



ADVANCED ON-NODE GPU COMMUNICATION

Steve Abbott, OLCF User Conference Call, February 2018



AGENDA

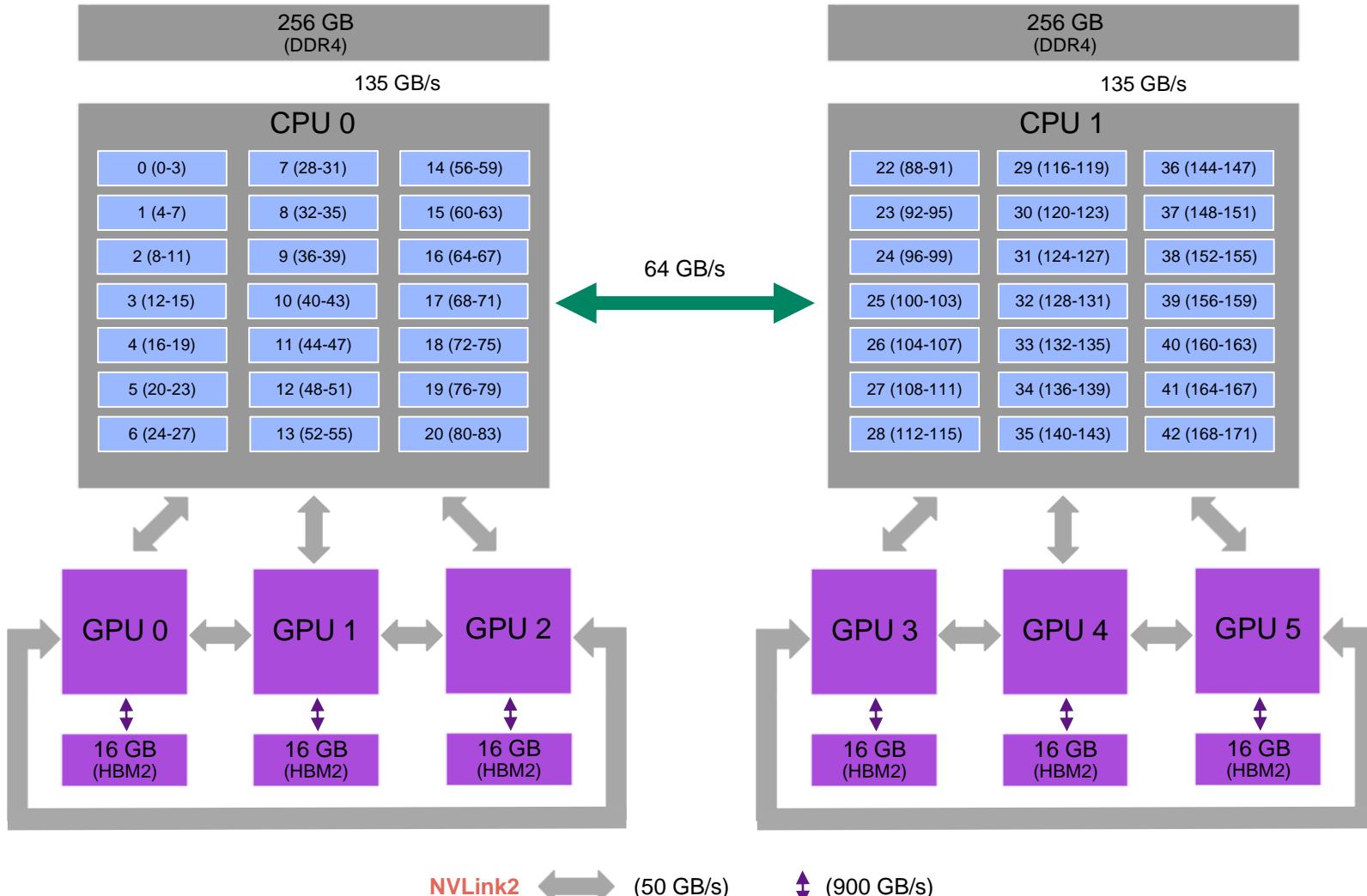
What is GPU Direct?
Intra-process Peer-to-peer
Inter-process Communication with CUDA IPC



