



Fakultät für
**Mathematik und
Informatik**

NCCL and Host-Initiated NVSHMEM

Lena Oden,
FernUniversität Hagen

So far we learned about

- 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

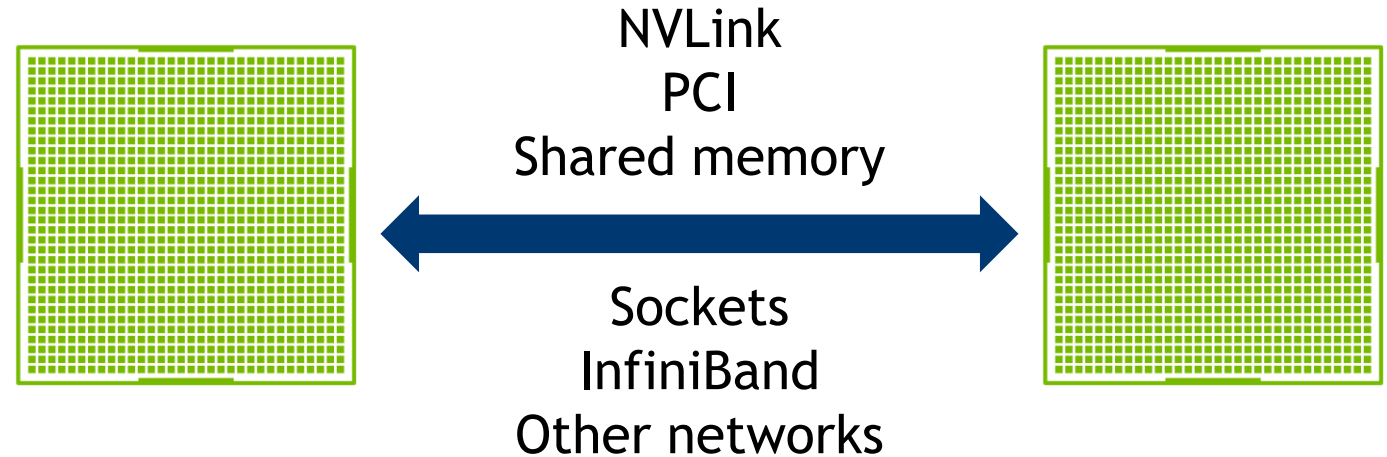
[1] MPI Forum Hybrid Working Group - Stream and Graph Based MPI Operations:
<https://github.com/mpiwg-hybrid/hybrid-issues/issues/5>

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, we 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

- Send/Recv

Supported for
NCCL 2.8+

```
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)
 - For more performance (can be more efficiently)

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<n ranks; 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();
```

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));
```

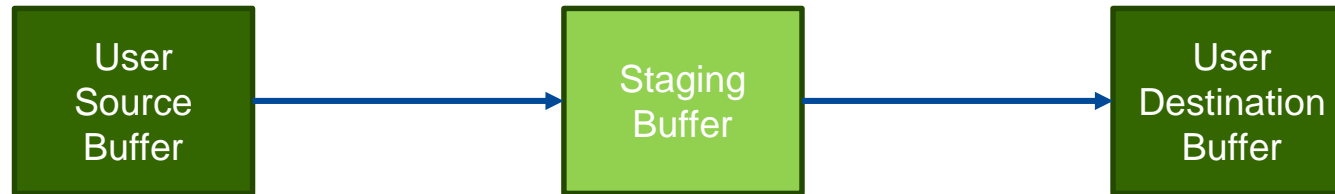
Jacobi using NCCL and Overlapping Communication and Computation

