High-Performance Computing

Daniele De Sensi

More Detailed Analysis of Assignment 4

(streams+priority version,
for the sake of clarity without CUDA_RT_CALL, NVTX PUSH/POP, etc..)

Analysis (pt. 1)

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Analysis (pt. 1)

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

The l2_norm_d is updated atomically by all threads (in all three kernels)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Analysis (pt. 1)

Enqueue also an event after that, so that we know when the reset of l2_norm_d happened

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Analysis (pt. 1)

Enqueue also an event after that, so that we know when the reset of l2_norm_d happened

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Compute on the inner domain (after reset done since on compute_stream)

Analysis (pt. 1)

Enqueue also an event after that, so that we know when the reset of l2_norm_d happened

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Compute on the inner domain (after reset done since on compute_stream)

Dependency between streams, nothing can start on push_top_stream if the l2_norm has not been reset

Analysis (pt. 1)

Enqueue also an event after that, so that we know when the reset of l2_norm_d happened

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Compute on the inner domain (after reset done since on compute_stream)

Dependency between streams, nothing can start on push_top_stream if the l2_norm has not been reset

Top row update on push_top_stream

Analysis (pt. 1)

Enqueue also an event after that, so that we know when the reset of l2_norm_d happened

Check if in this iteration we should compute the norm across all GPUs to determine if we need to terminate computation

The l2_norm_d is updated atomically by all threads (in all three kernels)

Enqueue on compute_stream a request to set l2_norm_d to 0 (all kernels executed on compute_stream will start only after this set to 0 happened)

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

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


    cudaStreamWaitEvent(push_top_stream, reset_l2norm_done, 0);
    launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm, push_top_stream);
    cudaEventRecord(push_top_done, push_top_stream);

```

Compute on the inner domain (after reset done since on compute_stream)

Dependency between streams, nothing can start on push_top_stream if the l2_norm has not been reset

Top row update on push_top_stream

Use an event to know when update of top row completed

Analysis (pt. 2)

Same as before, for bottom row (wait reset of l2_norm_d, update bottom row, enqueue an event to know where bottom row update finished)

```
cudaStreamWaitEvent(push_bottom_stream, reset_l2norm_done, 0);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, calculate_norm, push_bottom_stream);
cudaEventRecord(push_bottom_done, push_bottom_stream);

if (calculate_norm) {
    cudaStreamWaitEvent(compute_stream, push_top_done, 0);
    cudaStreamWaitEvent(compute_stream, push_bottom_done, 0);
    cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);
}
```

Analysis (pt. 2)

Same as before, for bottom row (wait reset of l2_norm_d, update bottom row, enqueue an event to know where bottom row update finished)

```
cudaStreamWaitEvent(push_bottom_stream, reset_l2norm_done, 0);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, calculate_norm, push_bottom_stream);
cudaEventRecord(push_bottom_done, push_bottom_stream);

if (calculate_norm) {
    cudaStreamWaitEvent(compute_stream, push_top_done, 0);
    cudaStreamWaitEvent(compute_stream, push_bottom_done, 0);
    cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);
}
```

Wait for the top and bottom rows to be updated

Analysis (pt. 2)

Same as before, for bottom row (wait reset of l2_norm_d, update bottom row, enqueue an event to know where bottom row update finished)

```
cudaStreamWaitEvent(push_bottom_stream, reset_l2norm_done, 0);
launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, calculate_norm, push_bottom_stream);
cudaEventRecord(push_bottom_done, push_bottom_stream);

if (calculate_norm) {
    cudaStreamWaitEvent(compute_stream, push_top_done, 0);
    cudaStreamWaitEvent(compute_stream, push_bottom_done, 0);
    cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);
}
```

Wait for the top and bottom rows to be updated

Copy the l2_norm_d to the host (we need it on the host to do an MPI_Allreduce with the other GPUs) when the kernel updating the inner part of the domain (running on compute_stream) completes

Analysis (pt. 3)

Send top/bottom rows to neighbors
(after the event on the relative streams happened)

```
const int top = rank > 0 ? rank - 1 : (size - 1);
const int bottom = (rank + 1) % size;

// Apply periodic boundary conditions
cudaStreamSynchronize(push_top_stream);
MPI_Sendrecv(a_new + iy_start * nx, nx, MPI_REAL_TYPE, top, 0,
             a_new + (iy_end * nx), nx, MPI_REAL_TYPE, bottom, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

cudaStreamSynchronize(push_bottom_stream);
MPI_Sendrecv(a_new + (iy_end - 1) * nx, nx, MPI_REAL_TYPE, bottom, 0, a_new, nx,
                         MPI_REAL_TYPE, top, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Analysis (pt. 4)

Wait for the update of the inner domain, and for the asyncmemcpy of l2_norm_d to l2_norm_h
(we already waited for completion on top/bottom row)

```
    if (calculate_norm) {
        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);
        }
    }
    std::swap(a_new, a);
    iter++;
}
```

Analysis (pt. 4)

Wait for the update of the inner domain, and for the asyncmemcpy of l2_norm_d to l2_norm_h
(we already waited for completion on top/bottom row)

```
    if (calculate_norm) {
        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);
        }
    }
    std::swap(a_new, a);
    iter++;
}
```

Compute the aggregated norm across alll GPUs

Analysis (pt. 5)

Wait for the update of the inner domain, and for the asyncmemcpy of l2_norm_d to l2_norm_h
(we already waited for completion on top/bottom row)

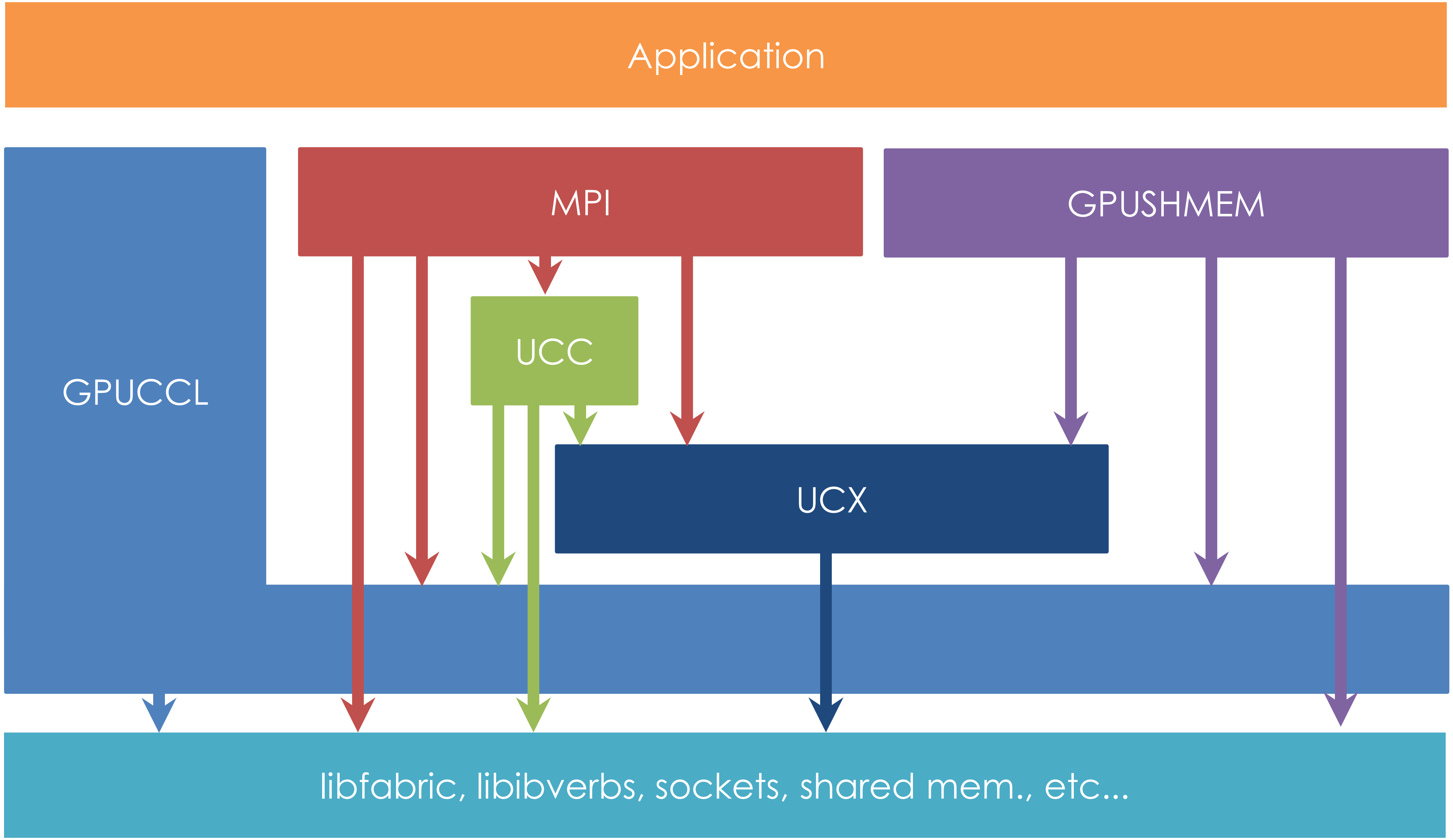
```
    if (calculate_norm) {
        cudaStreamSynchronize(compute_stream);
        MPI_Allreduce(l2_norm_h, &l2_norm, 1, MPI_REAL_TYPE, MPI_SUM, MPI_COMM_WORLD);
        l2_norm = std::sqrt(l2_norm);

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

Compute the aggregated norm across alll GPUs

Swap the old/new domains

Questions?

Software Stack



NCCL & NVSHMEM

Slide courtesy: Lena Oden & Markus Hrywniak

NCCL

Slide courtesy: Lena Oden & Markus Hrywniak

The road so far

- CUDA-Aware MPI
- What is behind (GPUDirect Technologies, Unified Memory)
- How to debug, profile and trace your Code
- How to navigate in the traces
- How to overlap communication and computation on the GPUs
- Streams and priority streams

Motivation

- MPI is **not** (yet [1]) aware of CUDA streams
- Explicit synchronization between GPU-compute kernel and CPU communication calls is required
- CUDA-aware MPI is *GPU-memory-aware* communication
- For better efficiency: *CUDA-stream-aware* communication
  - Communication, which is aware of CUDA-streams or use CUDA streams
  - NCCL and (Host-API) of NVSHMEM

What will you Learn?

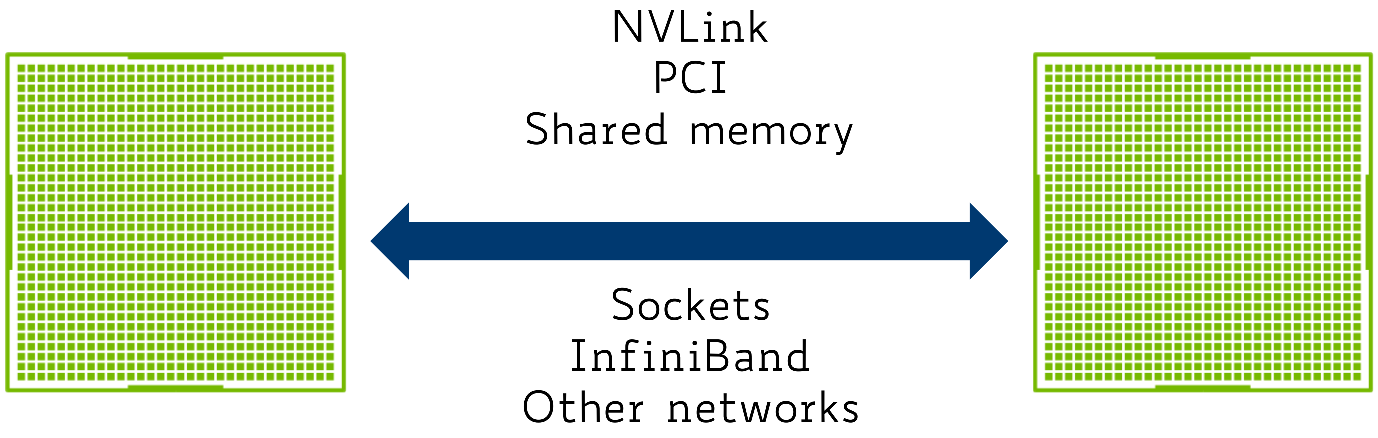
- How to use NCCL inside an MPI Application to use CUDA-stream-aware P2P communication
- NVSHMEM memory model
- How to use stream-aware NVSHMEM communication operations in MPI Programs

[1] MPI Forum Hybrid Working Group - Stream and Graph Based MPI Operations: https://github.com/mpiwg-hybrid/hybrid-issues/issues/5

Optimized inter-GPU communication
NCCL : NVIDIA Collective Communication Library

Communication library running on GPUs, for GPU buffers.

- Library for efficient communication with GPUs
- First: Collective Operations (e.g. Allreduce), as they are required for DeepLearning
- Since 2.8: Support for Send/Recv between GPUs
- Library running on GPU: Communication calls are translated to GPU a kernel (running on a stream)



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-API (With MPI) - Initialization

First, we need a NCCL-Communicator for, this, wee need a NCCL UID

We use MPI-Ranks and size for initialization