SUMMIT NODE OVERVIEW

SUMMIT NODE

(2) IBM POWER9 + (6) NVIDIA VOLTA V100

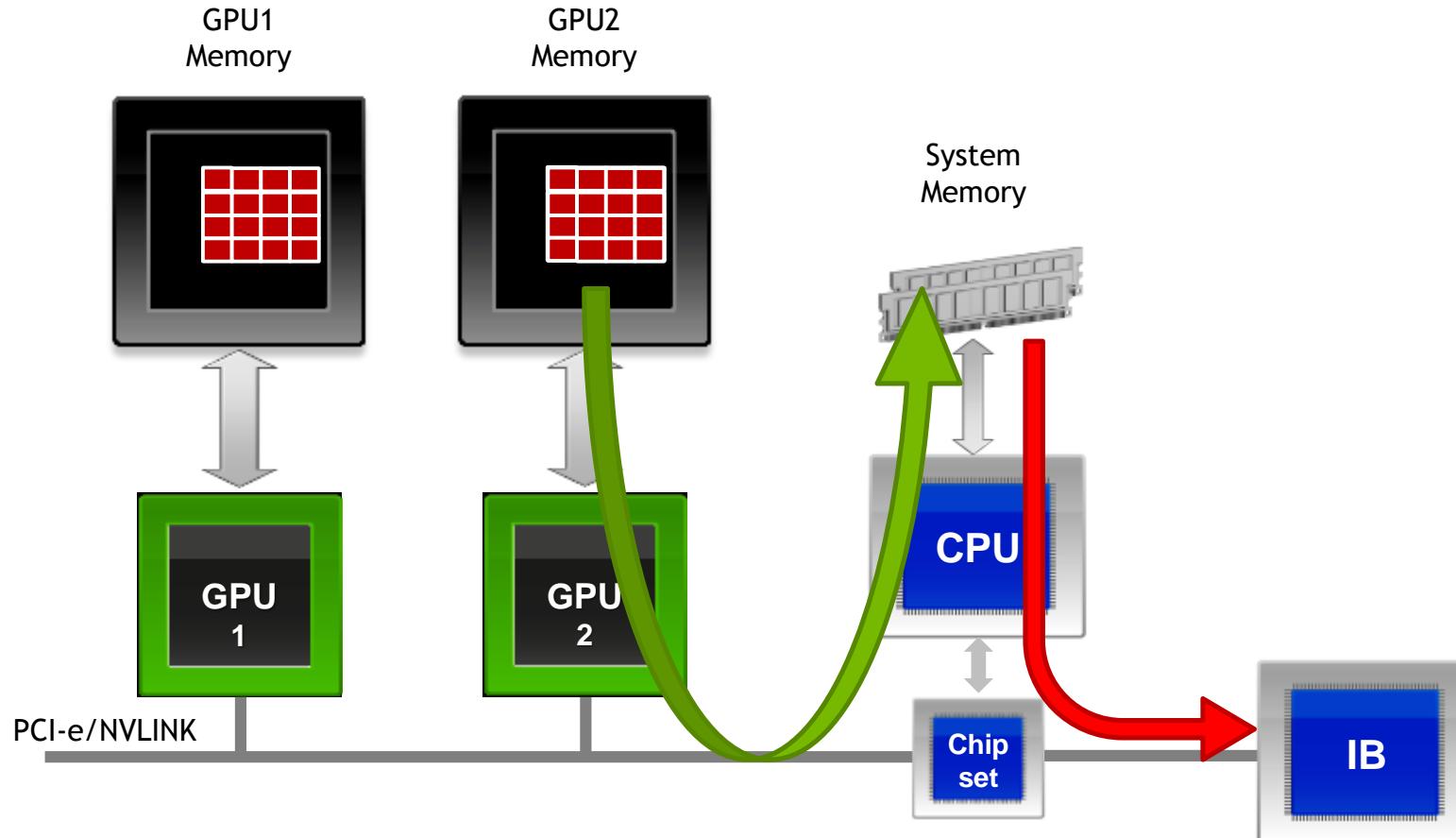




GPUDIRECT

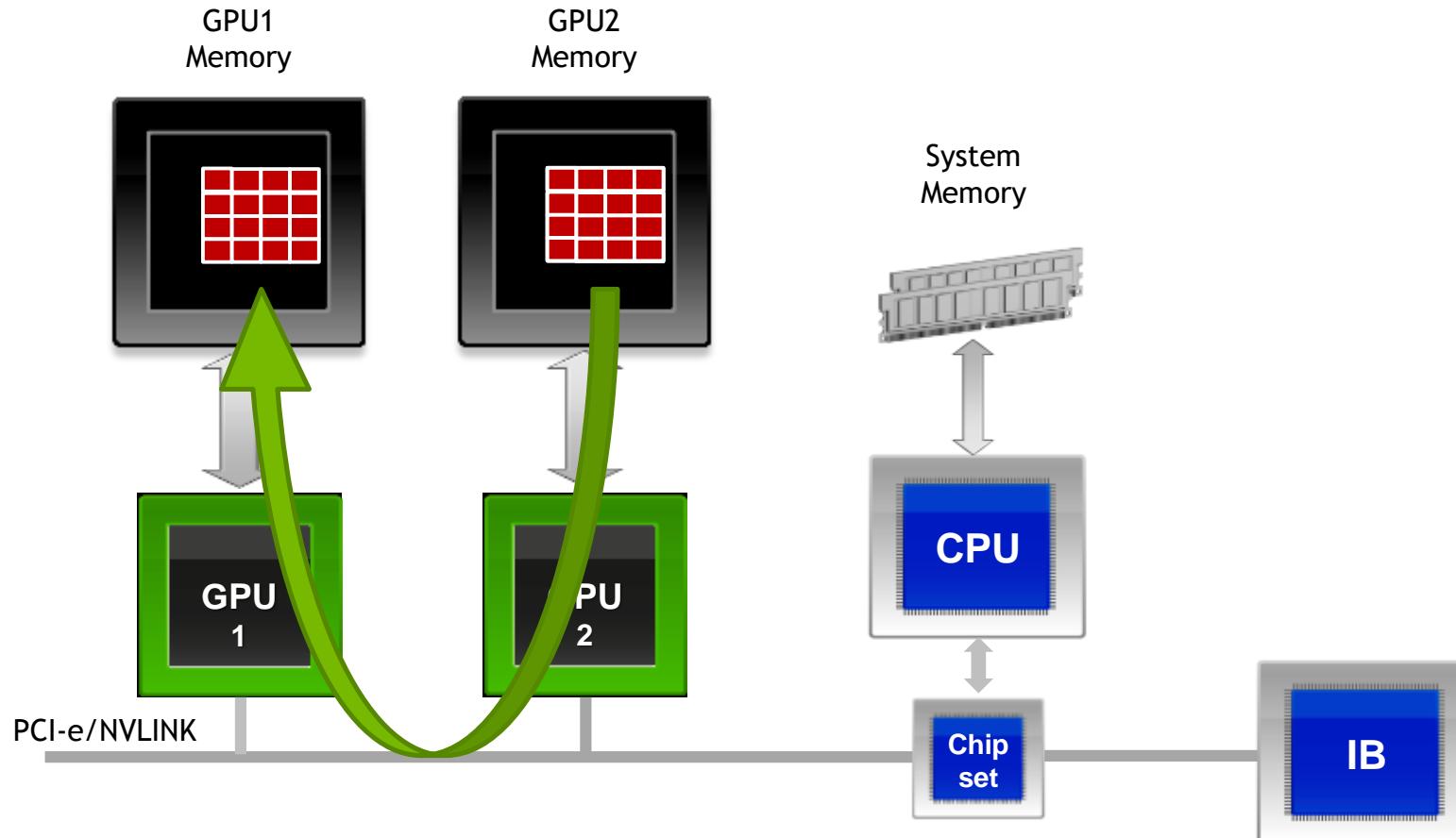
NVIDIA GPUDIRECT™

Accelerated Communication with Network & Storage Devices



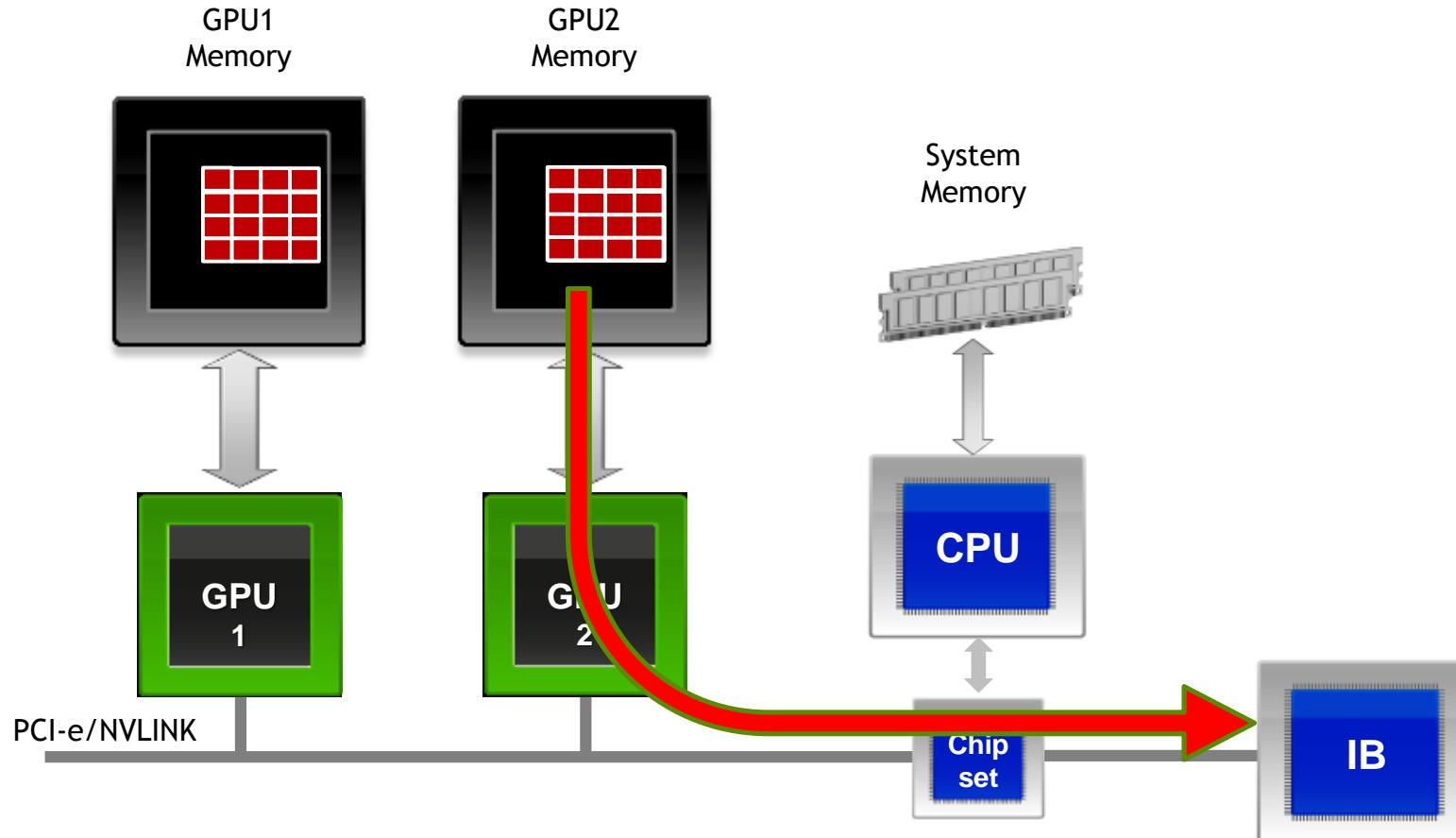
NVIDIA GPUDIRECT™

Peer to Peer Transfers



NVIDIA GPUDIRECT™

Support for RDMA





INTRA-PROCESS PEER-TO-PEER

SINGLE THREADED MULTI GPU PROGRAMMING

```
while ( l2_norm > tol && iter < iter_max ) {
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1); const int bottom = (dev_id+1)%num_devices;
        cudaSetDevice( dev_id );
        cudaMemsetAsync(l2_norm_d[dev_id], 0 , sizeof(real) );
        jacobi_kernel<<<dim_grid,dim_block>>>( a_new[dev_id], a[dev_id], l2_norm_d[dev_id],