```
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
```

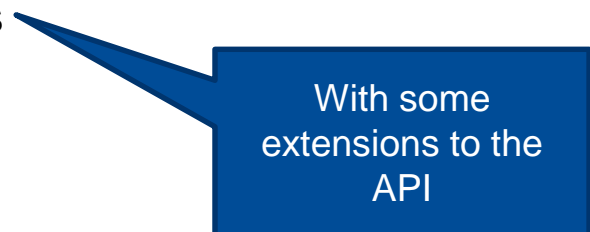
```
LD_FLAGS = -L$(CUDA_HOME)/lib64 -lcudart -lncccl
```

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

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

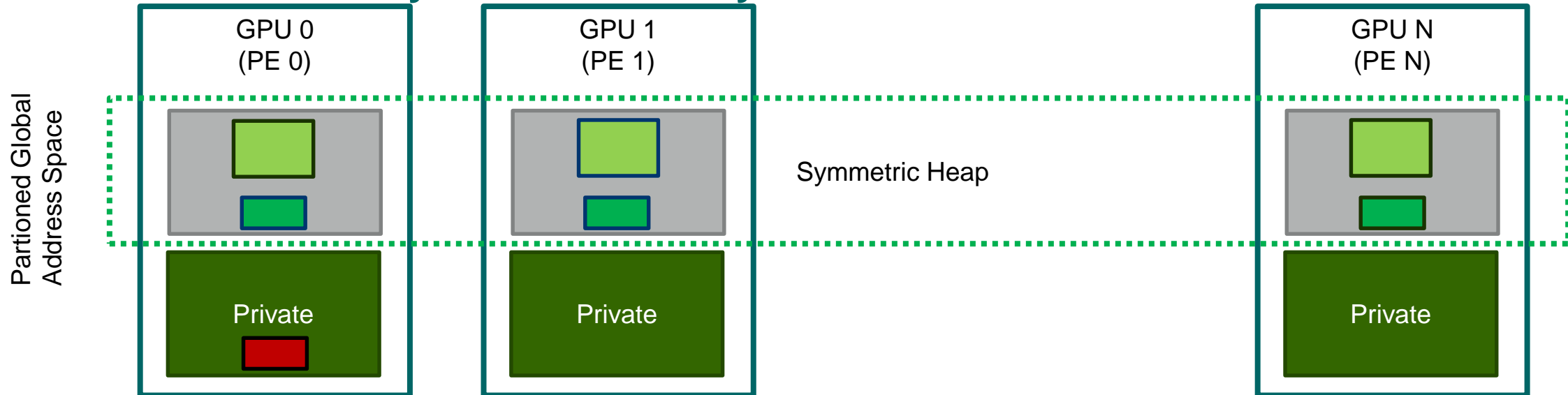
NVSHMEM – Overview

- 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
 - 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



With some
extensions to the
API

NVSHMEM Symmetric Memory Model



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

Private memory: `cudaMalloc(...)`

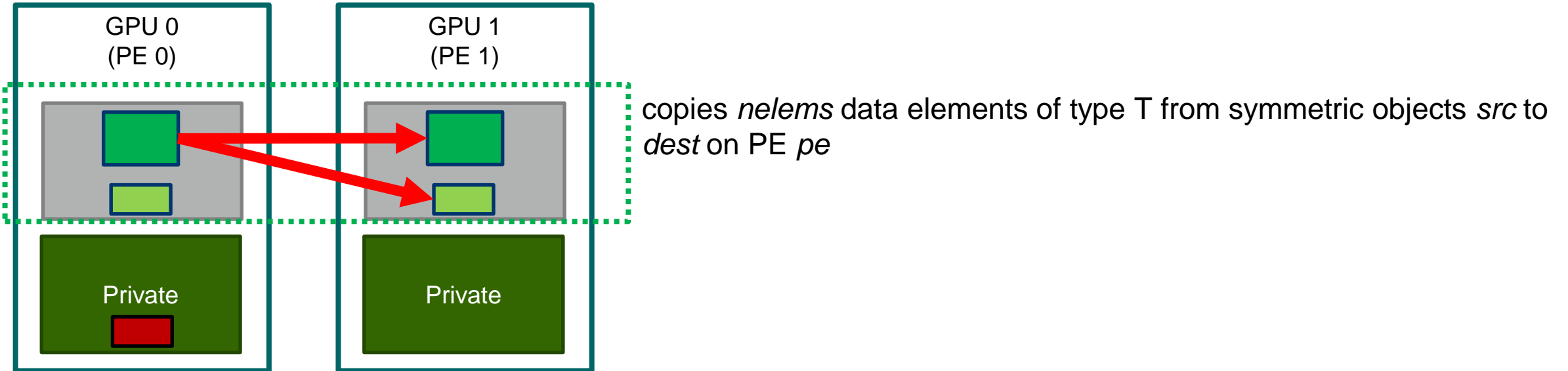
Must be the
same on all
PEs

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);
...
```

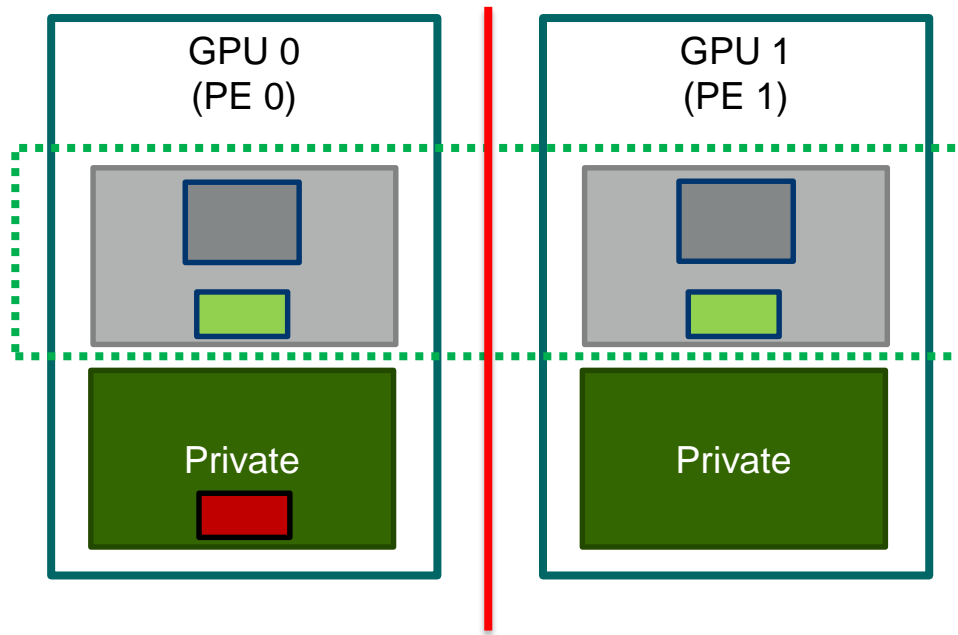
NVSHMEM Host API Put



```
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 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)
```

