Multi-GPU Programming in NCCL and NVSHMEM

Jeff Hammond
Principal Engineer
GPU Communication Software

Questions to Answer

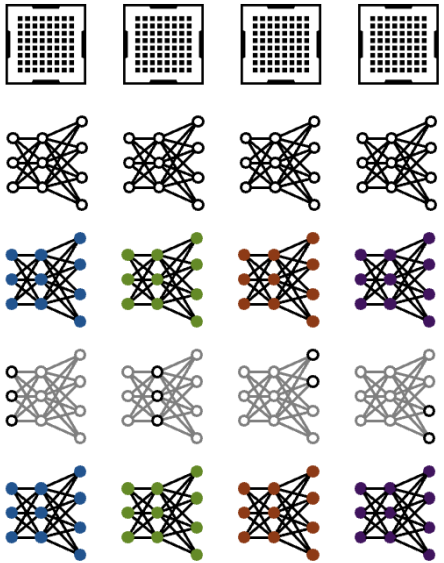
1. Why do we have two GPU communication libraries, NCCL and NVSHMEM? What is different about them?
2. Why do we need NCCL instead of MPI, if they are so similar?
3. How do various NCCL and NVSHMEM communication algorithms differ in usage and performance?
4. What are some forward-looking strategies for optimizing communication for GPU platforms?

Outline

1. Overview of NCCL and NVSHMEM
2. Historical context for NCCL and NVSHMEM
    a. Communication semantics and networking hardware
    b. GPU issues for MPI
3. Matrix transpose experiments
    a. Benchmark description, MPI/SHMEM examples
    b. All-to-all, point-to-point, one-sided, load-store with NCCL/NVSHMEM
    c. DGX-H100 performance
4. Take-away messages

Overview of NCCL and NVSHMEM

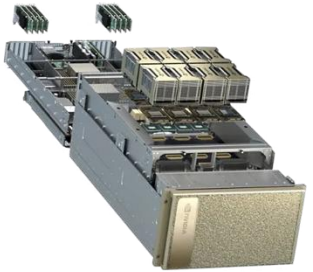
Overview of NCCL



Data Parallelism / FSDP	All-reduce, all-gather, reduce-scatter
Tensor Parallelism	All-reduce, all-gather, reduce-scatter
Pipeline Parallelism	Send / receive
Expert Parallelism	All-to-all

"NCCL: The Inter-GPU Communication Library Powering Multi-GPU AI", Sylvain Jeaugey. GTC25-S72583: https://www.nvidia.com/en-us/on-demand/session/gtc25-s72583/

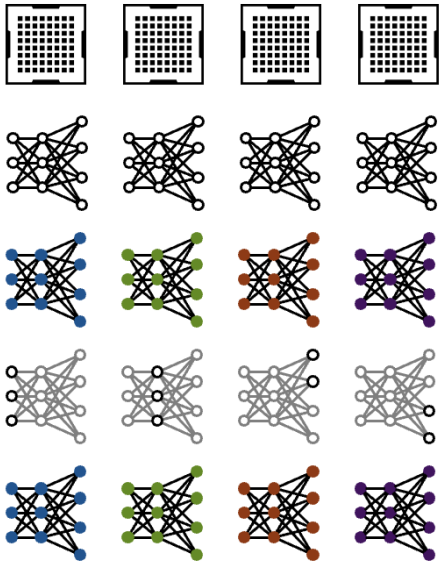
PCI Server



DGX/HGX

Large Systems

Overview of NCCL

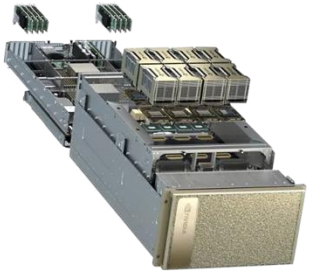


Data Parallelism / FSDP	All-reduce, all-gather, reduce-scatter
Tensor Parallelism	All-reduce, all-gather, reduce-scatter
Pipeline Parallelism	Send / receive
Expert Parallelism	All-to-all

Today's focus.

"NCCL: The Inter-GPU Communication Library Powering Multi-GPU AI", Sylvain Jeaugey. GTC25-S72583: https://www.nvidia.com/en-us/on-demand/session/gtc25-s72583/

PCI Server



DGX/HGX

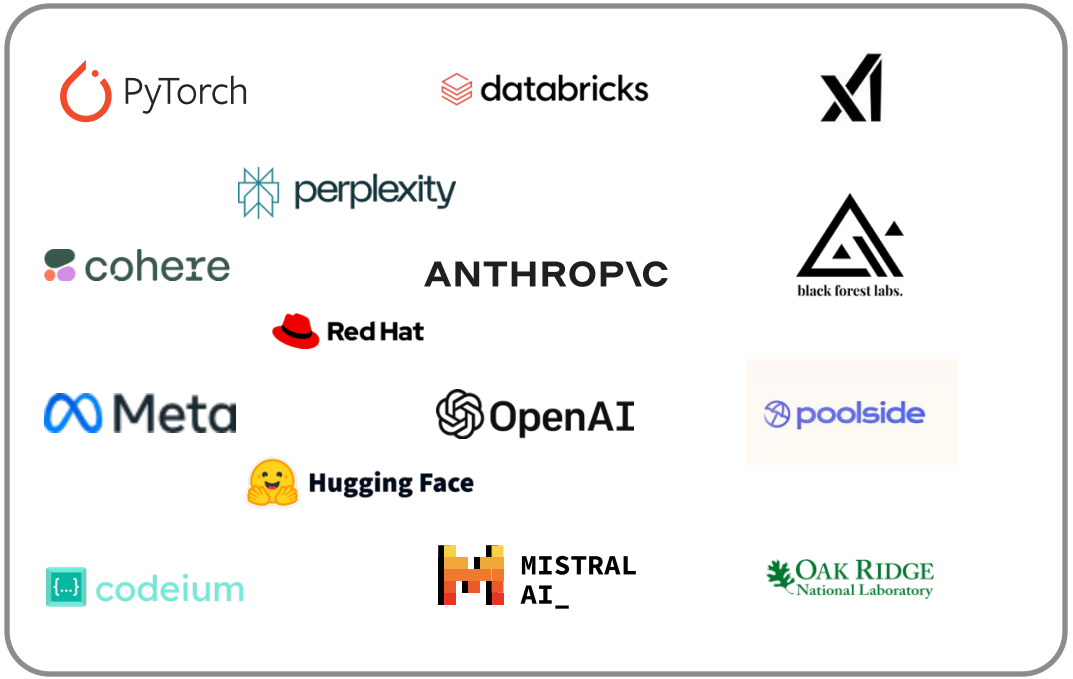
Large Systems

NCCL Ecosystem

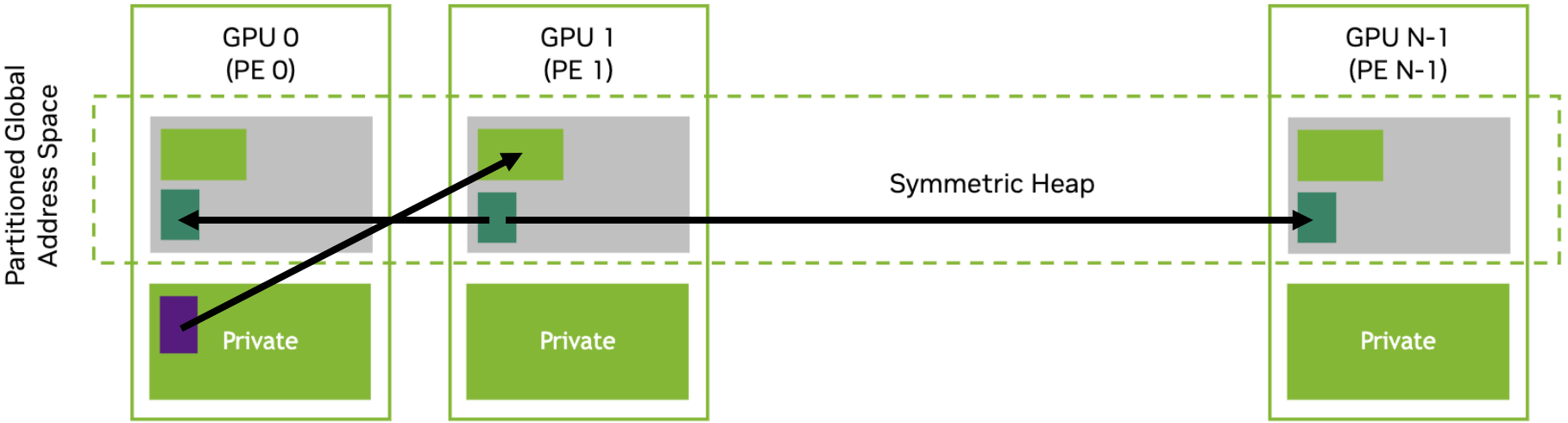
Enabling breakthrough AI research at unprecedented scales

CLOUD & INFRASTRUCTURE

AI INNOVATORS & RESEARCH



Overview of NVSHMEM



"NVSHMEM: GPU-Integrated Communication for NVIDIA GPU Clusters", Akhil Langer and Jim Dinan.
GTC21-S32515: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32515/

"GPU Communication Libraries for Accelerating HPC and AI Applications @ HotI 2025"
https://hoti.org/tutorials-nccl-nvshmem.html

NVSHMEM API Examples

```
// host APIs
void nvshmem_putmem(void *dest, const void * src, size_t nb, int pe)
void nvshmemx_putmem_on_stream(void *dest, const void * src, size_t nb,
                               int pe, cudaStream_t stream)

// device APIs
void nvshmem_putmem(void *dest, const void *src, size_t nb, int pe)
void nvshmemx_putmem_block(void *dest, const void * src, size_t nb, int pe)
void nvshmemx_putmem_warp(void *dest, const void * src, size_t nb, int pe)

// device-initiated collectives... (cooperative group launch required)
```

"NVSHMEM: GPU-Integrated Communication for NVIDIA GPU Clusters", Akhil Langer and Jim Dinan.
GTC21-S32515: https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32515/

"GPU Communication Libraries for Accelerating HPC and AI Applications @ HotI 2025"
https://hoti.org/tutorials-nccl-nvshmem.html

NVSHMEM AI Ecosystem

C Host and Device APIs

NVSHMEM4Py

Use-cases

Custom Communication Kernels (EP in MoE)

Fused compute-comm kernel

Zero-SM and low latency collectives

One-sided pt-to-pt comms

Organizations

deepseek

perplexity

ByteDance

NVIDIA

Meta

Google

Libraries/Frameworks

OpenXLA

FlashInfer

PyTorch

Google: JAX - Pallas & MosaicGPU
Perplexity: PPLX kernels
Deepseek: DeepEP library
Meta: PyTorch SymmetricMemory
ByteDance: Triton Distributed, TileLink, FLUX
Nvidia: NeMO, cuBLASMp, cuFFTMp, nvmath-python, Numba-CUDA (NVSHMEM 3.5)

A Brief History of Communication Libraries

The Two Paradigms of HPC Communication

- Two-sided aka message passing, as exemplified by MPI-1.
  - The MPI Forum started in 1993, to standardize what was then a large set of similar but incompatible messaging libraries.
  - MPI Send-Recv can be implemented using e.g. POSIX sockets and thus are considered universally portable.
  - MPI was designed when CPUs were (often) much faster than networks…
  - Two-sided communication combines synchronization and data movement.
- One-sided aka remote memory access (RMA), as exemplified by SHMEM.
  - SHMEM was created for the Cray T3D, which was a revolutionary MPP system that supported direct access to remote memory.
  - (Open)SHMEM's design assumes an SMP or RDMA network and discourages implementations that don't support asynchronous progress in hardware.
  - One-sided communication decouples synchronization and data movement.

Brief MPI history: https://www.hpcwire.com/2017/05/01/mpi-25-years-old/
Original SHMEM paper: https://cug.org/5-publications/proceedings_attendee_lists/1997CD/S95PROC/303_308.PDF

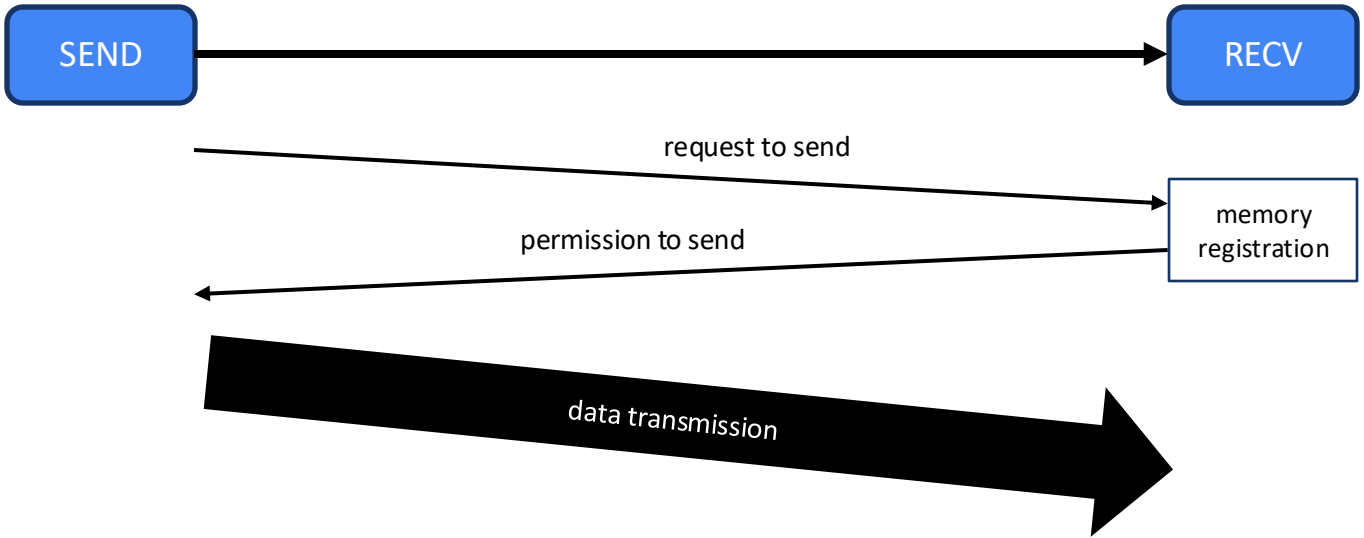
Two-sided communication

Sender knows:
Input buffer address
Input buffer size
Input data type
Message tag
Receiver ID

The matching protocol allows the input buffer to be written into the output buffer.