                                                    iy_start[dev_id], iy_end[dev_id], nx );
        cudaMemcpyAsync( l2_norm_h[dev_id], l2_norm_d[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( a_new[top]+(iy_end[top]*nx), a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ... );
        cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), ... );
    }
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        cudaSetDevice( dev_id ); cudaDeviceSynchronize();
        l2_norm += *(l2_norm_h[dev_id]);
    }
    l2_norm = std::sqrt( l2_norm );
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) std::swap(a_new[dev_id],a[dev_id]);
    iter++;
}
```

GPUDIRECT P2P

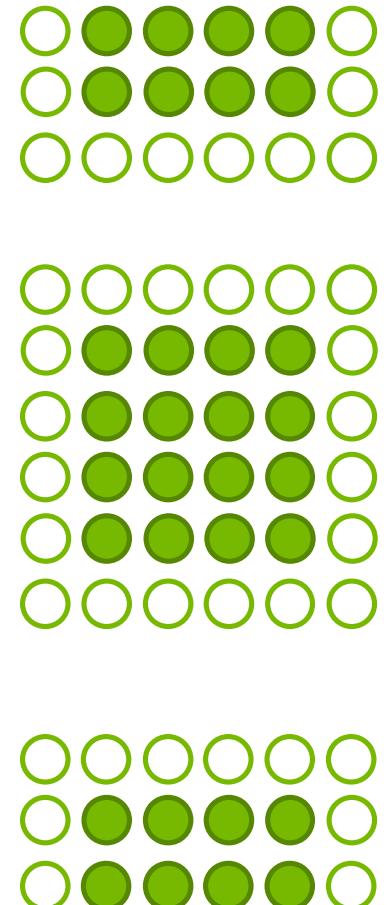
Enable P2P

```
