



# SUMMARY AND ADVANCED TOPICS ISC24 TUTORIAL SESSION 11

13 June 2025 | Andreas Herten | Jülich Supercomputing Centre, Forschungszentrum Jülich

# Overview

## Summary

1L: Intro/JUPITER

2L: MPI-Distributed GPU Computing

5L: Optimization Techniques

7L: NCCL, NVSHMEM

9L: CUDA Graphs, Device-Initiated  
NVSHMEM

*More: Other Languages/Models*

OpenACC, OpenMP; Kokkos

Python

*More: In-Network Computing*

Concept

Libraries

Other Vendors

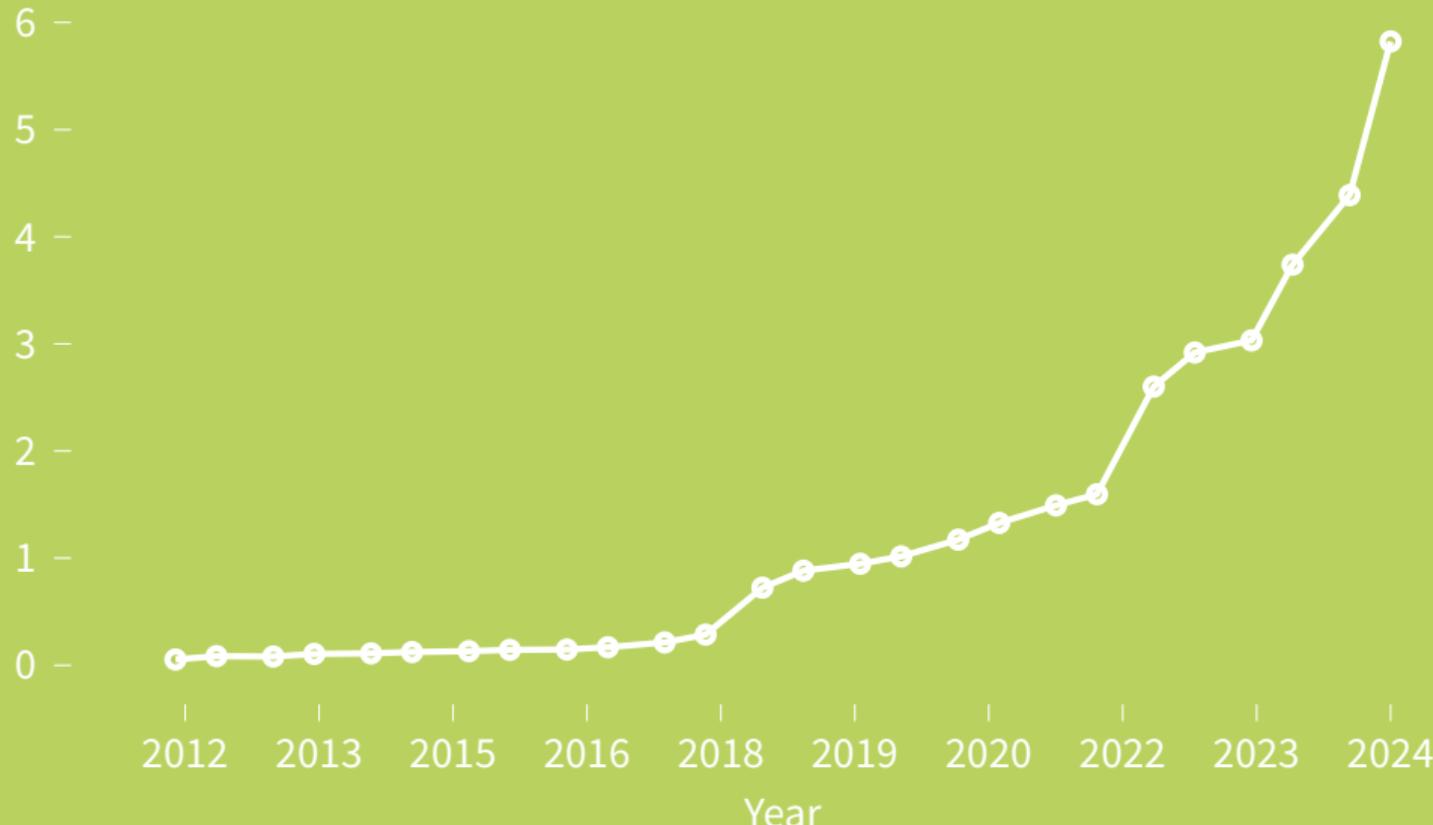
Summary, Conclusion

# Summary

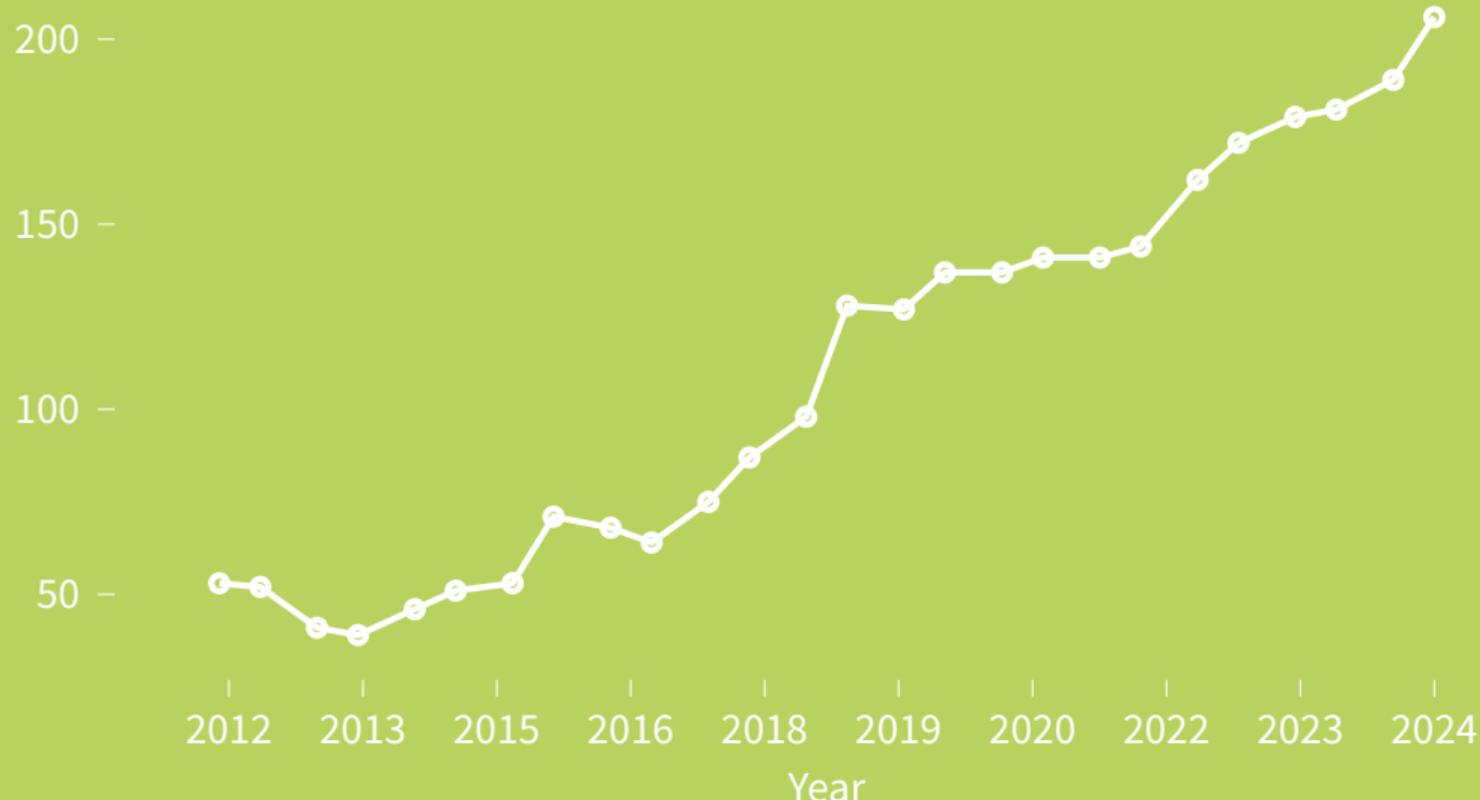
## *1L: Intro/JUPITER*

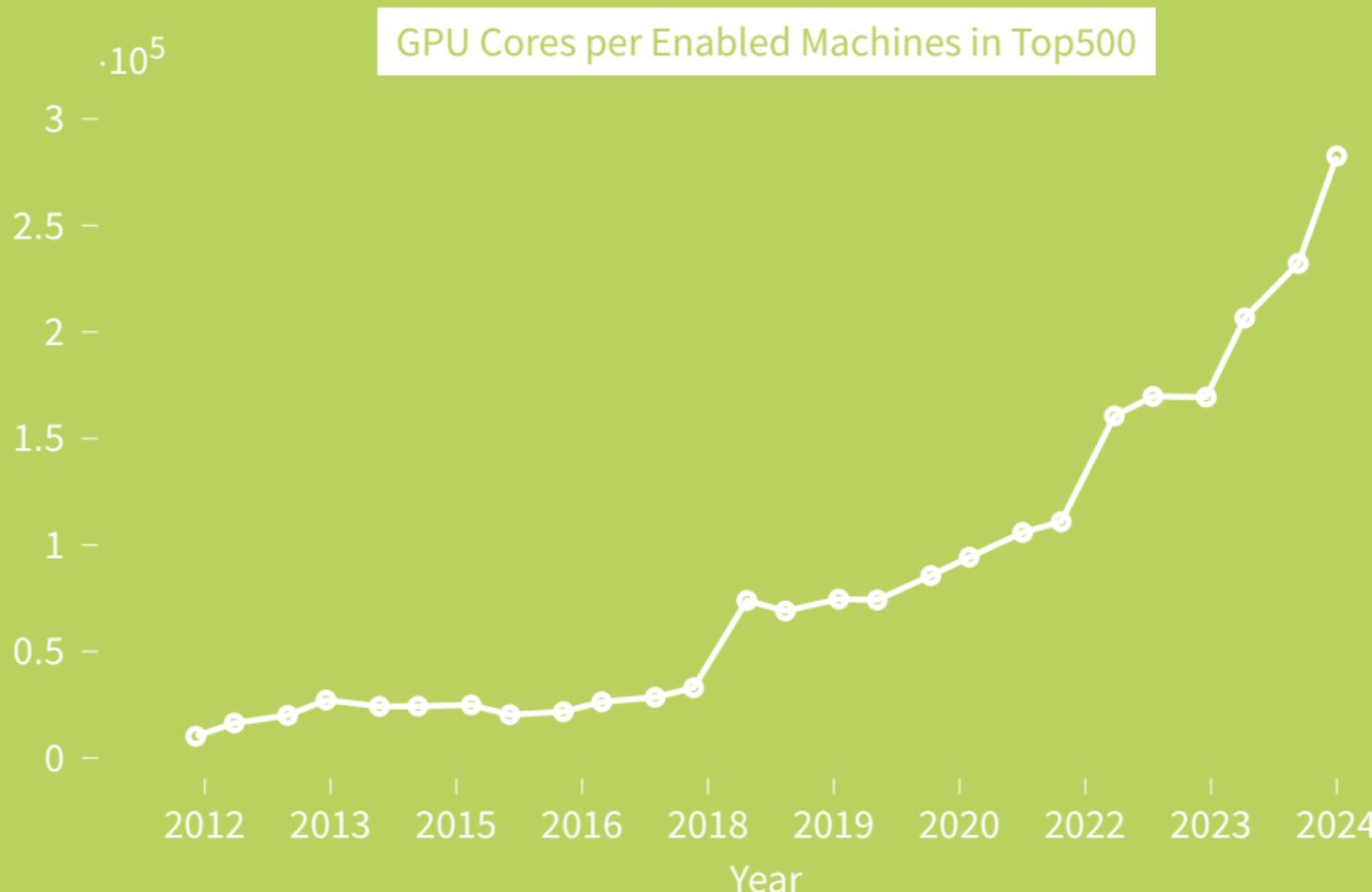
$\cdot 10^7$

### GPU Cores in Top500 (*SMs etc.*)



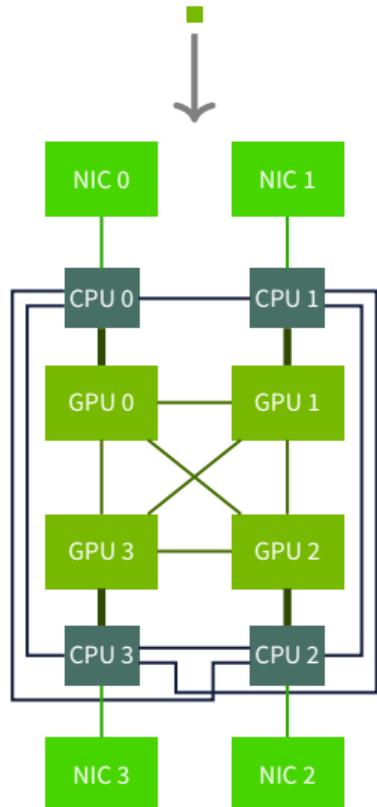
## GPU-enabled Machines in Top500



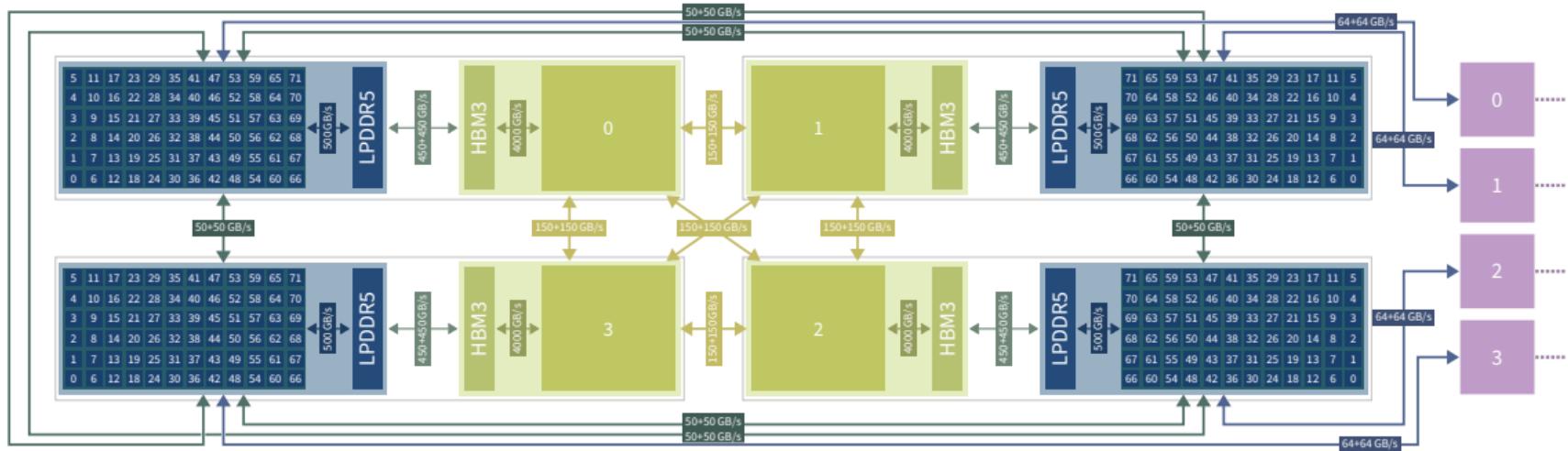


# JUPITER

- At Jülich Supercomputing Center, Germany; procured by EuroHPC JU
  - First European Exascale system *soon!*
  - **Booster**, Cluster
  - HPL: 0.8 EFLOP/s
  - GPU : 4× NVIDIA H100 *Grace-Hopper flavor*  
96 GB memory per GPU
  - CPU : 4× NVIDIA Grace, 4× 72 cores; 4× 120 GB LPDDR5X memory