Receiver knows:
Output buffer address
Output buffer size
Output data type
Message tag
Sender ID

Two-sided communication



OSU paper on MPI Send-Recv on IB with RDMA:
https://mvapich.cse.ohio-state.edu/static/media/publications/abstract/surs-ppopp06.pdf

One-sided communication

Initiator knows:
Input buffer address
Output buffer address/offset
Buffer size
Data type
Target ID

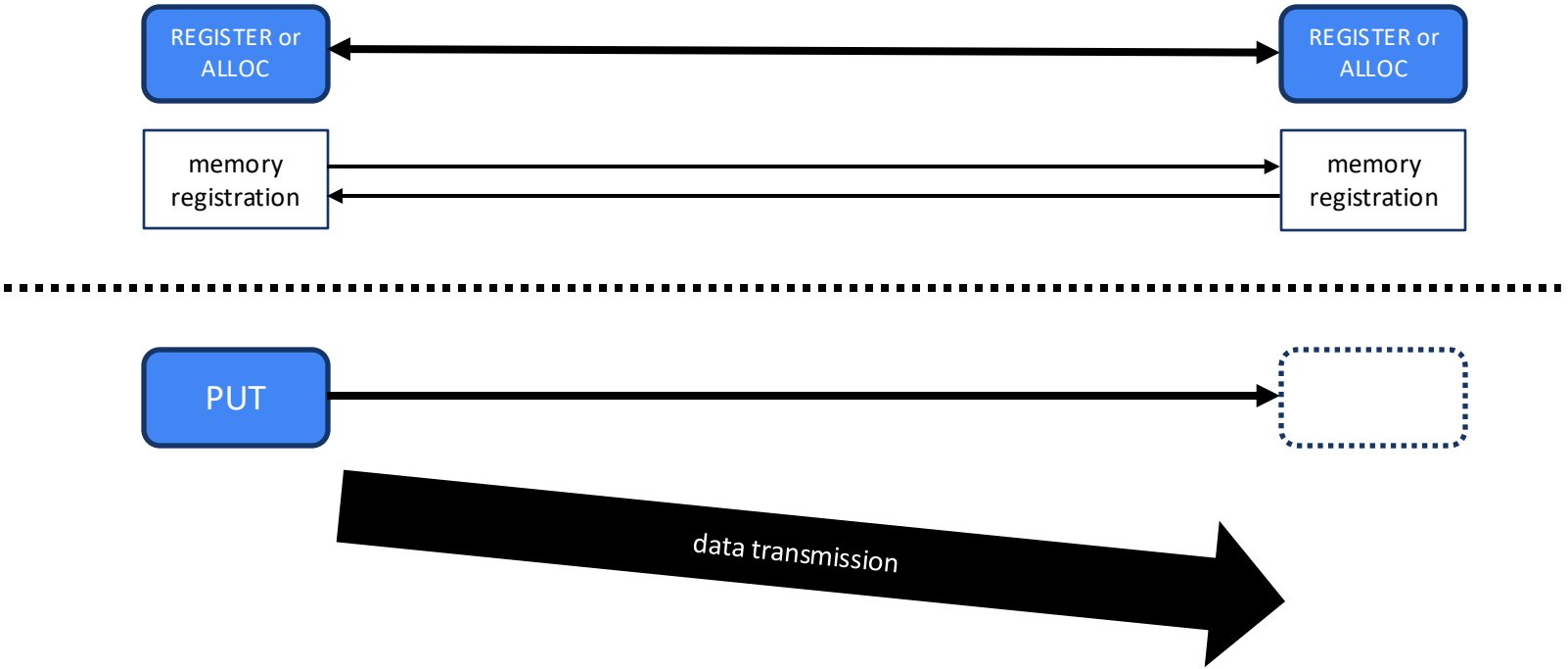
Before the PUT call is made, memory for the target buffers is registered with both sides.

One-sided communication

Before the GET call is made, memory for the target buffers is registered with both sides.

Initiator knows:
Input buffer address/offset
Output buffer address
Buffer size
Data type
Target ID

One-sided communication



Why does this matter?

- If the synchronizing steps do not line up, processors may idle while waiting.
- Sequential bottlenecks in GPU code should be avoided.
- Synchronizing GPU blocks/threads should also be avoided.
- Data transmission is much easier to offload to specialized hardware (RDMA interconnects like IB, GPU copy engines aka CE) than message matching logic.

Most important AI/HPC communication patterns are persistent, so amortizing away setup costs is beneficial.

MPI vs NCCL

The most obvious difference is stream support, but there are more differences:

- MPI allows underflow: send_count < recv_count.
- MPI datatypes allow arbitrary, nested, heterogeneous, noncontiguous data.
- MPI supports tags, which distinguish messages within a communicator.
- MPI supports “wildcards” (e.g. ANY_SOURCE), which make message matching hard.
- MPI supports multiple ranks per GPU.

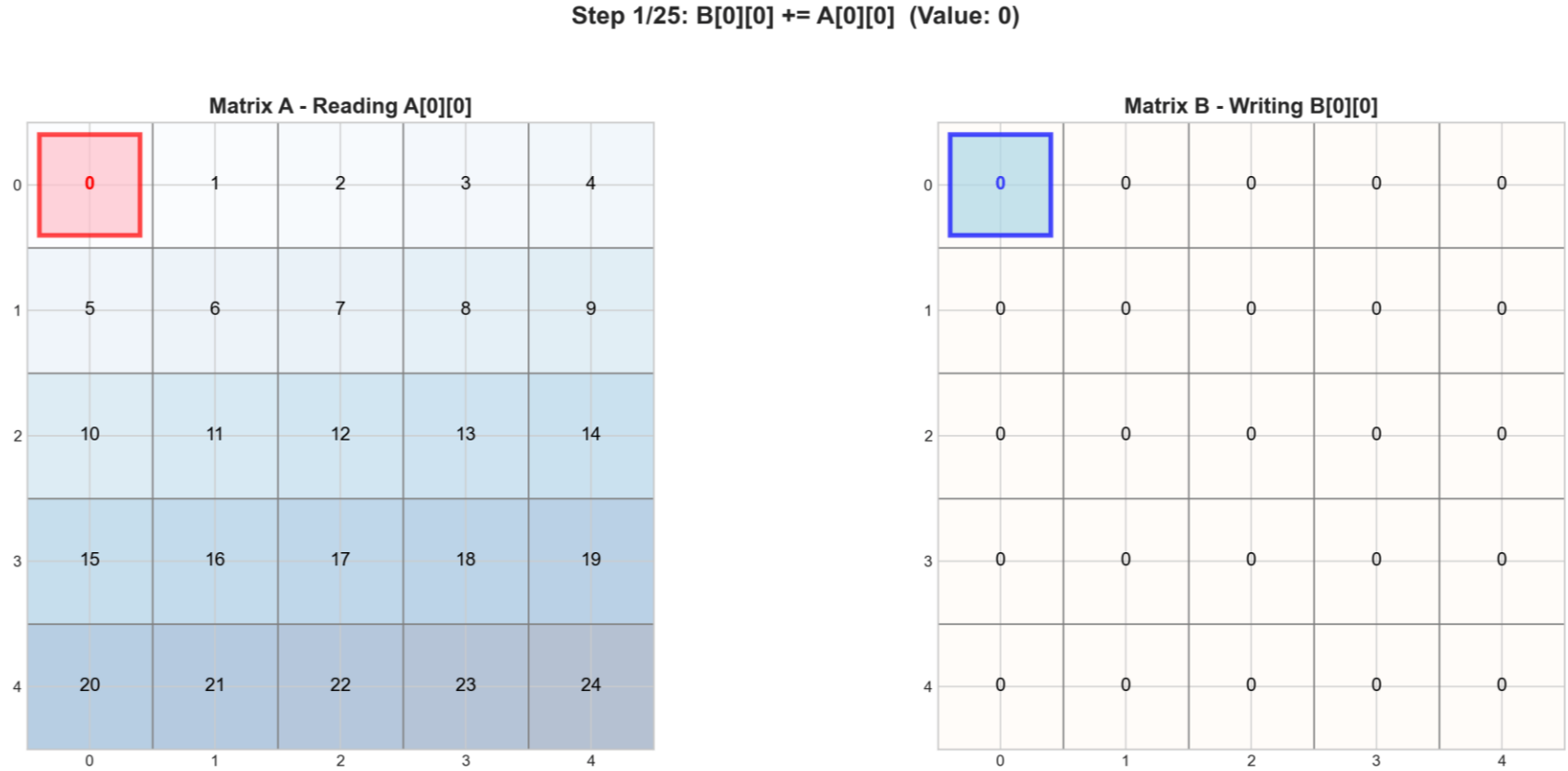
NCCL supports none of these features. MPI 4.0 allows some of them to be disabled, but as they are on by default, little to no effort has gone into optimizing for these scenarios.

NCCL also supports multiple ranks (i.e. GPUs) per process, which some AI workloads require.

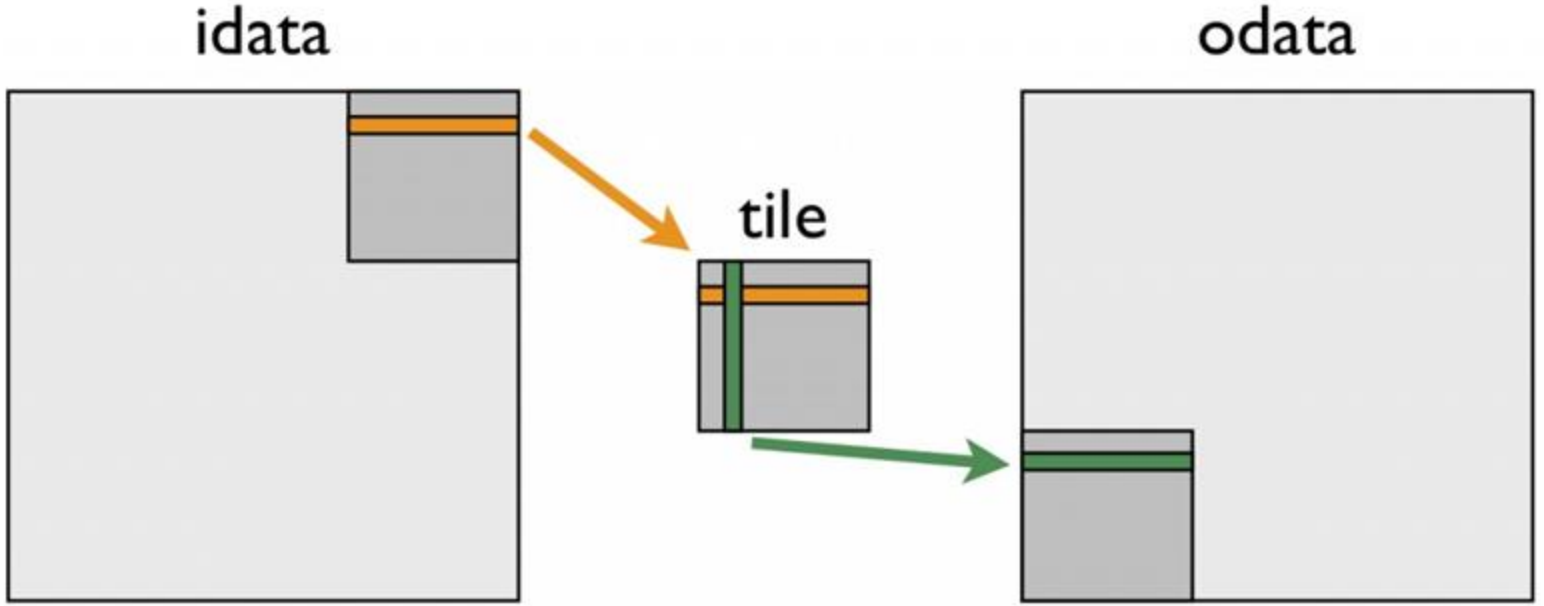
Mitigating MPI Message Matching Misery. ISC 2016. https://doi.org/10.1007/978-3-319-41321-1_15

Matrix Transpose

for (i,j) in matrix:
  B(i,j) += A(j,i)
  A(j,i) += 1.0 # increment A for verification because $B = (B^T)^T$