for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
    cudaSetDevice( dev_id );
    const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1);
    int canAccessPeer = 0;
    cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, top );
    if ( canAccessPeer )
        cudaDeviceEnablePeerAccess ( top, 0 );
    const int bottom = (dev_id+1)%num_devices;
    if ( top != bottom ) {
        cudaDeviceCanAccessPeer ( &canAccessPeer, dev_id, bottom );
        if ( canAccessPeer )
            cudaDeviceEnablePeerAccess ( bottom, 0 );
    }
}
```

EXAMPLE JACOBI

Top/Bottom Halo

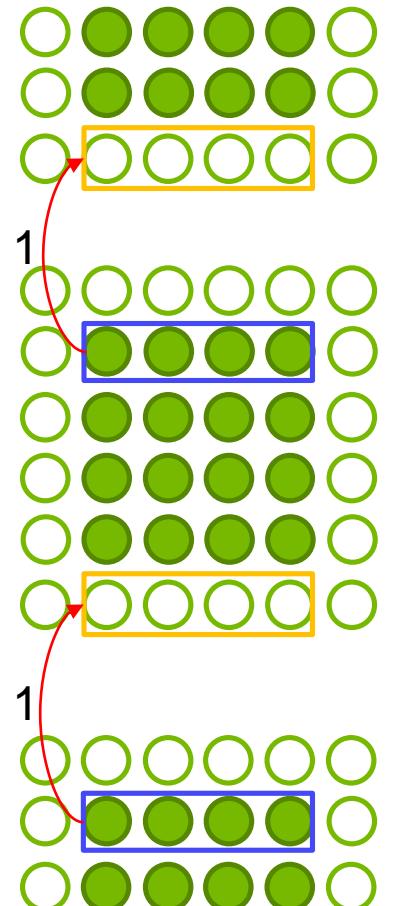
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx) ,  
    a_new[dev_id]+iy_start[dev_id]*nx ,  nx*sizeof(real) , ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

