



NCCL and Host-Initiated NVSHMEM

Jiri Kraus, Principal Devtech Compute | SC23/November 13th 2023

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 pt2pt communication
- NVSHMEM memory model
- How to use stream-aware NVSHMEM communication operations in MPI Programs

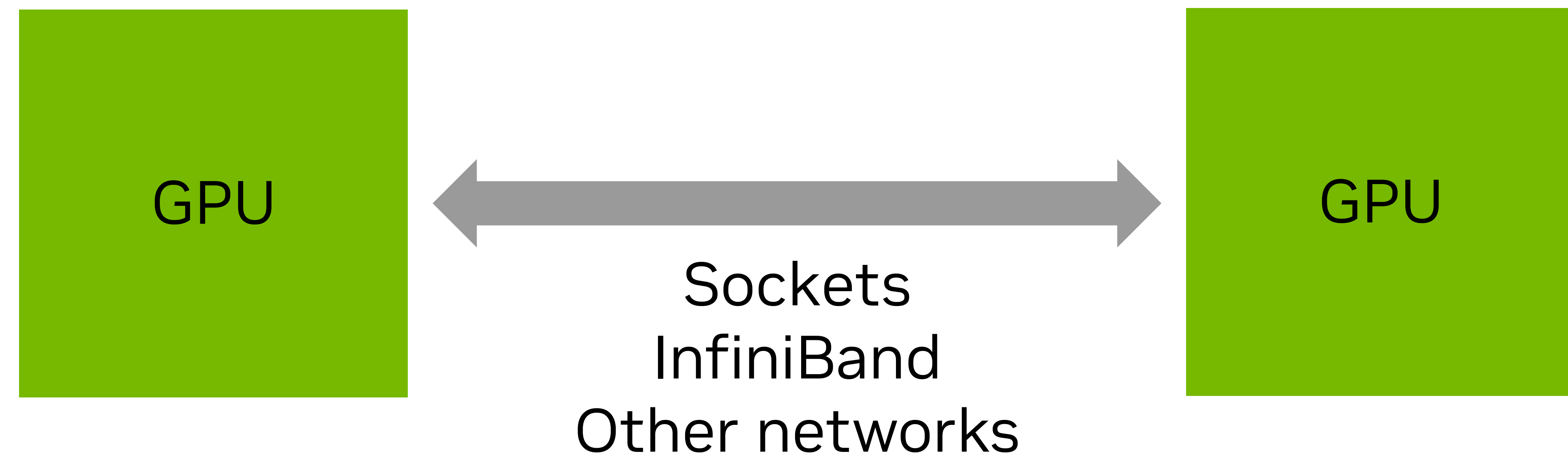
[1] MPI Forum Hybrid Working Group - Stream and Graph Based MPI Operations: <https://github.com/mpiwg-hybrid/hybrid-issues/issues/5>

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 Deep Learning
- Since 2.8: Support for Send/Recv between GPUs
- Library running on GPU: Communication calls are translated to a GPU 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

Initialization and Teardown with MPI

```
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();
```

NCCL API

Communication

- 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);  
ncclAllGather(void* sbuff, void* rbuff, size_t count, ncclDataType_t type,  
              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);
```


NCCL API

Fused Communication Calls

- Multiple calls to `ncc1Send()` and `ncc1Recv()` should be fused with `ncc1GroupStart()` and `ncc1GroupEnd()` to
 - Avoid deadlocks, e.g. if calls need to progress concurrently
 - For more performance: fused operations can be more efficient by better utilizing the available IO

Send/Recv

```
ncc1GroupStart();  
ncc1Send(sendbuff, sendcount, sendtype, peer, comm, stream);  
ncc1Recv(recvbuff, recvcount, recvtype, peer, comm, stream);  
ncc1GroupEnd();
```

Bcast:

```
ncc1GroupStart();  
if (rank == root) {  
    for (int r=0; r<n ranks; r++)  
        ncc1Send(sendbuff[r], size, type, r, comm, stream);  
}  
ncc1Recv(recvbuff, size, type, root, comm, stream);  
ncc1GroupEnd();
```

NCCL API

Jacobi Solver Communication