```
MPI_Init(&argc,&argv)
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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);
MPI_Finalize();
```

Communication Calls

"The NCCL call returns when the operation has been effectively enqueued to the given stream, or returns an error. The collective operation is then executed asynchronously on the CUDA device. The operation status can be queried using standard CUDA semantics, for example, calling cudaStreamSynchronize or using CUDA events."

- Send/Recv

```
ncclSend(void* sbuff, size_t count, ncclDataType_t type, int peer, ncclComm_t comm, cudaStream_t stream);
ncclRecv(void* rbuff, size_t count, ncclDataType_t type, int peer, ncclComm_t comm, cudaStream_t stream);
```

- Collective Operations

```
    ncclAllReduce(void* sbuff, void* rbuff, size_t count, ncclDataType_t type, ncclRedOp_t op,           ncclComm_t comm, cudaStream_t stream);
    ncclBroadcast(void* sbuff, void* rbuff, size_t count, ncclDataType_t type,                 int root, ncclComm_t comm, cudaStream_t stream);
       ncclReduce(void* sbuff, void* rbuff, size_t count, ncclDataType_t type, ncclRedOp_t op, int root, ncclComm_t comm, cudaStream_t stream);
ncclReduceScatter(void* sbuff, void* rbuff, size_t count, ncclDataType_t type, ncclRedOp_t op,           ncclComm_t comm, cudaStream_t stream);
    ncclAllGather(void* sbuff, void* rbuff, size_t count, ncclDataType_t type,                           ncclComm_t comm, cudaStream_t stream);
```

Fused Communication Calls

- Multiple calls to ncclSend() and ncclRecv() should be fused with ncclGroupStart() and ncclGroupEnd()to
  - Avoid deadlocks (if calls need to progress concurrently)
  - Reduces communication kernels launch overhead
  - Communication effectively start when ncclGroupEnd() is called
    - i.e., it guarantees it has been enqueued to the stream, not that it completed

SendRecv:

```
ncclGroupStart();
 ncclSend(sendbuff, sendcount, sendtype, peer, comm, stream);
ncclRecv(recvbuff, recvcount, recvtype, peer, comm, stream);
ncclGroupEnd();
```

Bcast:

```
ncclGroupStart();
if (rank == root) {
  for (int r=0; r<nranks; r++)
    ncclSend(sendbuff[r], size, type, r, comm, stream);}
ncclRecv(recvbuff, size, type, root, comm, stream);
ncclGroupEnd();
```

Neighbor exchange:

```
ncclGroupStart();
for (int d=0; d<ndims; d++) {
  ncclSend(sendbuff[d], sendcount, sendtype, next[d], comm, stream);
  ncclRecv(recvbuff[d], recvcount, recvtype, prev[d], comm, stream);}
ncclGroupEnd();
```

Fused Communication Calls - Attention to the order

```
RANK0/GPU0/Process0:
ncclGroupStart();
ncclAllReduce(sendbuff4, recvbuff4, count4, datatype, comm1, stream); // WRONG: reversed order
ncclBroadcast(sendbuff1, recvbuff1, count1, datatype, root, comm0, stream);
ncclAllReduce(sendbuff2, recvbuff2, count2, datatype, comm0, stream);
ncclAllReduce(sendbuff3, recvbuff3, count3, datatype, comm0, stream);
ncclGroupEnd();

RANK1/GPU1/Process1:
ncclGroupStart();
ncclBroadcast(sendbuff1, recvbuff1, count1, datatype, root, comm0, stream);
ncclAllReduce(sendbuff2, recvbuff2, count2, datatype, comm0, stream);
ncclAllReduce(sendbuff3, recvbuff3, count3, datatype, comm0, stream);
ncclAllReduce(sendbuff4, recvbuff4, count4, datatype, comm1, stream); // WRONG: reversed order
ncclGroupEnd();
```

Jacobi using NCCL

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

Performance Improvement

- So far, no overlap of communication and computation
- Use techniques from previous session to overlap communication and computation
- Make sure that communication streams are scheduled
  - CUDA high priority streams!

```
int leastPriority = 0;
int greatestPriority = leastPriority;
cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority));

cudaStream_t compute_stream;
cudaStream_t push_stream;

cudaStreamCreateWithPriority(&compute_stream, cudaStreamDefault, leastPriority));
cudaStreamCreateWithPriority(&push_top, cudaStreamDefault, greatestPriority));
```

Jacobi using NCCL and Overlapping Communication and Computation

```
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);

ncclGroupStart();
ncclRecv(a_new, nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream)
ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm,push_stream);
ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm,push_stream);
ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm,push_stream);
ncclGroupEnd();
```

Communication Buffers in NCCL

- Default: Data are staged in a communication buffer

- Read data from source buffer
- Write data to staging buffer
- Read data from staging buffer
- Write data to destination buffer

- Zero-Copy Communication

- Read data from source buffer
- Write data to destination buffer

Buffer Registration in NCCL

- To support zero copy, the buffer must be registered
- For this, the memory should be allocated with ncclMemAlloc

```
ncclResult_t ncclMemAlloc(void** buff, size_t count);
```

- Next, register the buffer for communication for a specific communicator with ncclCommRegister

```
ncclCommRegister(const ncclComm_t comm, void* buff, size_t size, void** handle);
```

- At the end, deregister the buffer, using the handle

```
ncclCommDeRegister(const ncclComm_t comm, void* handle);
```

- And free the buffer again

```
ncclResult_t ncclMemFree(void* buff);
```

How to Compile an MPI+NCCL Application

- Include header files and link against CUDA NCCL library

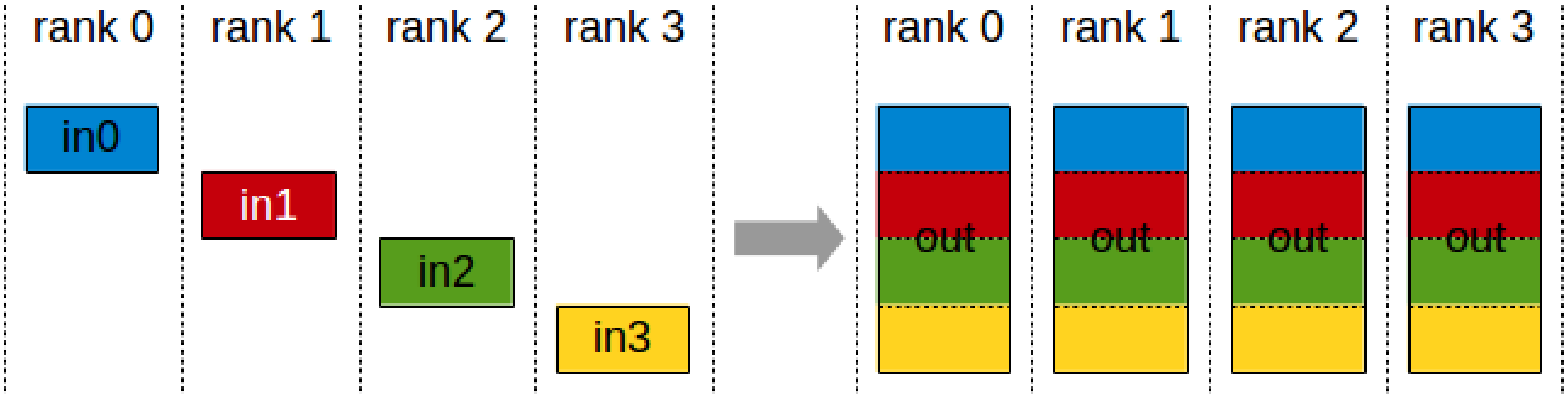
```
#include <nccl.h>
```

```
MPICXX_FLAGS = -I$(CUDA_HOME)/include -I$(NCCL_HOME)/include
LD_FLAGS = -L$(CUDA_HOME)/lib64 -lcudart -lnccl