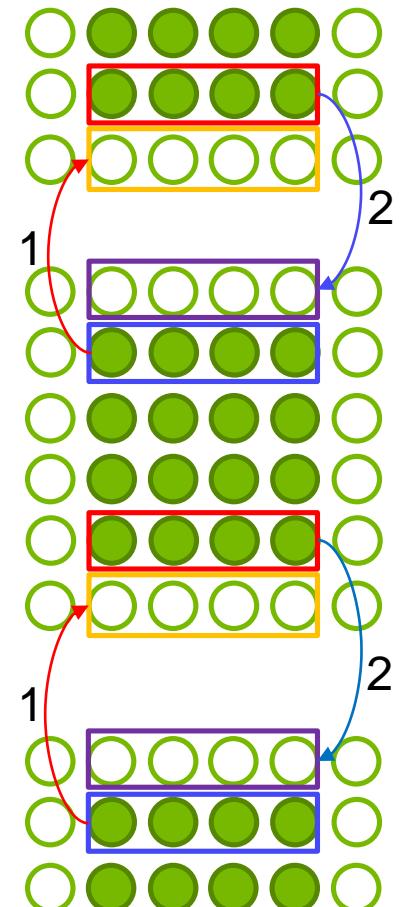
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);  
  
cudaMemcpyAsync(  
    a_new[bottom],  
    a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), ...);
```





CUDA IPC

MULTIPLE PROCESS, SINGLE GPU W/O MPI!

```
while ( l2_norm > tol && iter < iter_max ) {
    const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1); const int bottom = (dev_id+1)%num_devices;
    cudaSetDevice( dev_id );
    cudaMemsetAsync(l2_norm_d[dev_id], 0 , sizeof(real) );
    jacobi_kernel<<<dim_grid,dim_block>>>( a_new[dev_id], a[dev_id], l2_norm_d[dev_id],
                                                iy_start[dev_id], iy_end[dev_id], nx );
    cudaMemcpyAsync( l2_norm_h[dev_id], l2_norm_d[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
    cudaMemcpyAsync( a_new[top]+(iy_end[top]*nx), a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real) , ... );
    cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real) , ... );
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        l2_norm += *(l2_norm_h[dev_id]);
    }
    l2_norm = std::sqrt( l2_norm );
    std::swap(a_new[dev_id],a[dev_id]);
    iter++;
}
```

GPUDIRECT P2P

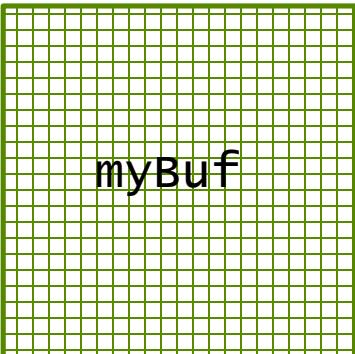
Enable CUDA Inter-Process Communication (IPC)!