  - Network : 4× NVIDIA InfiniBand NDR200, 4 × 25 GB/s
- ≈6000 nodes, ≈24 000 GPUs, ≈24 000 network devices



# Node Topology



- 4 GH200 superchips, each 120 GB LPDDR5X & 96 GB HBM3 memory → 4 NUMA domains
- 1:1:1 affinity of **CPU**, **GPU**, **HCA**; coherent access between **GPU-GPU**, **CPU-GPU**, **CPU-CPU**  
No fear: Slurm – and we – take care about affinity!

# Summary

**2L: MPI-Distributed GPU Computing**



## CUDA-aware MPI

CUDA-aware MPI allows you to use Pointers to GPU-Memory as source and destination

```
//MPI rank 0  
MPI_Send(s_buf_d, n, MPI_BYTE, size-1, tag, MPI_COMM_WORLD);  
  
//MPI size-1  
MPI_Recv(r_buf_d, n, MPI_BYTE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

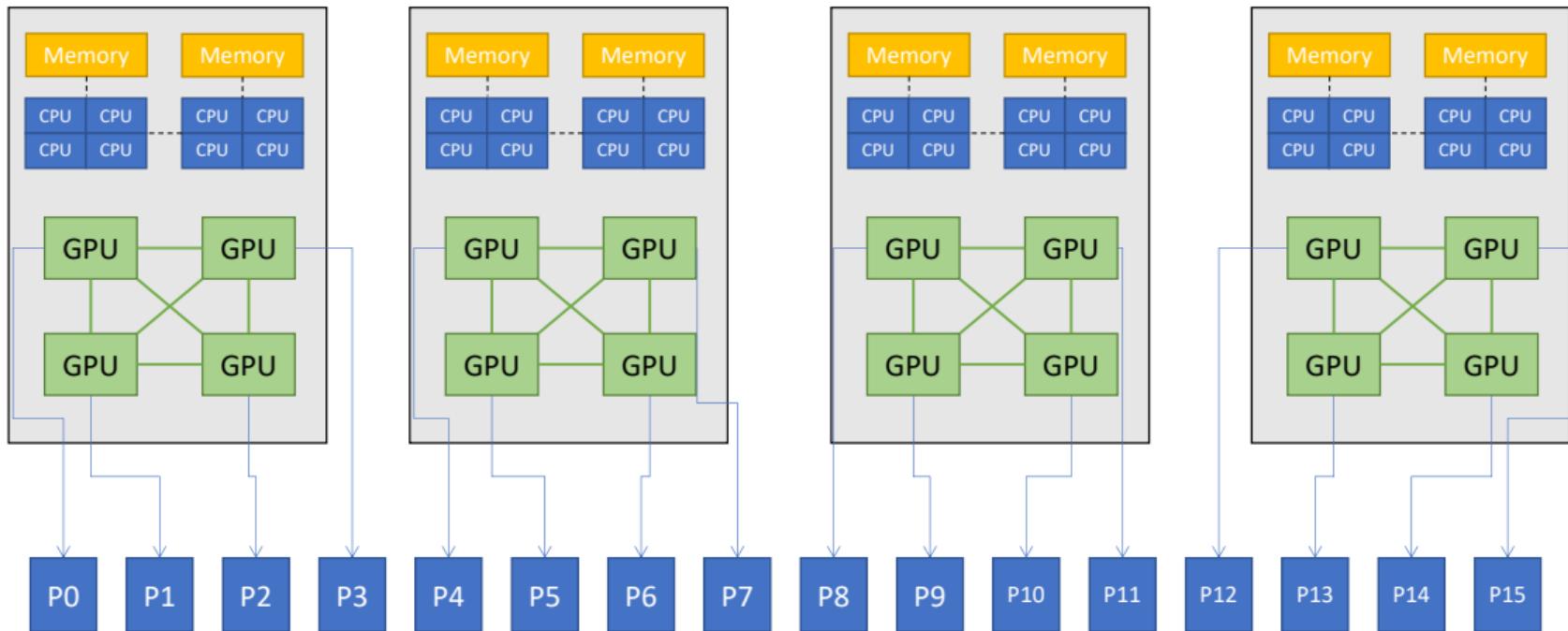


Pointer to  
GPU  
memory!



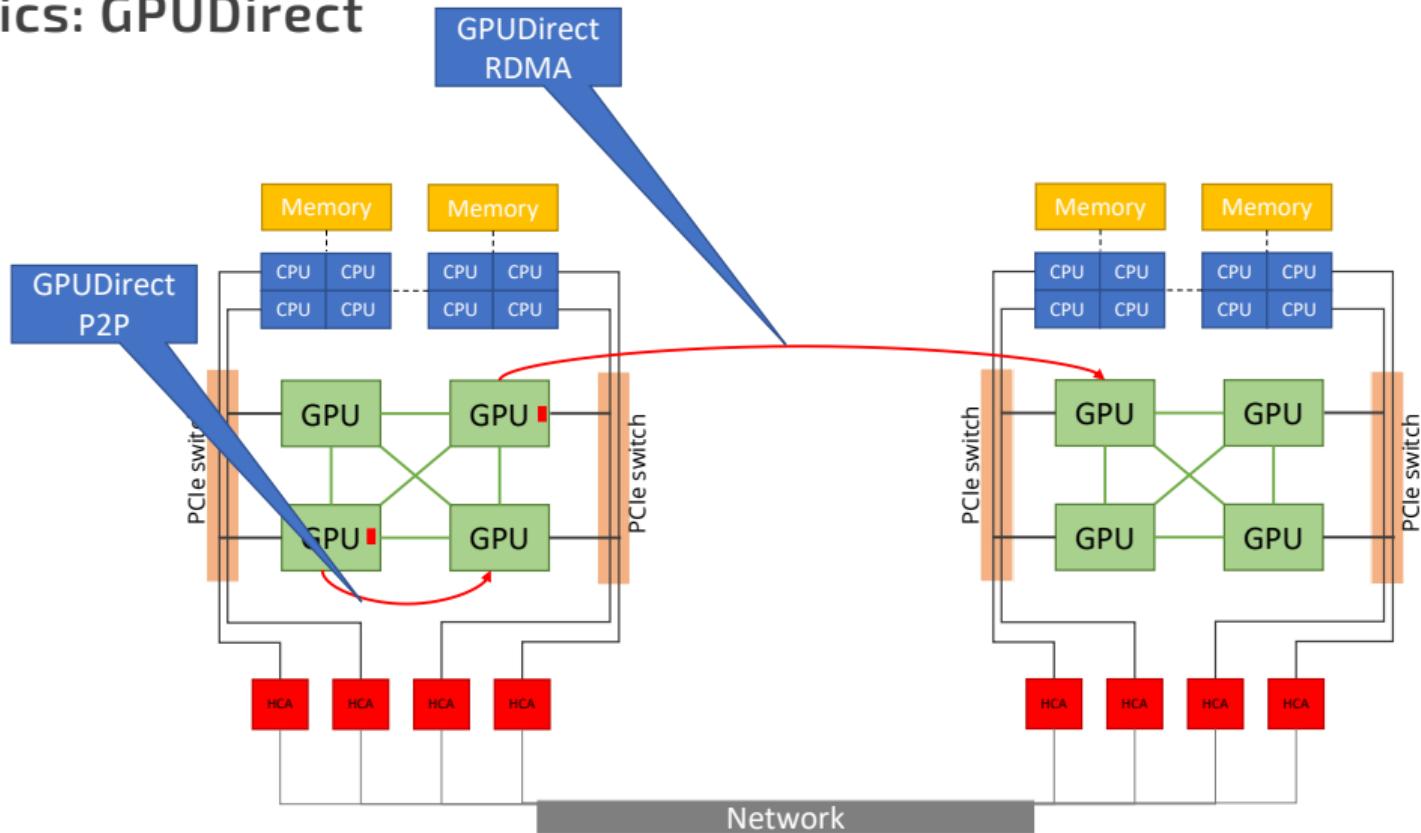
# Process Mapping on Multi GPU Systems

## One GPU per Process



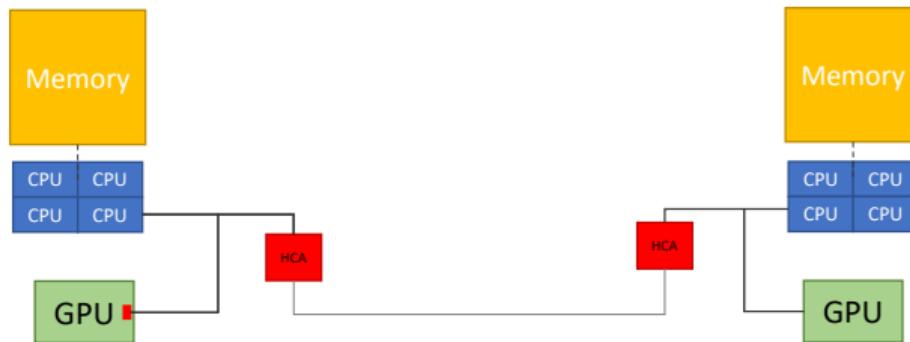
# Basics: GPUDirect

Slide quoted





# CUDA-aware MPI with GPUDirect RDMA

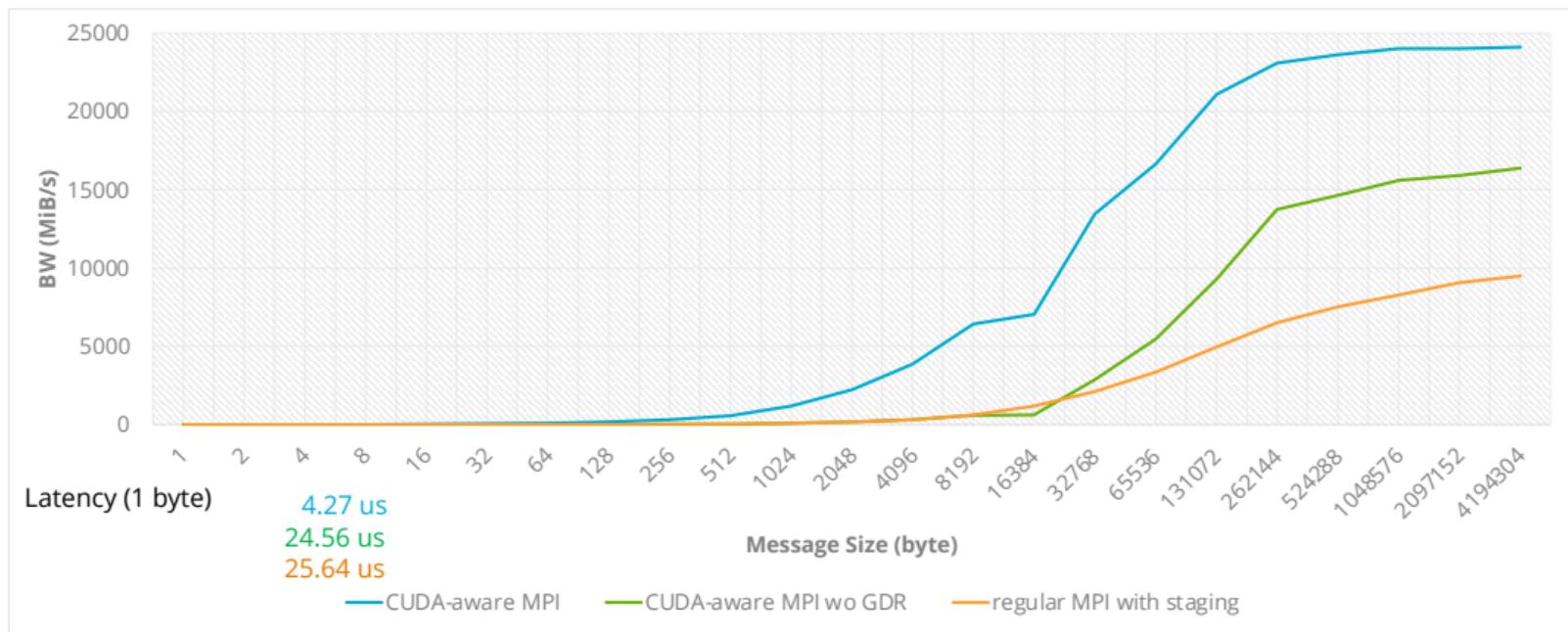


