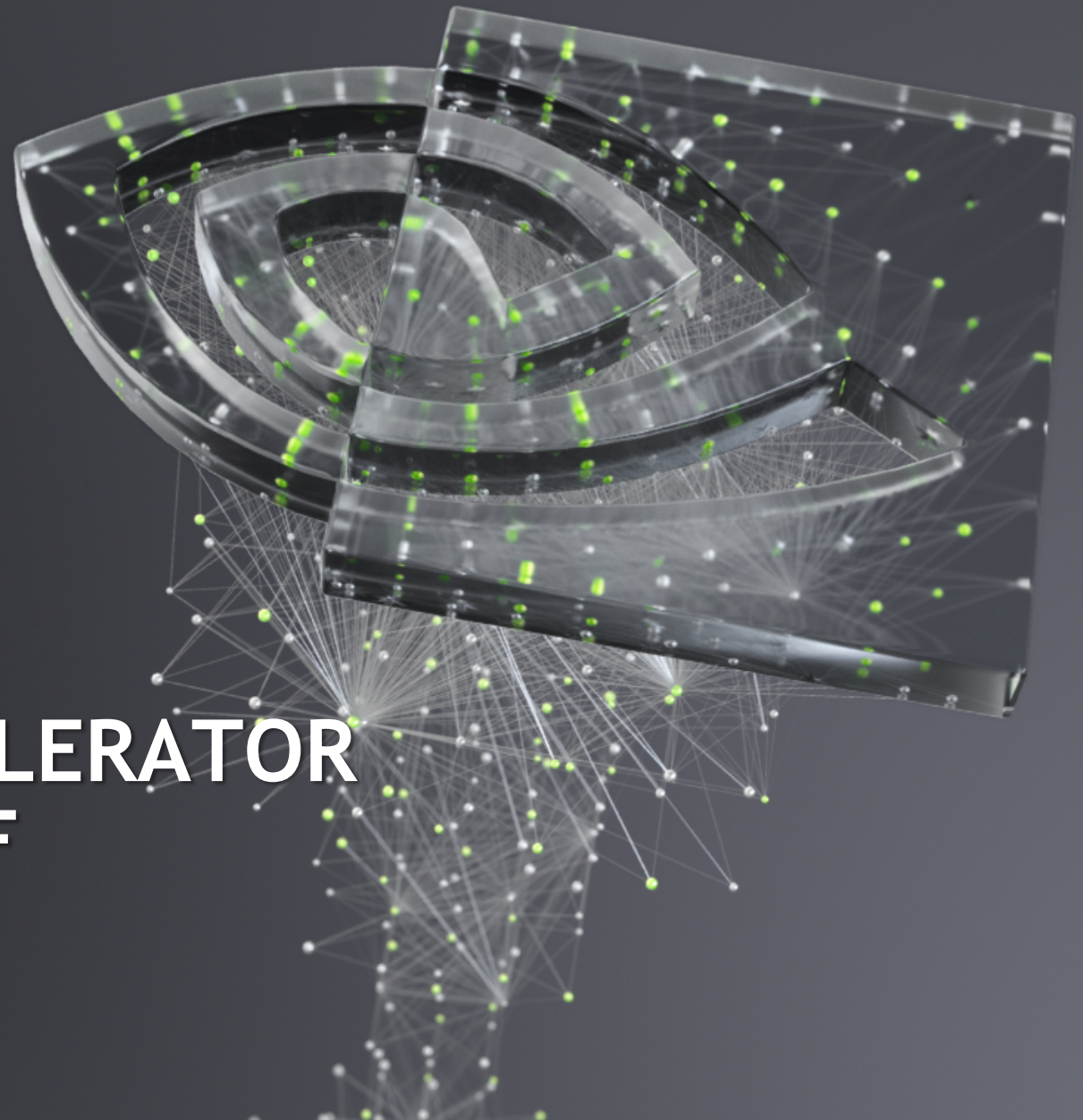


MPI HYBRID & ACCELERATOR (HACC) WG KICKOFF

James Dinan
December 2, 2020

HYBRID/ACCELERATOR WORKING GROUP

Goals

- Investigate support for MPI communication involving accelerators
- Hybrid programming of MPI + [CUDA, HIP, DPC++, OpenACC, ...]
- Host-initiated communication with accelerator memory
- Host-setup with accelerator triggering
- Host-setup, enqueued on a stream or queue
- Accelerator-initiated communication
- Investigate improved compatibility/efficiency for multithreaded MPI communication
- MPI + [Pthreads, OpenMP, C/C++ threading, TBB, ...]