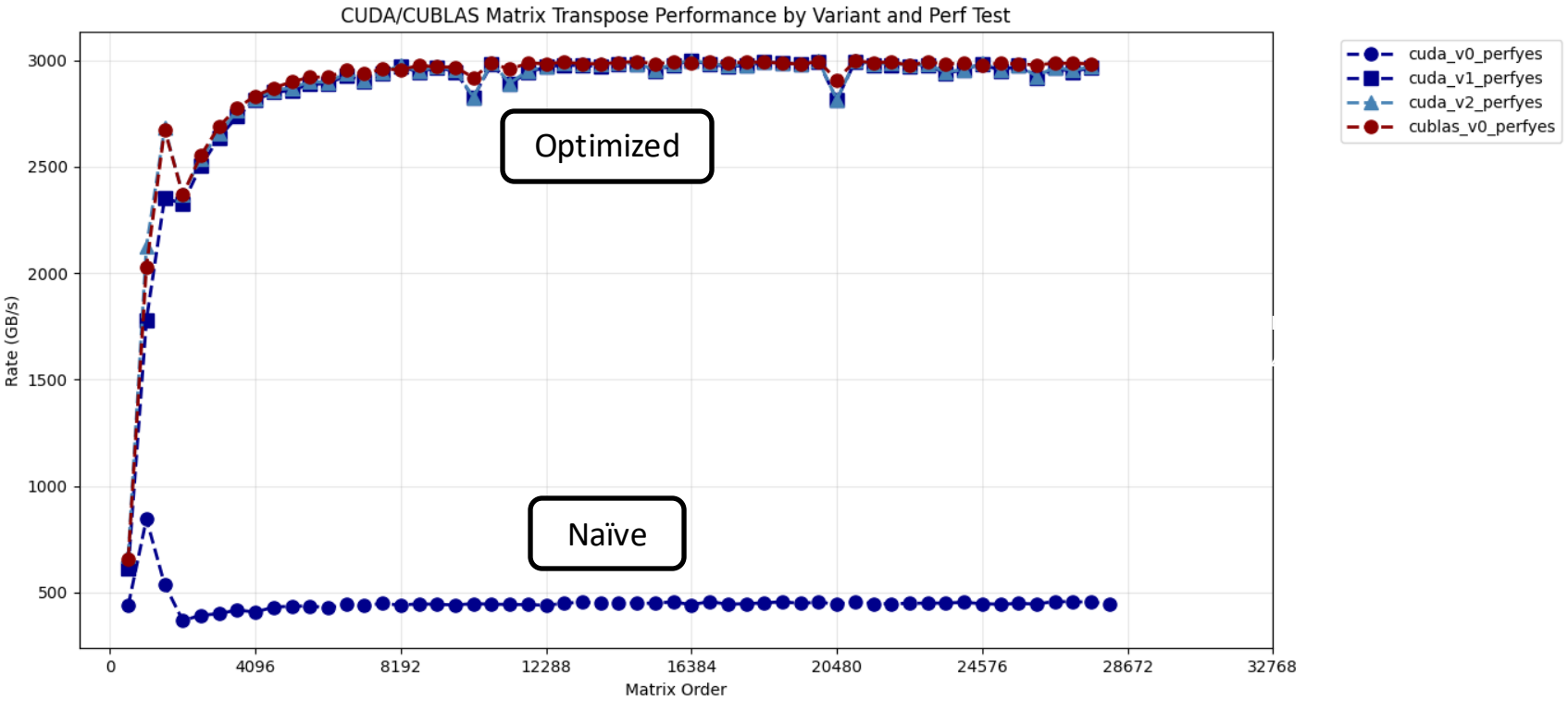


CUDA transpose



Mark Harris' blog post on matrix transpose:
https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/

CUDA transpose - single GPU performance



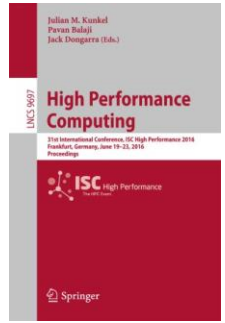
https://github.com/ParRes/Kernels/blob/main/Cxx11/transpose-cuda.cu

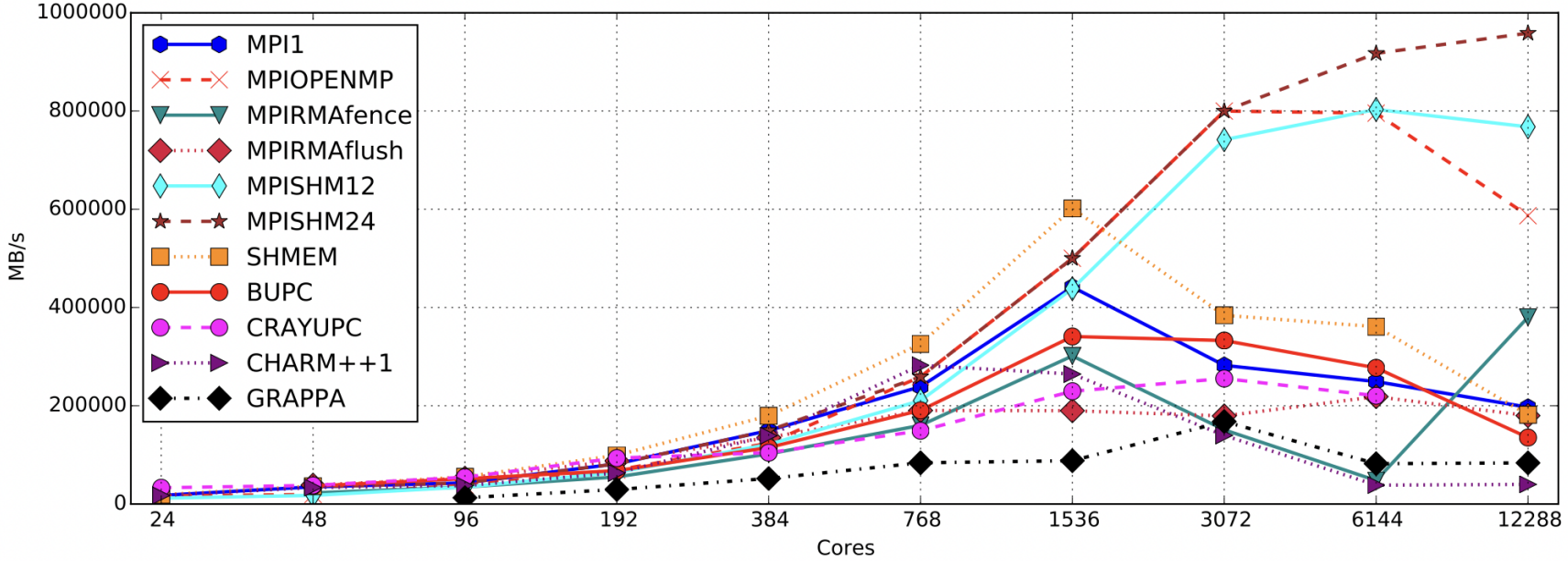
Distributed Matrix Transpose

Comparing runtime systems with exascale ambitions using the Parallel Research Kernels

R. F. Van der Wijngaart[1], A. Kayi[1], J. R. Hammond[1], G. Jost[1], T. St. John[1], S. Sridharan[1], T. G. Mattson[1], J. Abercrombie[2], and J. Nelson[2]

[1] Intel Corporation, Hillsboro, Oregon, USA.
[2] University of Washington, Seattle, WA, USA.

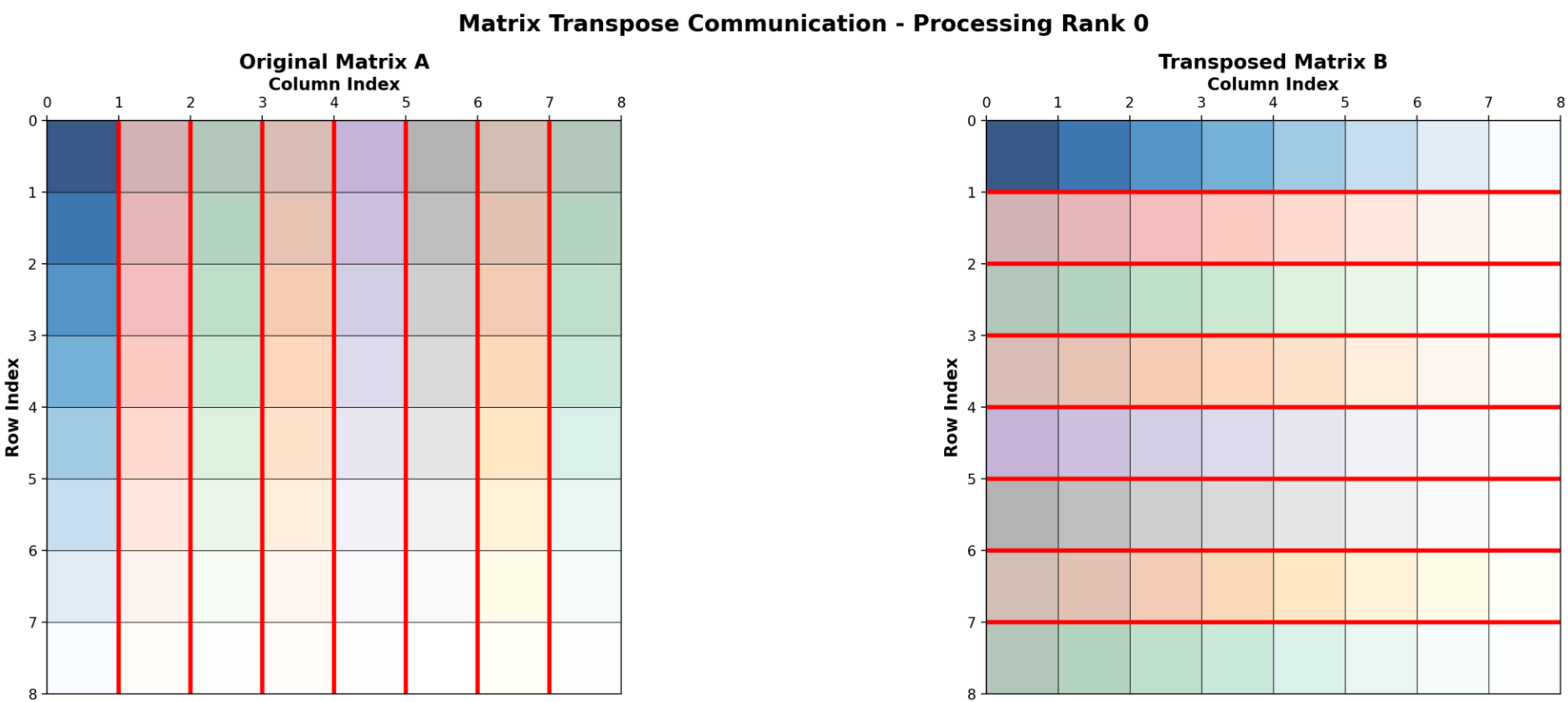




Source Code:
https://github.com/ParRes/Kernels/

Paper: https://link.springer.com/book/10.1007/978-3-319-41321-1
Presentation on the PRK project: https://youtu.be/HTbjM5GDIRM

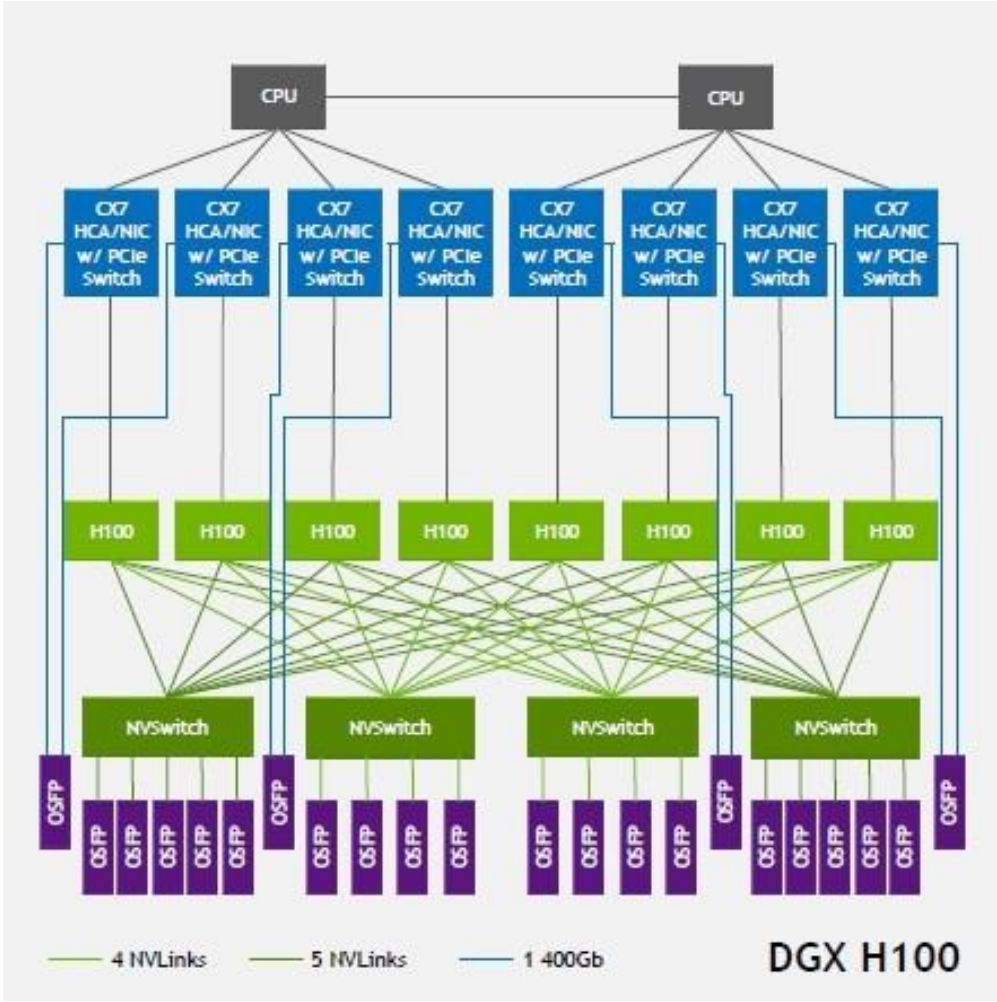
NVIDIA



Column blocks are mapped onto GPUs 0..7

DGX-H100

Spec	Per H100 GPU	System Total
FP64 Performance	34 TFLOPS	272 TFLOPS
FP64 Tensor Core	67 TFLOPS	536 TFLOPS
Memory Bandwidth	3.35 TB/s	26.8 TB/s
GPU Memory	80 GB HBM3	640 GB
NVLink Bandwidth	900 GB/s bidirectional	3.6 TB/s bisection