```
cudaSetDevice( dev_id );
// Allocate and fill my device buffer
cudaMalloc((void **) &myBuf, nbytes);
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);
// Get my IPC handle
cudaIpcMemHandle_t myIpc;
cudaIpcGetMemHandle(&myIpc, myBuf);
```

GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

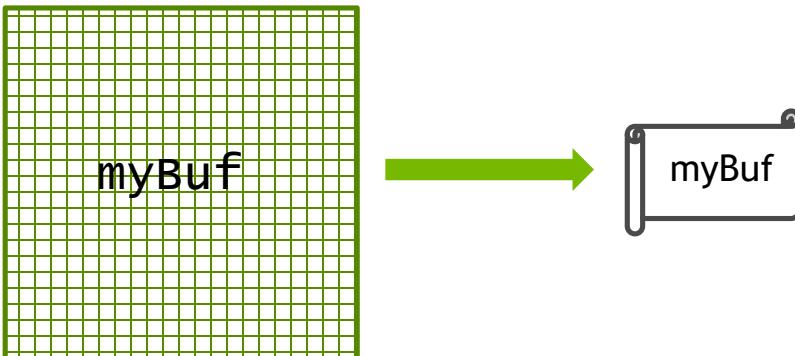
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

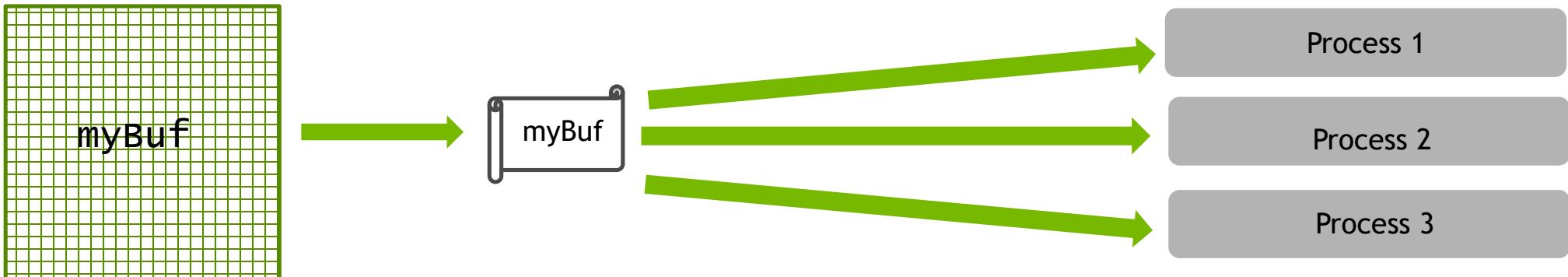
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

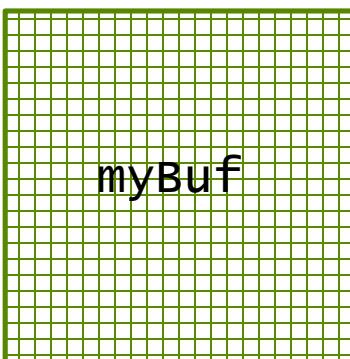
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```

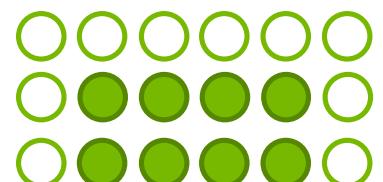
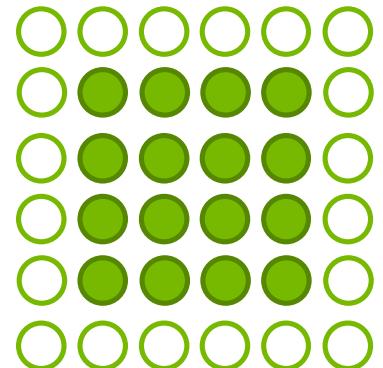
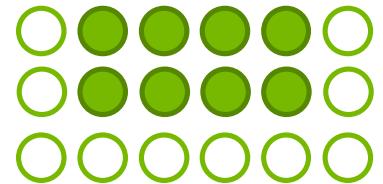


EXAMPLE JACOBI

Top/Bottom Halo

```
// Open their Ipc Handle onto a pointer
cudaIpcOpenMemHandle((void **) &a_new[top], topIpc,
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();

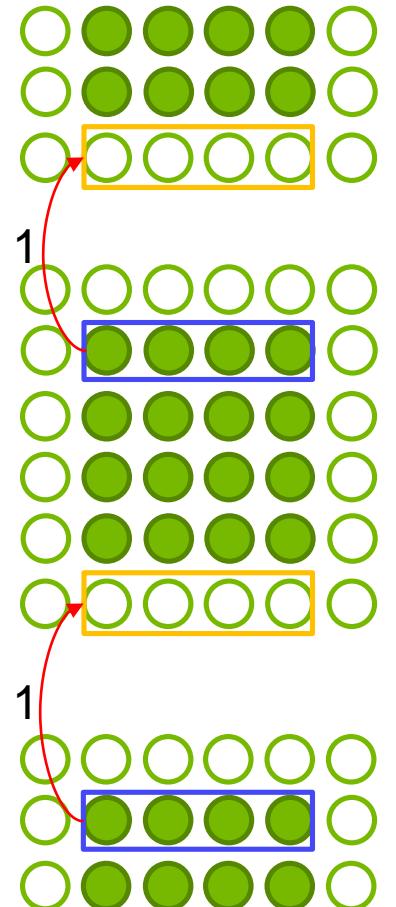
cudaMemcpyAsync(
    a_new[top]+(iy_end[top]*nx),
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

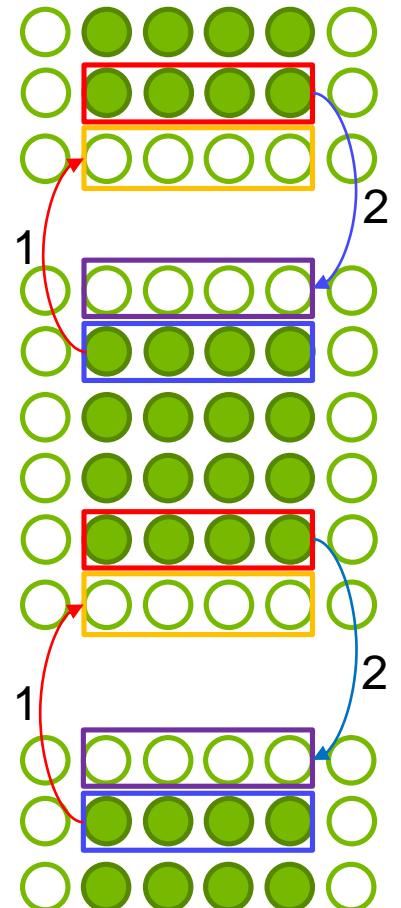
```
cudaIpcOpenMemHandle((void **) &a_new[top], topIpc,  
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();  
  
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