Active Proposals

- Accelerator pready/parrived: https://github.com/mpi-forum/mpi-standard/pull/264

Chairs

- James Dinan, NVIDIA
- Pavan Balaji, Argonne National Laboratory

Mailing List

- mpiwg-hybridpm@lists.mpi-forum.org
- https://github.com/mpiwg-hybrid/hybrid-issues/wiki

MEETING ORGANIZATION

Shared meeting time with Persistence WG, Wed 10-11am ET

Dec. 2 – HACC WG

Dec. 9 – MPI Forum

Dec. 16 – HACC WG

Dec. 23 – Holiday Break

https://github.com/mpiwg-hybrid/hybrid-issues/wiki

SPEAKERS

1. Jim Dinan – Kickoff
2. Olga Pearce – GPU communication bottlenecks @LLNL (e.g. COMB)
3. Ryan Grant – Challenges for applications
   - E.g. Determining how MPI library will handle GPU communications
4. Joseph Schuchart and Joachim Protze – Callback-based Completion Notification
   - *Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations*. EuroMPI '20.
   - *MPI Detach - Asynchronous Local Completion*. EuroMPI '20.
5. Jim Dinan – NCCL and NVSHMEM: GPU Comms Best Practices
6. Pavan Balaji – Proposal to pass stream to MPI

OUTLINE FOR TODAY

1. Current state of GPU communication
2. Hints to improve today's GPU support
3. Analysis of persistent GPU communication
   - Stream/graph communication
   - Partitioned communication
4. Brainstorming

IMPORTANCE OF COMMUNICATION EFFICIENCY

13 Years Of The CUDA Architecture

	G80 (2007)	A100 (2020)	A100/G80 Ratio
Compute Throughput	0.345 FP32 TFLOPS	19.5 FP32 TFLOPS	56x
Memory Bandwidth	86.4 GB/s	1,555 GB/s	18x
Memory Capacity	1.5 GB	40 GB	27x
PCIe Bandwidth	8 GB/s PCIe 2.0 x16, unidir	32 GB/s PCIe 4.0 x16, unidir	4x

Credit: Wen-mei Hwu

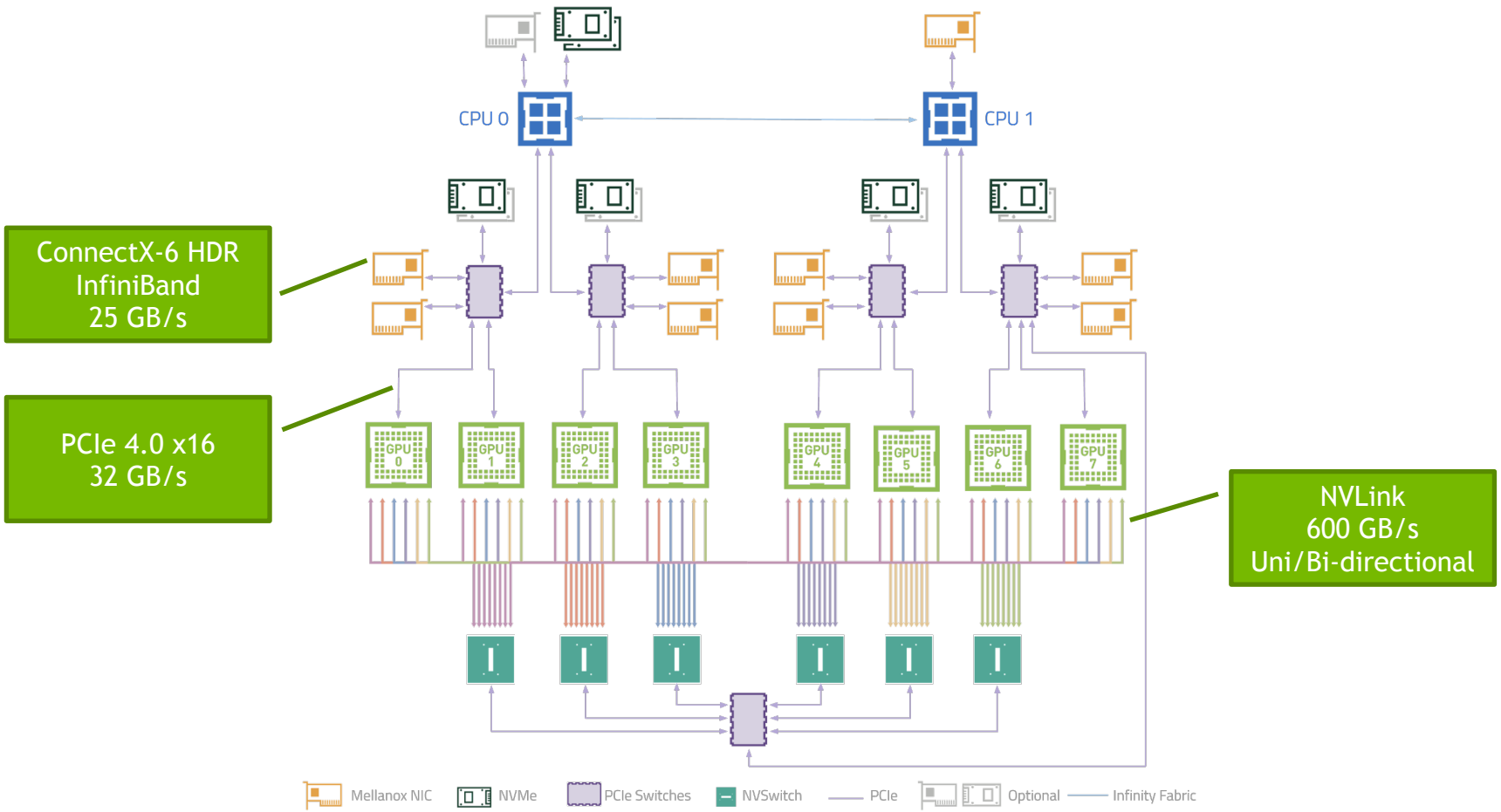
For every byte read through PCIe, A100 must perform 610 FP32 ops (13.9x increase)