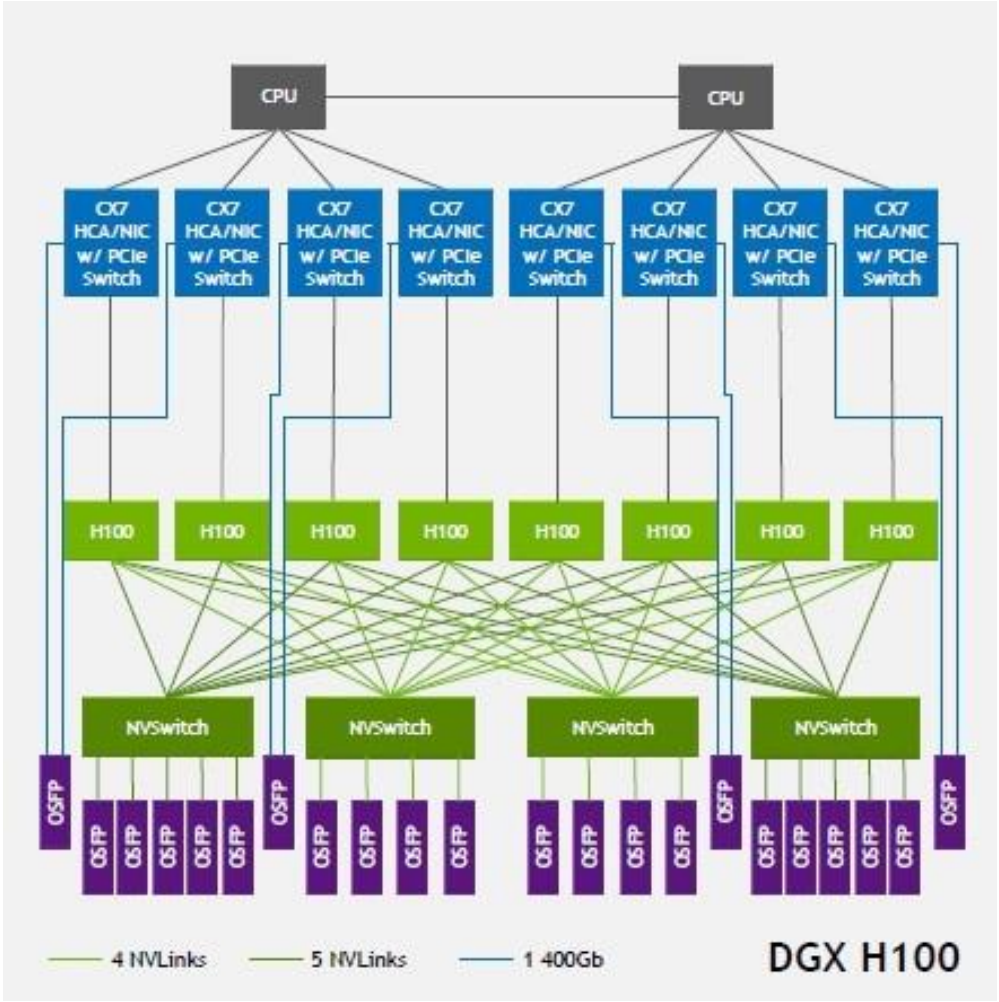


See NVLink @ https://hc34.hotchips.org/

DGX-H100

~24 TB/s and ~3 TB/s are the goal for local and remote bandwidth.

Spec	Per H100 GPU	System Total
FP64 Performance	34 TFLOPS	272 TFLOPS
FP64 Tensor Core	67 TFLOPS	536 TFLOPS
Memory Bandwidth	3.35 TB/s	26.8 TB/s
GPU Memory	80 GB HBM3	640 GB
NVLink Bandwidth	900 GB/s bidirectional	3.6 TB/s bisection

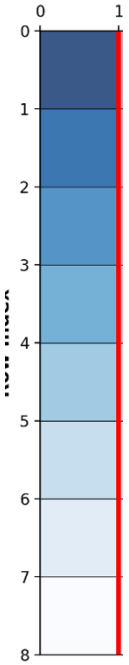


See NVLink @ https://hc34.hotchips.org/

Communication Strategies

MPI all-to-all implementation

```
// A is the source, B is the target, T is a temporary of the same size:

// prk::MPI wrappers infer type and default to MPI_COMM_WORLD
prk::MPI::alltoall(A.data(), block_order*block_order,
                   T.data(), block_order*block_order);

// transpose the matrix
for (int r=0; r<np; r++) {
  const size_t offset = block_order * block_order * r;
  transpose_block(B.data() + offset,
                  T.data() + offset,
                  block_order, tile_size);
}

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```



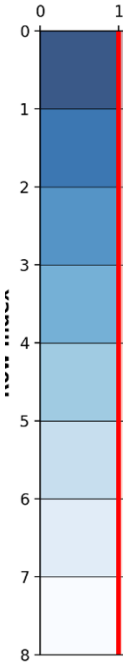
MPI all-to-all implementation

Step 1: redistribute A into T

T is a temporary of the same size:

```
// prk::MPI wrappers infer type and default to MPI_COMM_WORLD
prk::MPI::alltoall(A.data(), block_order*block_order,
                   T.data(), block_order*block_order);
```

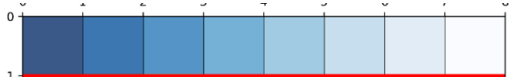
```
// transpose the matrix
for (int r=0; r<np; r++) {
  const size_t offset = block_order * block_order * r;
  transpose_block(B.data() + offset,
                  T.data() + offset,
                  block_order, tile_size);
}

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```



MPI all-to-all implementation

```
// A is the source, B is the target, T is a temporary of the same size:

// prk::MPI wrappers infer type and default to MPI_COMM_WORLD
                                                    ck_order,
                                                    er);
```

Step 2: transpose each block of T into B



```
// transpose the matrix
for (int r=0; r<np; r++) {
  const size_t offset = block_order * block_order * r;
  transpose_block(B.data() + offset,
                  T.data() + offset,
                  block_order, tile_size);
}
```

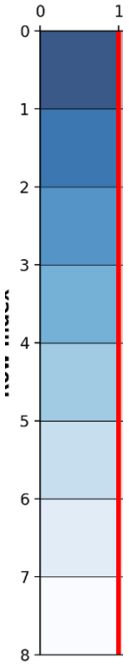
```
// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

MPI all-to-all implementation

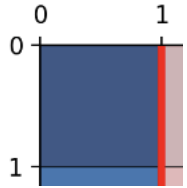
```
// A is the source, B is the target, T is a temporary of the same size:

// prk::MPI wrappers infer type and default to MPI_COMM_WORLD
prk::MPI::alltoall(A.data(), block_order*block_order,
                   T.data(), block_order*block_order);

// transpose the matrix
for (int r=0; r<np; r++) {
  const size_t offset = block_order * block_order * r;
  transpose_block(B.data() + offset,
                  T.data() + offset,
                  block_order, tile_size);
```

Step 3: update A

```
// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

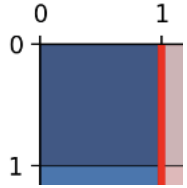


MPI point-to-point implementation




```
// A is the source, B is the target; T is a temporary for one block:

// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  const int send_to = (me - r + np) % np;
  size_t offset = block_order * block_order * send_to;
  prk::MPI::sendrecv(A.data() + offset, send_to,
                     T.data(), recv_from,
                     block_order*block_order);

  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
}

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

MPI point-to-point implementation



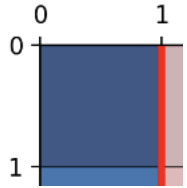

```
// A is the source, B is the target; T is a temporary for one block:
```

Step 1: redistribute A into T one block at a time

```
// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  const int send_to = (me - r + np) % np;
  size_t offset = block_order * block_order * send_to;
  prk::MPI::sendrecv(A.data() + offset, send_to,
                     T.data(), recv_from,
                     block_order*block_order);

  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
}

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

Circulant shift to avoid network hotspots.

MPI point-to-point implementation

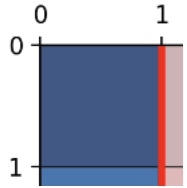



```
// A is the source, B is the target; T is a temporary for one block:

// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  const int send_to = (me - r + np) % np;
  size_t offset = block_order * block_order * send_to;
  prk::MPI::sendrecv(A.data() + offset, send_to,
```

Step 2: transpose the T block into a block of B

```
  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
}

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

MPI point-to-point implementation



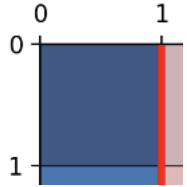

```
// A is the source, B is the target; T is a temporary for one block:

// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  const int send_to = (me - r + np) % np;
  size_t offset = block_order * block_order * send_to;
  prk::MPI::sendrecv(A.data() + offset, send_to,
                     T.data(), recv_from,
                     block_order*block_order);

  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
```

Increment A block could happen here…

Step 3: update A

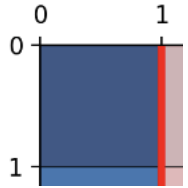
```
// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
```

OpenSHMEM one-sided (Get) implementation




```
// A is the source, B is the target; T is a temporary for one block:

// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  size_t offset = block_order * block_order * me;
  prk::SHMEM::get(T.data(), A.data() + offset,
                  block_order * block_order, recv_from);

  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
}
prk::SHMEM::barrier();

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
prk::SHMEM::barrier();
```

OpenSHMEM one-sided (Get) implementation




```
// A is the source, B is the target; T is a temporary for one block:

// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  size_t offset = block_order * block_order * me;
  prk::SHMEM::get(T.data(), A.data() + offset,
                  block_order * block_order, recv_from);

  offset = block_order * block_order * recv_from;
  transpose_block(B.data() + offset, T.data(), block_order, tile_size);
}
prk::SHMEM::barrier();

// increment A
std::transform(A.begin(), A.end(), A.begin(), [](auto a) { return a + 1; });
prk::SHMEM::barrier();
```

With one-sided, we need additional synchronization to prevent reading or writing data before it is ready.

GPU Communication

NCCL “all-to-all” implementation

```
// Details on the next slide
prk::NCCL::alltoall(A, T, block_order*block_order, nccl_comm_world);

// transpose the matrix
if (bulk) {
  // bulk: iterate over all np blocks on the GPU using CUDA Z-dim
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```

NCCL “all-to-all” implementation

```
// Details on the next slide
prk::NCCL::alltoall(A, T, block_order*block_order, nccl_comm_world);
```

```
// transpose the matrix
if (bulk) {
  // bulk: iterate over all np blocks on the GPU using CUDA Z-dim
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```

NCCL “all-to-all” implementation

```
// Details on the next slide
prk::NCCL::alltoall(A, T, block_order*block_order, nccl_comm_world);

// transpose the matrix
if (bulk) {
  // bulk: iterate over all np blocks on the GPU using CUDA Z-dim
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```