```
cudaIpcOpenMemHandle((void **) &a_new[top], topIpc,  
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();  
  
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);  
  
cudaIpcOpenMemHandle((void **) &a_new[bottom], bottomIpc,  
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();  
  
cudaMemcpyAsync(  
    a_new[bottom],  
    a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), ...);
```





PROFILING NVLINK USAGE

PROFILING NVLINK USAGE

Using nvprof+NVVP

Run nvprof multiple times to collect metrics

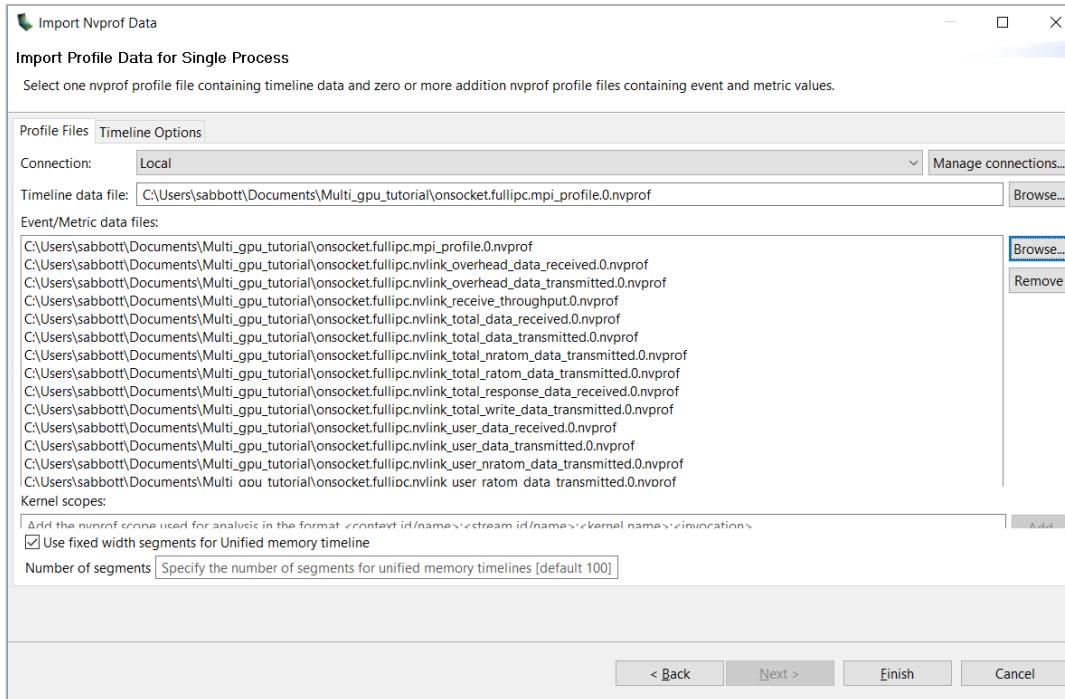
```
jsrun <args> nvprof --output-profile profile.<metric>.%(OMPI_COMM_WORLD_RANK) \  
    --aggregate-mode off --event-collection-mode continuous \  
    --metrics <metric> -f
```

Use `--query-metrics` and `--query-events` for full list of metrics (-m) or events (-e)

Combine with an MPI annotated timeline file for full picture

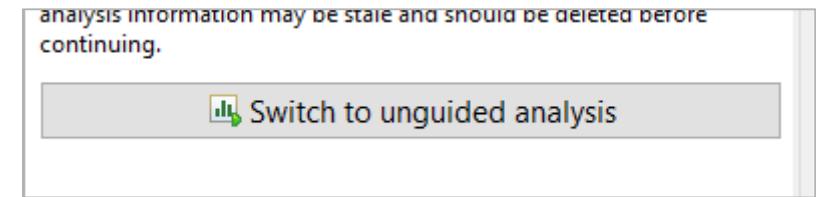
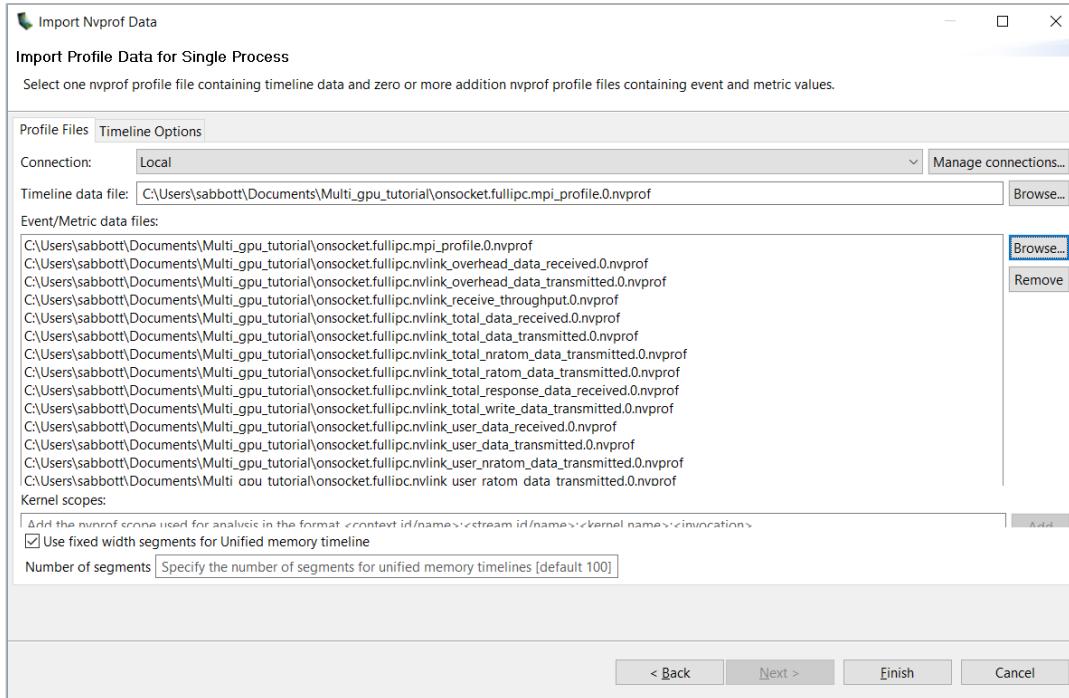
PROFILING NVLINK USAGE

Using nvprof+NVVP



PROFILING NVLINK USAGE

Using nvprof+NVVP



PROFILING NVLINK USAGE

Using nvprof+NVVP

Import Nvprof Data

Import Profile Data for Single Process
Select one nvprof profile file containing timeline data and zero or more addition nvprof profile files containing event and metric values.

Profile Files Timeline Options

Connection: Local

Timeline data file: C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.mpi_profile.0.nvprof

Browse... Remove

Event/Metric data files:

C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.mpi_profile.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_overhead_data_received.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_overhead_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_receive_throughput.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_total_data_received.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_total_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_total_ratom_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_total_response_data_received.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_total_write_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_data_received.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_ratom_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_ratom_data_transmitted.0.nvprof