```
MPI_Send(s_buf_d, size, MPI_BYTE, 1, tag, MPI_COMM_WORLD);  
MPI_Recv(r_buf_d, size, MPI_BYTE, 0, tag, MPI_COMM_WORLD, &stat);
```



# Performance Results GPUDirect RDMA

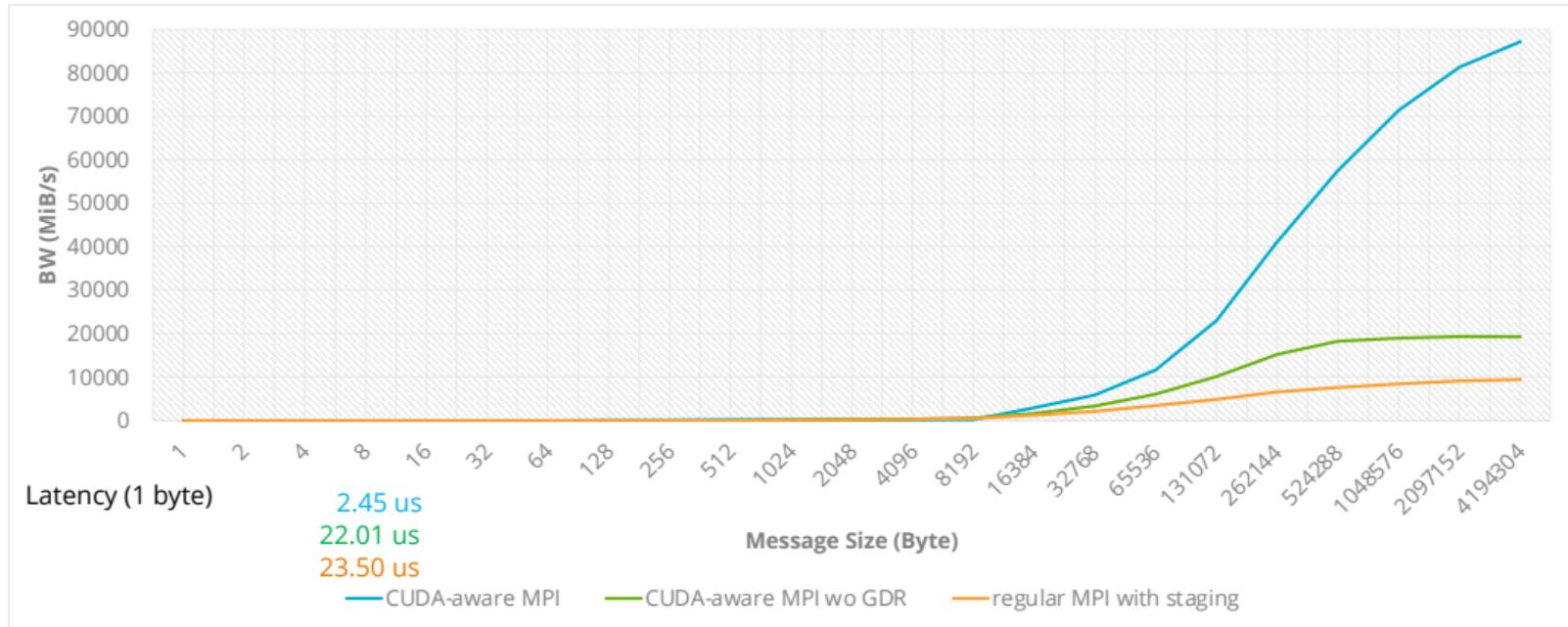
OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS Booster





# Performance Results GPUDirect P2P

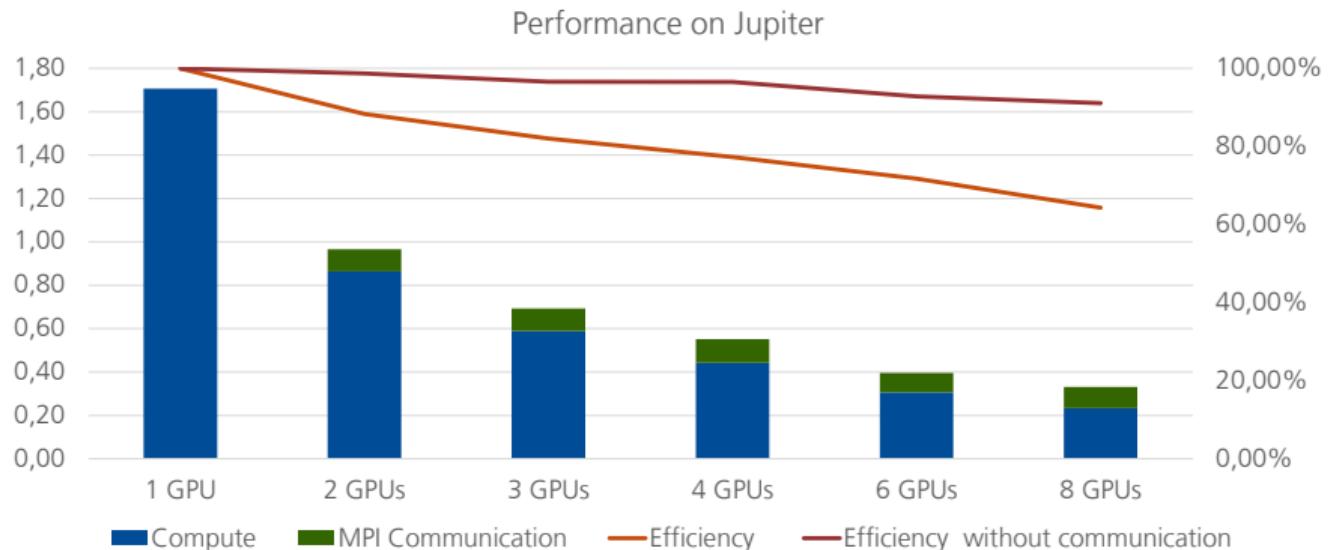
OpenMPI 4.1.0RC1 + UCX 1.9.0 on JUWELS-Booster



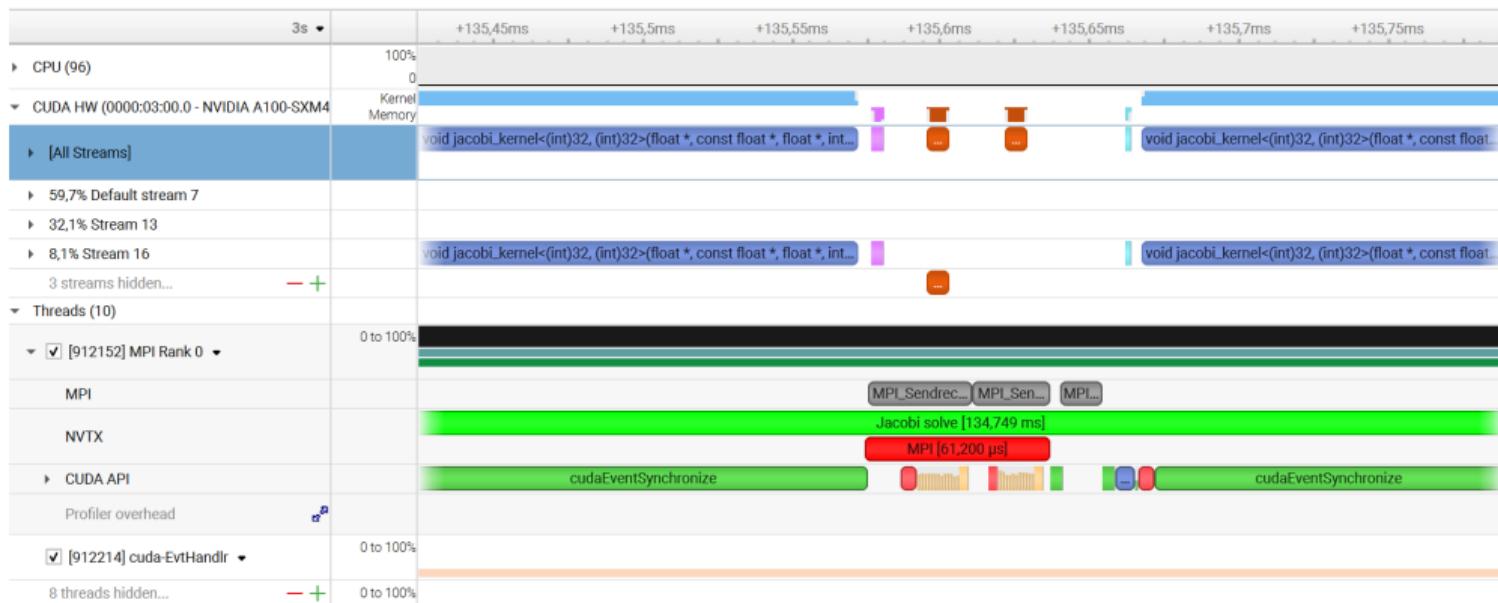
# Summary

## *5L: Optimization Techniques*

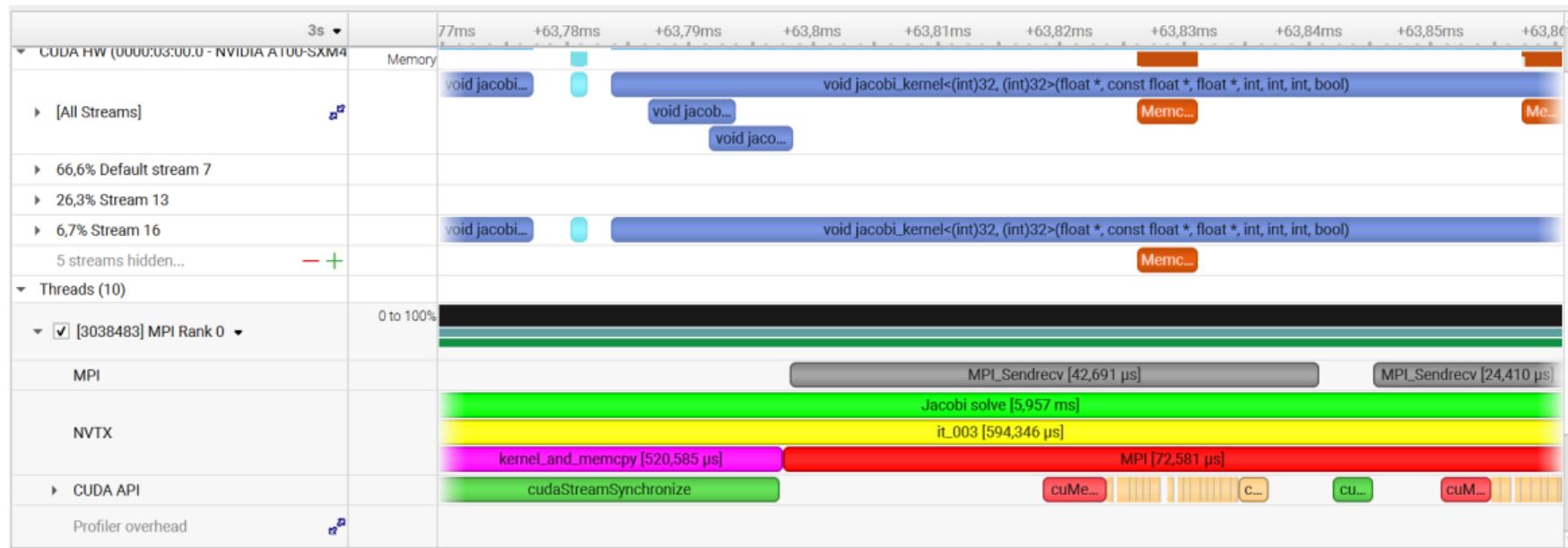
## Performance of our first multi-GPU application



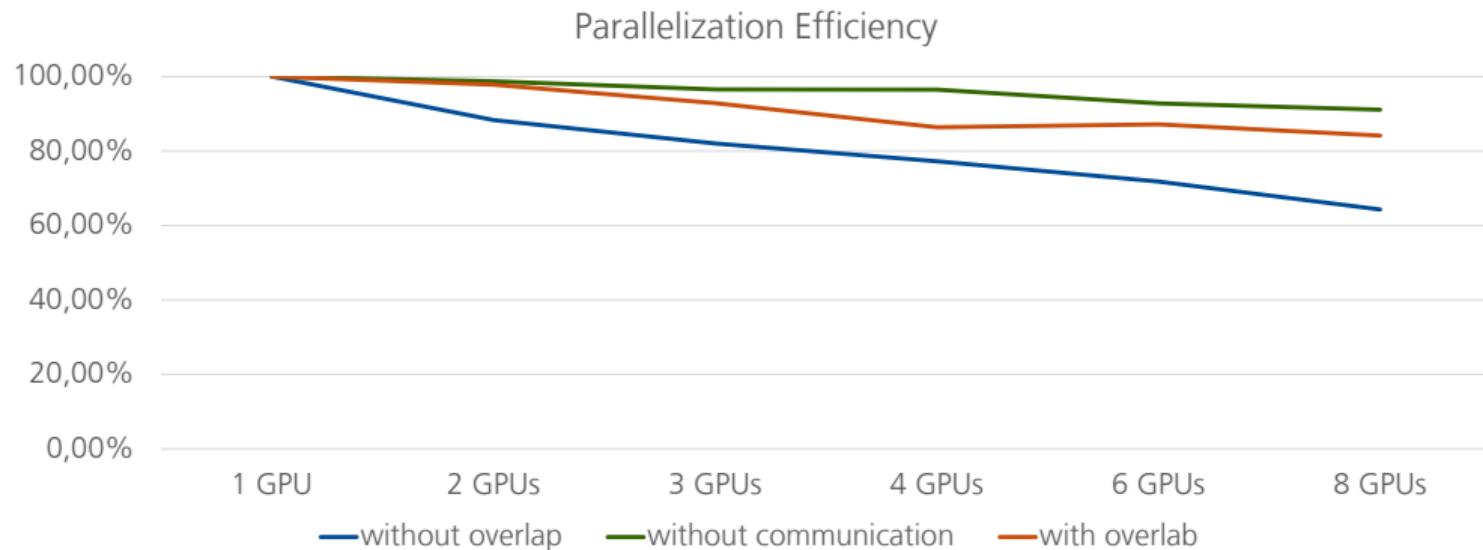
## Analysis with Nsight System



## Avoid Synchronisation



## Performance comparison on Jupiter

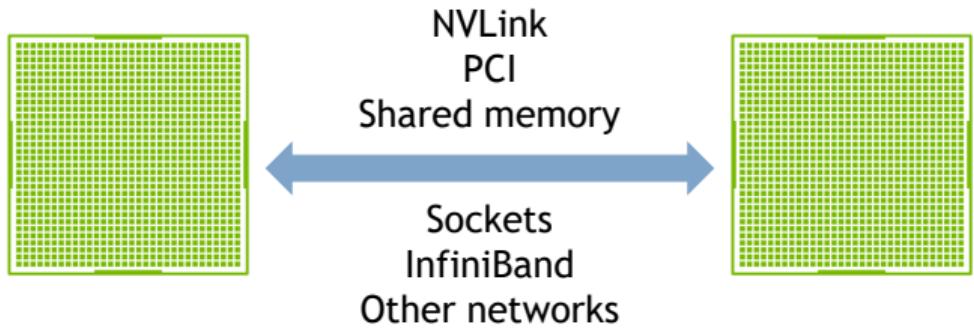


## Optimized inter-GPU communication

### 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>



## 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();
```