NCCL “all-to-all” implementation

```
// Details on the next slide
prk::NCCL::alltoall(A, T, block_order*block_order, nccl_comm_world);

// transpose the matrix
if (bulk) {
  // bulk: iterate over all np blocks on the GPU using CUDA Z-dim
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```

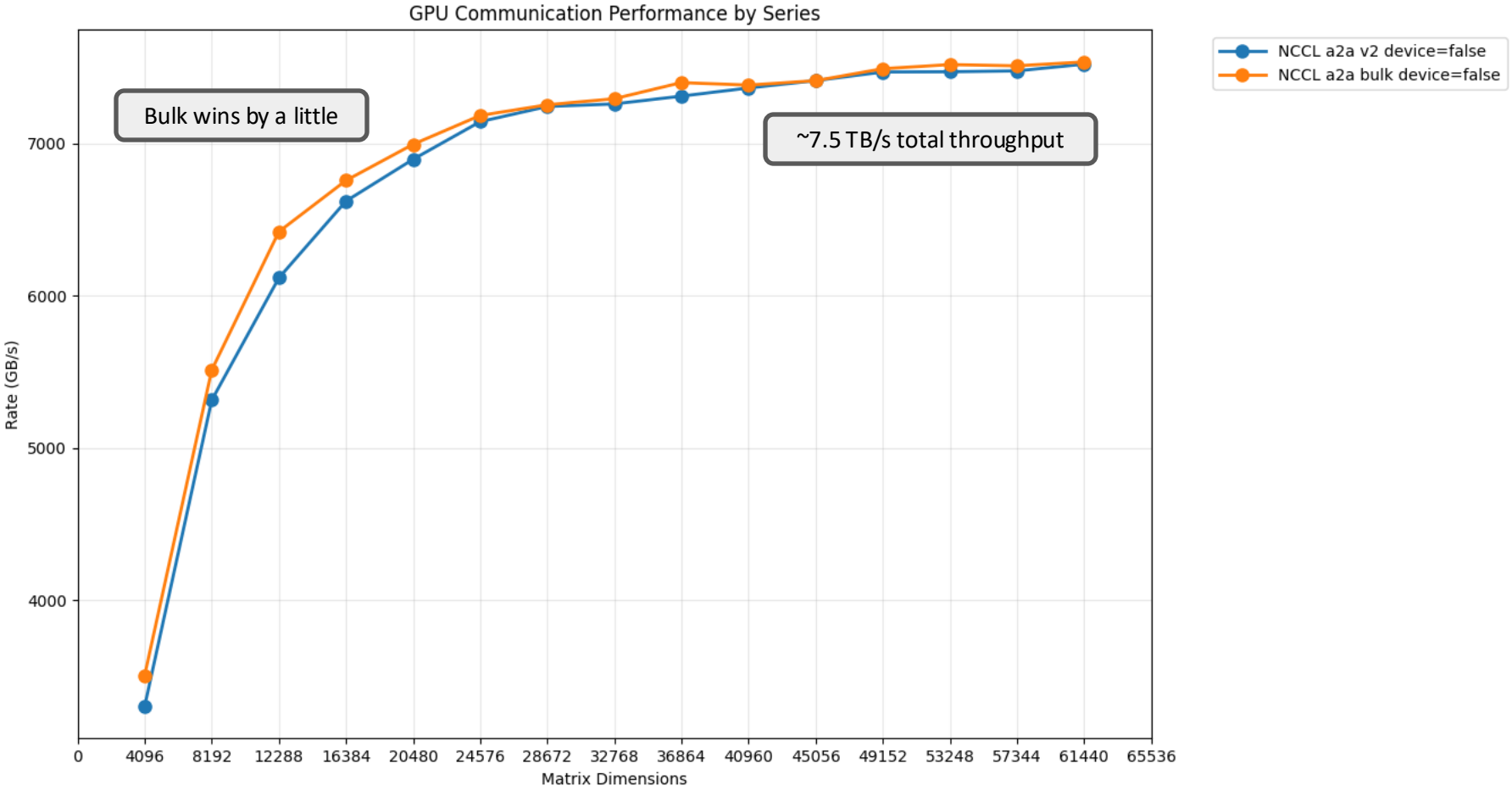
NCCL “all-to-all” implementation

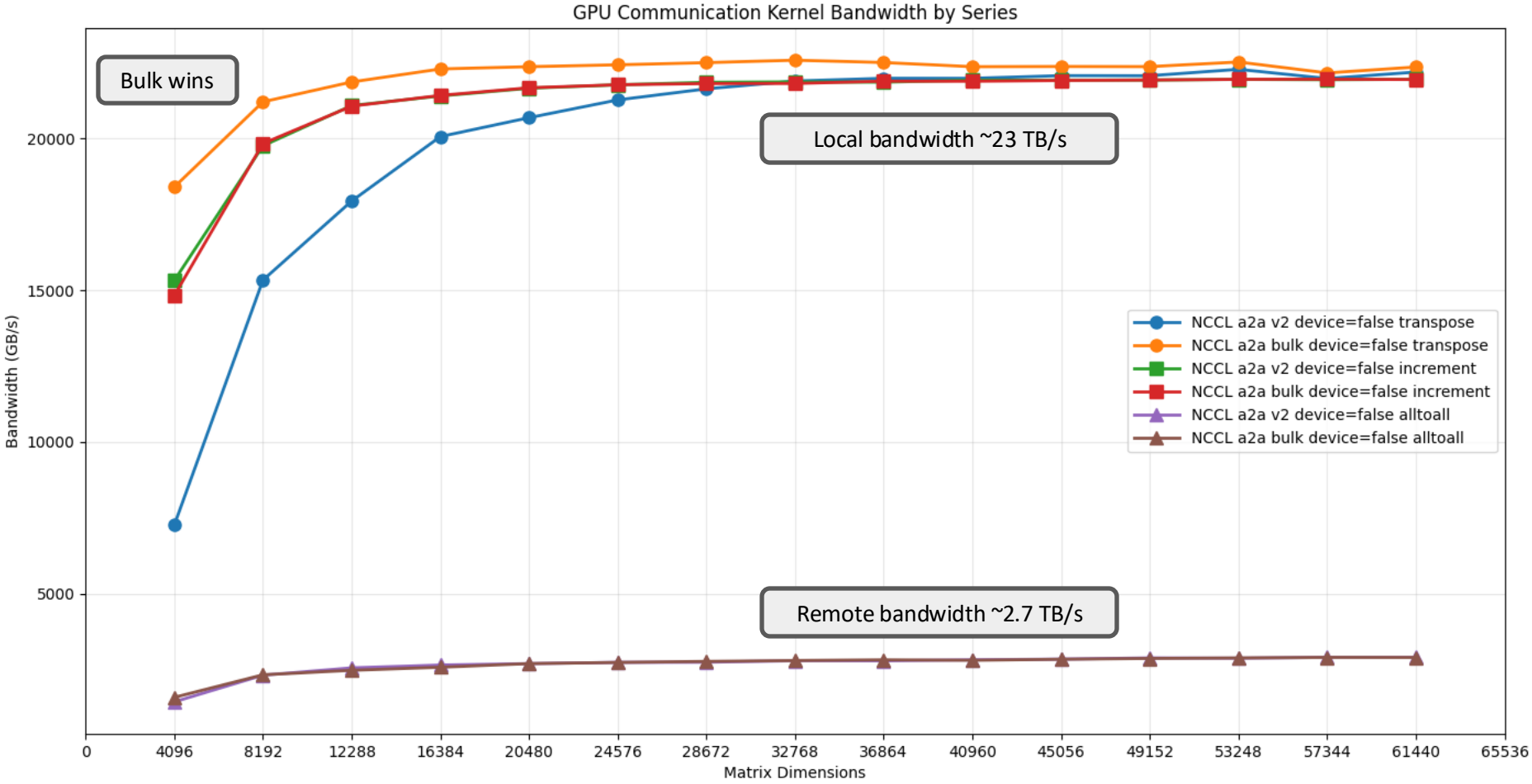
```
// Details on the next slide
prk::NCCL::alltoall(A, T, block_order*block_order, nccl_comm_world);

// transpose the matrix
if (bulk) {
  // bulk: iterate over all np blocks on the GPU using CUDA Z-dim
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}
```

```
// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```

NCCL “all-to-all” implementation

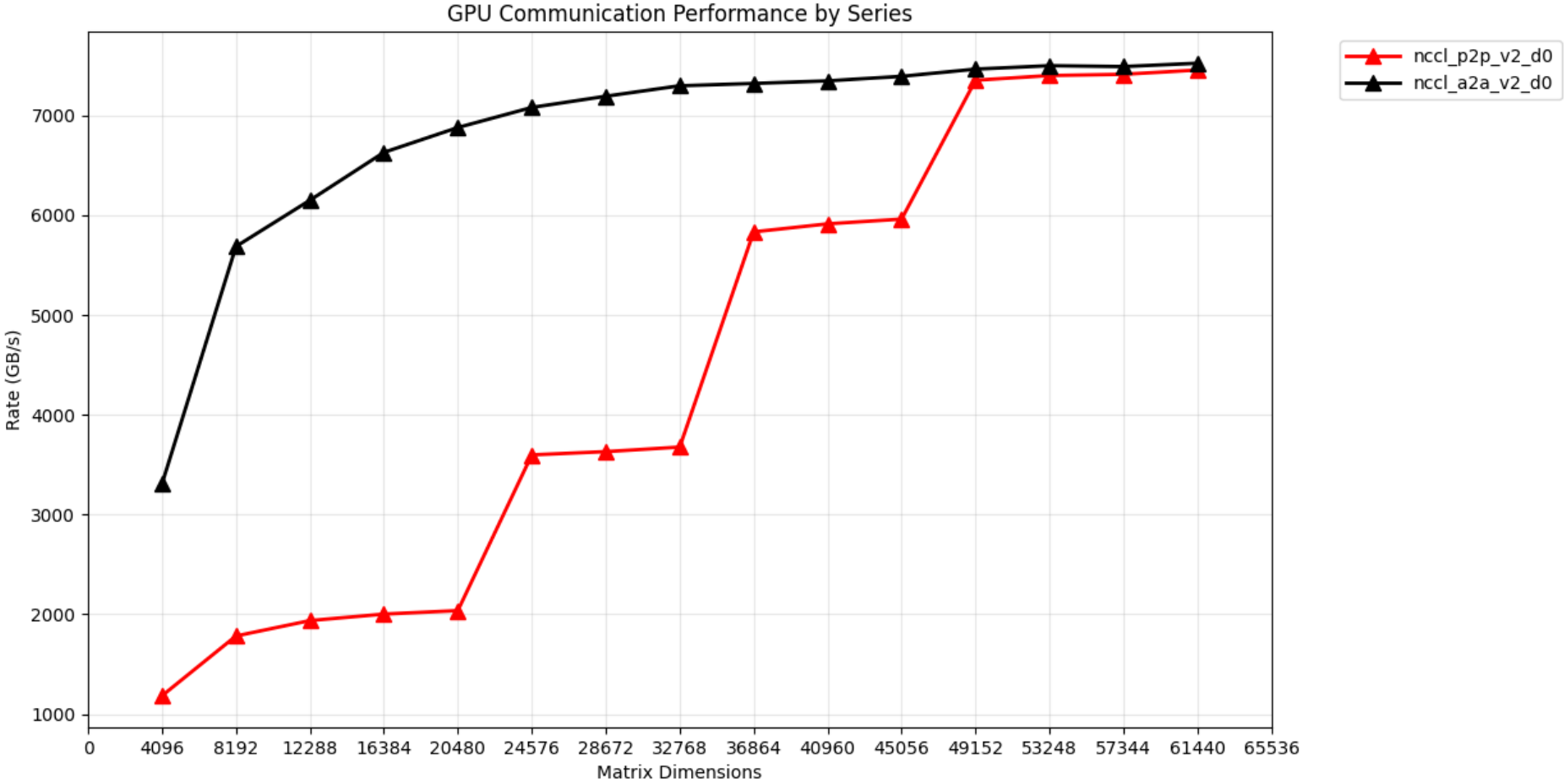
```
template <typename T>
void alltoall(const T * sbuffer, T * rbuffer, size_t count,
              ncclComm_t comm, cudaStream_t stream = 0)
{
  ncclDataType_t type = get_NCCL_Datatype(*sbuffer);

  int np;
  prk::check( ncclCommCount(comm, &np) );

  prk::check( ncclGroupStart() );
  for (int r=0; r<np; r++) {
    prk::check( ncclSend(sbuffer + r*count, count, type, r, comm, stream) );
    prk::check( ncclRecv(rbuffer + r*count, count, type, r, comm, stream) );
  }
  prk::check( ncclGroupEnd() );
}
```

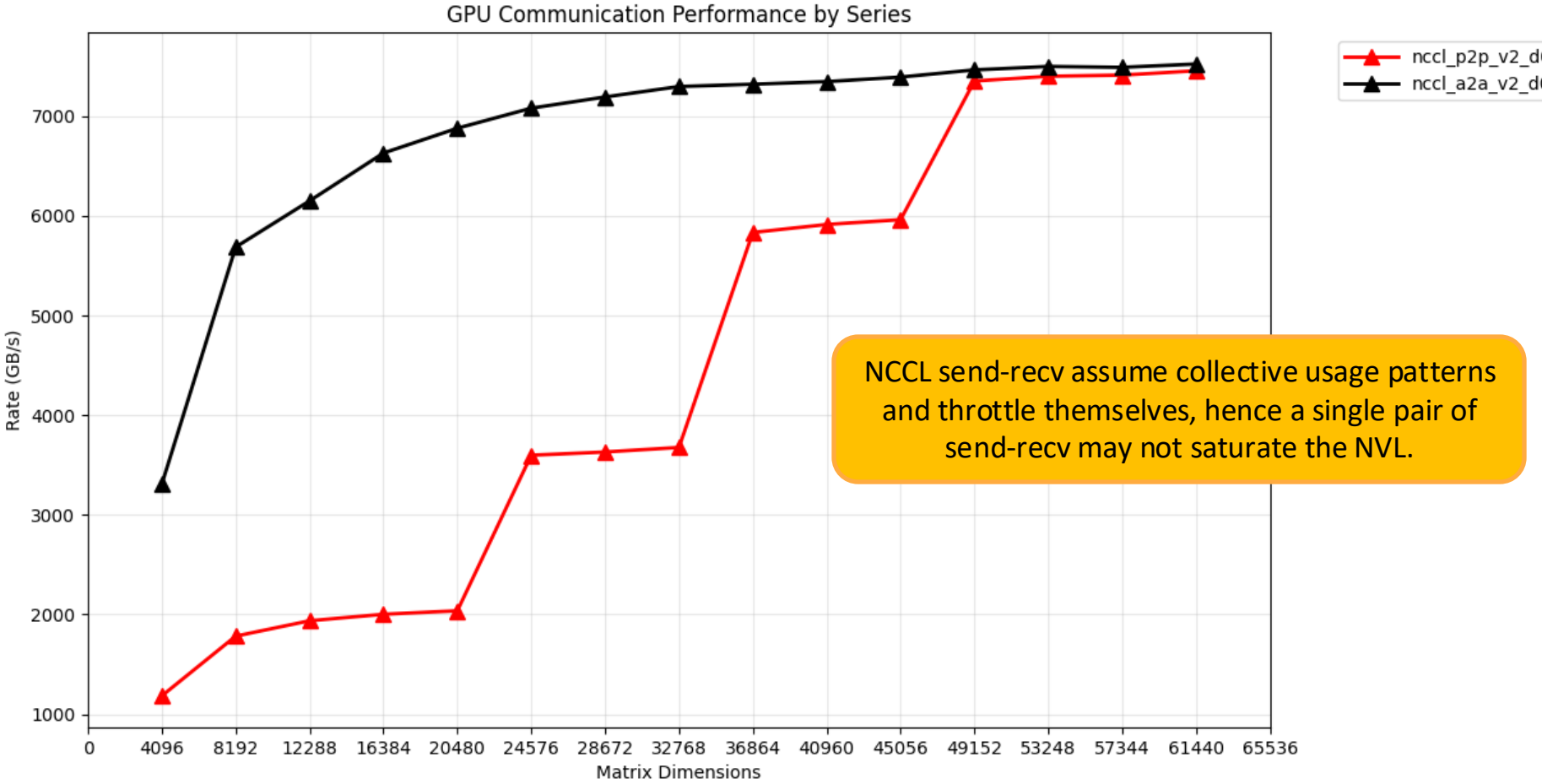

GPU Communication Performance by Series
Bulk wins by a little
~7.5 TB/s total throughput
NCCL a2a v2 device=false
NCCL a2a bulk device=false
Rate (GB/s)
7000
6000
5000
4000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
Matrix Dimensions