$(NVCC) $(NVCC_FLAGS) jacobi_kernels.cu -c -o jacobi.o
$(MPICXX) $(MPICXX_FLAGS)  jacobi.cpp jacobi_kernels.o $(LD_FLAGS) -o jacobi
```

Questions?

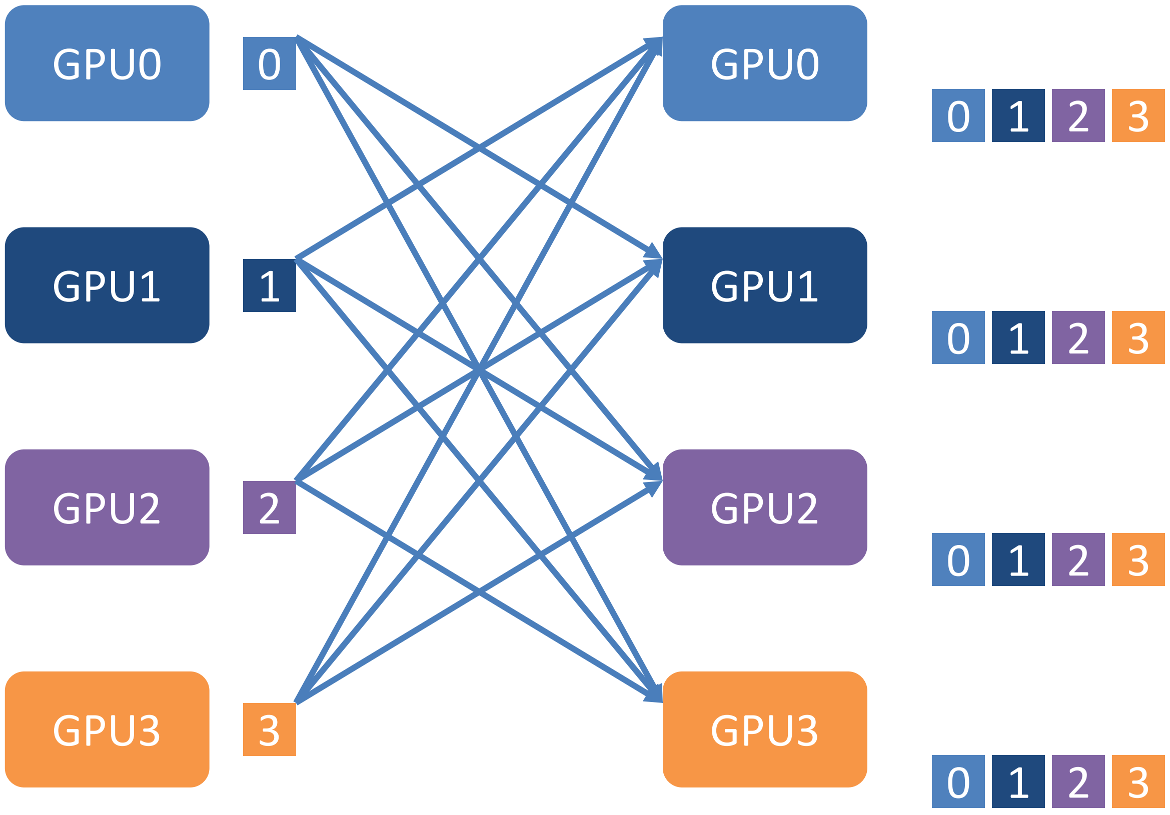
NCCL Collectives

Allgather



rank 0
rank 1
rank 2
rank 3
in0
in1
in2
in3
rank 0
rank 1
rank 2
rank 3
out
out
out
out

Allgather: Naïve Algorithm

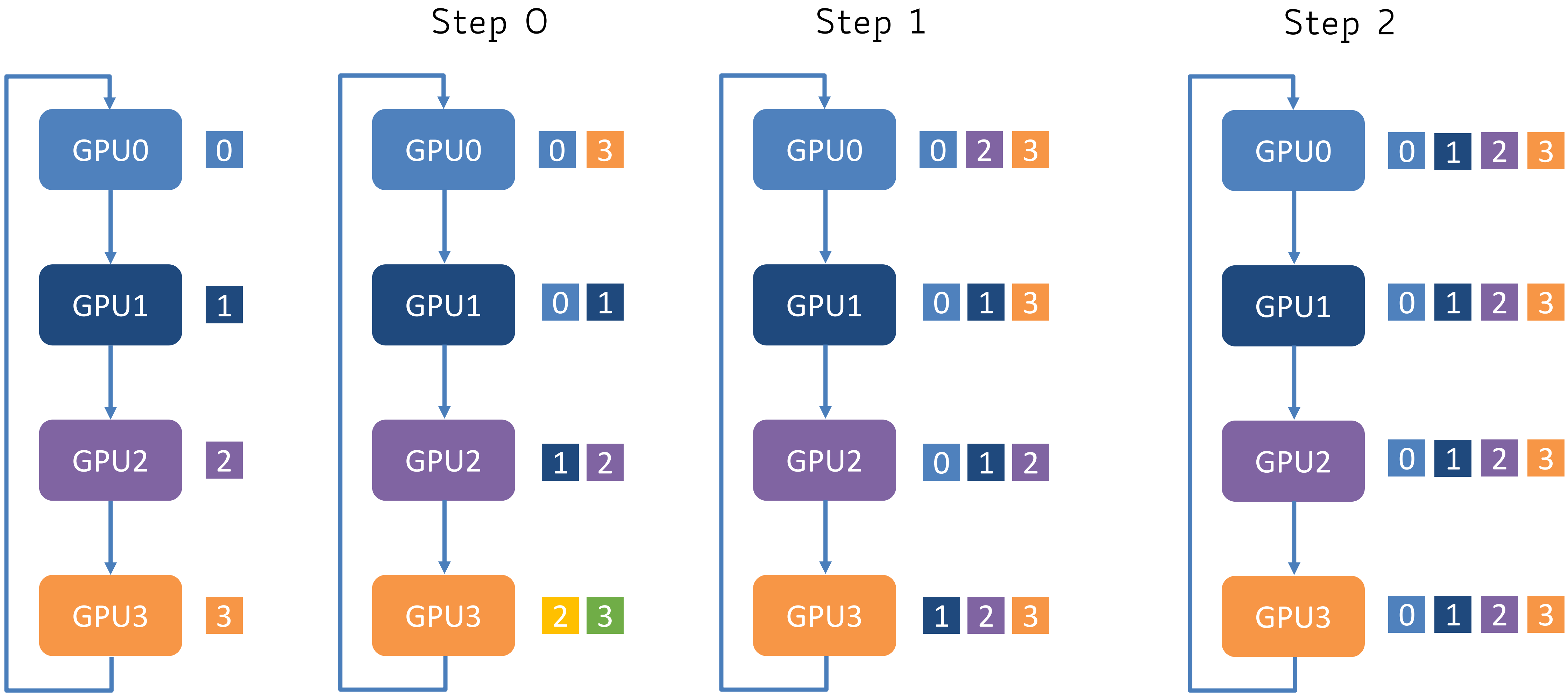


Size of the final data,
not the starting data

$$T(p, n) = p * \alpha + n * \beta$$

We still p (actually p-1) steps. We assume to have a single network interface card, and thus we cannot transmit to all the other p-1 GPUs in parallel

Allgather: Ring Algorithm

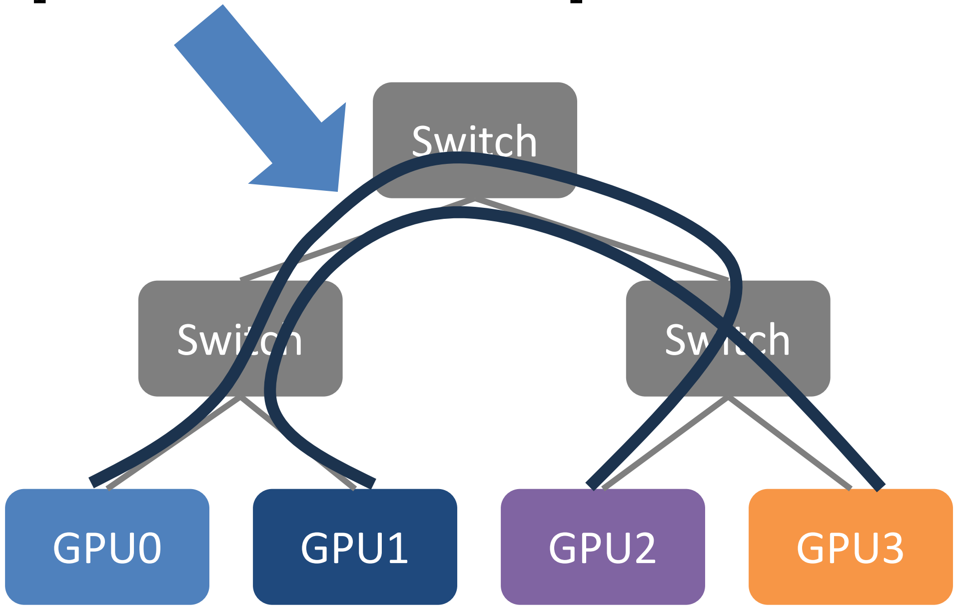


$$T(p, n) = p^*\alpha + n^*\beta$$

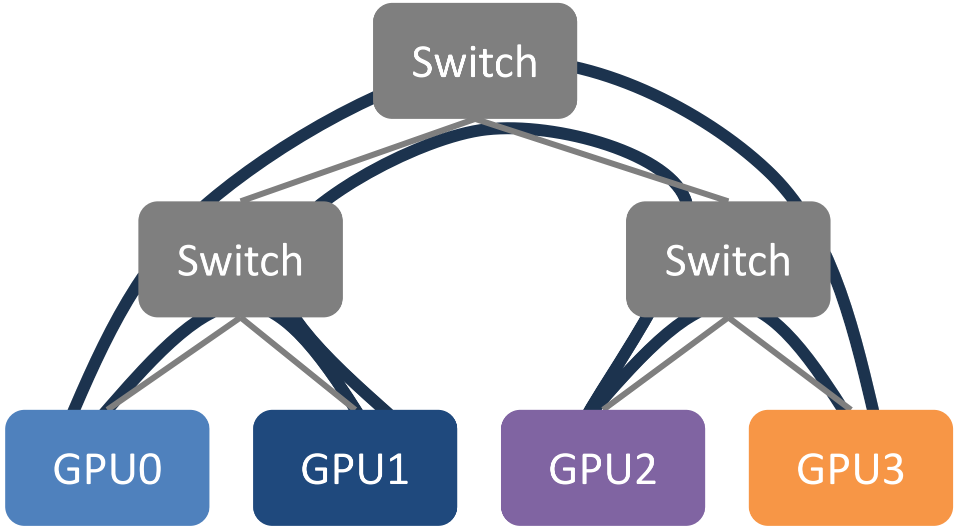
Ring vs. Naïve Algorithm

In practice, the ring algorithm is used instead of the Naive algorithm (albeit they have the same cost): Why?

Contention, performance will drop



Naive



Ring

Take-home message: Performance modelling only tell us a part of the story, and do not (fully) take into account the system/network they are running on

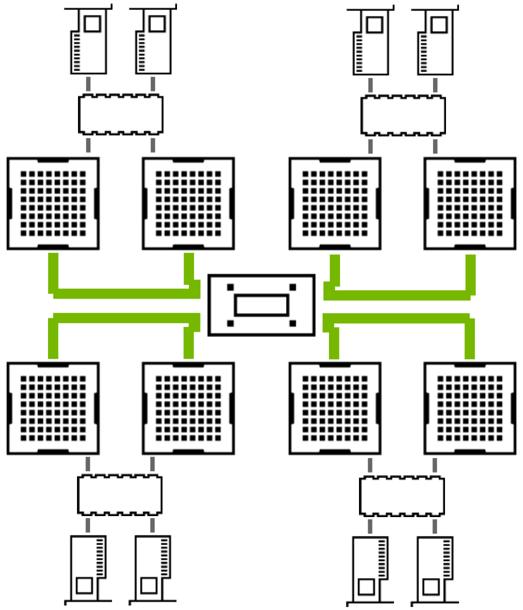
NCCL Background - Topology Detection

Mapping algorithms to the hardware

Topology detection

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

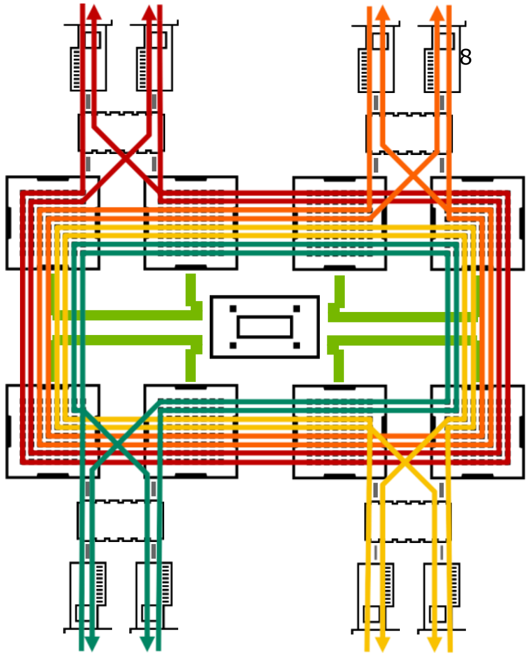
Topology injection for VMs.



Graph search

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

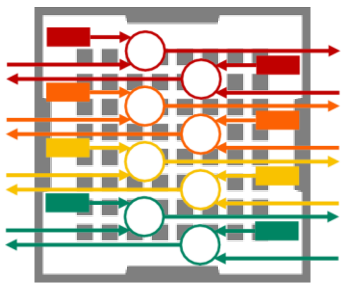