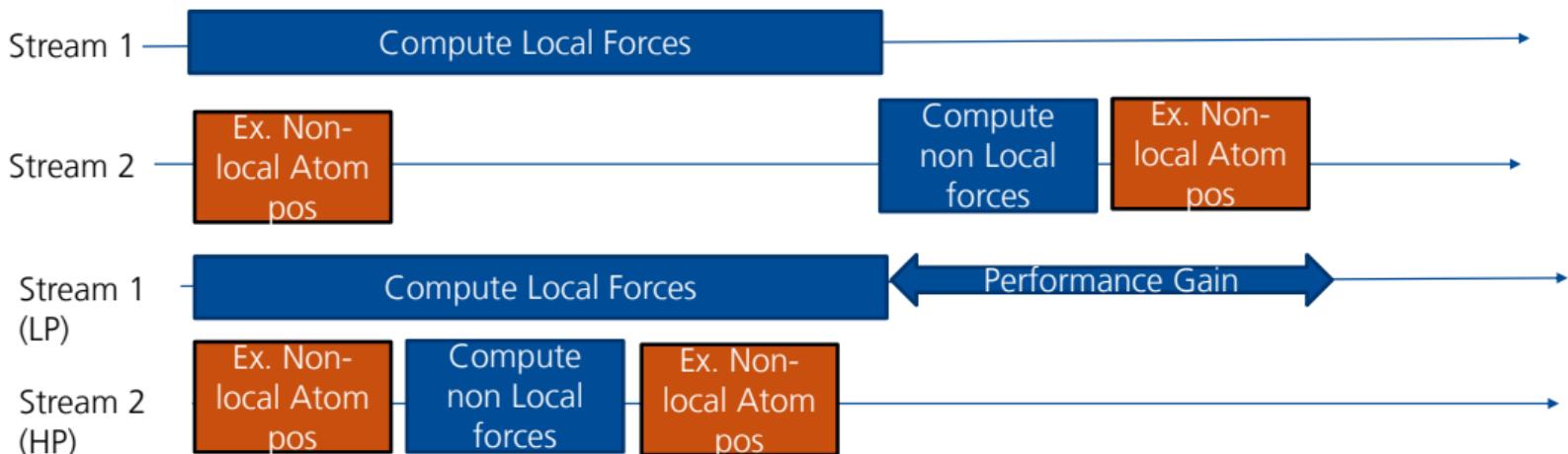
So far, no Overlap of communication and computation!

## High Priority Streams

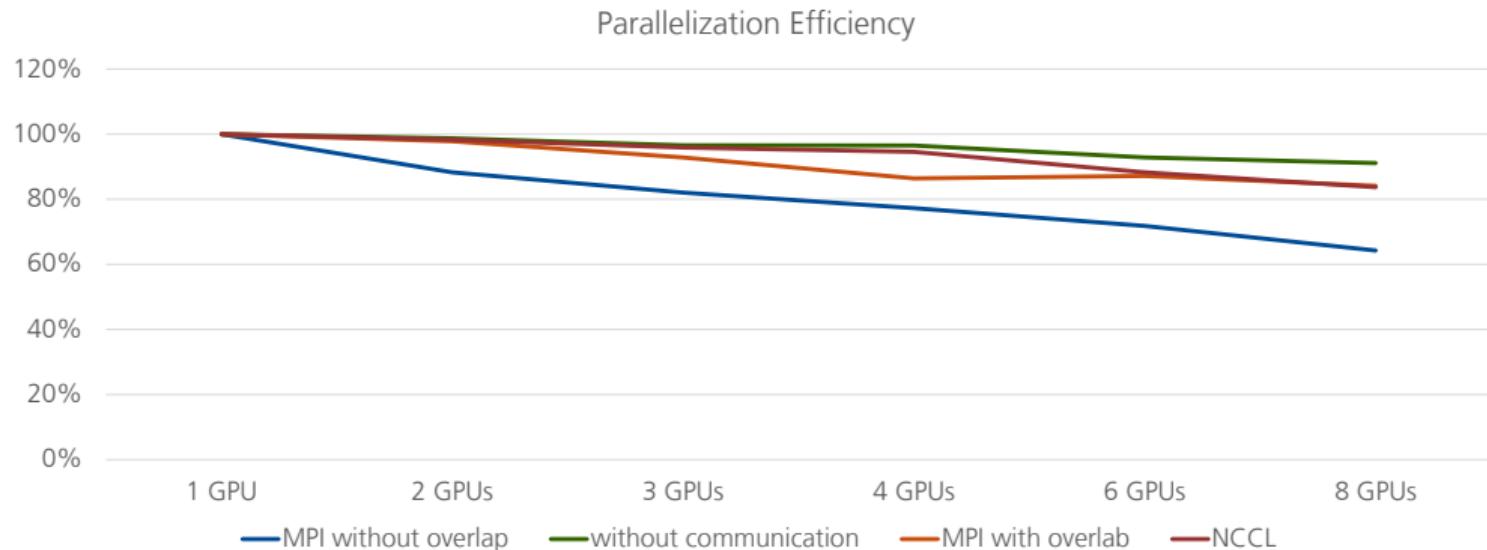
Improve scalability with high priority streams (available on CC 3.5+)

```
cudaStreamCreateWithPriority ( cudaStream_t* pStream, unsigned int flags, int priority )
```