For TF32 math, A100 must perform 4,875 FP32 ops (100x increase)

Data transfer efficiency is extremely important

NVIDIA DGX A100 SYSTEM TOPOLOGY

Record Breaking A100 GPUs, Fastest I/O Architecture, Focus on Efficient Data Movement



NVIDIA

A TAXONOMY OF GPU COMMUNICATON

CPU-Initiated

- Application calls MPI routines on the CPU

GPU-Initiated

- Application calls MPI routines on the GPU

Stream/Graph-Based

- MPI operations enqueued on CUDA streams/graphs
- Stream ordering captures data dependencies

GPU-Triggered

- MPI calls on CPU perform initiation/completion
- MPI calls on GPU trigger data movement

CURRENT STATE OF GPU-AWARE MPI

Highly Optimized CPU-Initiated Communication

MPI libraries support GPU memory

- Memory type detection

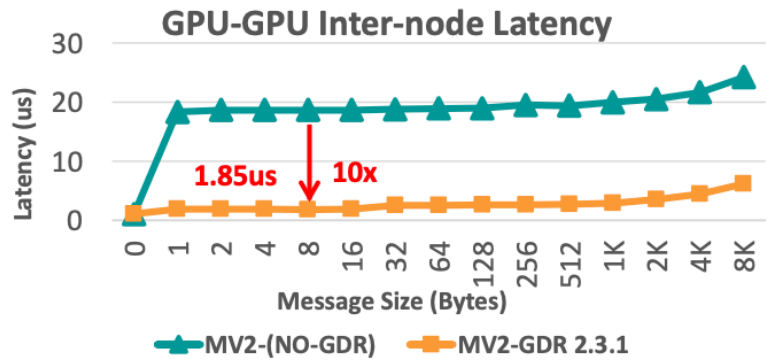
Efficient intranode protocols

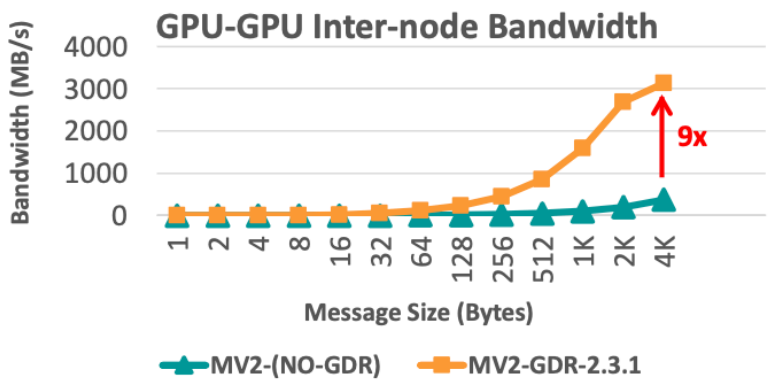
- Peer-to-peer data transfers via PCIe and NVLink using CUDA IPC

Efficient Internode protocols

- Pipelined copy-based transfers
- GPUDirect RDMA

Support for MPI datatype packing kernels





"MVAPICH2-GDR: High-Performance and Scalable CUDA-Aware MPI Library for HPC and AI." DK Panda and Hari Subramoni, GTC 2019