GPU Communication Kernel Bandwidth by Series
Bulk wins
Local bandwidth ~23 TB/s
Remote bandwidth ~2.7 TB/s
NCCL a2a v2 device=false transpose
NCCL a2a bulk device=false transpose
NCCL a2a v2 device=false increment
NCCL a2a bulk device=false increment
NCCL a2a v2 device=false alltoall
NCCL a2a bulk device=false alltoall
Bandwidth (GB/s)
20000
15000
10000
5000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
Matrix Dimensions

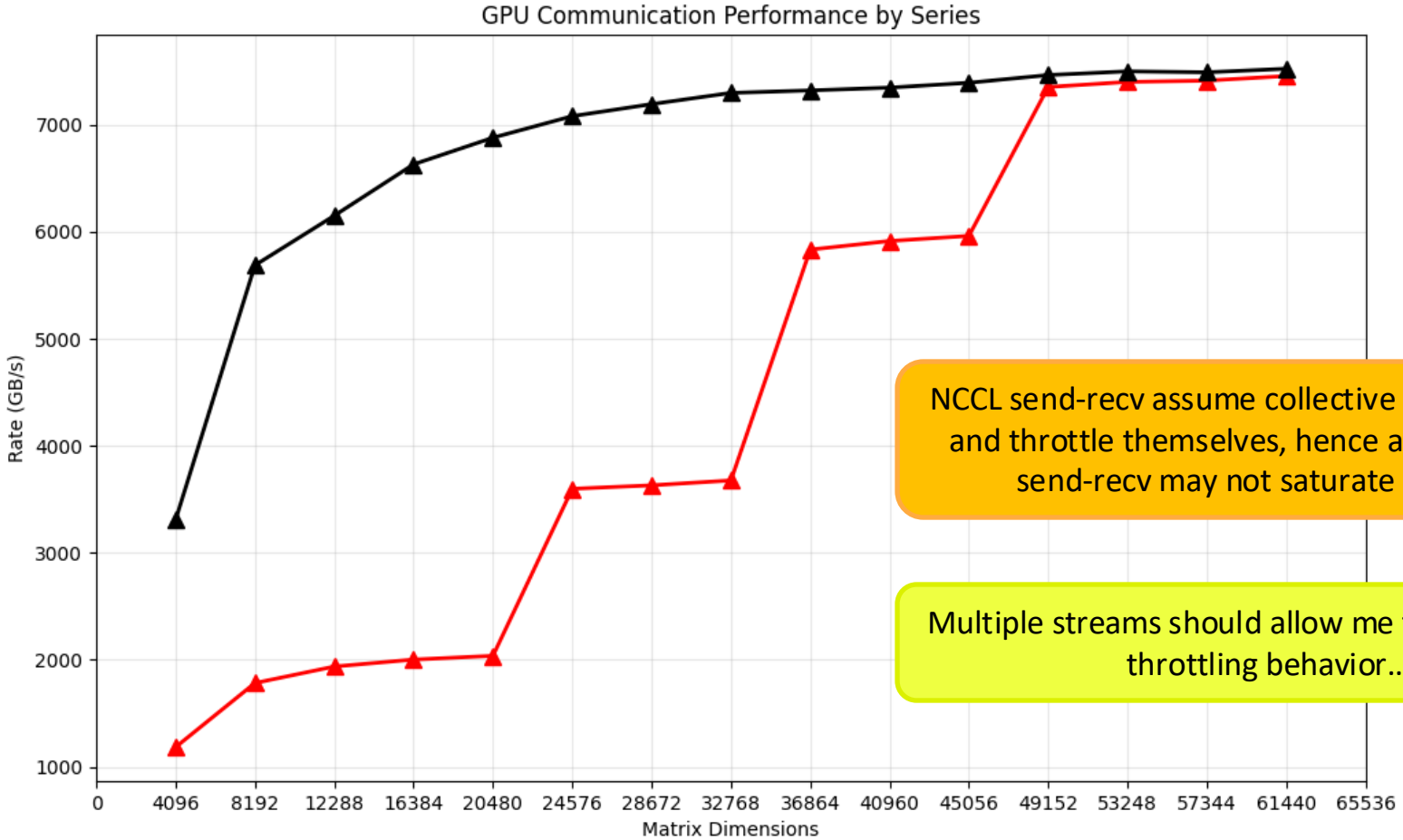
NCCL point-to-point implementation

```
// transpose the matrix
for (int r=0; r<np; r++) {
  const int recv_from = (me + r) % np;
  const int send_to = (me - r + np) % np;
  size_t offset = block_order * block_order * send_to;
  // this is just NCCL group { send, recv }
  prk::NCCL::sendrecv(A + offset, send_to, T, recv_from,
                      block_order*block_order, nccl_comm_world);

  offset = block_order * block_order * recv_from;
  transpose<<<dimGrid, dimBlock>>>(block_order, T, B + offset);
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```


GPU Communication Performance by Series
nccl_p2p_v2_d0
nccl_a2a_v2_d0
Rate (GB/s)
1000
2000
3000
4000
5000
6000
7000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
Matrix Dimensions



NCCL send-recv assume collective usage patterns and throttle themselves, hence a single pair of send-recv may not saturate the NVL.



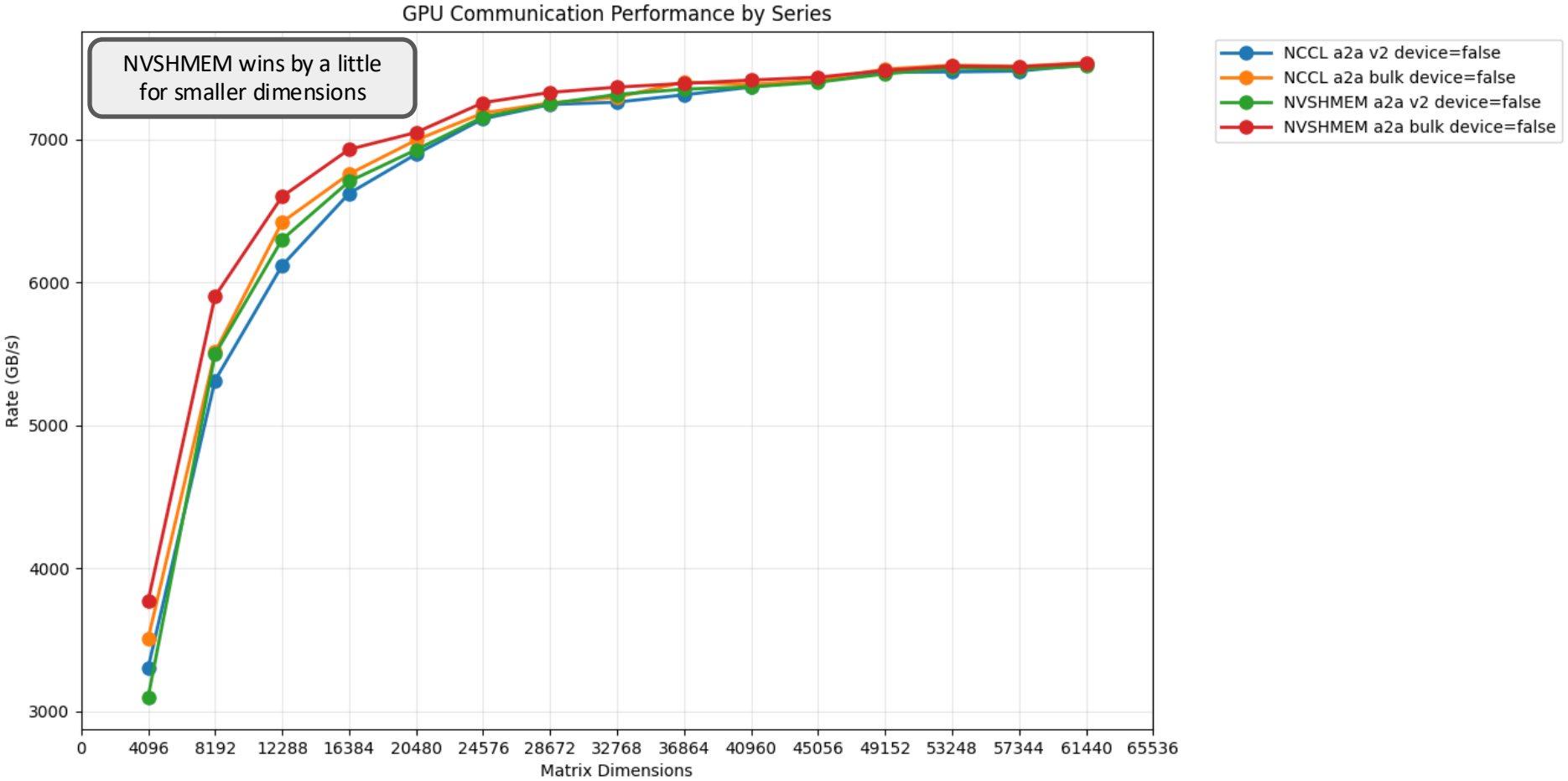
NCCL send-recv assume collective usage patterns and throttle themselves, hence a single pair of send-recv may not saturate the NVL.

Multiple streams should allow me to override the throttling behavior…

Josh Romero's cuDecomp uses auto-tuning to produce optimal communication schedules for alltoall: https://github.com/NVIDIA/cuDecomp

NVSHMEM all-to-all implementation

```
// alltoall does not fully synchronize; as-if put-to-all targets
prk::NVSHMEM::barrier(false /* no memory barrier */ );
prk::NVSHMEM::alltoall(T, A, block_order*block_order);

// transpose the matrix
if (bulk) {
  transposeBulk<<<dimGrid, dimBlock>>>(np, block_order, T, B);
} else {
  for (int r=0; r<np; r++) {
    const size_t offset = block_order * block_order * r;
    transpose<<<dimGrid, dimBlock>>>(block_order, T + offset, B + offset);
  }
}

// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
```


GPU Communication Performance by Series
NVSHMEM wins by a little for smaller dimensions
NCCL a2a v2 device=false
NCCL a2a bulk device=false
NVSHMEM a2a v2 device=false
NVSHMEM a2a bulk device=false
Rate (GB/s)
3000
4000
5000
6000
7000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
Matrix Dimensions

NVSHMEM one-sided (Get) implementation

```
// transpose the matrix
if (on_device) {
  transpose_nvshmem_get<<<dimGrid, dimBlock>>>(variant, block_order*block_order,
                                               me, np, block_order, A, B, T);
} else {
  for (int r=0; r<np; r++) {
    const int recv_from = (me + r) % np;
    size_t offset = block_order * block_order * me;
    prk::NVSHMEM::get(T, A+offset, block_order * block_order, recv_from);
    offset = block_order * block_order * recv_from;
    transpose<<<dimGrid, dimBlock>>>(block_order, T, B+offset);
  }
}
prk::NVSHMEM::barrier(false);
// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
prk::NVSHMEM::barrier(false);
```

NVSHMEM one-sided (Get) implementation

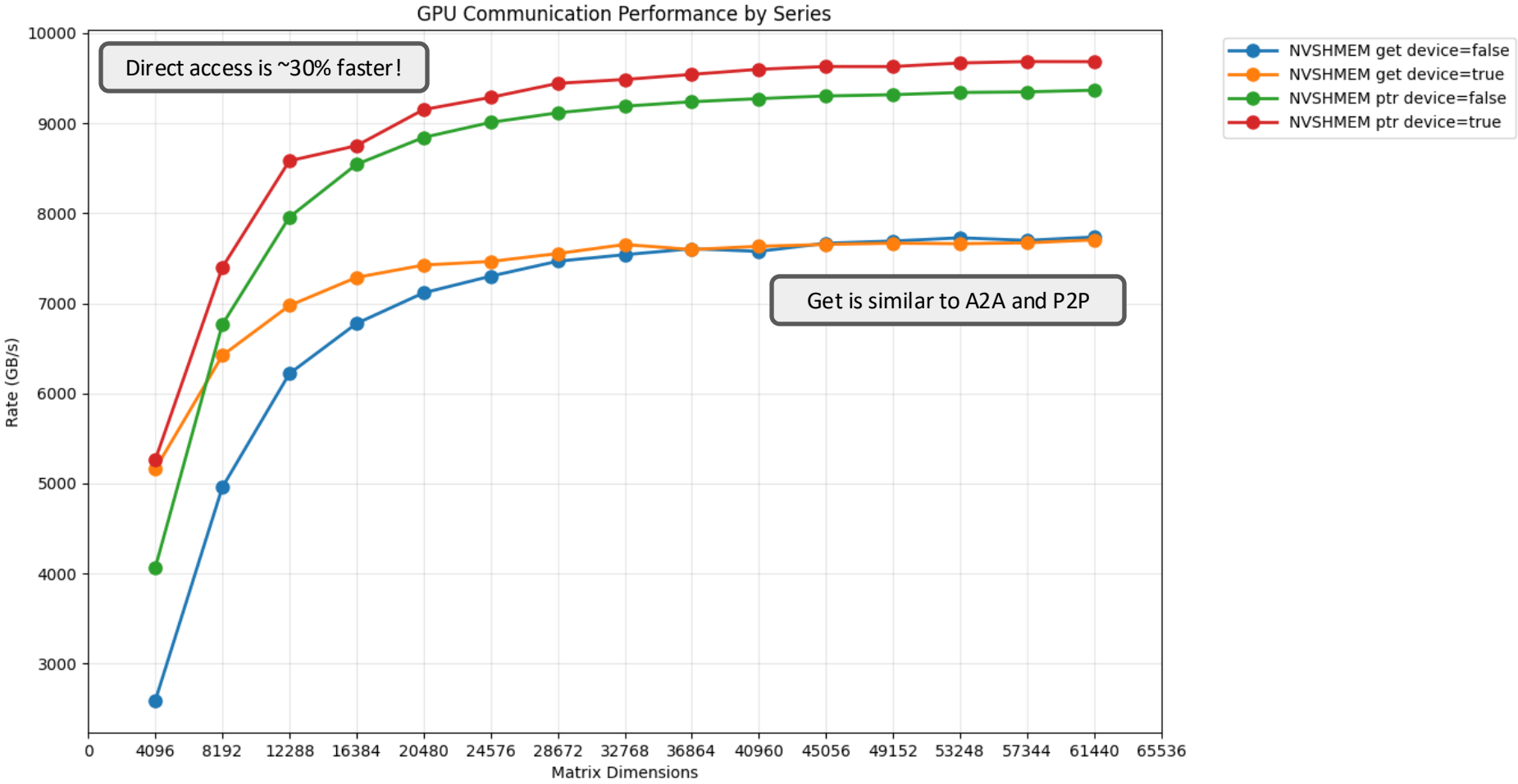
```
__device__ void transposeDevice_get(unsigned order, const double * A, double * B, int recv_from)
{
 __shared__ double tile[tile_dim][tile_dim+1];