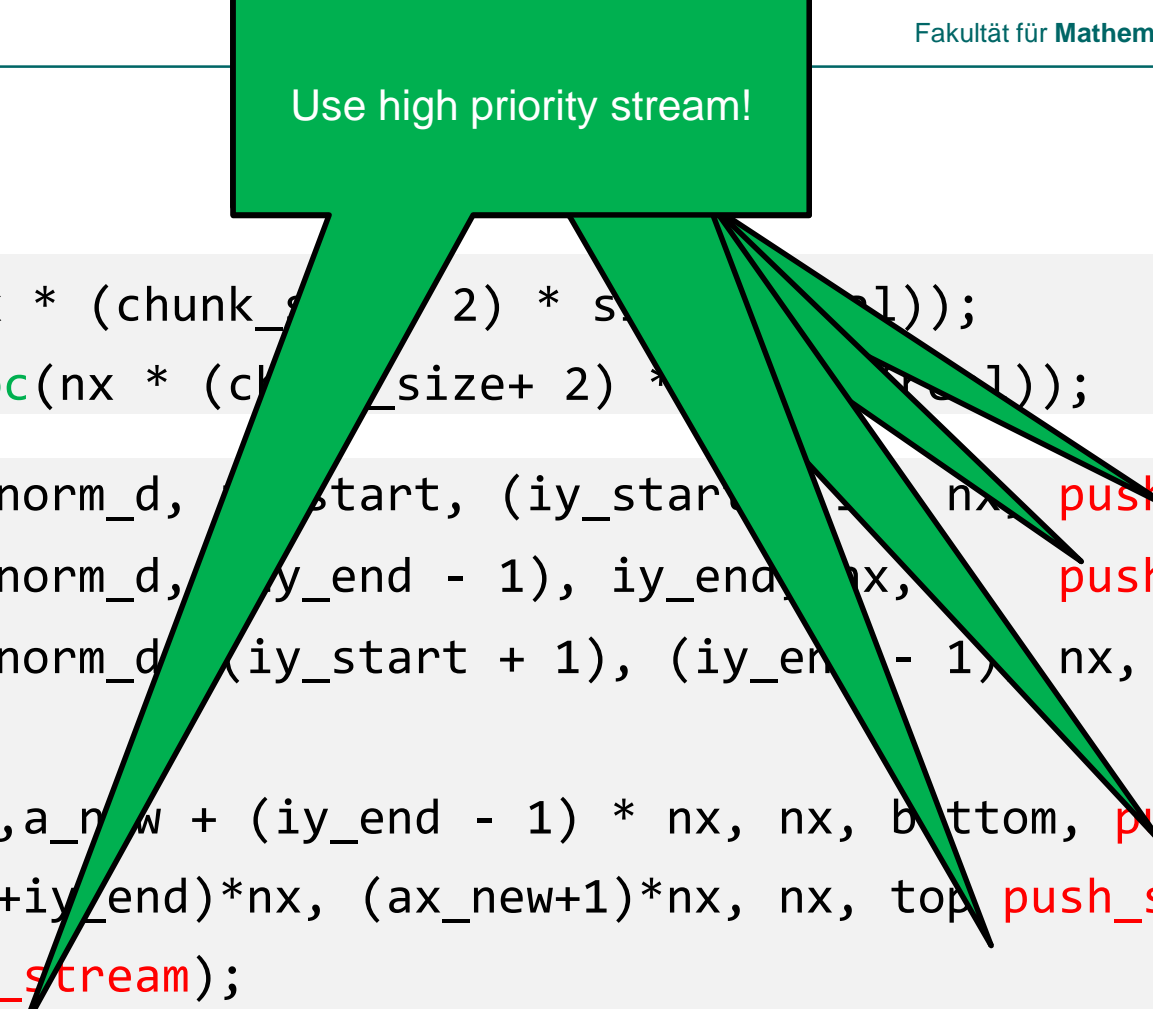
Chunk size must be the same
on all PEs. Otherwise, you get
Undefined Behavior!

Jacobi with NVSHMEM

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

Jacobi with NVSHMEM



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

How to compile NVSHMEM + MPI applications

- Compile CUDA-kernel
 - Use the `-rdc=true` compile flag due to the device interface
 - Link against the nvshmem library `-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