Performance model for each algorithm, tuning.



CUDA Kernels

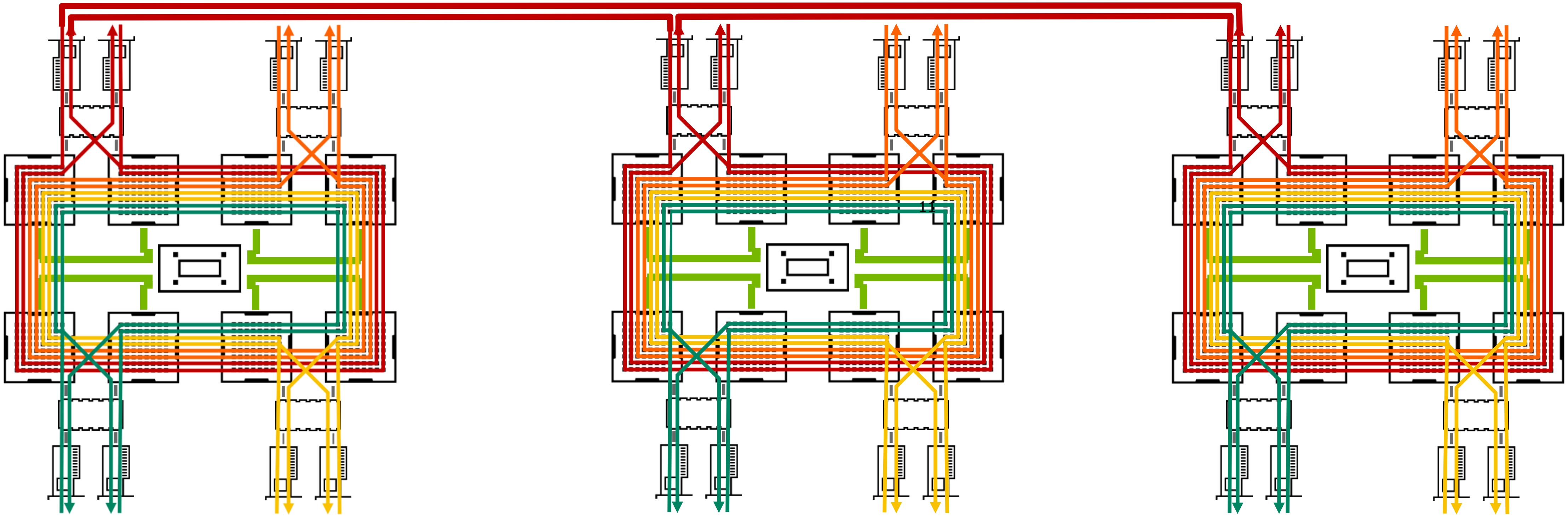
Optimized reductions and copies for a minimal SM usage.

CPU threads for network communication.



Ring algorithm

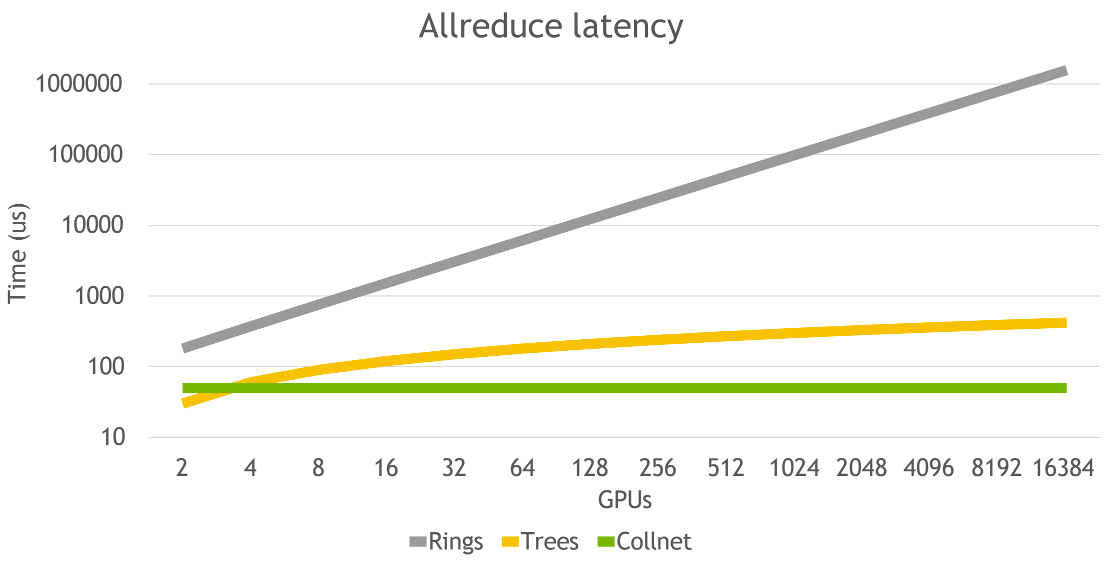
Multi-rings on DGX

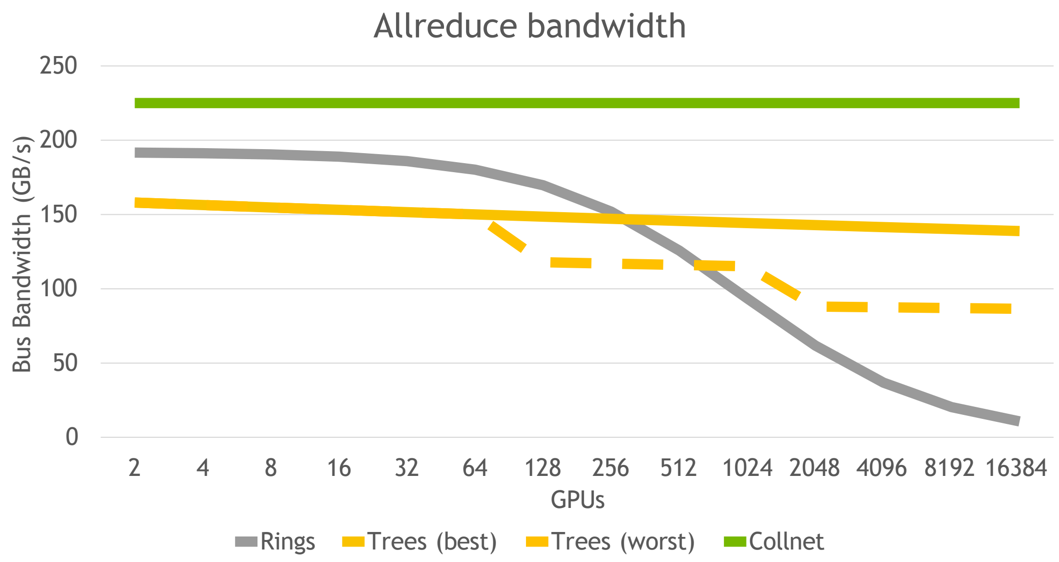


Algorithms Summary

Pros and cons

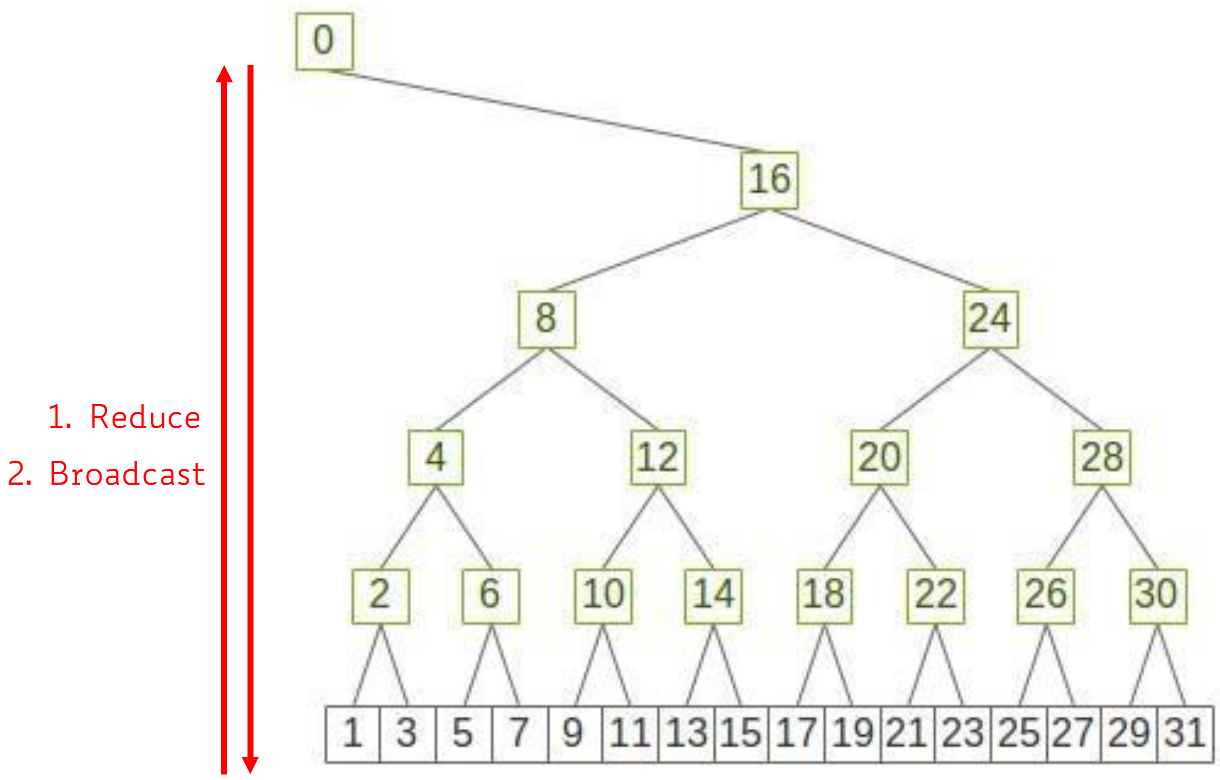
Algorithm	Latency	Bandwidth	Computing	Network Pattern
Rings	Linear	Perfect	Uniform	Few flows
Trees	Log	Close to perfect	Imbalanced	Many flows
Collnet	Constant	Close to 2x (may be limited by NVLink)	Almost uniform	Minimal flows





TREE Algorithm

For allreduce

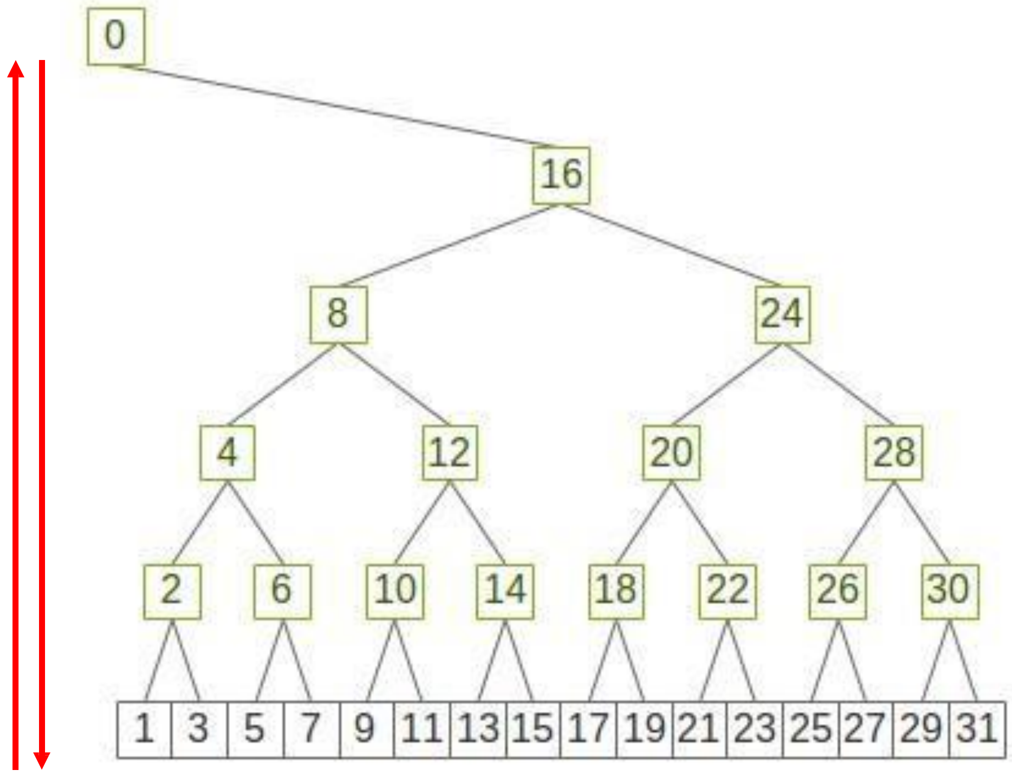


This works if each node has at least three NICs (two to send data to the children, one to send data to the parent)

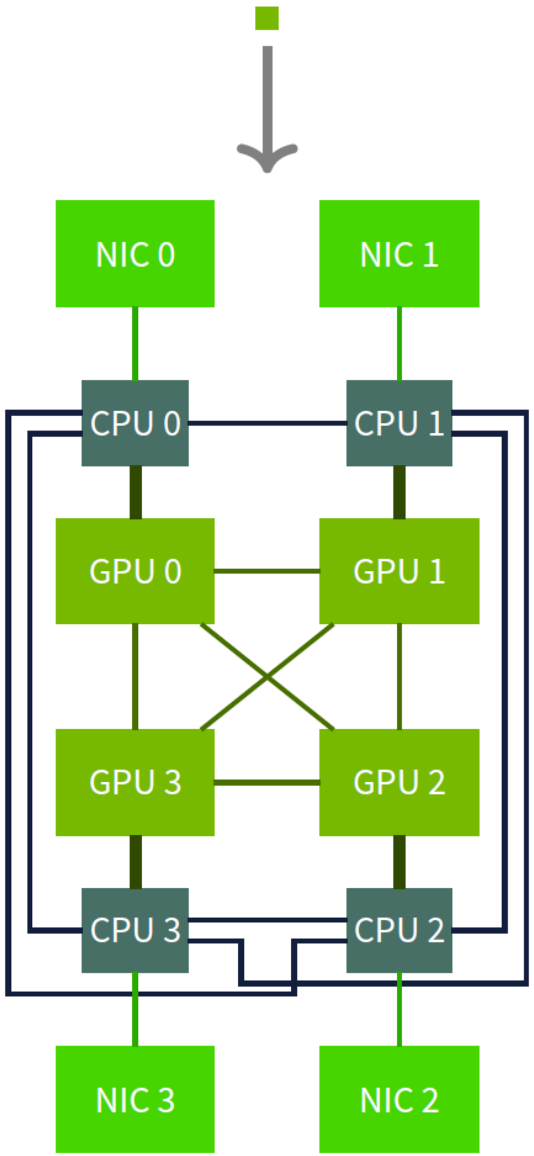
TREE Algorithm

For allreduce

1. Reduce
2. Broadcast

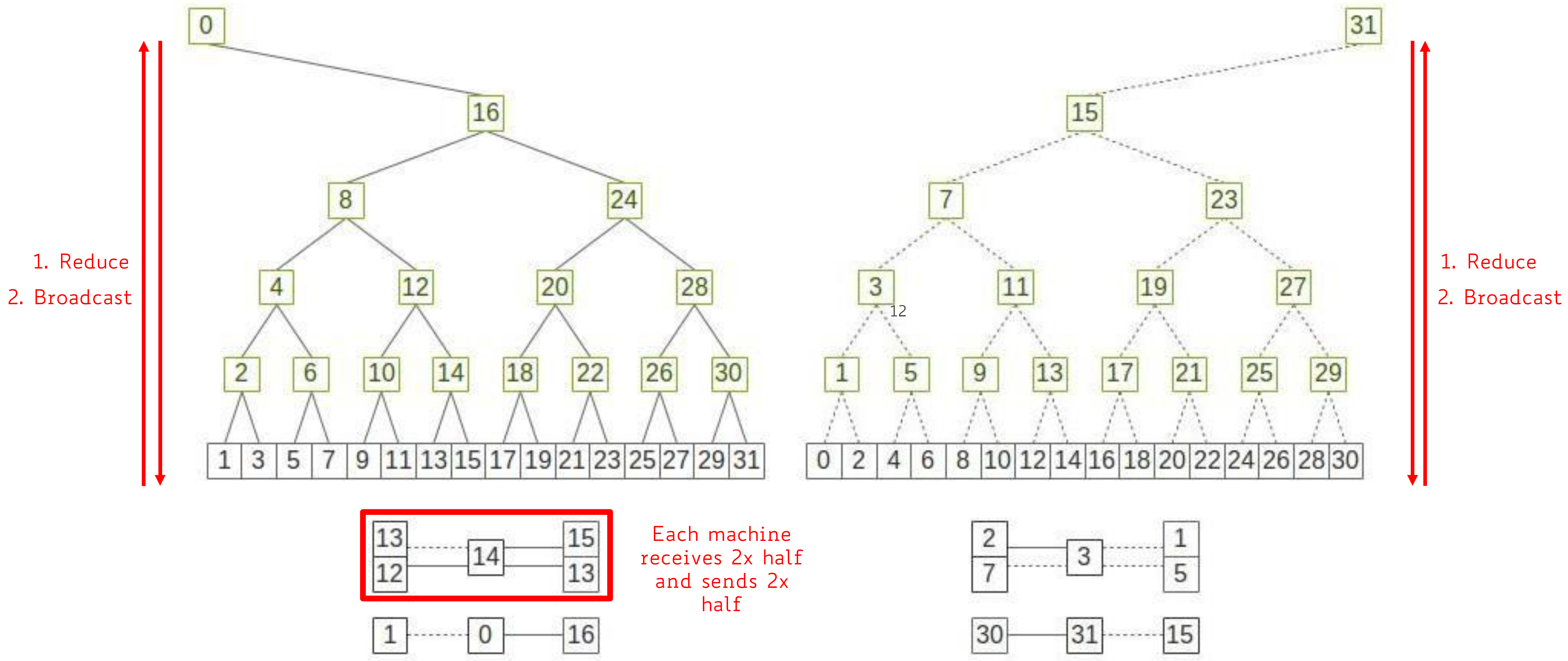


What if each GPU has 4 «NICs» (e.g., the node on the left, GPU 0 is connected to GPU1, GPU2, GPU3, and NIC0)?

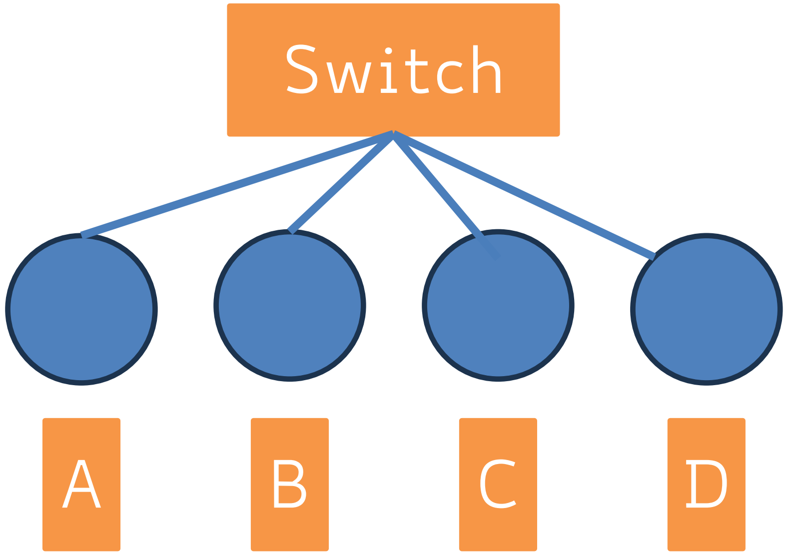


TREE Algorithm

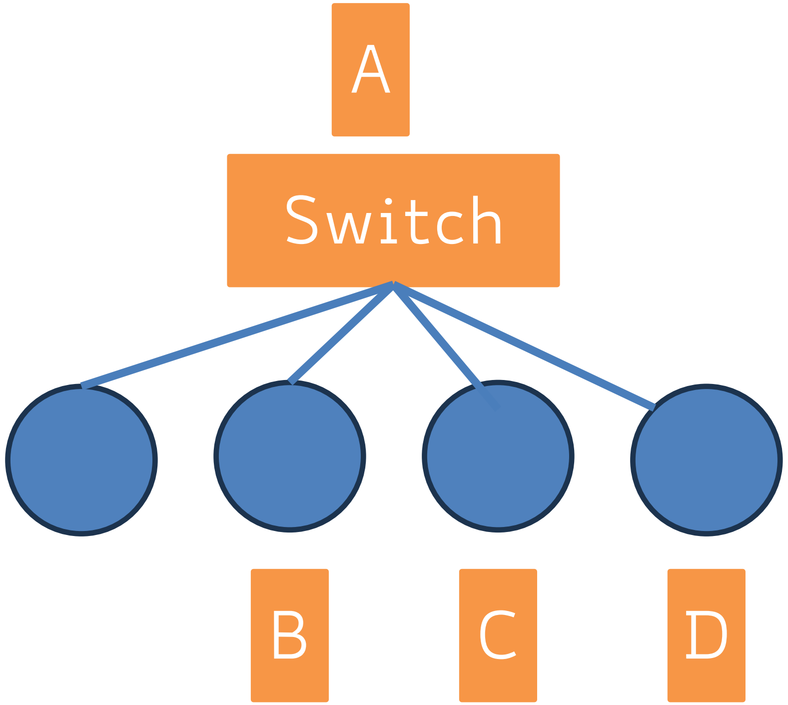
For allreduce



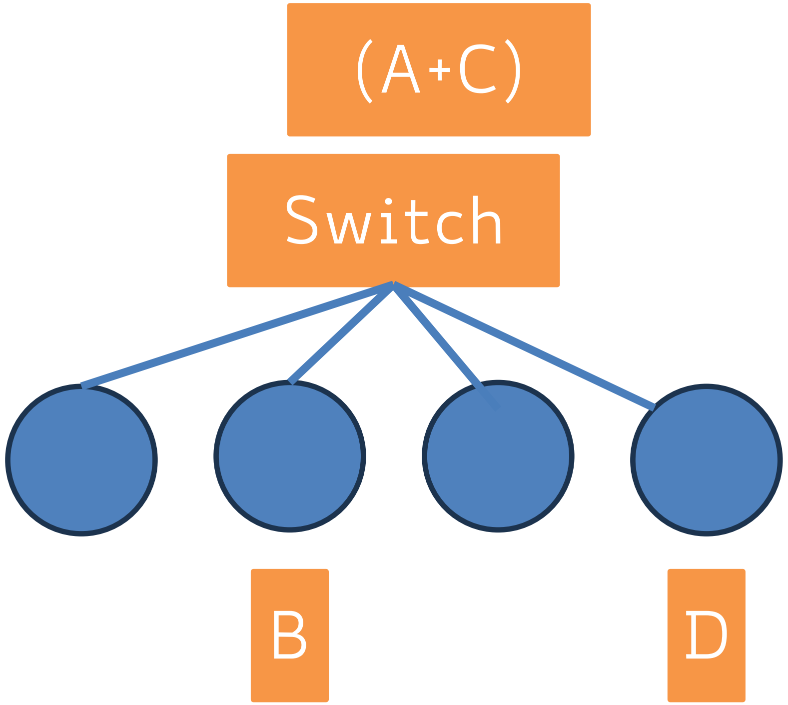
Collnet (In-Network Collectives, e.g. Allreduce)



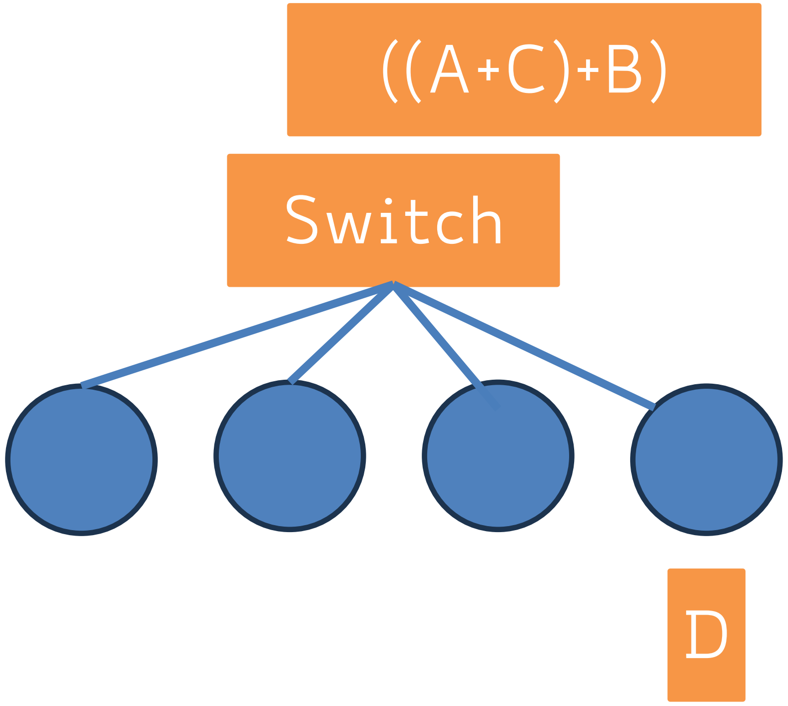
Collnet (In-Network Collectives, e.g. Allreduce)



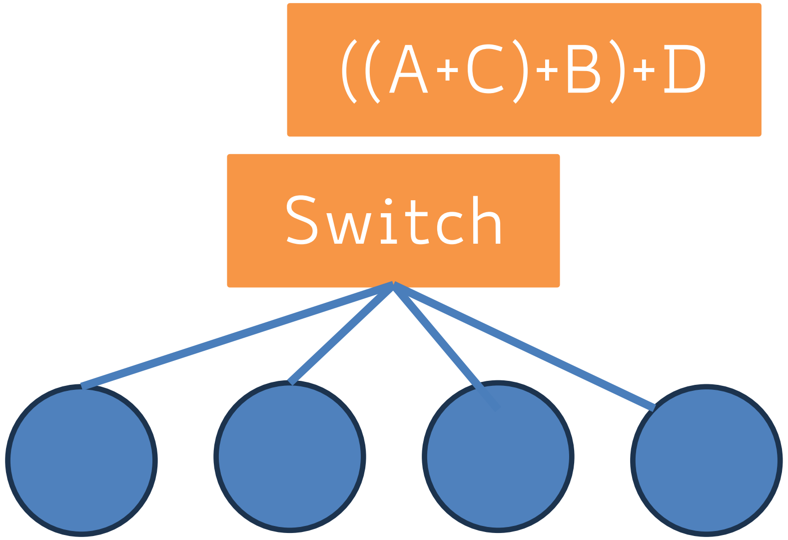
Collnet (In-Network Collectives, e.g. Allreduce)