Motivating Example: MD- Simulations



## Performance on Jupiter



# Summary

## *7L: NCCL, NVSHMEM*

## 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

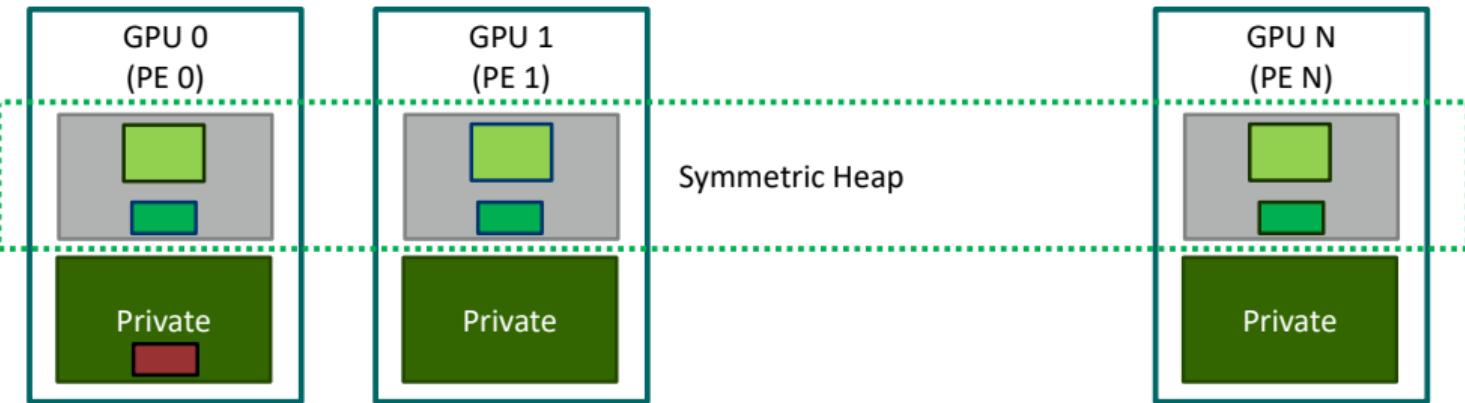
With some  
extensions to  
the API

# NVSHMEM Symmetric Memory Model

FernUniversität in Hagen

Fakultät für Mathematik und Informatik

Partitioned Global Address Space



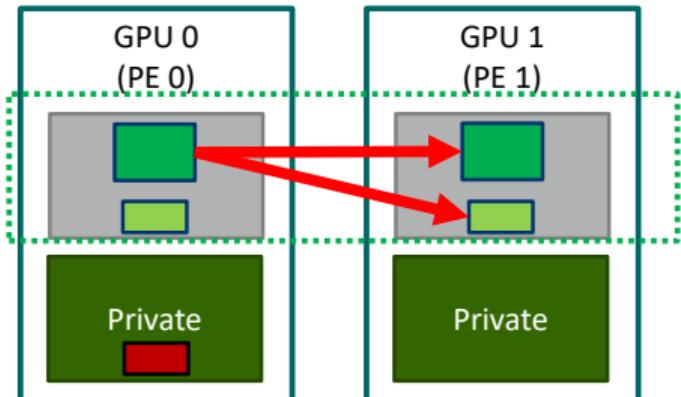
Symmetric objects are allocated collectively with the same size on every PESymmetric memory:

`nvshmem_malloc( shared_size);`

Private memory: `cudaMalloc(...)`

Must be the  
same on all  
PEs

## 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*source, size_t nelems, int pe);  
void nvshmem_x<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

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

## Jacobi solver communication 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));
```

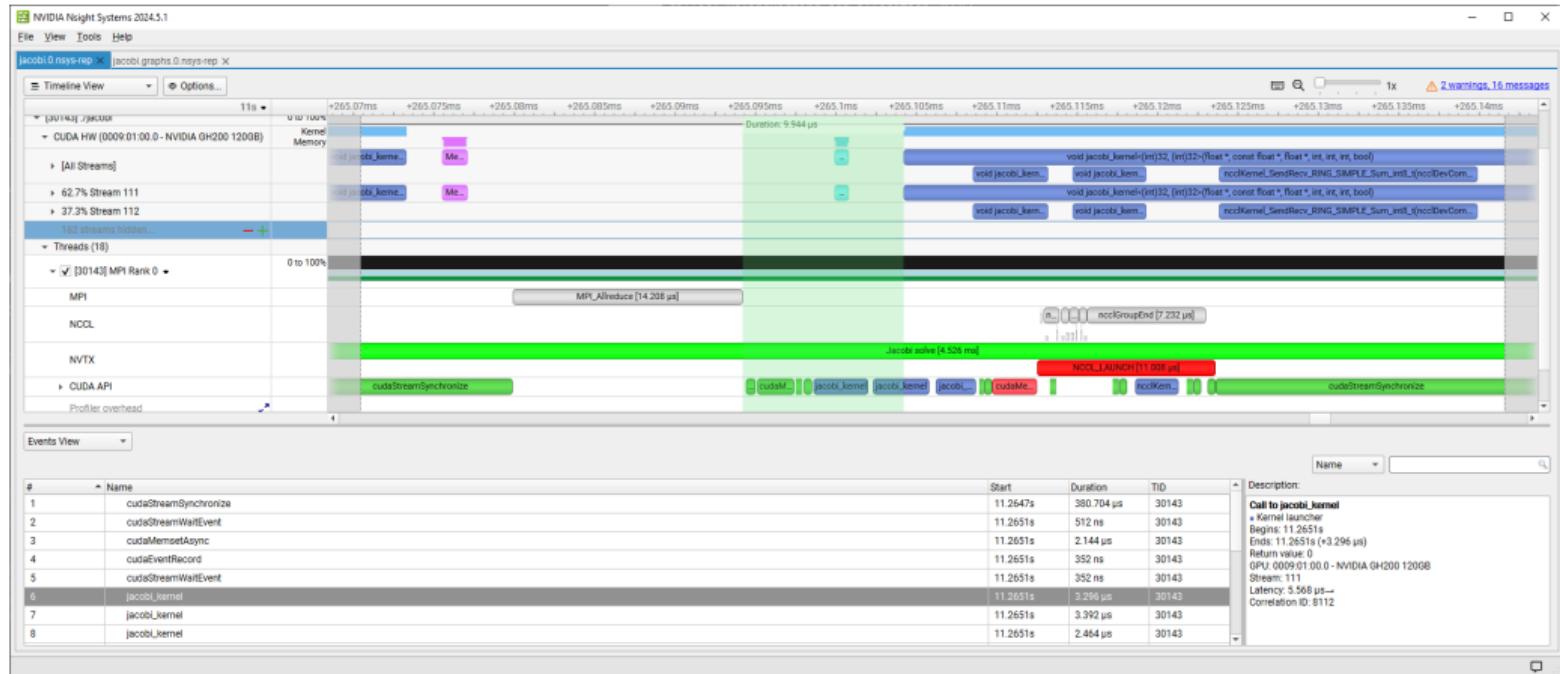
```
launch_jacobi_kernel(a_new, a, l2_norm_d, iy_start, iy_end, nx, compute_stream);  
nvshmemx_float_put_on_stream(a_new, a_new +iy_end - 1) * nx, nx, btm, compute_stream);  
nvshmemx_float_put_on_stream((a_new+iy_end)*nx, (ax_new+1)*nx, nx, top, compute_stream);  
nvshmemx_barrier_all_on_stream(compute_stream);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

# Summary

**9L: CUDA Graphs, Device-Initiated NVSHMEM**

# Multi GPU Jacobi Nsight Systems Timeline

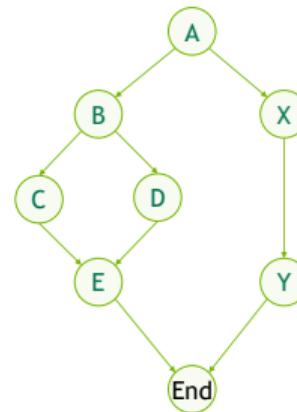
NCCL 4 NVIDIA GH200 120GB on JEDI



# Asynchronous Task Graph

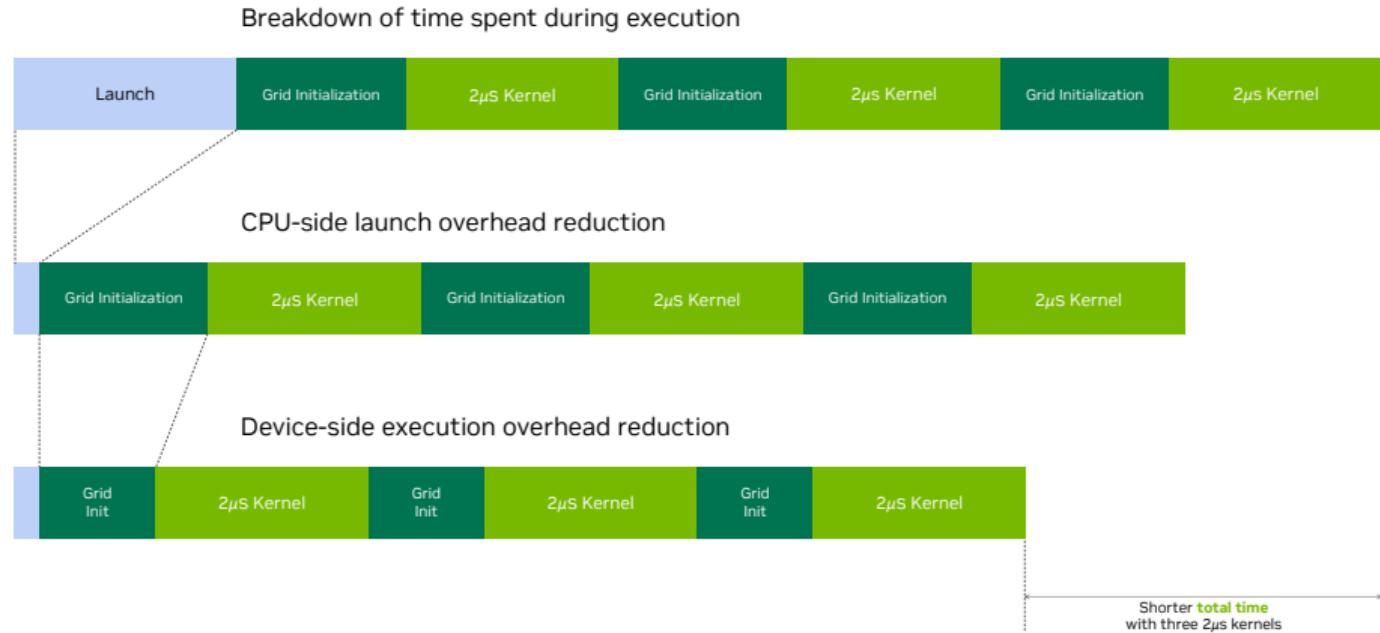
A Graph Node Is A CUDA Operation

- Sequence of operations (nodes), connected by dependencies
- Operations are one of:
  - Kernel Launch CUDA kernel running on GPU
  - CPU Function Call Callback function on CPU
  - Memcopy/Memset GPU data management
  - Mem Alloc/Free Memory management
  - External Dependency External semaphores/events
  - Sub-Graph Graphs are hierarchical
- Nodes within a graph can also span multiple devices



