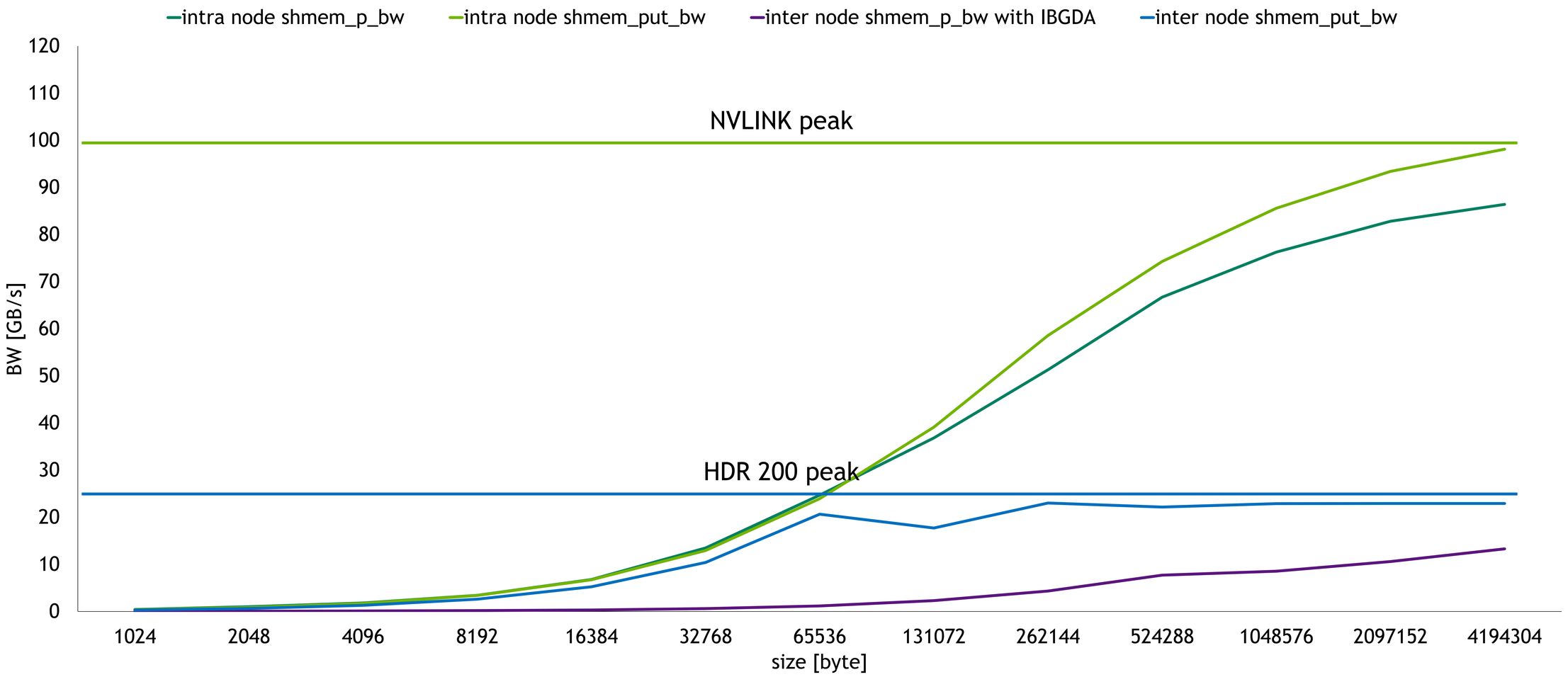
- **IB GPUD**irect **A**sync (IBGDA) over InfiniBand
- Using GPUDirect RDMA (data plane)
- GPU directly initiates network transfers involving the CPU only for the setup of control data structures