CPU-INITIATED API CHALLENGES

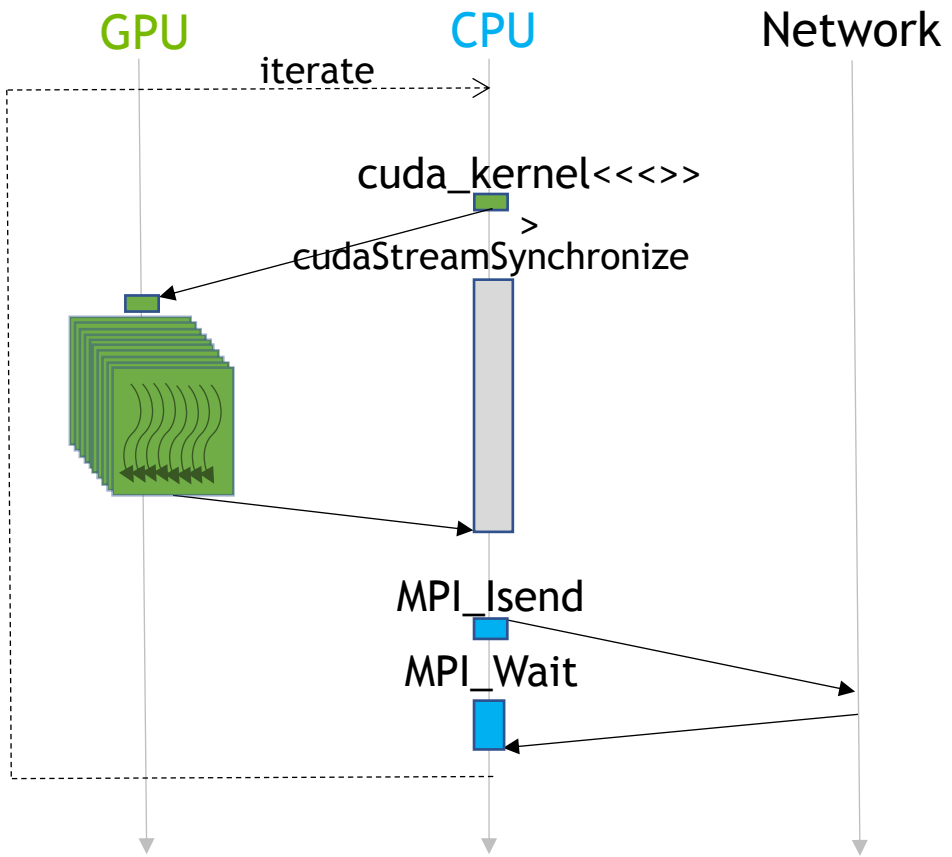
GPU/CPU Control Transfer Exposes Offload Overheads

Computation and communication don't naturally overlap

Prevents application from using streams/graphs

User can't tell MPI how to use GPU

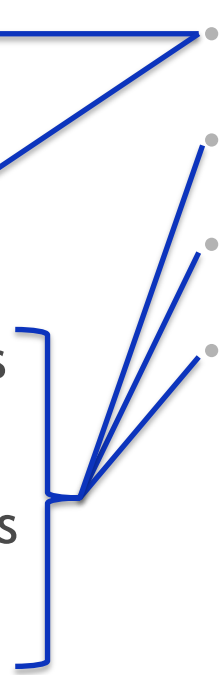
- Which memory is a buffer in?
- Are resources available to launch kernels?
- Which GPUs does a process use?



HOW CAN WE IMPROVE TODAY'S GPU-AWARE MPI?

Reduce Overheads Through Info Assertions

- MPI Process to GPU mapping (assert 1:1 fixed mapping)
- Query GPU support (set query key, check if it was accepted)
- Guide the use of packing kernels (assert GPU cores available)
- Memory type checking overheads (assert buffer kind)



- Session
- Communicator, window, file
- Persistent operation
- Datatype
  - `MPI_Type_commit_with_info( MPI_Datatype type, MPI_Info info)`

PERSISTENCE: STREAM AND DEVICE TRIGGERED

ANALYSIS OF GPU COMMUNICATION

Control and Data Planes

Control plane:

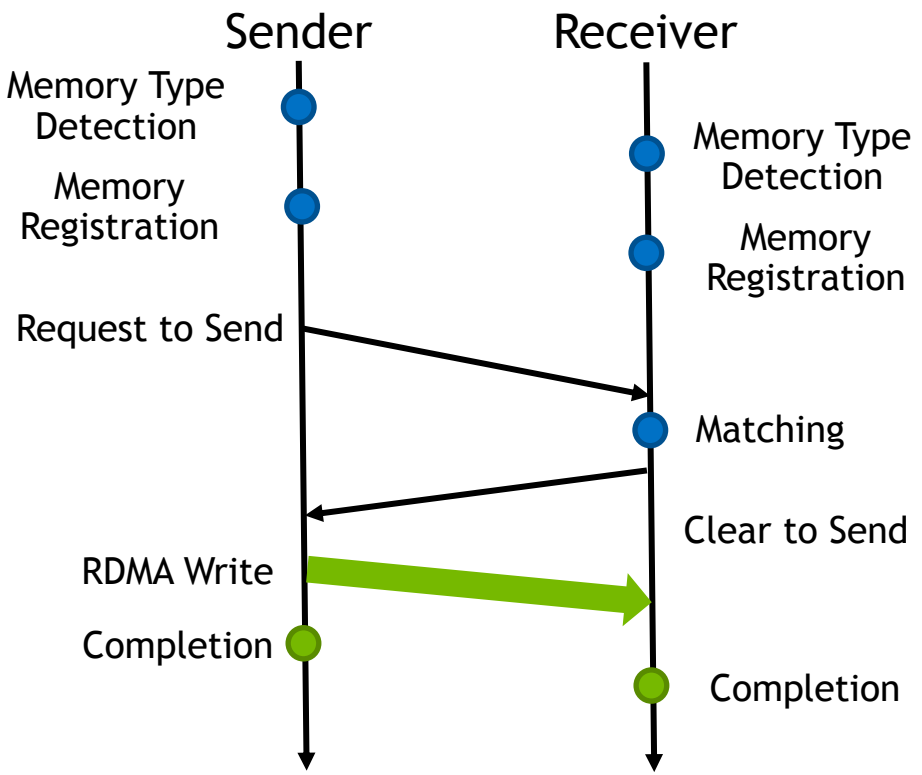
- Memory registration, protocol selection (RTS/CTS), matching, pipeline management, data packing, event handling