# Where is Performance Coming From?

## Reducing System Overheads Around Short-Running Kernels



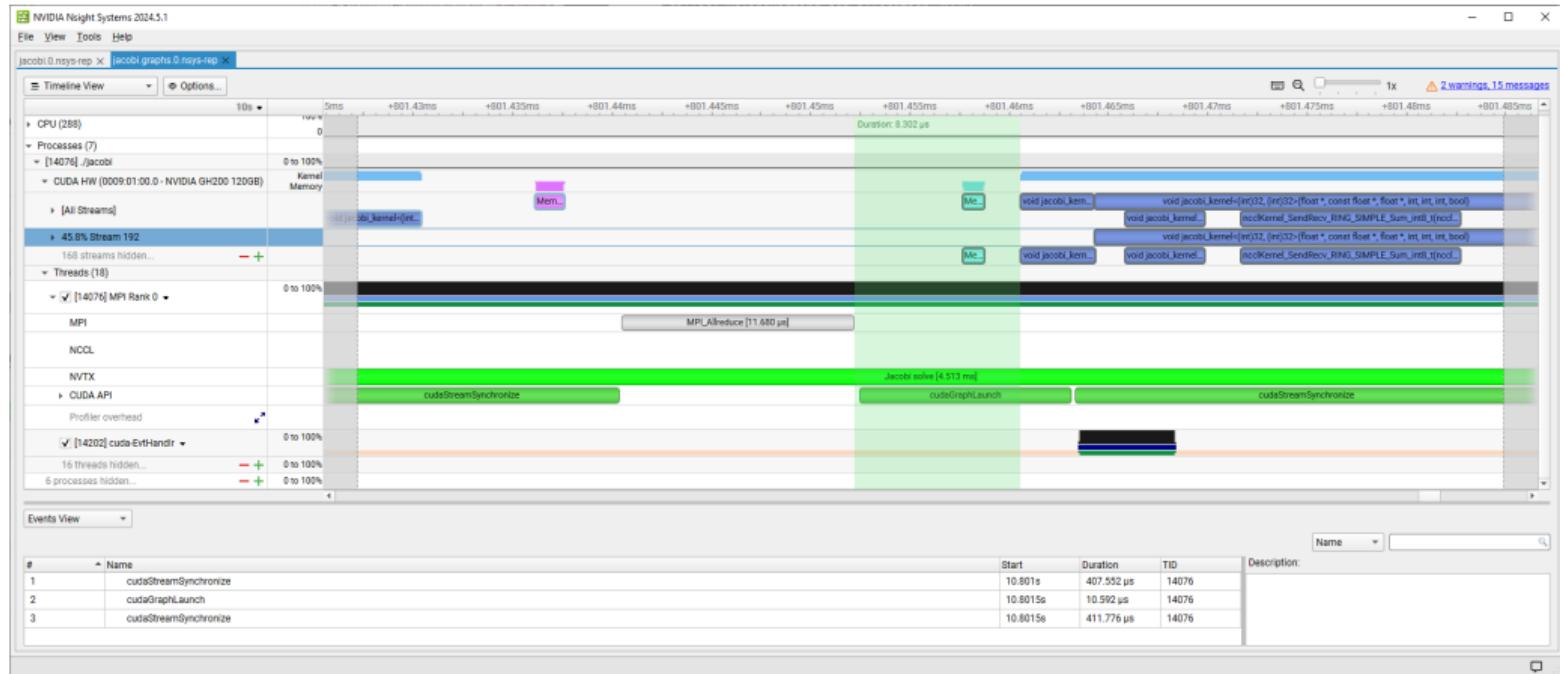
# Capture Stream Work into a Graph

Create A Graph With Two Lines of Code

```
{  
    cudaStreamBeginCapture(compute_stream, cudaStreamCaptureModeGlobal);  
  
    cudaMemsetAsync(l2_norm_d, 0, sizeof(real), compute_stream));  
    cudaEventRecord(reset_l2norm_done, compute_stream);  
  
    ...  
  
    cudaStreamWaitEvent(compute_stream, push_done, 0);  
  
    cudaStreamEndCapture(compute_stream, graphs[calculate_norm]+is_even);  
  
    std::swap(a_new, a);  
    iter++;  
}
```

# Multi GPU Jacobi Nsight Systems Timeline

NCCL with CUDA Graphs 4 NVIDIA GH200 120GB on JEDI



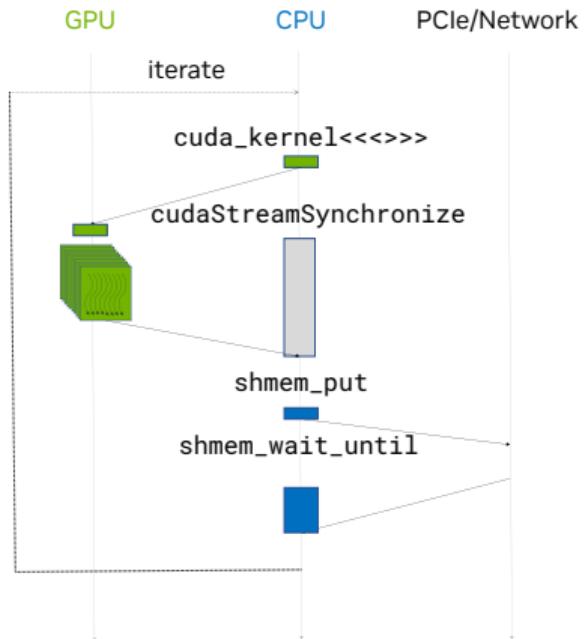
## CPU-Initiated Communication

- Compute on GPU
  - Communication from CPU
- Synchronization at boundaries

Commonly used model, but -

- Offload latencies in critical path
- Communication is not overlapped

Hiding increased code complexity, not hiding limits strong scaling

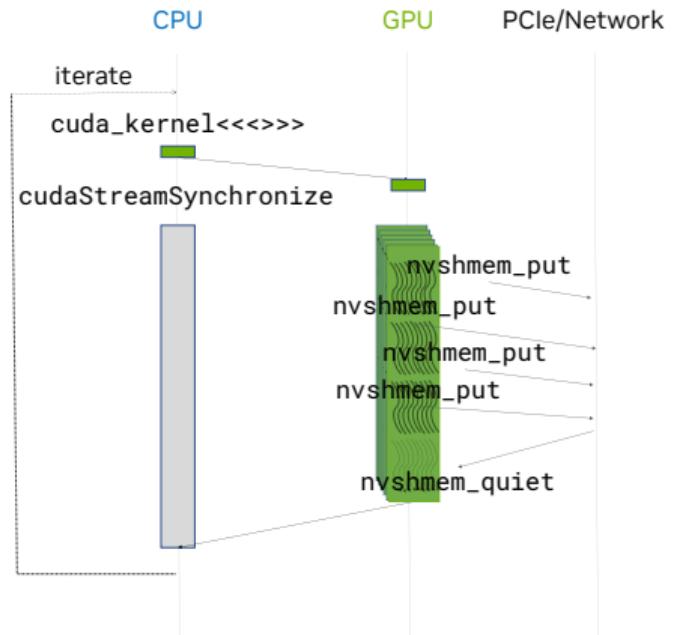


# GPU-Initiated Communication

- Compute on GPU
- Communication from GPU

## Benefits

- Eliminates offloads latencies
- Compute and communication overlap by threading
- Easier to express algorithms with inline communication

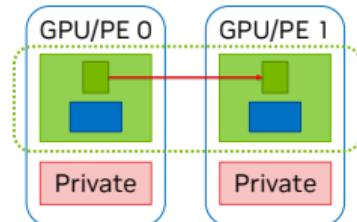


# NVSHMEM API

## 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,..., 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)
```

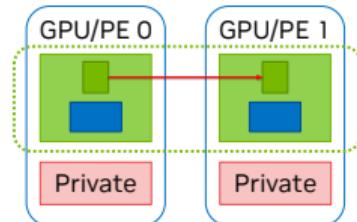
- dest [OUT]: Symmetric address of the destination data object.
- source [IN]: Symmetric address of the object containing the data to be copied.
- nelems [IN]: Number of elements in the dest and source arrays.
- pe [IN]: The number of the remote PE.

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

x in nvshmemx 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](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, ..., ptrdiff  
(see: <https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes>)



# 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.

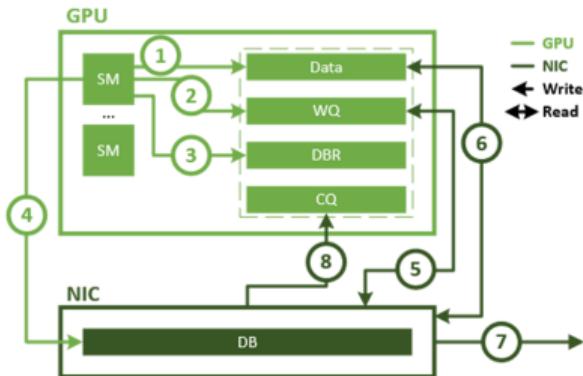
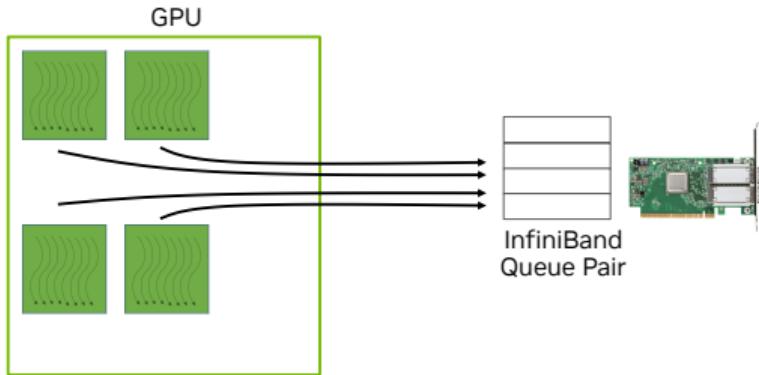
These routines perform an atomic increment operation on the dest data object on PE.

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,..., ptrdiff  
(see: <https://docs.nvidia.com/hpc-sdk/nvshmem/api/docs/gen/api/rma.html#stdrmatypes>)

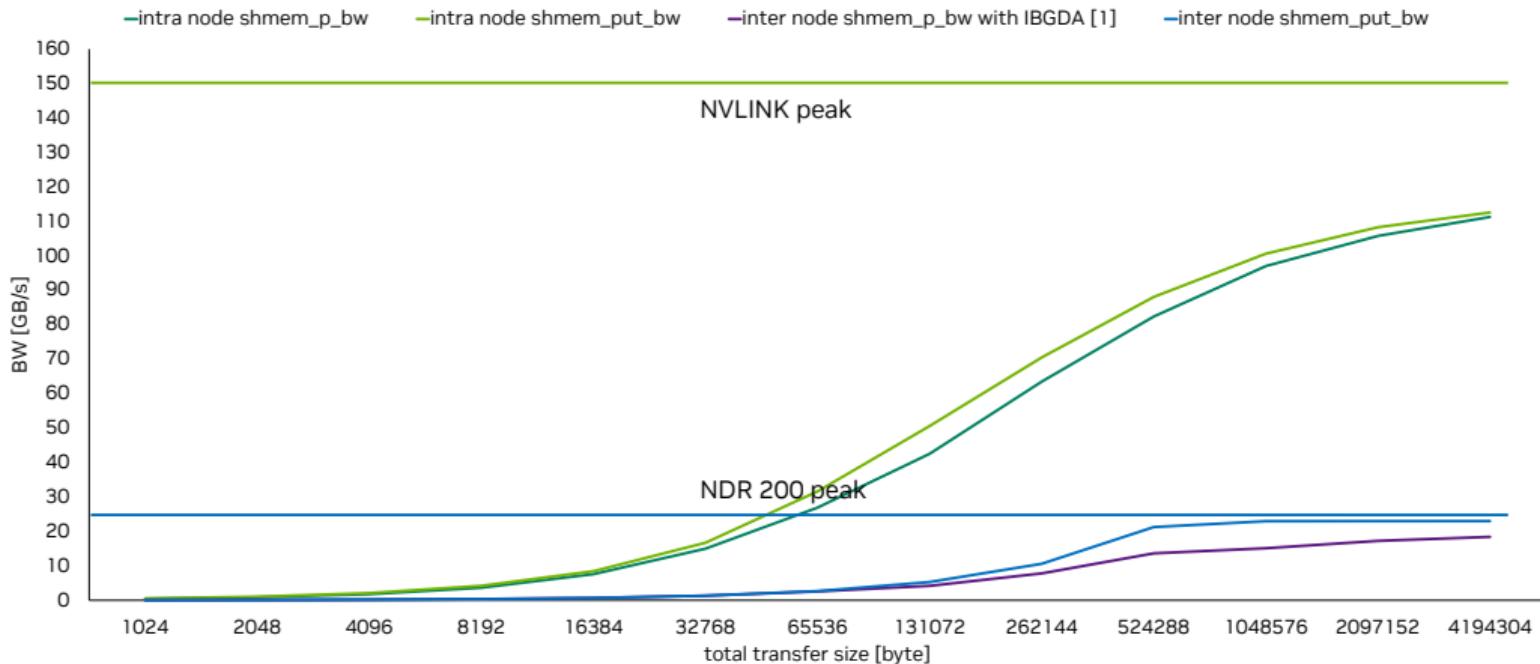
## Optimized Inter-Node Communication Improved

- IB GPUDirect Async (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 Perfests with IBGDA

shmem\_p\_bw and shmem\_put\_bw on JEDI – NVIDIA GH200 120GB



[1] shmem\_p\_bw with IBGDA using experimental version of NVSHMEM on NVIDIA internal NVIDIA GH200 480GB cluster with NDR400

*More: Other Languages/Models*

# OpenACC, OpenMP; Kokkos

- Directive-based GPU programming models work analogously to CUDA
- GPU-awareness via MPI configuration, no need to copyout or map( from)
- Using explicit device pointer necessary: host\_data use\_device / use\_device\_addr