```
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, btm, nccl_comm, compute_stream);
ncclRecv(a_new + (iy_end * nx), nx, NCCL_REAL_TYPE, btm, nccl_comm, compute_stream);
ncclSend(a_new + iy_start * nx, nx, NCCL_REAL_TYPE, top, nccl_comm, compute_stream);
ncclGroupEnd();
```

NCCL

Overlapping Communication and Computation

- 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_stream, cudaStreamDefault, greatestPriority);
.
```


NCCL

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, btm, nccl_comm, push_stream);
ncclRecv(a_new + (iy_end * nx),      nx, NCCL_REAL_TYPE, btm, nccl_comm, push_stream);
ncclSend(a_new + iy_start * nx,      nx, NCCL_REAL_TYPE, top, nccl_comm, push_stream);
ncclGroupEnd();
```

NCCL

Compiling MPI+NCCL Applications

Include the NCCL header file and link against NCCL

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

```
MPICXX_FLAGS = -I$(CUDA_HOME)/include -I$(NCCL_HOME)/include
```

```
LD_FLAGS = -L$(CUDA_HOME)/lib64 -lcudart -lnccl
```

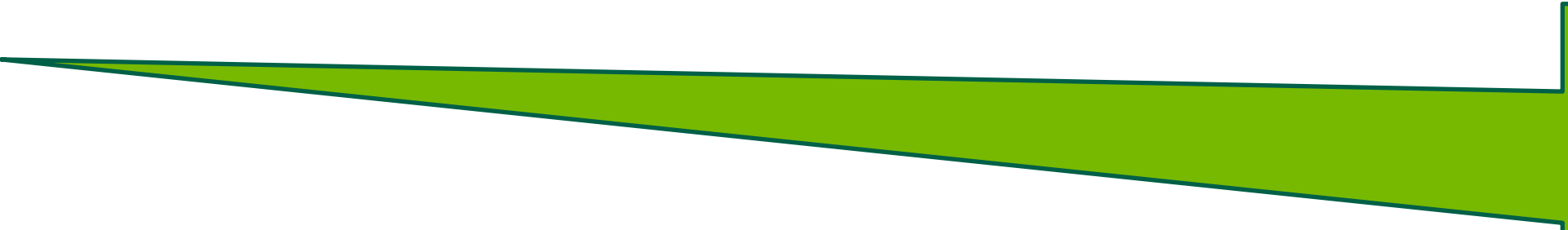
```
$(NVCC) $(NVCC_FLAGS) jacobi_kernels.cu -c -o jacobi.o
```