Data plane:

- Data transfer
- Completion notification

Most MPI operations contain both control and data plane actions

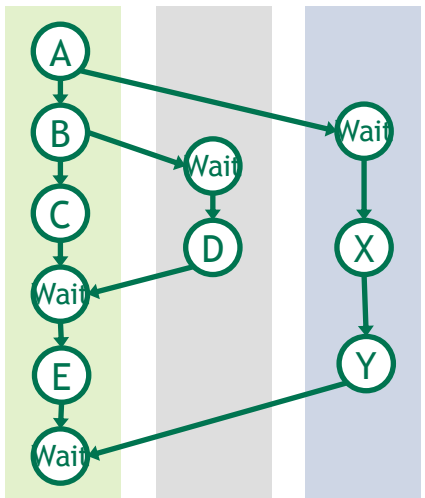
- Control plane tasks are challenging on GPU



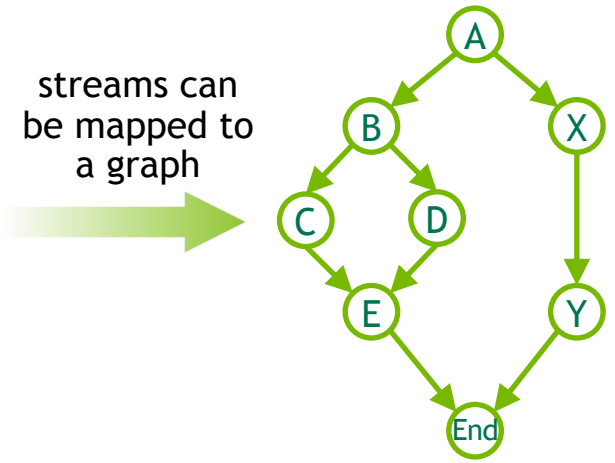
CUDA STREAMS/GRAPHS

Optimize Workflows and Reduce Launch Overheads

CUDA Work in Streams



Graph of Dependencies



CUDA Streams are sequences of asynchronous operations that execute in issue-order on the GPU

CUDA Graphs are captured from streams and can be replayed multiple times

- Operations in different streams may run concurrently
- Enables data copy pipelining, hiding launch overheads
- Eliminates potential for deadlock across streams by construction

STREAM AND GRAPH COMMUNICATION

Approaches to Extending MPI

Introduce Stream-Based APIs

- APIs ala NCCL: Good GPU integration, but significant API churn

Inject Stream reference using Info

- Set stream on a communicator using MPI_Info object
- Pro: Minor changes to extend MPI_Info to support binary data
- Con: Info is just a hint. Using multiple streams requires multiple communicators

Persistent communication on Streams

- Introduce start/wait operations on streams
- Pro: Flexibility, can use multiple streams and CUDA Graphs
- Pro: Persistence provides separation of control and data planes
- Con: Only applies to persistent operations (or is this a pro?)

PERSISTENT COMMUNICATION ON STREAMS

Generic API Compatible with Multiple Accelerator Models

```
int MPI_Start_enqueue(MPI_Request *req, int kind, ...);
int MPI_Wait_enqueue(MPI_Request *req, MPI_Status *status, int kind, ...);

int MPI_Startall_enqueue(MPI_Count count, MPI_Request requests[], int kind, ...);
int MPI_Waitall_enqueue(MPI_Count count, MPI_Request requests[], MPI_Status *status, int kind, ...);
```

1. Persistent APIs set up future communication operations
   - Allows MPI library to manage control and data planes separately

2. Communication request is started and waited on from stream
   - Captures data dependence through stream ordering
   - Can be restarted to repeat the same operation again enabling CUDA graph replay