 auto x = blockIdx.x * tile_dim + threadIdx.x;
 auto y = blockIdx.y * tile_dim + threadIdx.y;

 for (int j = 0; j < tile_dim; j += block_rows) {
  double T;
  // element-wise get is not the most efficient code
  // it will inline, so on NVL it is a remote load, but it will be very bad over the network
  nvshmem_getmem(&T, &A[(y+j)*order + x], sizeof(double), recv_from);
  tile[threadIdx.y+j][threadIdx.x] = T;
 }

 __syncthreads();

 x = blockIdx.y * tile_dim + threadIdx.x;
 y = blockIdx.x * tile_dim + threadIdx.y;

 for (int j = 0; j < tile_dim; j+= block_rows) {
  B[(y+j)*order + x] += tile[threadIdx.x][threadIdx.y + j];
 }
}
```

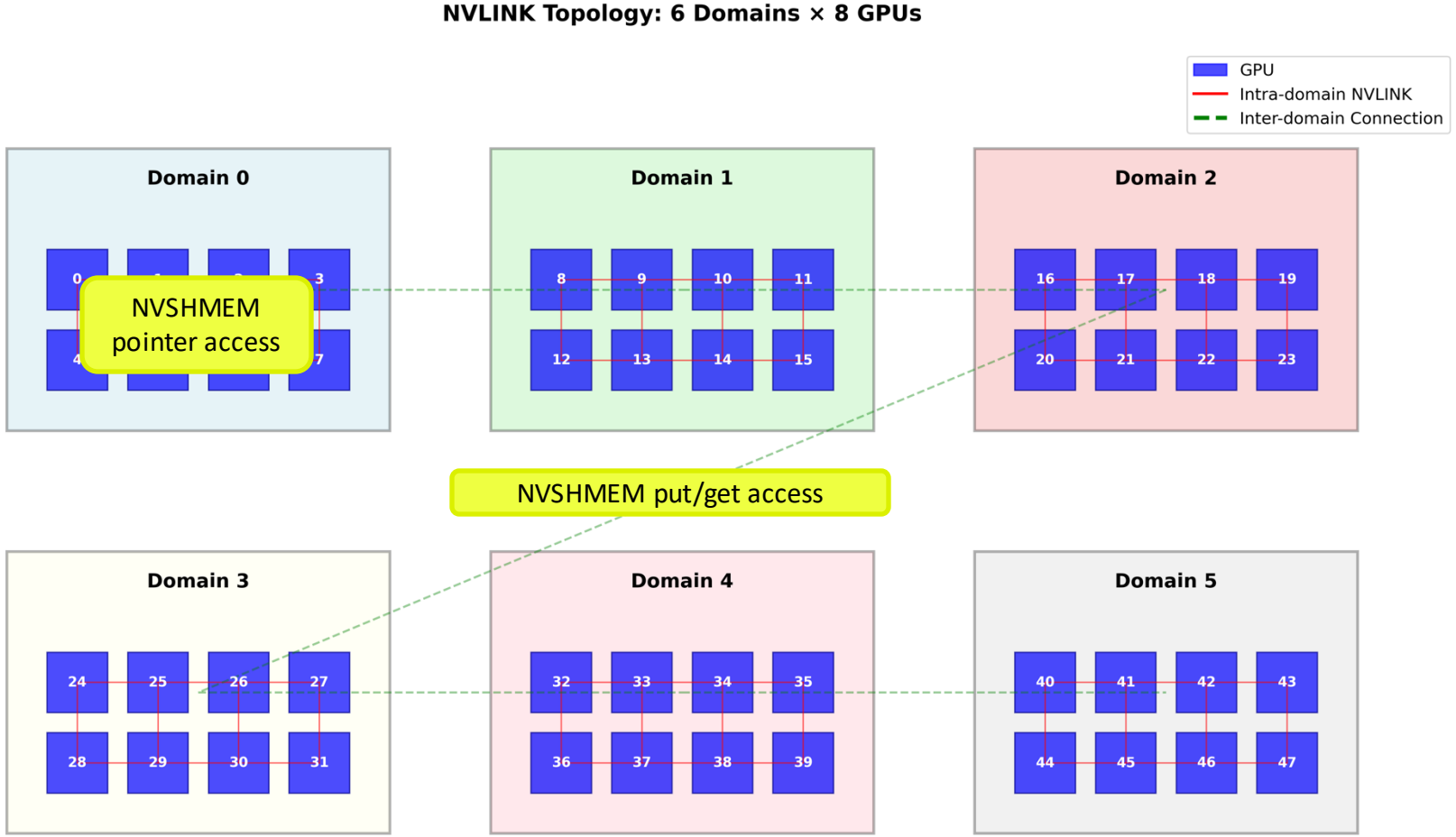
NVSHMEM direct (pointer) implementation

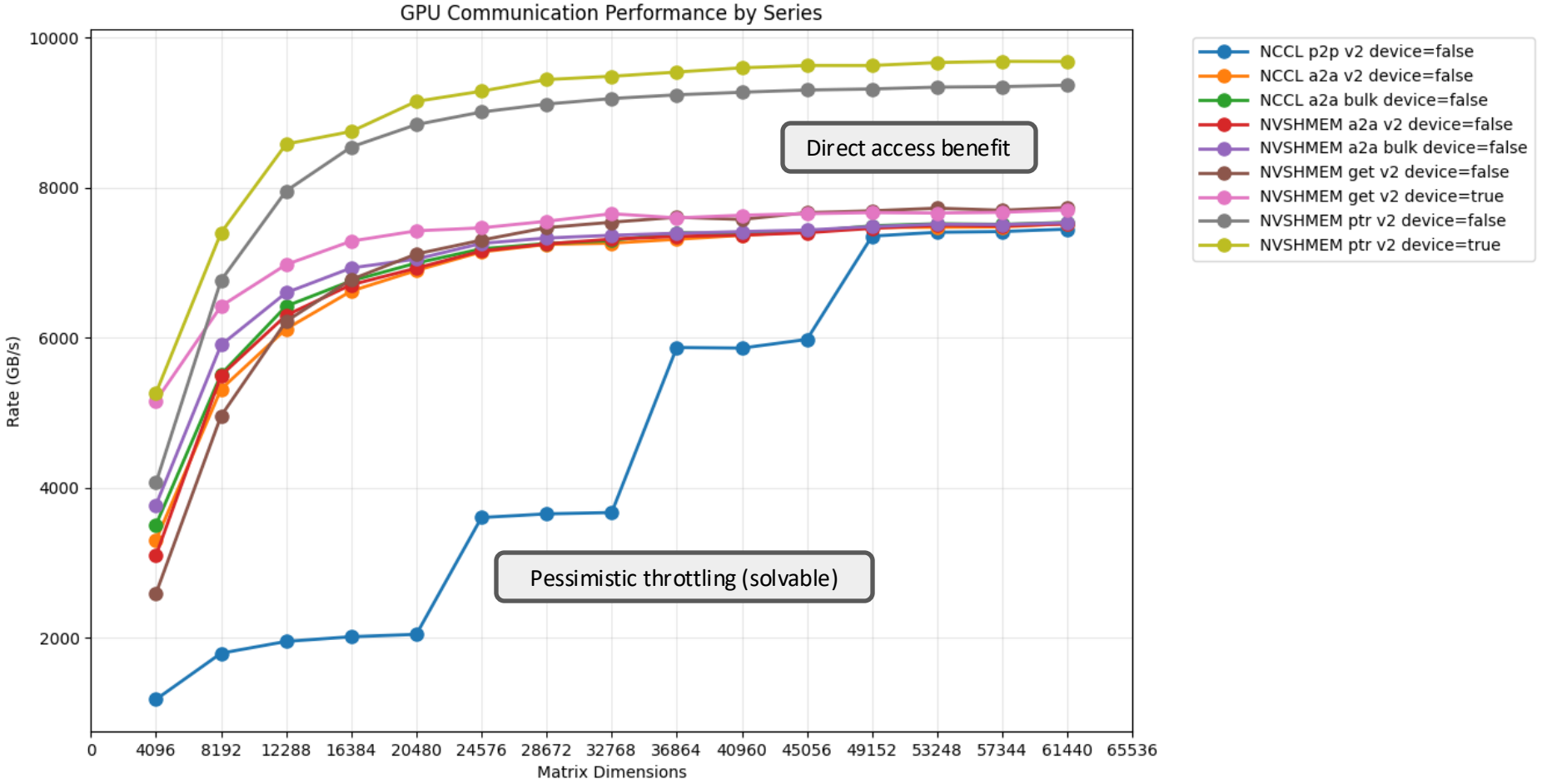
```
// transpose the matrix
if (on_device) {
  transpose_nvshmem_ptr<<<dimGrid, dimBlock>>>(variant, block_order*block_order,
                                               me, np, block_order, A, B);
} else {
  for (int r=0; r<np; r++) {
    const int recv_from = (me + r) % np;
    size_t offset = block_order * block_order * me;
    const double * T = (double*)nvshmem_ptr(A + offset, recv_from);
    transpose<<<dimGrid, dimBlock>>>(block_order, T, B+offset);
  }
}

prk::NVSHMEM::barrier(false);
// increment A
cuda_increment<<<blocks_per_grid, threads_per_block>>>(order * block_order, A);
prk::NVSHMEM::barrier(false);
```

NVSHMEM one-sided (ptr) implementation

```
__global__ void transpose_nvshmem_ptr(int variant, size_t block_size,
                                      int me, int np, unsigned block_order,
                                      const double * A, double * B)
{
  for (int r=0; r<np; r++) {
    const int recv_from = (me + r) % np;
    const size_t soffset = block_size * me;
    const size_t roffset = block_size * recv_from;
    const double * T = (double*)nvshmem_ptr(A + soffset, recv_from);