NVSHMEM perftests with IBGDA

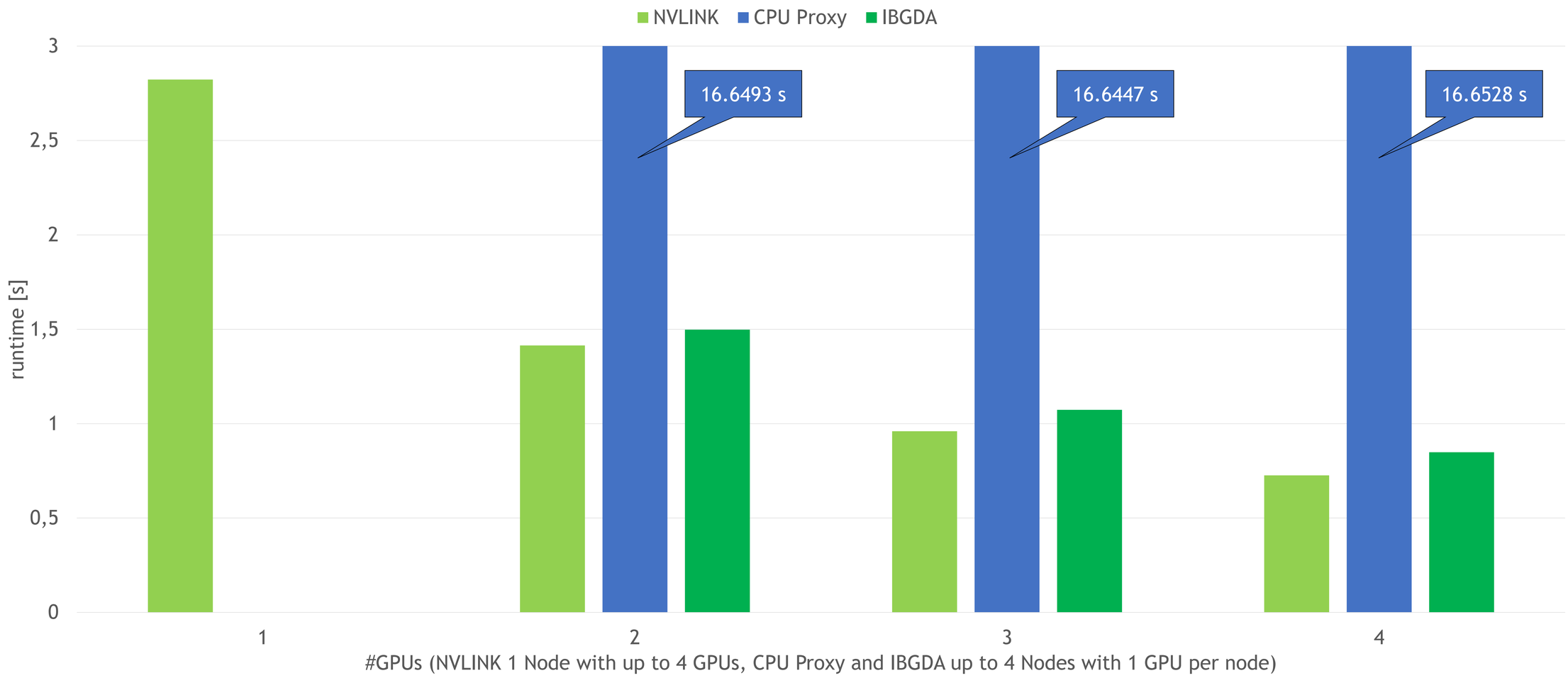
shmem_p_bw and shmem_put_bw on JUWELS Booster – NVIDIA A100 40 GB



shmem_p_bw|shmem_put_bw from NVSHMEM perftest
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