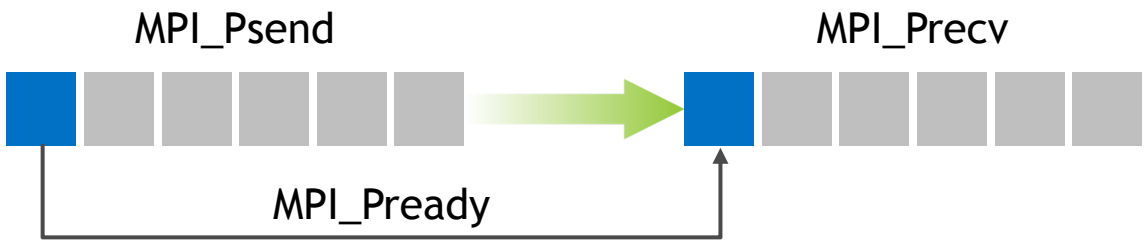
PERSISTENT STREAM/GRAPH COMMUNICATION

Split operation into control (init) and data (start/wait)

```
MPI_Send_init(send_buf, ..., &send_req);
MPI_Recv_init(recv_buf, ..., &recv_req);

for (i = 0; i < NITER; i++) {
  if (i > 0) {
    MPI_Wait_enqueue(recv_req, MPI_CUDA_STREAM, stream);
    MPI_Wait_enqueue(send_req, MPI_CUDA_STREAM, stream);
  }

  kernel<<<..., stream>>>(send_buf, recv_buf, …);

  if (i < NITER – 1) {
    MPI_Start_enqueue(recv_req, MPI_CUDA_STREAM, stream);
    MPI_Start_enqueue(send_req, MPI_CUDA_STREAM, stream);
  }
}
cudaStreamSynchronize(stream);
```

GPU TRIGGERED OPERATIONS

MPI 4.0 Persistent Partitioned Communication to the Rescue



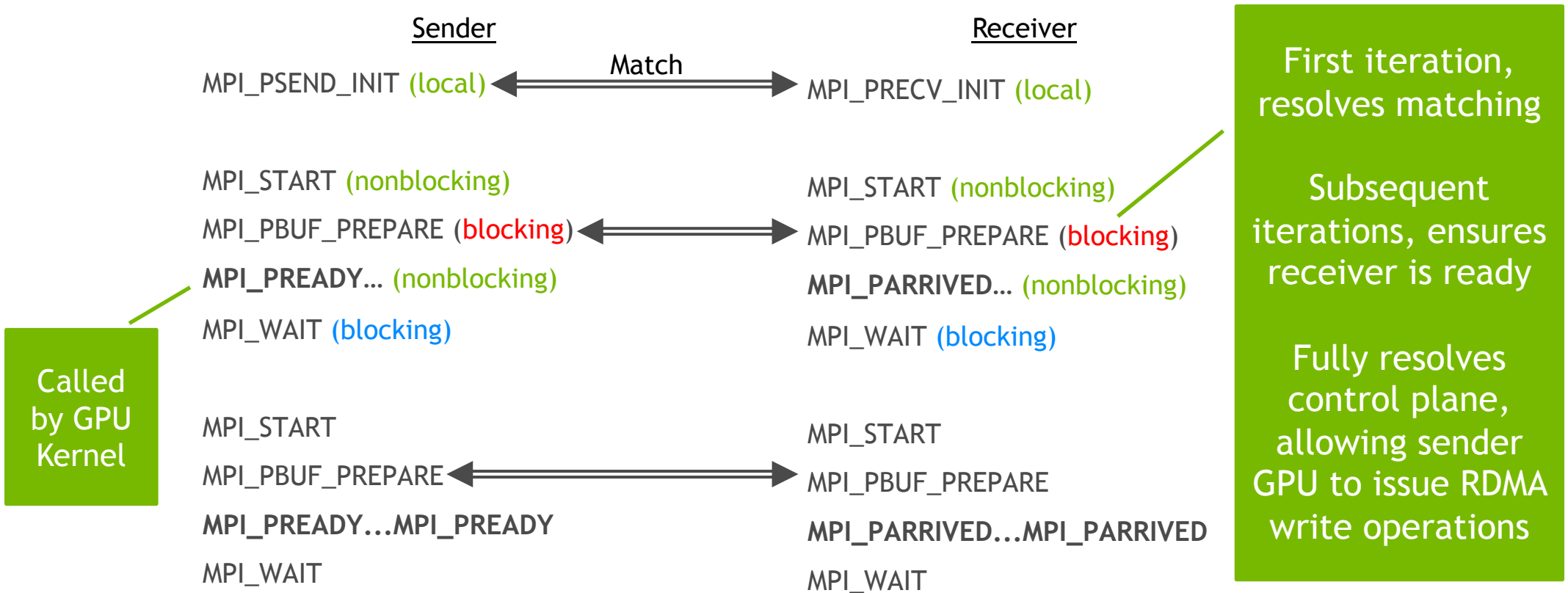
Send/Recv data buffers are broken into equal-sized partitions

MPI_Pready: mark partition as ready to send

MPI_Parrived: query if partition has arrived

SENDER/RECEIVER INTERACTION

Buffer Preparation Step

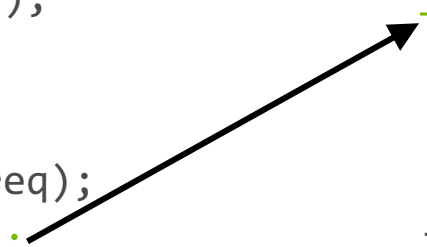


KERNEL TRIGGERED COMMUNICATION USAGE

Partitioned Neighbor Exchange

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall(2, req);
  MPI_Pbuf_prepare_all(2, req);
  kernel<<<...>>>(..., req);
  MPI_Waitall(2, req);
}
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```



Device Code

```
__device__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  // Compute and fill partition i
  // then mark i as ready
  MPI_Pready(i, req[0]);
}
```

KERNEL TRIGGERED COMMUNICATION USAGE

Paired with Stream-Based Wait/Start

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall_enqueue(2, req, …);
  MPI_Pbuf_prepare_all_enqueue(2, req, …);
  kernel<<<...>>>(..., req);
  MPI_Waitall_enqueue(2, req, …);
}
cudaStreamSynchronize(stream);
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Device Code

```
__device__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  // Compute and fill partition i
  // then mark i as ready
  MPI_Pready(i, req[0]);
}
```

WHAT ABOUT MPI_PARRIVED?