    // this is just the variant 2 (optimized) kernel
    transposeDevice(block_order, T, B + roffset);
    __syncthreads();
  }
}
```

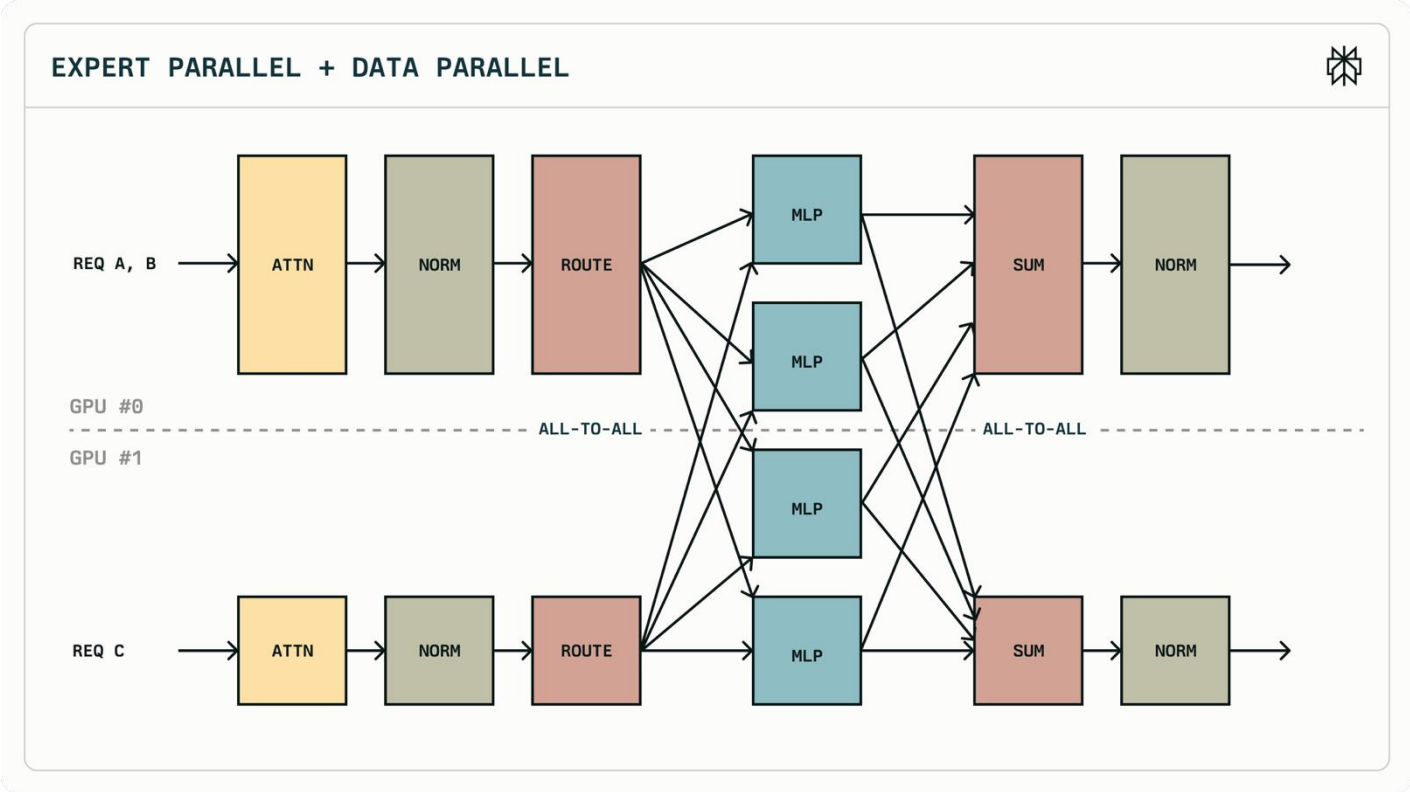

GPU Communication Performance by Series
Direct access is ~30% faster!
Get is similar to A2A and P2P
NVSHMEM get device=false
NVSHMEM get device=true
NVSHMEM ptr device=false
NVSHMEM ptr device=true
Rate (GB/s)
Matrix Dimensions
10000
9000
8000
7000
6000
5000
4000
3000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536


NVLINK Topology: 6 Domains × 8 GPUs
GPU
Intra-domain NVLINK
Inter-domain Connection
Domain 0
Domain 1
Domain 2
Domain 3
Domain 4
Domain 5
0
3
4
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
NVSHMEM pointer access
NVSHMEM put/get access


GPU Communication Performance by Series
Rate (GB/s)
Matrix Dimensions
10000
8000
6000
4000
2000
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
Direct access benefit
Pessimistic throttling (solvable)
NCCL p2p v2 device=false
NCCL a2a v2 device=false
NCCL a2a bulk device=false
NVSHMEM a2a v2 device=false
NVSHMEM a2a bulk device=false
NVSHMEM get v2 device=false
NVSHMEM get v2 device=true
NVSHMEM ptr v2 device=false
NVSHMEM ptr v2 device=true

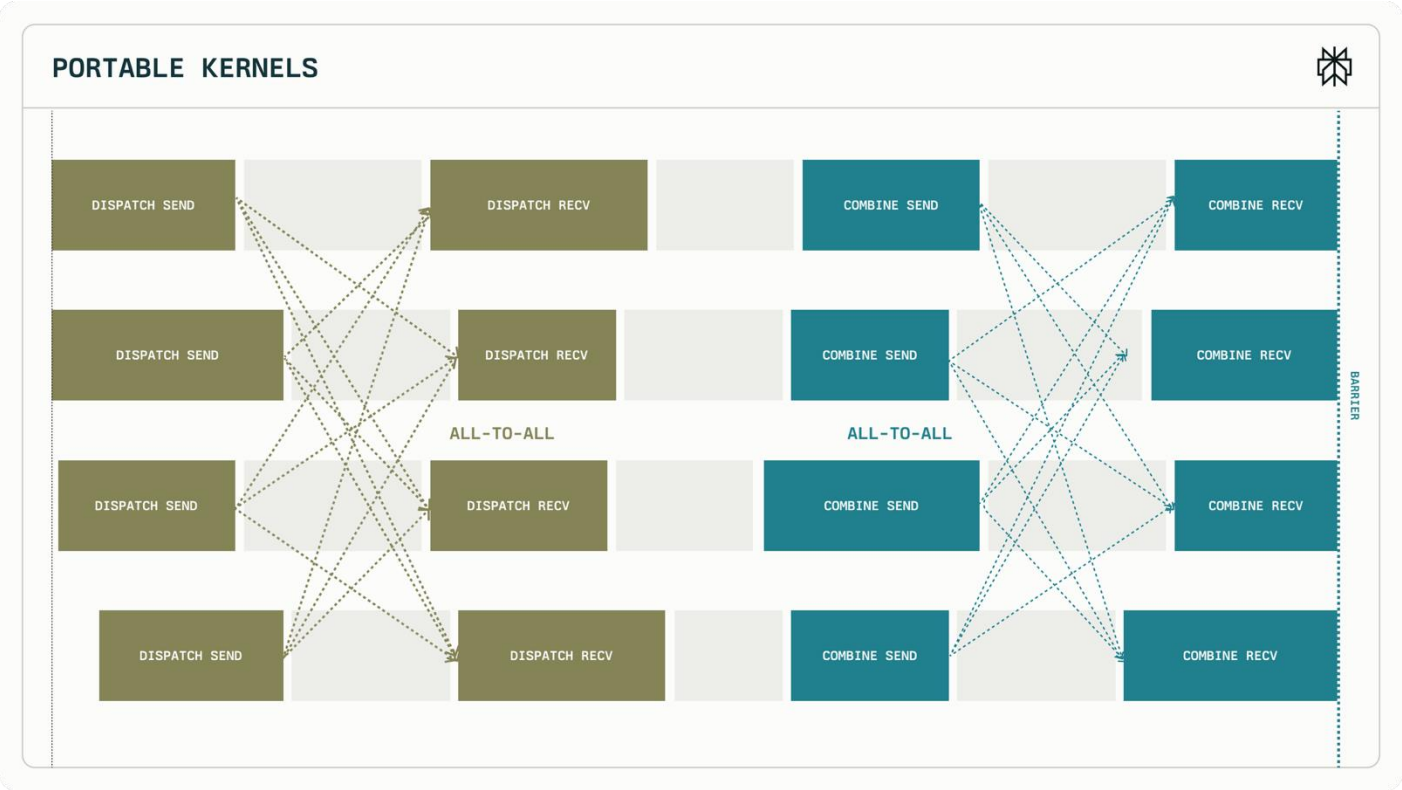
Why it matters

Mixture-of-Experts



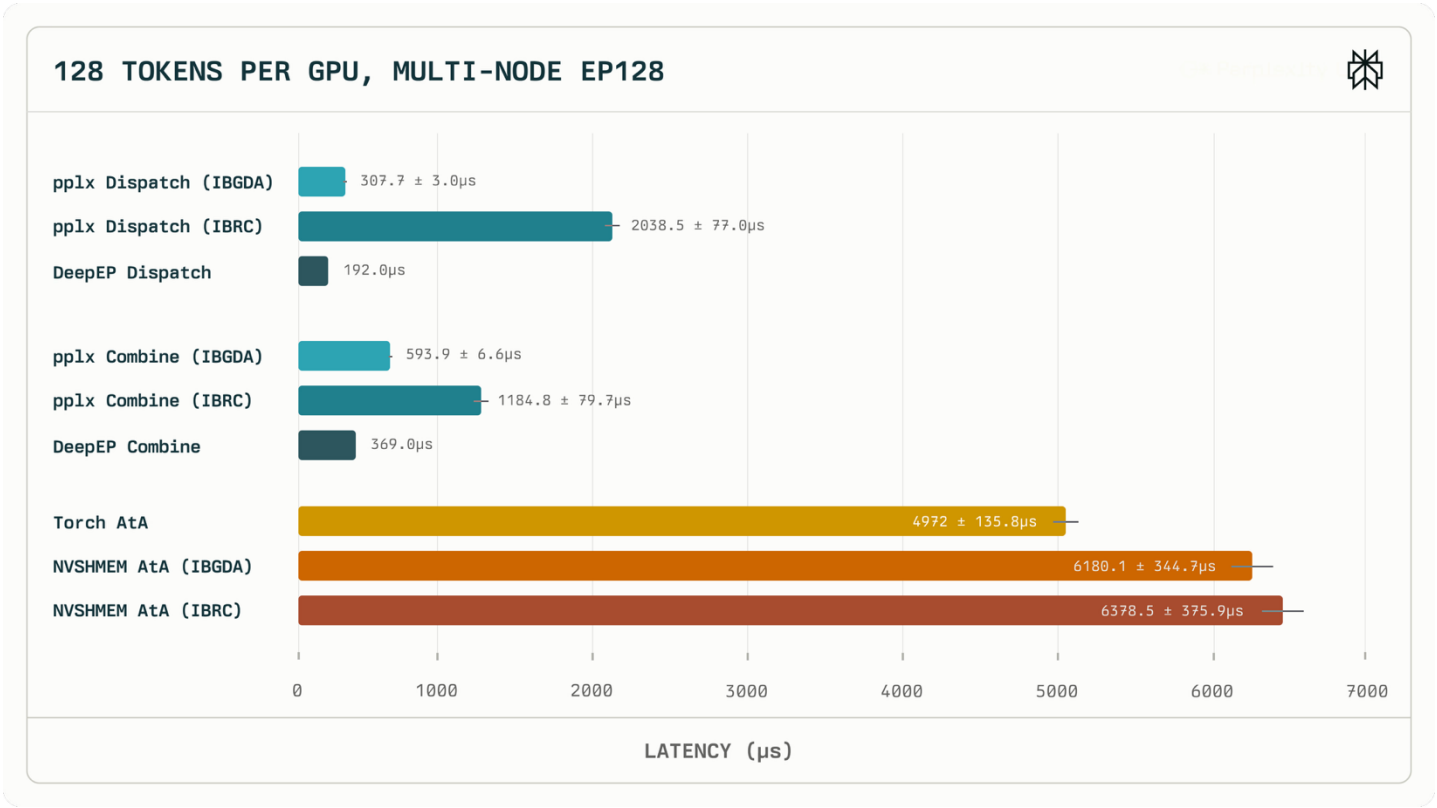
“Efficient and Portable Mixture-of-Experts Communication” by Perplexity.
https://www.perplexity.ai/hub/blog/efficient-and-portable-mixture-of-experts-communication

Mixture-of-Experts



“Efficient and Portable Mixture-of-Experts Communication” by Perplexity.
https://www.perplexity.ai/hub/blog/efficient-and-portable-mixture-of-experts-communication

Mixture-of-Experts



“Efficient and Portable Mixture-of-Experts Communication” by Perplexity.
https://www.perplexity.ai/hub/blog/efficient-and-portable-mixture-of-experts-communication

NCCL Roadmap

Feedback & Prioritization Welcome

NCCL v2.27 May '25	Github Preview – Aug'25 NCCL v2.28 Sept '25	NCCL v2.29 Q4'25
Low latency kernel and algos	**CE Collectives**	MNNVL CE Collectives
Symmetric Memory	**Device API Support**	**Python Host API support (NCCL4Py)**
NCCL Communicator Shrink	MNNVL Symmetric memory support	NCCL Put/Get Host API
NVL SHARP with IB SHARP and UB registration	Extend PAT Support	**NCCL Communicator Grow**
Profiler Enhancements	New APIs for A2A, Gather, Scatter	New API for A2Av
Improved Cost Model & Tuning	Performance tuning improvements	**More latency optimizations**
User-buffer Optimization	NCCL inspector support	MIG support
Direct NIC GB300 / CX-8 Enablement	CMake support	
DGX Spark Enablement	Multiple ranks per GPU	
Cross-DC Communication Support		

Subject to Change

Prior Release Notes Available on ***docs.nvidia.com***

NCCL Roadmap

Feedback & Prioritization Welcome

NCCL v2.27 May '25	Github Preview – Aug'25 NCCL v2.28 Sept '25	NCCL v2.29 Q4'25
Low latency kernel and algos	**CE Collectives**	MNNVL CE Collectives
Symmetric Memory	**Device API Support**	**Python Host API support (NCCL4Py)**
NCCL Communicator Shrink	MNNVL Symmetric memory support	NCCL Put/Get Host API
NVL SHARP with IB SHARP and UB registration	Extend PAT Support	**NCCL Communicator Grow**
Profiler Enhancements	New APIs for A2A, Gather, Scatter	New API for A2Av
Improved Cost Model & Tuning	Performance tuning improvements	**More latency optimizations**
User-buffer Optimization	NCCL inspector support	MIG support
Direct NIC GB300 / CX-8 Enablement	CMake support	
DGX Spark Enablement	Multiple ranks per GPU	
Cross-DC Communication Support		

Subject to Change

*Prior Release Notes Available on **docs.nvidia.com***

Take-away messages

NVSHMEM

- Easy: NVLINK mapping allows remote memory to look like local memory.
- Fast: NVLINK direct access is the fastest way to move data within a domain.
- Portable: Put/Get works over NVLINK, IB, EFA, etc.

NCCL

- All the collectives, at scale, with extensive support for AI workflows.
- Trust NCCL. Don’t try to be too smart.
- NCCL symmetric memory is here; the device API is coming.

The end

Extras

Why MPI on GPUs is Hard

- Is this a CUDA GPU or an OpenCL GPU?
  - Forward-progress guarantees are important.
  - Blocking, synchronization and ordering are all performance hazards on GPUs. RMA is a good model for GPUs…
  - MPI support for NUMA doesn't easily extend to GPUs.
- NCCL is MPI for GPUs
  - Stream semantics in everything.
  - Only implements patterns that make sense.
  - Supports GPU endpoints…
- Hopefully, MPI-6 will catch up to NCCL…

https://github.com/NVIDIA/mpi-acx