Collnet (In-Network Collectives, e.g. Allreduce)



Collnet (In-Network Collectives, e.g. Allreduce)

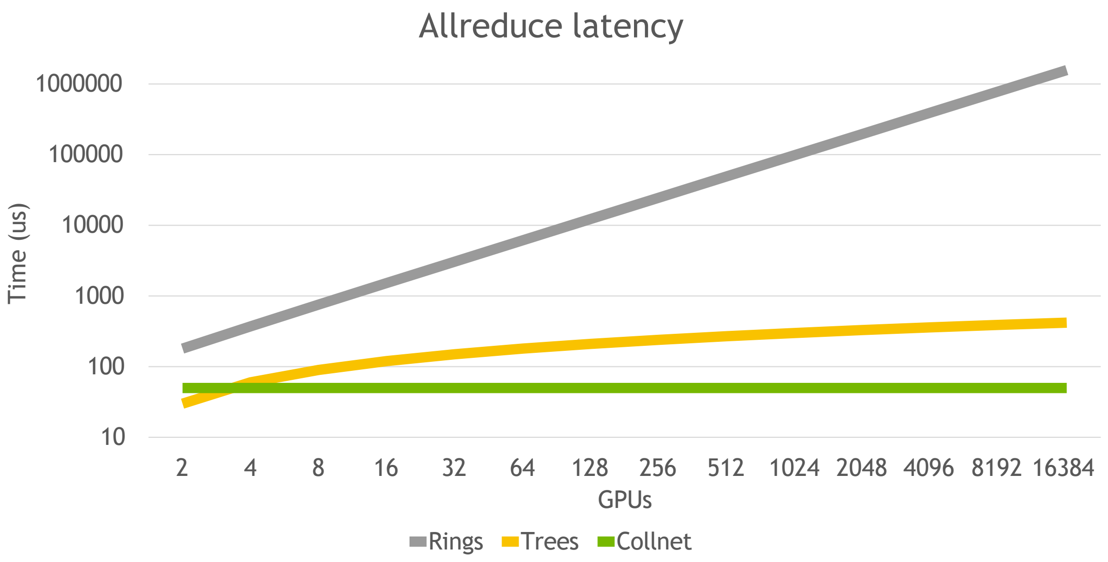


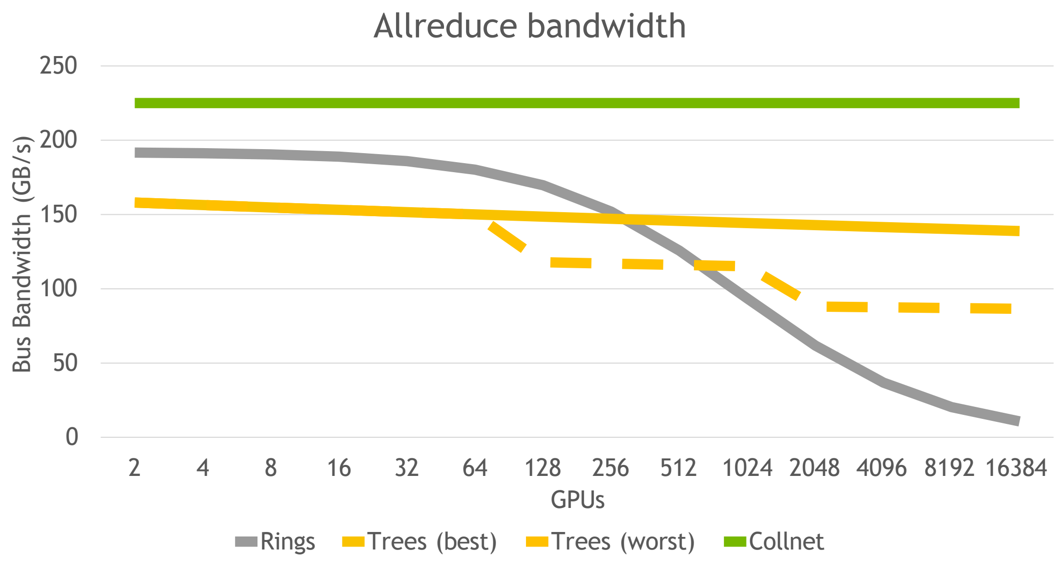
2x Gain wrt. Host-Based Allreduce

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

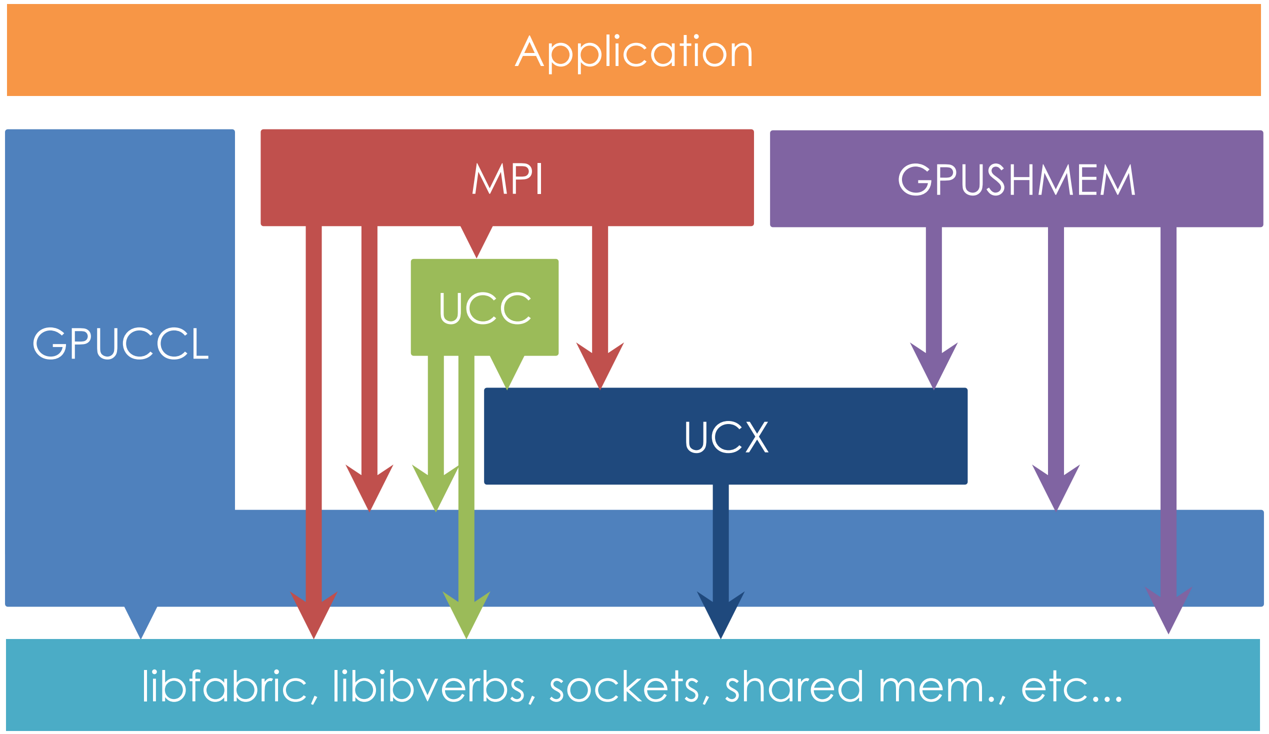




Using NCCL below MPI

Unified Collective Communication (UCC)

- Some MPI implementation can run on top of NCCL natively
- Otherwise, use UCC+NCCL
- 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

Extra References

- Comparison between NCCL and MPI: *"Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects"*
- Details on how NCCL works: *"Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms"*

NCCL Hands-On

https://github.com/FZJ-JSC/tutorial-multi-gpu/

Instructions

- Join the assignment on Github Classroom: https://classroom.github.com/a/cU1ICCEV
  - At the moment, there is no auto-grading yet, check correctness by yourself
- Access the cluster and clone the repository
- Fill the «TODOs» in the jacobi.cu code
- Run "source /home/guest/init-hpc.sh" to load the MPI, CUDA compilers etc...
- Get the GPUs allocation with «salloc --nodes=4 --gpus-per-task=1 --gpus=4 --cpus-per-task=1 --qos=students_limit»
- Run «make run» to compile and run the code (by default on 4 GPUs, one GPU per node)
- When you are done, release the allocation with CTRL-D or «exit»

Solution (pt. 1) – Without NCCL_CALL, NVTX etc for readability