```
$(MPICXX) $(MPICXX_FLAGS) jacobi.cpp jacobi_kernels.o $(LD_FLAGS) -o jacobi
```


NVSHMEM

Overview

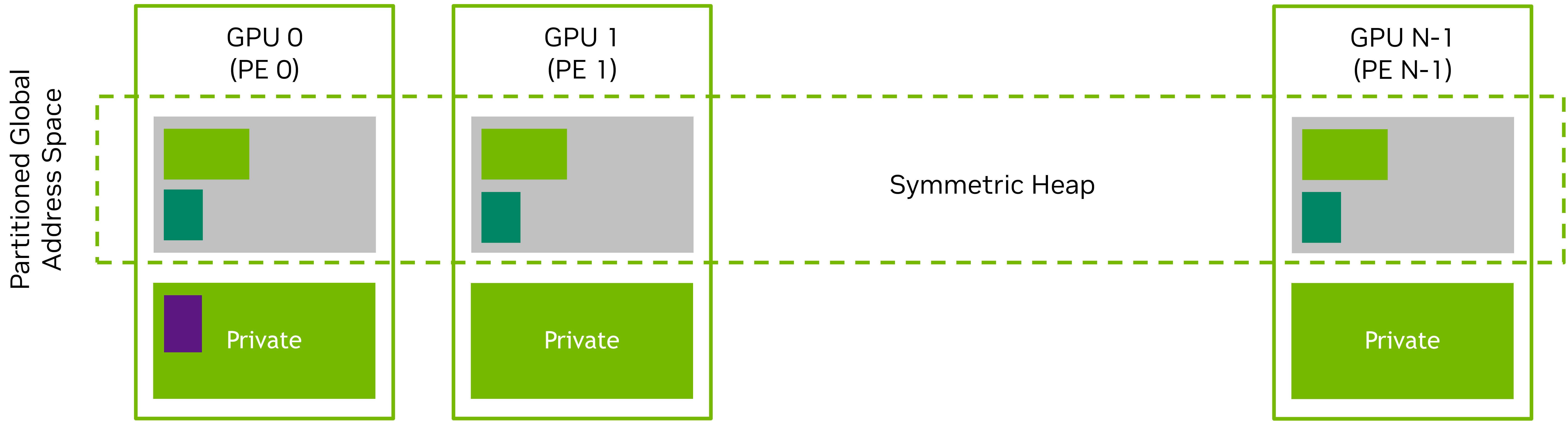
- Implements the OpenSHMEM API for clusters of NVIDIA GPUs
- Partitioned Global Address Space (PGAS) programming model
 - One sided Communication with put/get
 - Shared memory Heap
- GPU Centric communication APIs
 - GPU Initiated: thread, warp, block
 - CPU Initiated: Stream/Graph-Based (communication kernel or cudaMemcpyAsync)
- prefixed with “nvshmem” to allow use with a CPU OpenSHMEM library
- Interoperability with OpenSHMEM and MPI



With some
extensions to
the API

NVSHMEM

Memory Model



Symmetric objects are allocated collectively with the same size on every PE

- Symmetric memory: `nvshmem_malloc(shared_size);`
- Private memory: `cudaMalloc(...)`

Must be the
same on all PEs

NVSHMEM API

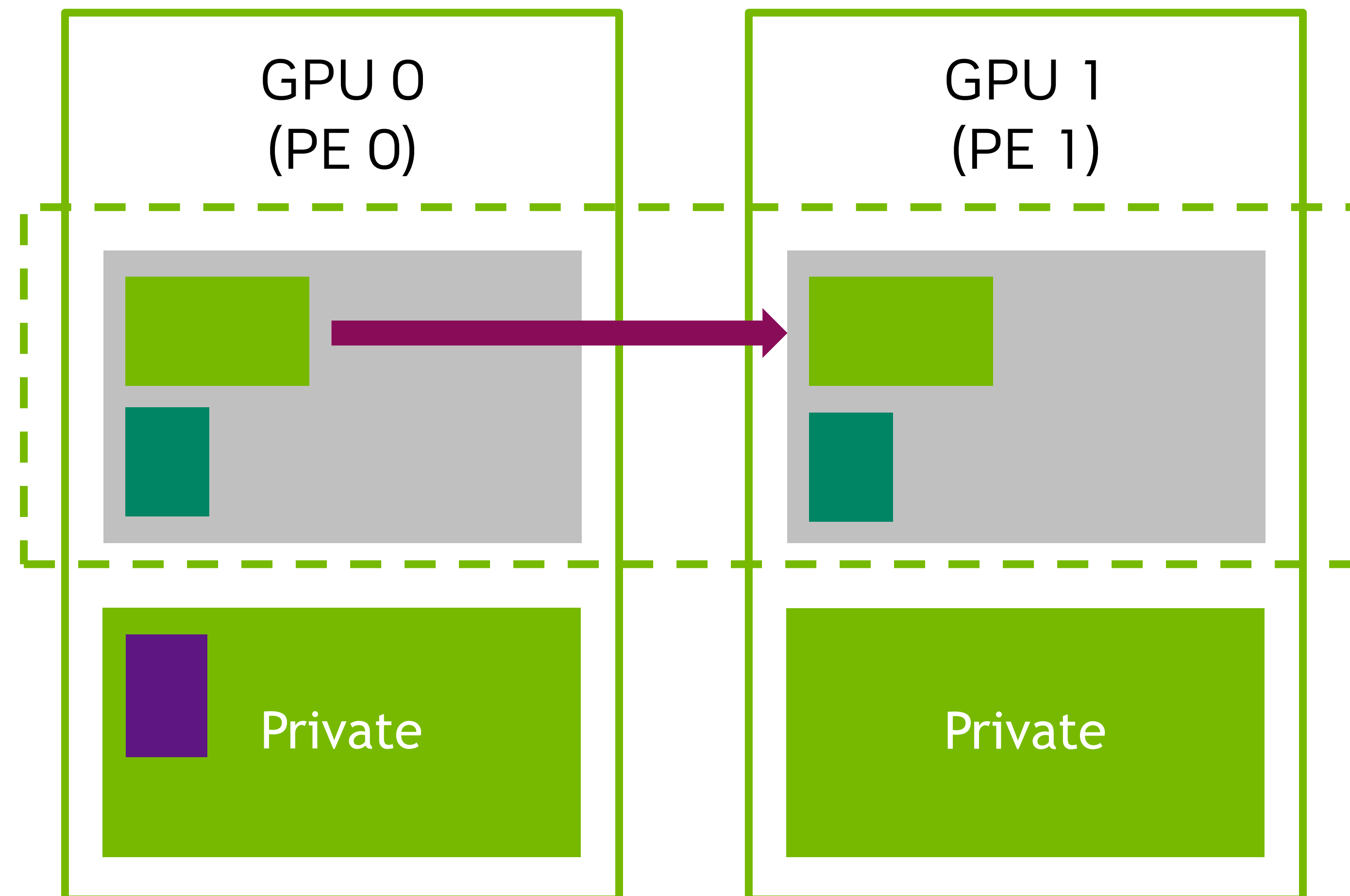
Interoperability with MPI and OpenSHMEM