Kernel scopes:
Δ Add the number of cores used for analysis in the format <context id>/names<stream id>/names<kernel names>/<iterations>
 Use fixed width segments for Unified memory timeline

Number of segments [Specify the number of segments for unified memory timelines [default 100]]

< Back Next > Finish

analysis information may be stale and should be deleted before continuing.

Switch to unguided analysis

Analysis GPU Details (Summary) CPU Details OpenACC Details Console Settings

To enable kernel analysis stages select a host-launched kernel instance in the timeline.

Application

Data Movement And Concurrency

Compute Utilization

Kernel Performance

Dependency Analysis

NVLink

Unified Memory

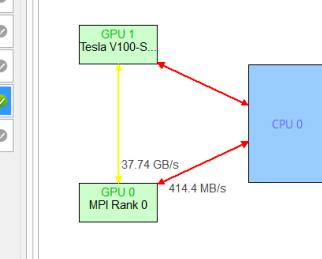
i NVLink Analysis
The following NVLink topology diagram shows logical NVLink connections between GPUs and CPUs. A logical NVLink can contain one or more physical links. When two throughput of device B. The tables on right hand side show the properties for each logical NVLink.
* NVLink utilization may vary in accuracy, because any activity within the sampling period is treated as active, even though most of that period could be idle.

Logical NVLink Properties

Logical NVLink	PeakBandwidth	Phy
GPU0<->CP...	100 GB/s	
GPU0<->GP...	100 GB/s	
GPU1<->CP...	100 GB/s	

Logical NVLink Throughput

Logical NVLink	Avg Throughput	Max
GPU0-->CPU0	n/a	
GPU0-->CPU1	414.398 MB/s	5
GPU0-->GP...	n/a	
GPU0-->GPU1	37.738 GB/s	5
GPU1-->CPU0	n/a	
GPU1-->CPU0	n/a	





SUMMARY

GPU TO GPU COMMUNICATION

- ▶ CUDA aware MPI functionally portable
 - ▶ OpenACC/MP interoperable
 - ▶ Performance may vary between on/off node, socket, HW support for GPU Direct
 - ▶ Unified memory support varies between implementations, but it becoming common
- ▶ Single-process, multi-GPU
 - ▶ Enable peer access for straight forward on-node transfers
- ▶ Multi-process, single-gpu
 - ▶ Pass CUDA IPC handles for on-node copies
- ▶ Combine for more flexibility/complexity!



NVIDIA®