```
while (l2_norm > tol && iter < iter_max) {
    calculate_norm = (iter % nccheck) == 0 || (!csv && (iter % 100) == 0);

    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream);
    cudaEventRecord(reset_l2norm_done, compute_stream);

    cudaStreamWaitEvent(push_stream, reset_l2norm_done, 0);

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

    if (calculate_norm) {
        cudaStreamWaitEvent(compute_stream, push_prep_done, 0);
        cudaMemcpyAsync(l2_norm_h, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream);
    }
```

As for MPI, but we use only one stream for both top and bottom row. Top and bottom rows updates are fast enough that we can serialize them. As for MPI:

- We can't start update top/bottom rows if the l2_norm has not been reset (cudaStreamWaitEvent)
- We have an event to track when the top/bottom rows update finishes so that we can copy back the l2norm from device to host for doing MPI_Allreduce (for the moment we only use NCCL for sending top and bottomr ows)

Solution (pt. 2) – Without NCCL_CALL, NVTX etc for readability

```
    const int top = rank > 0 ? rank - 1 : (size - 1);
    const int bottom = (rank + 1) % size;

    ncclGroupStart();
    ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclGroupEnd();
    cudaEventRecord(push_done, push_stream);

    if (calculate_norm) {
        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);
        }
    }
    cudaStreamWaitEvent(compute_stream, push_done, 0);

    std::swap(a_new, a);
    iter++;
}
```

Wait for the update of the inner domain before doing the allreduce
What about the top/bottom rows? (On compute stream, we are waiting for the memcpyAsync termination, that in turn was waiting for the top/bottom rows update)

Solution (pt. 2) - Without NCCL_CALL, NVTX etc for readability

```
    const int top = rank > 0 ? rank - 1 : (size - 1);
    const int bottom = (rank + 1) % size;

    ncclGroupStart();
    ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclGroupEnd();
    cudaEventRecord(push_done, push_stream);

    if (calculate_norm) {
        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);
        }
    }
    cudaStreamWaitEvent(compute_stream, push_done, 0);

    std::swap(a_new, a);
    iter++;
}
```

Wait for the update of the inner domain before doing the allreduce
What about the top/bottom rows? (On compute stream, we are waiting for the memcpyAsync termination, that in turn was waiting for the top/bottom rows update)

ncclSend/ncclRecv are happening asynchronously (on the GPU), so the next activities on compute_stream (in the next iteration) should not start before the ncclSend/Recv are completed.
(otherwise I could update the domain before it is sent/recvd)

Performance (streams+priority)

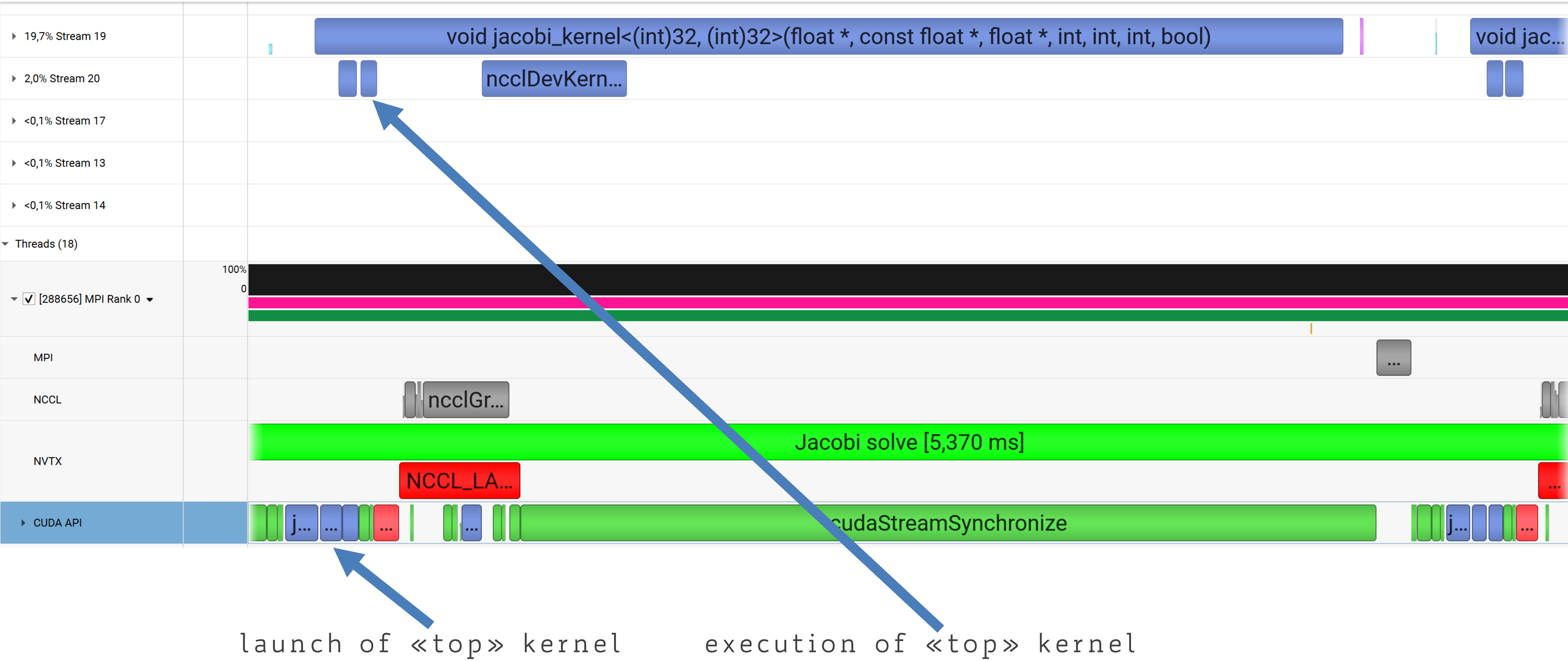
MPI:

```
Num GPUs: 4.
8192x8192: 1 GPU:   1.5205 s, 4 GPUs:   0.4006 s, speedup:     3.80, efficiency:    94.90
```

NCCL:

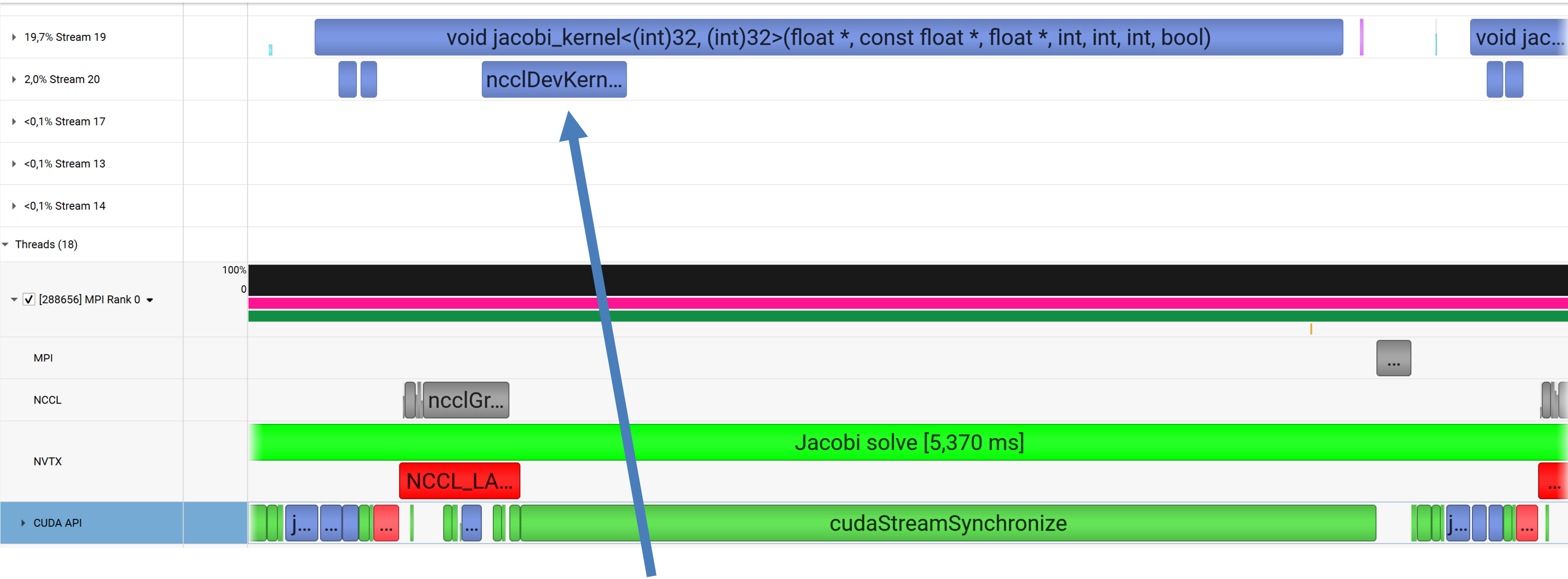
```
Num GPUs: 4.
8192x8192: 1 GPU:   1.5212 s, 4 GPUs:   0.4091 s, speedup:     3.72, efficiency:    92.97
```

Profile



19,7% Stream 19
2,0% Stream 20
<0,1% Stream 17
<0,1% Stream 13
<0,1% Stream 14
Threads (18)
[288656] MPI Rank 0
100%
0
MPI
NCCL
NVTX
CUDA API
void jacobi_kernel<(int)32, (int)32>(float *, const float *, float *, int, int, int, bool)
void jac...
ncclDevKern...
...
ncclGr...
Jacobi solve [5,370 ms]
NCCL_LA...
...
j...
...
...
...
cudaStreamSynchronize
j...
...
launch of «top» kernel
execution of «top» kernel

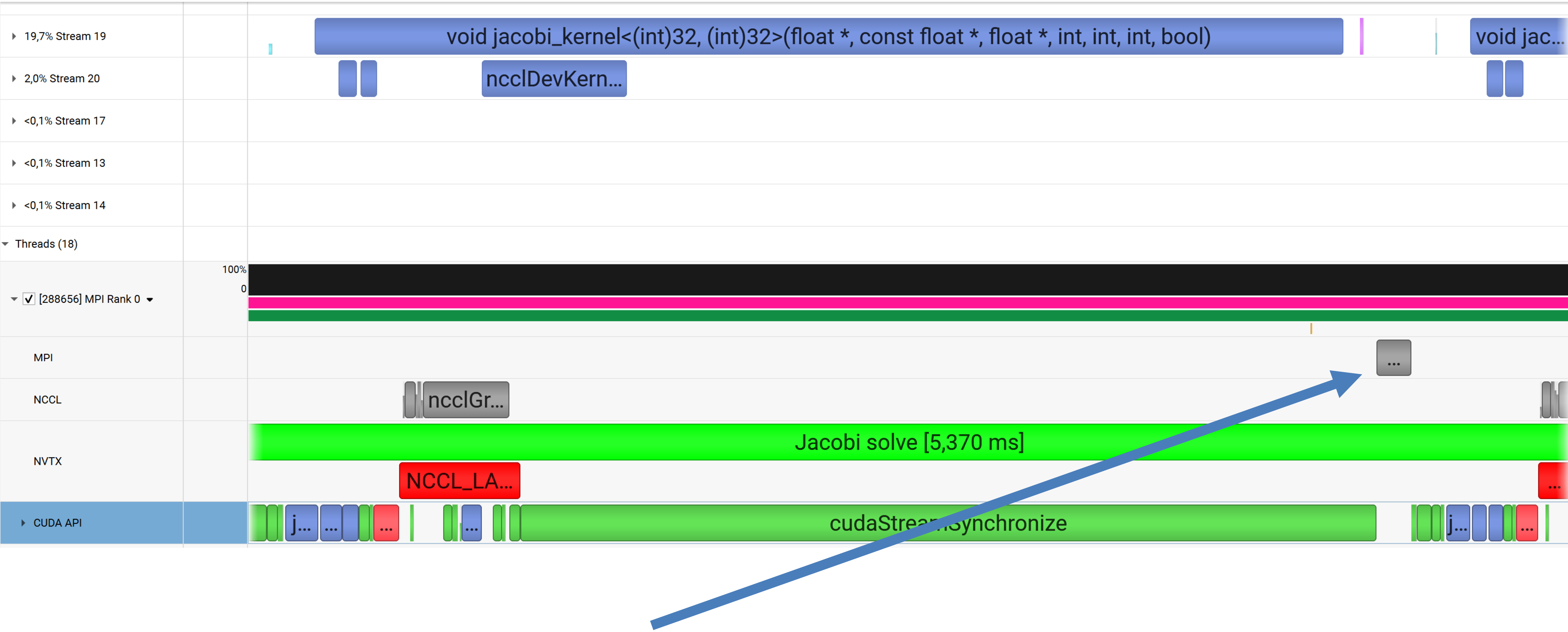
Profile



ncclDevKernel_SendRecv

(communication ran on device as soon as data is ready, without the need to syncrhonize with the host)

Profile



Only MPI communication happening is the Allreduce

Using ncclAllReduce

```
    const int top = rank > 0 ? rank - 1 : (size - 1);
    const int bottom = (rank + 1) % size;

    ncclGroupStart();
    ncclRecv(a_new,                     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclSend(a_new + (iy_end - 1) * nx, nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclRecv(a_new + (iy_end * nx),     nx, NCCL_REAL_TYPE, bottom, nccl_comm, push_stream);
    ncclSend(a_new + iy_start * nx,     nx, NCCL_REAL_TYPE, top,    nccl_comm, push_stream);
    ncclGroupEnd();
    cudaEventRecord(push_done, push_stream);

    if (calculate_norm) {
        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);
        }
    }
    cudaStreamWaitEvent(compute_stream, push_done, 0);

    std::swap(a_new, a);
    iter++;
}
```

Using ncclAllReduce – Solution (pt. 1)

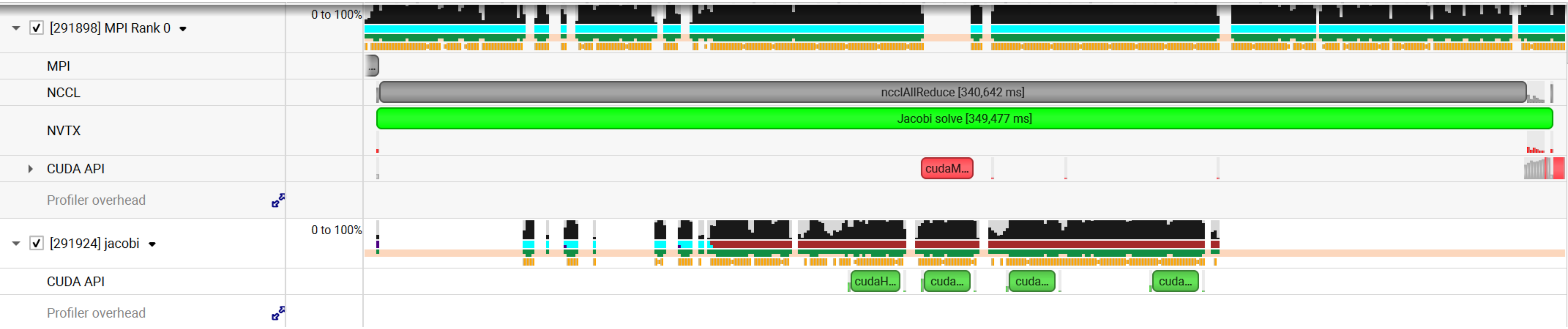
```
            launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_start + 1), (iy_end - 1), nx, calculate_norm,
                                 compute_stream);

            launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, (iy_start + 1), nx, calculate_norm,
                                 push_stream);

            launch_jacobi_kernel(a_new, a, l2_norm_d, (iy_end - 1), iy_end, nx, calculate_norm,
                                 push_stream);

            if (calculate_norm){
                CUDA_RT_CALL(cudaEventRecord(push_prep_done, push_stream));
                CUDA_RT_CALL(cudaStreamWaitEvent(compute_stream, push_prep_done, 0));
                NCCL_CALL(ncclAllReduce(l2_norm_d, l2_norm_d, 1, NCCL_REAL_TYPE, ncclSum, nccl_comm, compute_stream)); // In-place allreduce
                CUDA_RT_CALL(cudaMemcpyAsync(l2_norm, l2_norm_d, sizeof(real), cudaMemcpyDeviceToHost, compute_stream));
            }
```

Using ncclAllReduce – Solution (pt. 2)

```
        if (calculate_norm){
            CUDA_RT_CALL(cudaStreamSynchronize(compute_stream));
            *l2_norm = std::sqrt(*l2_norm);

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

Using ncclAllReduce – First attempt

```
Num GPUs: 4.
8192x8192: 1 GPU:   1.5250 s, 4 GPUs:   0.9953 s, speedup:     1.53, efficiency:    38.30
```



ncclAllReduce taking way too long the first time is called. Why?