```
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();
```

```
shmem_init();
nvshmemx_init_attr_t attr;
nvshmemx_init_attr(NVSHMEMX_INIT_WITH_SHMEM, &attr);
my_pe_node = nvshmem_team_my_pe(NVSHMEMX_TEAM_NODE);
```

NVSHMEM API

Host Put



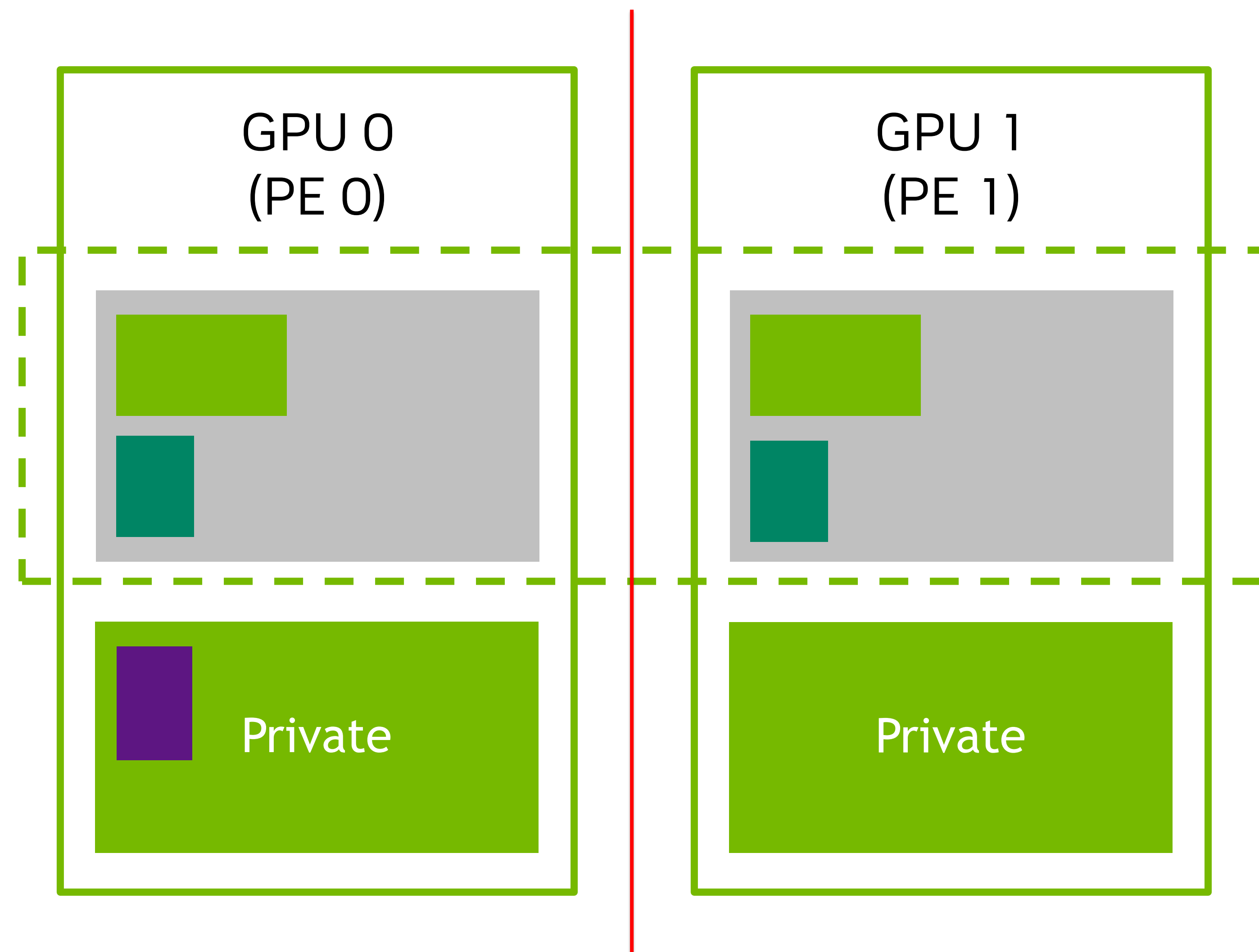
Copies `nelems` data elements of type `T` from symmetric object `src` to `dest` on PE `pe`