MPI_Parrived carries a synchronization between sender and receiver

- Requires sending thread to make progress
- But, there's a catch (feature) …

SYNCHRONIZING SAFELY IN RUNNING KERNELS

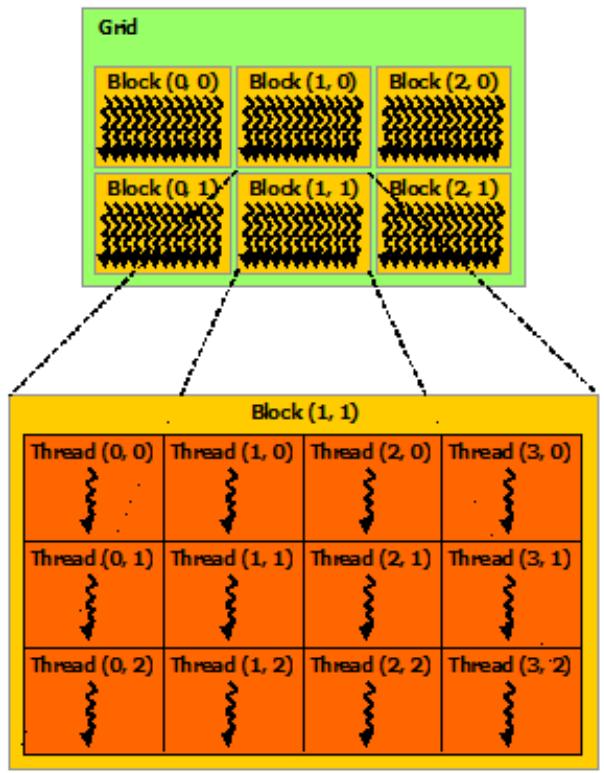
Ensuring Forward Progress

K<<<**blocks**, threads, smem, stream>>>(…)

- Grid of **blocks** expresses work to be done
- Blocks are not guaranteed to be coresident
- Sending from one block and receiving in another can lead to deadlock

Can use CUDA cooperative launch to ensure coresidency, or relax synchronization

- NVSHMEM provides a collective launch API to make this easy



KERNEL TRIGGERED COMMUNICATION USAGE

Partitioned Neighbor Exchange With Receive

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall_enqueue(2, req, …);
  MPI_Pbuf_prepare_all_enqueue(2, req, …);
  kernel<<<...>>>(..., req);
  MPI_Waitall_enqueue(2, req, …);
  complete_remaining<<<...>>>(...);
}
cudaStreamSynchronize(stream);
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Device Code

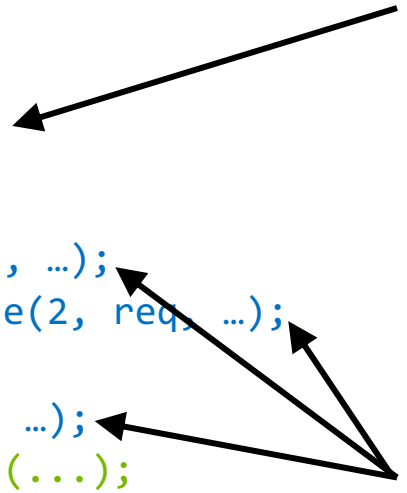
```
__device__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  MPI_Pready(i, req[0]);
  if (MPI_Parrived(i, req[1])) {
    process(i);
    mark_done(i);
  }
}

__device__ complete_remaining(...) {
  int i = my_partition(...);
  if (!done(i)) process(i);
}
```

KERNEL TRIGGERED COMMUNICATION IMPLEMENTATION

Host Side

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall_enqueue(2, req, …);
  MPI_Pbuf_prepare_all_enqueue(2, req, …);
  kernel<<<...>>>(..., req);
  MPI_Waitall_enqueue(2, req, …);
  complete_remaining<<<...>>>(...);
}
cudaStreamSynchronize(stream);
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```



Need to set up request object on the device

- E.g. Request could be an integer that is valid for lookup on both host/device
- Helpful to know where pready/parrived will be called at init time?
- E.g. could add an info key

CPU pushes updates to GPU request object

KERNEL TRIGGERED COMMUNICATION IMPLEMENTATION

Device Side

All threads in the block call

- This is invalid for `MPI_Pready`
- Application can use `threadIdx` to choose one thread to call `MPI_Pready`
- If we want Pready to perform P2P copy, block/warp call is more efficient