Probably topology discovery to optimize allreduce calls

Using ncclAllReduce - Second attempt, Warmup Done

```
Num GPUs: 4.
8192x8192: 1 GPU:   1.5624 s, 4 GPUs:   0.7135 s, speedup:     2.19, efficiency:    54.74
```

```
Num GPUs: 4.
8192x8192: 1 GPU:   1.5326 s, 4 GPUs:   0.5322 s, speedup:     2.88, efficiency:    72.00
```

Why?

We are launching a kernel for doing an allreduce on a single value (the norm)
It might be faster to copy that value to the host and run an MPI_Allreduce

Questions?

NVSHMEM

Slide courtesy: Lena Oden & Markus Hrywniak

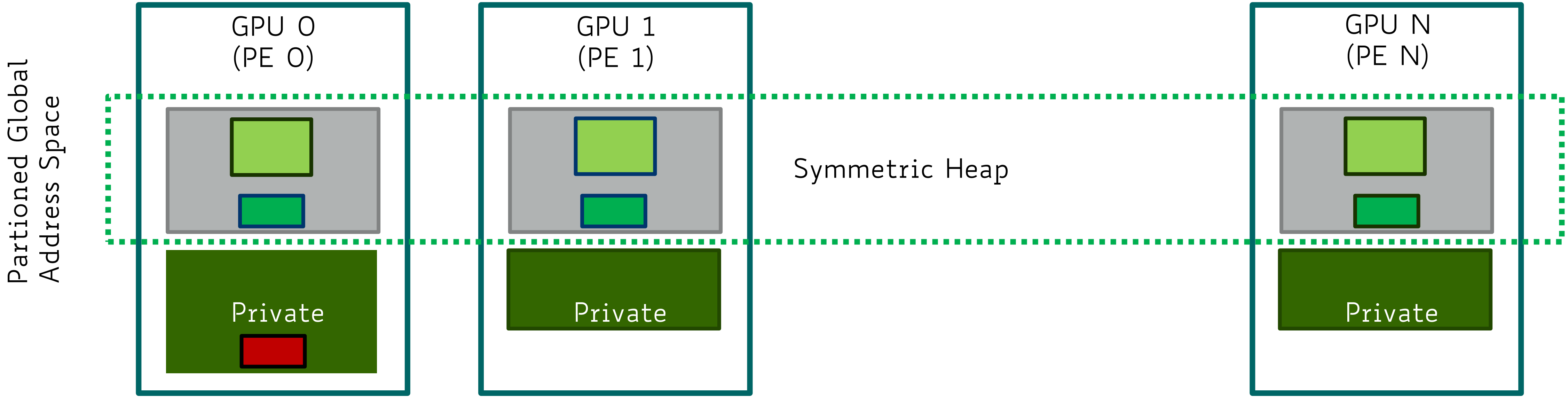
NVSHMEM – Overview

- Implements the OpenSHMEM API for clusters of NVIDIA GPUs
- Partitioned Global Address Space (PGAS) programming model
    - One sided Communication with put/get
    - Shared memory Heap

With some extensions to the API

- GPU Centric communication APIs
    - GPU Initiated: thread, warp, block
    - Stream/Graph-Based (communication kernel or cudaMemcpyAsync)
    - CPU Initiated
- prefixed with "*nvshmem*" to allow use with a CPU OpenSHMEM library
- Interoperability with OpenSHMEM and MPI

NVSHMEM Symmetric Memory Model



Symmetric objects are allocated collectively with the same size on every PESymmetric memory: nvshmem_malloc(shared_size);

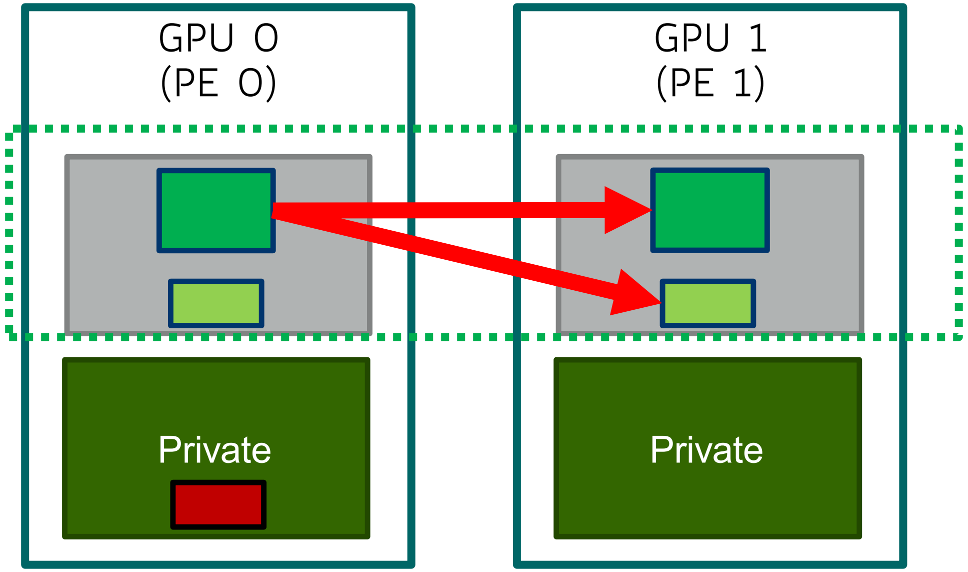
Must be
the same
on all PEs

Private memory:
cudaMalloc(...)

Interoperability with MPI and OpenSHMEM

```
MPI_Init(&argc, &argv);
MPI_Comm mpi_comm = MPI_COMM_WORLD;
nvshmemx_init_attr_t attr;
attr.mpi_comm& = mpi_comm;
nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, &attr);
assert( size == nvshmem_n_pes() );
assert( rank == nvshmem_my_pe() );
…
nvshmem_finalize();
MPI_Finalize();
```

NVSHMEM Host API Put



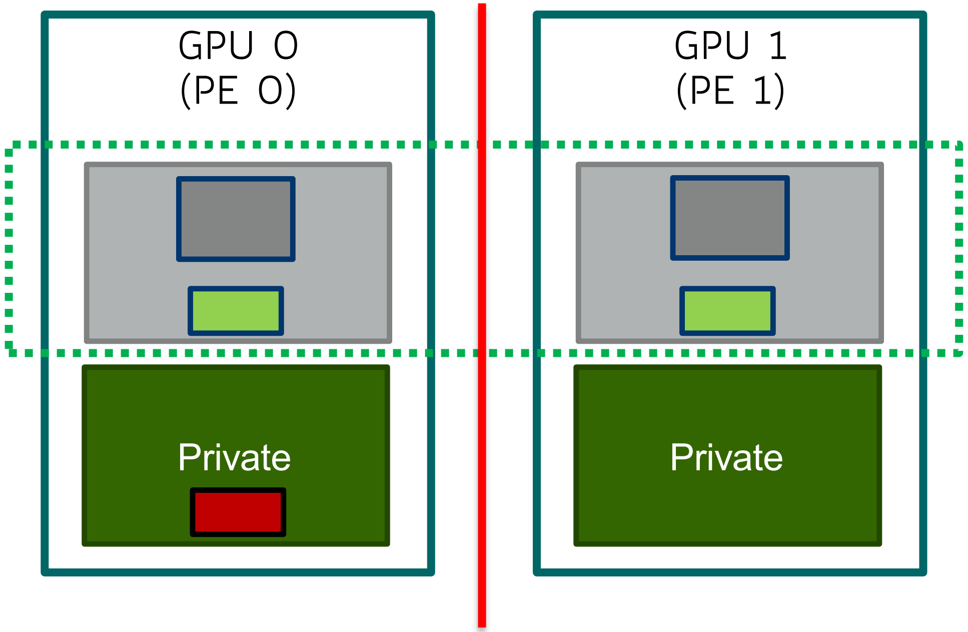
copies *nelems* data elements of type T from symmetric objects *src* to *dest* on PE *pe*

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

The x marks extensions to the OpenSHMEM API

ATTENTION: Are asynchronous calls, when they return, I do not know if data was written in remote memory already

NVSHMEM Barrier (on Host)



Synchronizes all PEs and ensures communication performed prior to the barrier has completed

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

Jacobi with NVSHMEM

Chunk size must me the same on all PEs. Otherwise, you get Undefined Behavior!

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

Jacobi with NVSHMEM

Source
- We should send the second row (the first one is the halo)
- iy_start=1, nx is the number of columns -> a_new + nx

```
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, compute_stream);
nvshmemx_float_put_on_stream(a_new + iy_top_lower_boundary_idx * nx, a_new + iy_start * nx, nx, top,
compute_stream);
nvshmemx_float_put_on_stream(a_new, a_new + (iy_end - 1) * nx, nx, bottom, compute_stream);
nvshmemx_barrier_all_on_stream(compute_stream);
```

We are writing to top neighbor's memory

Destination
- We should write in destination's halo (last row) – **IMPORTANT: We always refer to local buf**
- iy_top_lower_boundary_idx is the number of rows assigned to each rank, plus 1
- A bit complicated to manage the case where the number of rows cannot be divided evenly by the number of ranks (not all ranks have the same number of rows assigned)
- For simplicity, you can force the number of rows to be divisible by the number of ranks

Jacobi with NVSHMEM

Source
- We should send the second-last row (the last one is the halo)

```
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, compute_stream);
nvshmemx_float_put_on_stream(a_new + iy_top_lower_boundary_idx * nx, a_new + iy_start * nx, nx, top,
compute_stream);
nvshmemx_float_put_on_stream(a_new, a_new + (iy_end - 1) * nx, nx, bottom, compute_stream);
nvshmemx_barrier_all_on_stream(compute_stream);
```

We are writing to bottom neighbor's memory

Destination
- We should write in destination's halo (first row)
- **IMPORTANT: we always refer to local buf**

Jacobi with NVSHMEM

Use high priority on push_stream

```
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, c_stream);

nvshmemx_float_put_on_stream(a_new + iy_top_lower_boundary_idx * nx, a_new + iy_start * nx, top, push_stream);
nvshmemx_float_put_on_stream(a_new + iy_bottom_upper_boundary_idx * nx, a_new + (iy_end - 1) * nx, nx, bottom,
push_stream);
nvhmemx_barrier_all_on_stream(push_stream);
```

How to compile NVSHEM + MPI applications

- Compile CUDA-kernel
  - Use the -rdc=true compile flag due to the device interface
  - Link againt the nvshmem libray -lnvshmem

```
#include <nvshmem.h>
#include <nvshmemx.h>
```

```
nvcc -rdc=true -ccbin g++ -gencode=$NVCC_GENCODE -I $NVSHMEM_HOME/include nvshmem_hello.cu
-o nvshmem_hello  -L $NVSHMEM_HOME/lib -lnvshmem -lcuda
```

```
nvcc -rdc=true -ccbin g++ -gencode=$NVCC_GENCODE -I $NVSHMEM_HOME/include -c jacobi_kernels.cu –o jacobi_kernels.o

$mpixx  -I $NVSHMEM_HOME/include  jacobi.cpp jacobi_kernels.o –lnvshmem –lcuda  -o jacobi
```

Summary

- NCCL and NVSHMEM support CUDA stream aware communication
- Both are interoperable with MPI
- NCCL support send/receive semantics
- NVSHMEM supports the OpenSHMEM library, supporting one sided communication operation
- Both allow to issue communication request asynchronous with respect to the CPU-thread, but synchronous to CUDA streams
- High priority streams are required to overlap communication and computation

NVSHMEM Hands-On

https://github.com/FZJ-JSC/tutorial-multi-gpu/

Instructions

- Join the assignment on Github Classroom: https://classroom.github.com/a/rsxvVcwP
  - At the moment, there is no auto-grading yet, check correctness by yourself
- Access the cluster and clone the repository
- Fill the «TODOs» in the jacobi.cu code
- Run "source /home/guest/init-hpc.sh" to load the MPI, CUDA compilers etc…
- Get the GPUs allocation with «salloc --nodes=4 --gpus-per-task=1 --gpus=4 --cpus-per-task=1 --qos=students_limit»
- Run «make run» to compile and run the code (by default on 4 GPUs, one GPU per node)
- When you are done, release the allocation with CTRL-D or «exit»