```
#pragma acc host_data use_device( A )
MPI_Sendrecv( A+iy_start*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top , 0,
              A+iy_end*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

- Advanced communication libraries can be used like any other library
- Kokkos similar: Use Kokkos::View and Kokkos::View::data() (see [Wiki](#))

```
Kokkos::View<double*> A("A", nx*ny);
MPI_Send(A.data(), int(A.size()), MPI_DOUBLE, bottom_rank, 0, COMM_WORLD);
```

# Python

- CUDA-awareness in MPI in Python available via `mpi4py`
- Higher level: `CuPy`, `CuTe DSL`, `Numba`

# Python

- CUDA-awareness in MPI in Python available via `mpi4py`
- Higher level: `CuPy`, `CuTe DSL`, `Numba`
- Very pythonic (and versatile): `cuNumeric`

# Python

- CUDA-awareness in MPI in Python available via mpi4py
- Higher level: CuPy, CuTe DSL, Numba
- Very pythonic (and versatile): cuNumeric

```
import numpy as np
```

```
A = np.random.rand(N, N)
x = np.zeros(A.shape[1])
d = np.diag(A)
```

# Python

- CUDA-awareness in MPI in Python available via mpi4py
- Higher level: CuPy, CuTe DSL, Numba
- Very pythonic (and versatile): cuNumeric

```
import cunumeric as np

A = np.random.rand(N, N)
x = np.zeros(A.shape[1])
d = np.diag(A)
```

# Python

- CUDA-awareness in MPI in Python available via mpi4py
- Higher level: CuPy, CuTe DSL, Numba
- Very pythonic (and versatile): cuNumeric

```
import cunumeric as np
```

```
A = np.random.rand(N, N)
x = np.zeros(A.shape[1])
d = np.diag(A)
```

- cuNumeric [2]: transparently accelerates / distributes Numpy (and others)
  - Acceleration: Numpy kernel implementations for single-core CPU, multi-core CPU (OpenMP), and GPU (via libraries)
  - Distribution: OpenMP or MPI (via GASNet)
  - Type / size of task pool determined at start time via launcher script

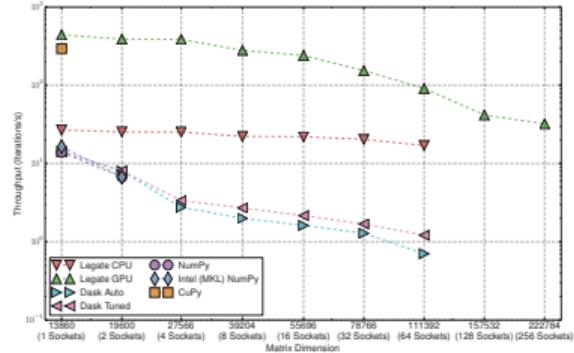
# Python

- CUDA-awareness in MPI in Python available via mpi4py
- Higher level: CuPy, CuTe DSL, Numba
- Very pythonic (and versatile): cuNumeric

```
import cunumeric as np
```

```
A = np.random.rand(N, N)
x = np.zeros(A.shape[1])
d = np.diag(A)
```

- cuNumeric [2]: transparently accelerates / distributes Numpy (and others)
  - Acceleration: Numpy kernel implementations for single-core CPU, multi-core CPU (OpenMP), and GPU (via libraries)
  - Distribution: OpenMP or MPI (via GASNet)
  - Type / size of task pool determined at start time via launcher script



# Python

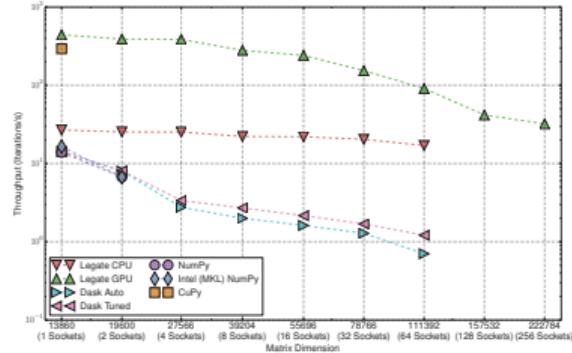
- CUDA-awareness in MPI in Python available via mpi4py
- Higher level: CuPy, CuTe DSL, Numba
- Very pythonic (and versatile): cuNumeric

```
import cunumeric as np
```

```
A = np.random.rand(N, N)
x = np.zeros(A.shape[1])
d = np.diag(A)
```

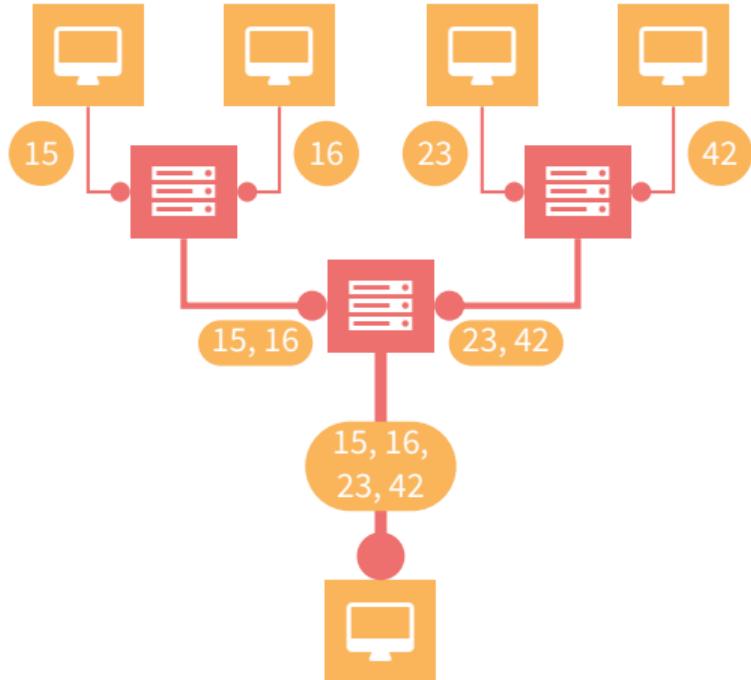
- cuNumeric [2]: transparently accelerates / distributes Numpy (and others)
  - Acceleration: Numpy kernel implementations for single-core CPU, multi-core CPU (OpenMP), and GPU (via libraries)
  - Distribution: OpenMP or MPI (via GASNet)
  - Type / size of task pool determined at start time via launcher script

→ <https://github.com/nv-legate/cunumeric/>



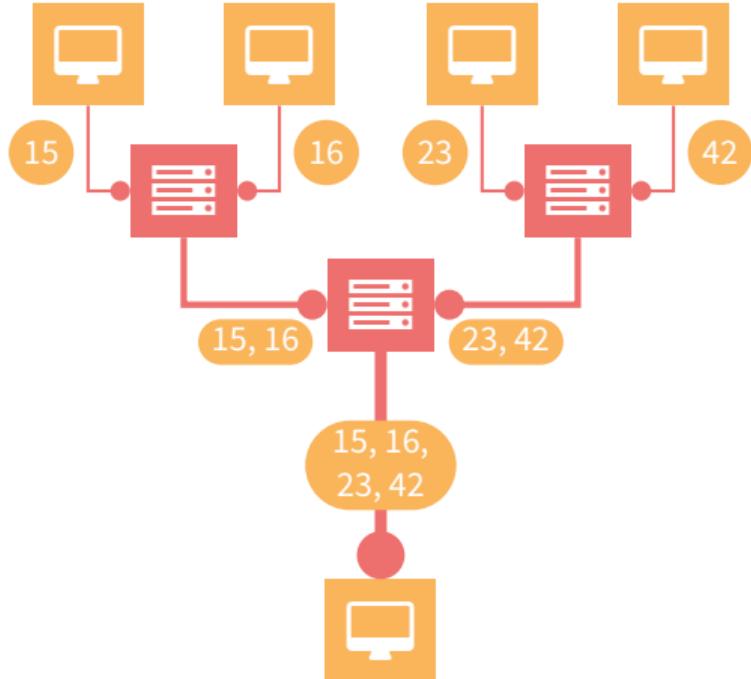
*More: In-Network Computing*

# In-Network Computing

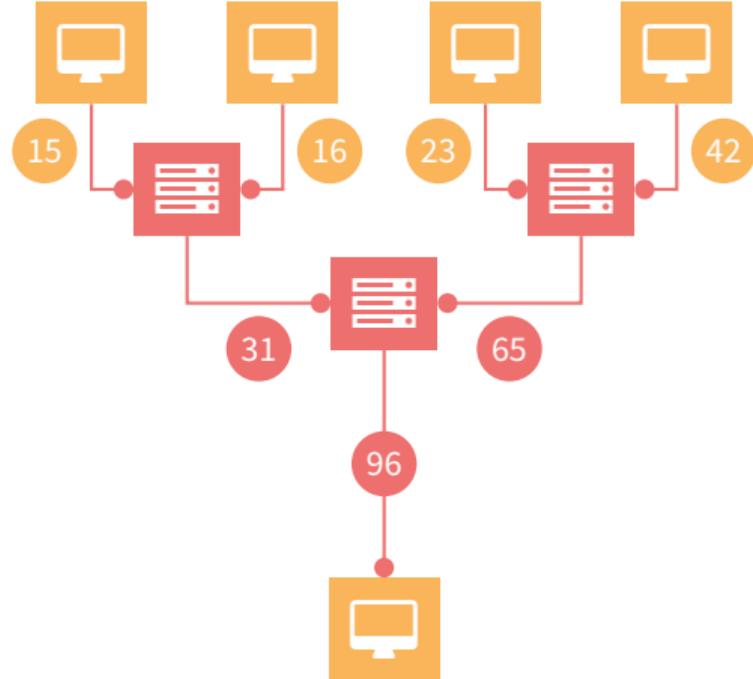


*Traditional Reduce()*

# In-Network Computing



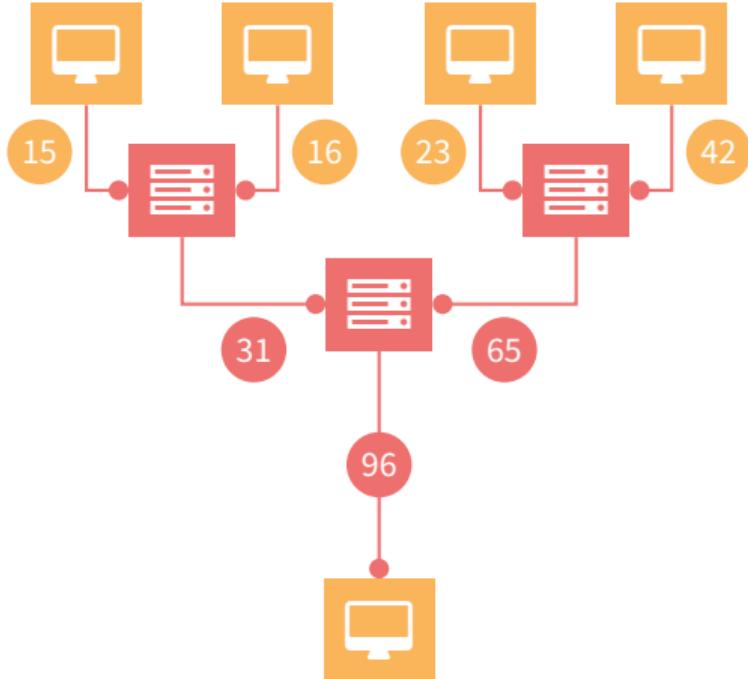
*Traditional Reduce()*



*Switch-supported Reduce()*

# In-Network Computing

- Usually, network devices (switches, HCAs) just forward to computing devices
- Modern hardware offers in-network computation
- Works also with GPUs  
→ Less latency, less traffic
- Especially for communication-intensive collectives like `AllReduce()`