NVSHMEM Version with NVL, CPU Proxy and IBGDA

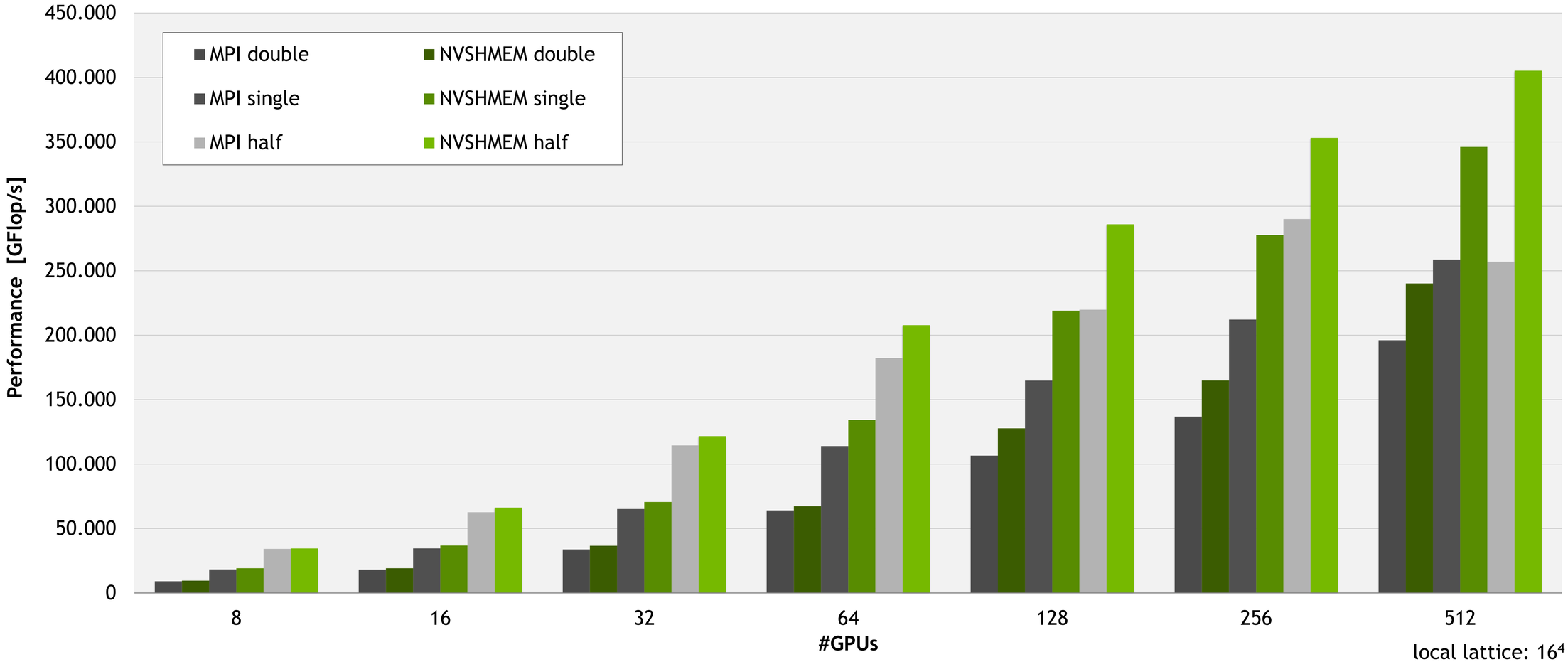
NVSHMEM 2.8.0 prerelease – JUWELS Booster – NVIDIA A100 40 GB – Jacobi on 17408x17408



Source: https://github.com/NVIDIA/multi-gpu-programming-models
JUWELS Booster: https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html

QUDA Strong Scaling on Selene

Lattice Quantum ChromoDynamics



local lattice: 16^4

up to 1.6x Speedup (512 GPUs half precision)

https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21673/

Content Sources and Details

Further reading and watching

- NCCL team GTC Talks
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51111/
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41784/
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31880/
- NVSHMEM GTC Talks
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51705/
  - https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41044/
  - https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21673/
- Multi-GPU Programming series, for example https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41018/
- More hands-on content and examples
  - https://github.com/FZJ-JSC/tutorial-multi-gpu - living repository for our onsite tutorial with exercises and slides
  - https://github.com/NVIDIA/multi-gpu-programming-models/ - Jacobi solver multi-GPU implementation with many different approaches, NCCL and NVSHMEM amongst them

Optional Hands-on: Device-initiated Communication with NVSHMEM

Location: 02-NCCL_NVSHMEM/03-NVSHMEM_Device, follow instructions from tasks/Instructions.ipynb

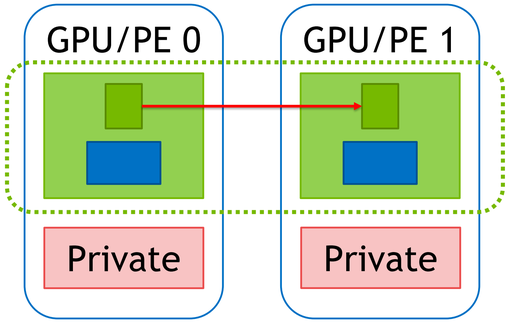
- Use NVSHMEM device API instead of MPI to implement a multi-GPU Jacobi solver. Work on TODOs in jacobi.cu:
  - Initialize NVSHMEM (this is same as in previous hands-on 2):
  - Include NVSHMEM headers.
  - Initialize and shutdown NVSHMEM using MPI_COMM_WORLD.
  - Allocate work arrays a and a_new from the NVSHMEM symmetric heap.
    Again, ensure the size you pass in is the same value on all participating ranks!
  - Calculate halo/boundary row index of top and bottom neighbors.
  - Add necessary inter PE synchronization.
  - Modify jacobi_kernel (this is the device-initiated part):
  - Pass in halo/boundary row index of top and bottom neighbors.
  - Use nvshmem_float_p to directly push values needed by top and bottom neighbors from the kernel.
  - Remove no longer needed MPI communication.
- Compile with
  - make
- Submit your compiled application to the batch system with
  - make run
- Study the performance
  - make profile
- The environment variable NP can change the number of processes

NVSHMEM API - Device side

single element put

```
__device__ void nvshmem_TYPENAME_p(TYPE *dest, TYPE value, int pe)
```

- `dest` [OUT]: Symmetric address of the destination data object.
- `value` [IN]: The value to be transferred to `dest`.
- `pe` [IN]: The number of the remote PE.