Inefficient for one thread per block to call MPI_Parrived

- Would require block-level bcast
- This is allowed

Device Code

```
__device__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  MPI_Pready(i, req[0]);
  if (MPI_Parrived(i, req[1])) {
    process(i);
    mark_done(i);
  }
}

__device__ complete_remaining(...) {
  int i = my_partition(...);
  if (!done(i)) process(i);
}
```

HYBRID & ACCELERATOR WG

Discussion

Communication efficiency is critical, important work in front of HACC WG

- Focus on improving efficiency through improved GPU awareness
- WG open to all Hybrid issues for MPI 4.1

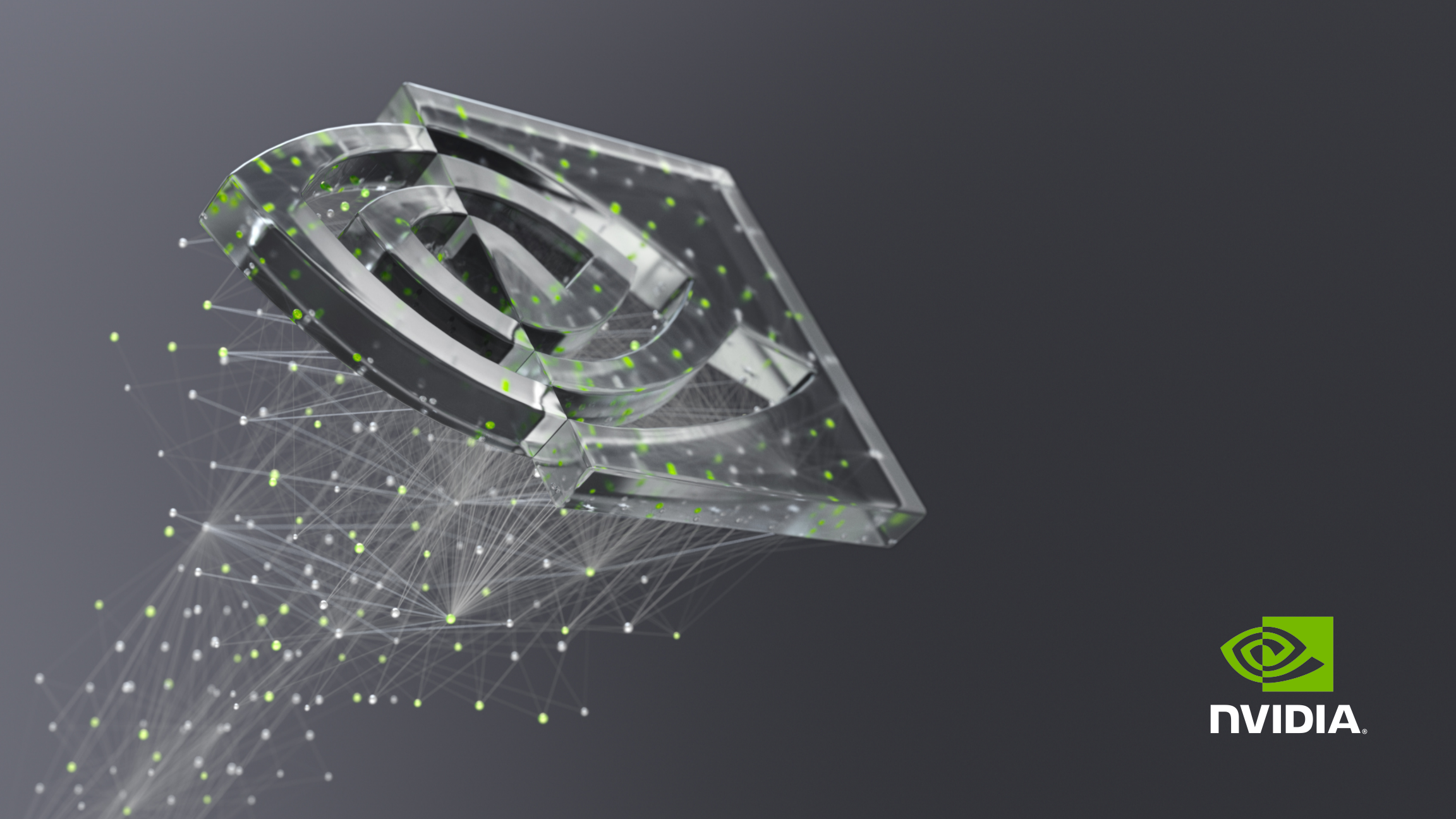
Partitioned communication API incorporated in MPI 4.0

- Further work to integrate with accelerators in MPI 4.1

Steam/Graph/Queue integration is best practice in GPU communication libraries

- E.g. NCCL, NVSHMEM, and others
- Can hide offloading latencies, pipeline data movement, and improve overlap
- Proposed On-Queue API to extend to MPI

Look forward to many interesting discussions!


NVIDIA