*Switch-supported Reduce()*

# In-Network Computing Libraries

MPI

- MPI** MPI runtime transparently offloads specific collective operations to network, if enabled  
(OpenMPI, e.g. bundled in NVIDIA's HPC-X; MVAPICH2-X; also NCCL via plugin)

# In-Network Computing Libraries



**MPI** MPI runtime transparently offloads specific collective operations to network, if enabled  
(OpenMPI, e.g. bundled in NVIDIA's HPC-X; MVAPICH2-X; also NCCL via plugin)

**SHARP** Lowest / base level API (*Scalable Hierarchical Aggregation and Reduction Protocol*)  
`libsharp_coll`: interface, `libsharp`: backend

# In-Network Computing Libraries



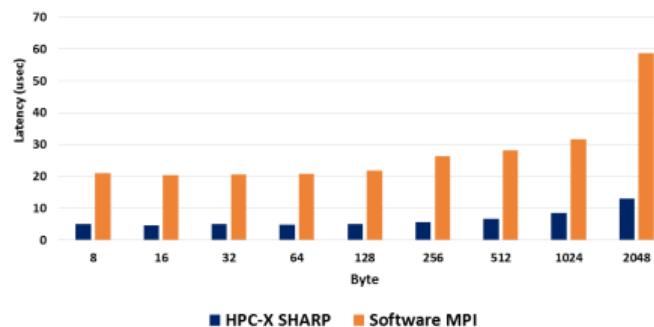
- MPI** MPI runtime transparently offloads specific collective operations to network, if enabled  
(OpenMPI, e.g. bundled in NVIDIA's HPC-X; MVAPICH2-X; also NCCL via plugin)
- UCC** New intermediate layer (from UCF initiative (UCX)) for *Unified Collective Communication*  
→ <https://github.com/openuxc/ucc>
- SHARP** Lowest / base level API (*Scalable Hierarchical Aggregation and Reduction Protocol*)  
`libsharp_coll`: interface, `libsharp`: backend

# In-Network Computing Libraries

- MPI** MPI runtime transparently offloads specific collective operations to network, if enabled  
(OpenMPI, e.g. bundled in NVIDIA's HPC-X; MVAPICH2-X; also NCCL via plugin)
- UCC** New intermediate layer (from UCF initiative (UCX)) for *Unified Collective Communication*  
→ <https://github.com/openucx/ucc>
- SHARP** Lowest / base level API (*Scalable Hierarchical Aggregation and Reduction Protocol*)  
`libsharp_coll`: interface, `libsharp`: backend



**MPI AllReduce Latency**  
1500 Nodes, 1PPN



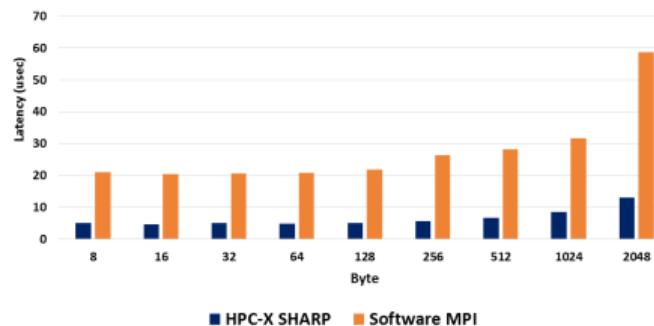
Graph by Gil Bloch / Mellanox (2019)[9]

# In-Network Computing Libraries

- MPI** MPI runtime transparently offloads specific collective operations to network, if enabled  
(OpenMPI, e.g. bundled in NVIDIA's HPC-X; MVAPICH2-X; also NCCL via plugin)
- UCC** New intermediate layer (from UCF initiative (UCX)) for *Unified Collective Communication*  
→ <https://github.com/openuxc/ucc>
- SHARP** Lowest / base level API (*Scalable Hierarchical Aggregation and Reduction Protocol*)  
`libsharp_coll`: interface, `libsharp`: backend



**MPI AllReduce Latency**  
1500 Nodes, 1PPN



Graph by Gil Bloch / Mellanox (2019)[9]

# Other Vendors

# AMD

- AMD Instinct GPUs entered HPC with a boom
- Multi-node ecosystem maturing rapidly
- Key technology already developed, mimicking NVIDIA's strategy
- UCX is ROCm enabled ([how-to ↗](#)); MVAPICH2-GDR [10] also optimized

Technology	NVIDIA	AMD
RDMA Support	GPUDirect RDMA	ROCmRDMA
Peer to Peer	GPUDirect P2P	ROCm IPC
Direct CPU Access (PCIe BAR)	GDRCopy BAR1	LargeBar
Accelerated Collectives	NCCL	RCCL
OpenSHMEM	NVSHMEM	ROC_SHMEM

# AMD HIP Jacobi MPI Example

- Procedure: hipify-perl → fix errors → compile
- Code example

```
hipGetDeviceCount(&num_devices);
hipSetDevice(local_rank%num_devices);
real* a_ref_h;
hipHostMalloc(&a_ref_h, nx * ny * sizeof(real));
```

- Compilation example

```
HIP_PLATFORM=amd hipcc --offload-arch=gfx90a -std=c++14 -munsafe-fp-atomics -O3 -fopenmp
↪ -I${MPI_HOME}/include -c -o jacobi.cu.hip.o jacobi.cu.hip
```

```
HIP_PLATFORM=amd hipcc --offload-arch=gfx90a -std=c++14 -munsafe-fp-atomics -O3
↪ -I${MPI_HOME}/include -L${MPI_HOME}/lib -lmpi --gcc-toolchain=${EBROOTGCCCORE} -o
↪ jacobi.amd jacobi.cu.hip.o
```

- Needed: ROCm-aware UCX (`UCX_TLS=rc_x,self,sm,rocm_copy,rocm_ipc`)

# Intel GPUs with SYCL

- SYCL: Native model for Intel GPU (also OpenMP); can also be executed on NVIDIA, AMD GPUs
- Very different programming model to CUDA, much more C++ish
- MPI supported as manual step
- Example: See [Codeplay's documentation](#)
- More *SYCLic*: Celerity, with distributed queues [celerity.github.io/](https://celerity.github.io/)

```
queue q{property::queue::in_order()};
q.submit([&](handler& h) {
    h.parallel_for(num_items, [=](id<1> k) {
        // jacobi_compute, fill result*
    });
});
MPI_Sendrecv(&result, ...);
```

Greatly reduced sketch of code based [on Intel documentation](#)

# Summary, Conclusion

# Summary, Conclusion

Efficient multi-node GPU computing is efficient multi-node computing with least possible amount of CPU

- Many advanced technologies and techniques in place to enable large-scale GPU applications
- GPU-aware MPI is key enabler
- On top / orthogonal: NCCL, NVSHMEM, ...→homework?
- Remember to [rate tutorial](#), give feedback! 
- *Appendix: [Links](#), [references](#)*

Efficient Distributed GPU Programming

isc.app.swapcard.com/event/isc-high-performance-2025/planning/Uxgbm5pbmdfMjU4MTc

ISC Events > ISC High Performance 2025

Startseite CONTENT ON-DEMAND MY ISC COMMUNITY SCHEDULE SPEAKERS EXHIBITION GIVEAWAYS HPC MARKETPLACE HPC CAREER CENTER EXHIBITOR EVENTS HELP CENTER VENUE MAP MY DIGITAL TICKET

students, aiming to scale their applications efficiently across many GPUs. Attendees, interested in identifying, understanding, and resolving performance bottlenecks in multi-GPU applications. Established researchers, familiar with multi-GPU applications but wanting to learn new techniques and use the latest software and hardware features.

Beginner Level  
20%

Intermediate Level  
80%

Advanced Level  
30%

[Tutorial Website](#)

[Materials](#)

[Tutorial Evaluation](#)

Research Software Engineer  
Deutsches Klimarechenzentrum (DKRZ)

Alex Knigge  
Undergraduate Researcher  
University of New Mexico

Alexander Sinn  
Student  
Deutsches Elektronen-Synchrotron DESY

Alle anzeigen (18)

Member of the Helmholtz Association

# Summary, Conclusion

Efficient multi-node GPU computing is efficient multi-node computing with least possible amount of CPU

- Many advanced technologies and techniques in place to enable large-scale GPU applications
- GPU-aware MPI is key enabler
- On top / orthogonal: NCCL, NVSHMEM, ...→homework?
- Remember to [rate tutorial](#), give feedback! 
- *Appendix: [Links](#), [references](#)*

# Summary, Conclusion

Efficient multi-node GPU computing is efficient multi-node computing with least possible amount of CPU

- Many advanced technologies and techniques in place to enable large-scale GPU applications
- GPU-aware MPI is key enabler
- On top / orthogonal: NCCL, NVSHMEM, ... → homework
- Remember to [rate tutorial](#), give feedback!
- Appendix: [Links](#), [references](#)



Thank you  
for your attention!  
[a.herten@fz-juelich.de](mailto:a.herten@fz-juelich.de)

# Appendix

## References

# Links I

- [1] *Support of GPU-aware MPI in mpi4py.* URL:  
<https://mpi4py.readthedocs.io/en/stable/overview.html#support-for-gpu-aware-mpi>.
- [3] *Legate (Numpy).* URL: <https://github.com/nv-legate/legate.numpy>.
- [4] *NVIDIA: HPC-X.* URL: <https://docs.mellanox.com/category/hpcx>.
- [5] *MVAPICH2.* URL: <https://mvapich.cse.ohio-state.edu/>.
- [6] *NVIDIA: NCCL SHARP Plugin.* URL:  
<https://github.com/Mellanox/nccl-rdma-sharp-plugins>.
- [7] *Unified Communication Framework (UCF) Consortium.* URL:  
<https://ucfconsortium.org/>.
- [8] *NVIDIA: SHARP.* URL: <https://docs.mellanox.com/category/mlnxsharp>.

# References I

- [2] Michael Bauer and Michael Garland. “Legate NumPy: Accelerated and Distributed Array Computing.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. doi: [10.1145/3295500.3356175](https://doi.org/10.1145/3295500.3356175). URL: <https://doi.org/10.1145/3295500.3356175> (pages 45–51).
- [9] Gil Bloch. “SHARP Tutorial.” In: *HPC Advisory Council 2019 Lugano Workshop*. 2019. URL: [http://www.hpcadvisorycouncil.com/events/2019/swiss-workshop/pdf/020419/G\\_Bloch\\_Mellanox\\_SHARP\\_02042019.pdf](http://www.hpcadvisorycouncil.com/events/2019/swiss-workshop/pdf/020419/G_Bloch_Mellanox_SHARP_02042019.pdf) (pages 56–60).

# References II

- [10] Kawthar Shafie Khorassani et al. “Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences.” In: *High Performance Computing*. Ed. by Bradford L. Chamberlain et al. Cham: Springer International Publishing, 2021, pp. 118–136. ISBN: 978-3-030-78713-4. URL: [https://link.springer.com/chapter/10.1007%2F978-3-030-78713-4\\_7](https://link.springer.com/chapter/10.1007%2F978-3-030-78713-4_7) (page 62).