See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#nvshmem-p

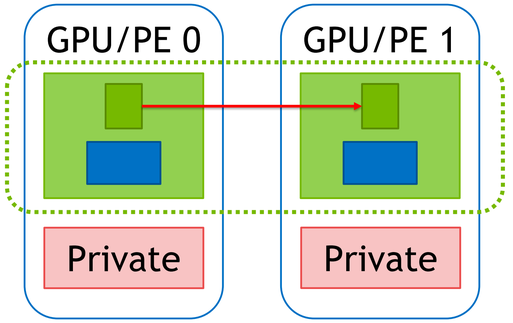
`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff`
(see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

nonblocking block cooperative put

```
__device__ void nvshmemx_TYPENAME_put_nbi_block(TYPE *dest, const TYPE *source, size_t nelems, int pe)
```

- `dest` [OUT]: Symmetric address of the destination data object.
- `source` [IN]: Symmetric address of the object containing the data to be copied.
- `nelems` [IN]: Number of elements in the `dest` and `source` arrays.
- `pe` [IN]: The number of the remote PE.



Cooperative call: Needs to be called by all threads in a block. thread and warp are also available.

x in `nvshmem`**x** marks API as extension of the OpenSHMEM APIs.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html?highlight=nvshmemx_typename_put_nbi_block#nvshmem-put-nbi

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff`
(see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

ordering and completion

```
__device__ void nvshmem_quiet(void)
```

Ensures completion of all operations on symmetric data objects issued by the calling PE.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/ordering.html#nvshmem-quiet

NVSHMEM API

signal operation

```
__device__ inline void nvshmemx_signal_op(uint64_t *sig_addr, uint64_t signal, int sig_op, int pe)
```

- `sig_addr` [OUT]: Symmetric address of the signal word to be updated.
- `signal` [IN]: The value used to update `sig_addr`.
- `sig_op` [IN]: Operation used to update `sig_addr` with `signal`. (`NVSHMEM_SIGNAL_SET` or `NVSHMEM_SIGNAL_ADD`)
- `pe` [IN]: The number of the remote PE.

x in `nvshmem`**x** marks API as extension of the OpenSHMEM APIs.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/signal.html#nvshmemx-signal-op

NVSHMEM API

atomic operation

```
__device__ void nvshmem_TYPENAME_atomic_inc(TYPE *dest, int pe)
```

- `dest` [OUT]: Symmetric address of the signal word to be updated.
- `pe` [IN]: The number of the remote PE.

These routines perform an atomic increment operation on the `dest` data object on PE.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/amo.html#nvshmem-atomic-inc

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff` (see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)

NVSHMEM API

wait operations

```
__device__ void nvshmem_TYPENAME_wait_until_all(TYPE *ivars, size_t nelems, const int *status,
                                                int cmp, TYPE cmp_value)

__device__ void nvshmem_TYPENAME_wait_until(TYPE *ivar, int cmp, TYPE cmp_value)
```

- `ivars|ivar` [IN]: Symmetric address of an array of remotely accessible data objects.|Symmetric address of a remotely accessible data object.
- `nelems` [IN]: The number of elements in the `ivars` array.
- `status` [IN]: Local address of an optional mask array of length `nelems` that indicates which elements in `ivars` are excluded from the wait set. Set to `NULL` when not used.
- `cmp` [IN]: A comparison operator (`NVSHMEM_CMP_EQ`, `NVSHMEM_CMP_NE`, `NVSHMEM_CMP_GT`, `NVSHMEM_CMP_GE`, `NVSHMEM_CMP_LT`, `NVSHMEM_CMP_LE`) that compares elements of `ivars|ivar` with `cmp_value`.
- `cmp_value` [IN]: The value to be compared with the objects pointed to by `ivars`.

See: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/sync.html#nvshmem-wait-until-all and https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/sync.html#nvshmem-wait-until

`TYPENAME` can be: `float`, `double`, `char`, `schar`, `short`, `int`, `long`, `longlong`, `uchar`, `ushort`, `uint`, `ulong`, `ulonglong`, …, `ptrdiff` (see: https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes)


NVIDIA