```
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

NVSHMEM API

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)
```

NVSHMEM API

Jacobi Solver

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

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

```
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start,      iy_end,      nx, compute_stream);
```

```
nvshmemx_float_put_on_stream(a_new,      a_new + (iy_end - 1) * nx, nx, btm, compute_stream);  
nvshmemx_float_put_on_stream((a_new+iy_end)*nx, (a_new+1)*nx,      nx, top, compute_stream);  
nvshmemx_barrier_all_on_stream(compute_stream);
```


NVSHMEM API

Jacobi Solver with Communication Computation overlap

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



Use high priority stream!

```
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,                      a_new + (iy_end - 1) * nx, nx, btm, push_stream);  
nvshmemx_float_put_on_stream((a_new+iy_end)*nx, (a_new+1)*nx, nx, top, push_stream);  
nvshmemx_barrier_all_on_stream(push_stream);
```

NVSHMEM

Compiling MPI+NVSHMEM Applications

Include the NVSHMEM header files

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

Compile with `-rdc=true` and link against the NVSHMEM library `-lnvshmem`

```
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 might be required to overlap communication and computation



Backup

NCCL

User Buffer Registration

- NCCL uses library managed GPU resident staging buffers for any network or inter process communication.
 - Staging avoids overheads for pinning and registering of user allocations and restrictions caused by that. NCCLs only requirement for user allocations is that the memory is accessible by the GPU. I.e. it can be pinned CPU (cudaMallocHost) or GPU (cudaMalloc) memory, unified (malloc on HMM or coherent systems) or managed (cudaMallocManaged) memory
 - Staging enables optimizations like PXN improving all to all and other communication patterns on rail optimized networks: <https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/>
 - In most cases the necessary required writes and reads of the staging buffers are not limiting the performance

- With version 2.19 NCCL introduced user buffer registration

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

(see <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/api/comms.html#ncclcommregister>)

Enabling applications to register the buffers passed into NCCL APIs to avoid staging buffers for pinned GPU memory (cudaMalloc) when they are a performance bottleneck:

- When using NVLINK SHARP (2.19)
- Specifically for communication not requiring computation avoiding staging buffer reads/writes improves performance when NCCL is overlap with memory bandwidth bound compute kernels (planned for 2.20)
- For small transfers added latency for staging can matter (planned for 2.20)