GPUDIRECT, CUDA AWARE MPI, & CUDA IPC

Steve Abbott, Summit Training Workshop, December 2018

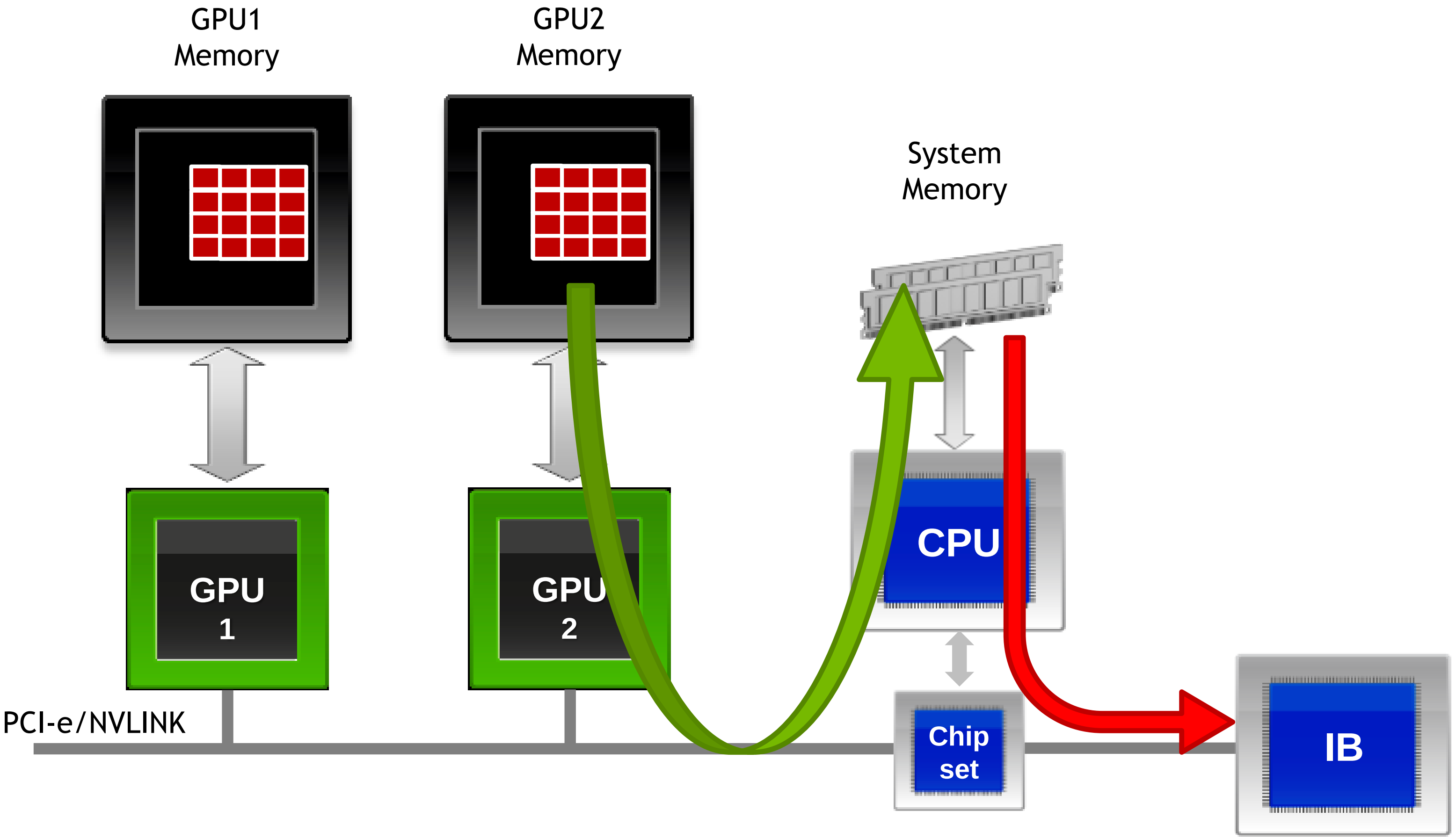
AGENDA

What is GPU Direct?
CUDA Aware MPI
Advanced On Node Communication

GPUDIRECT

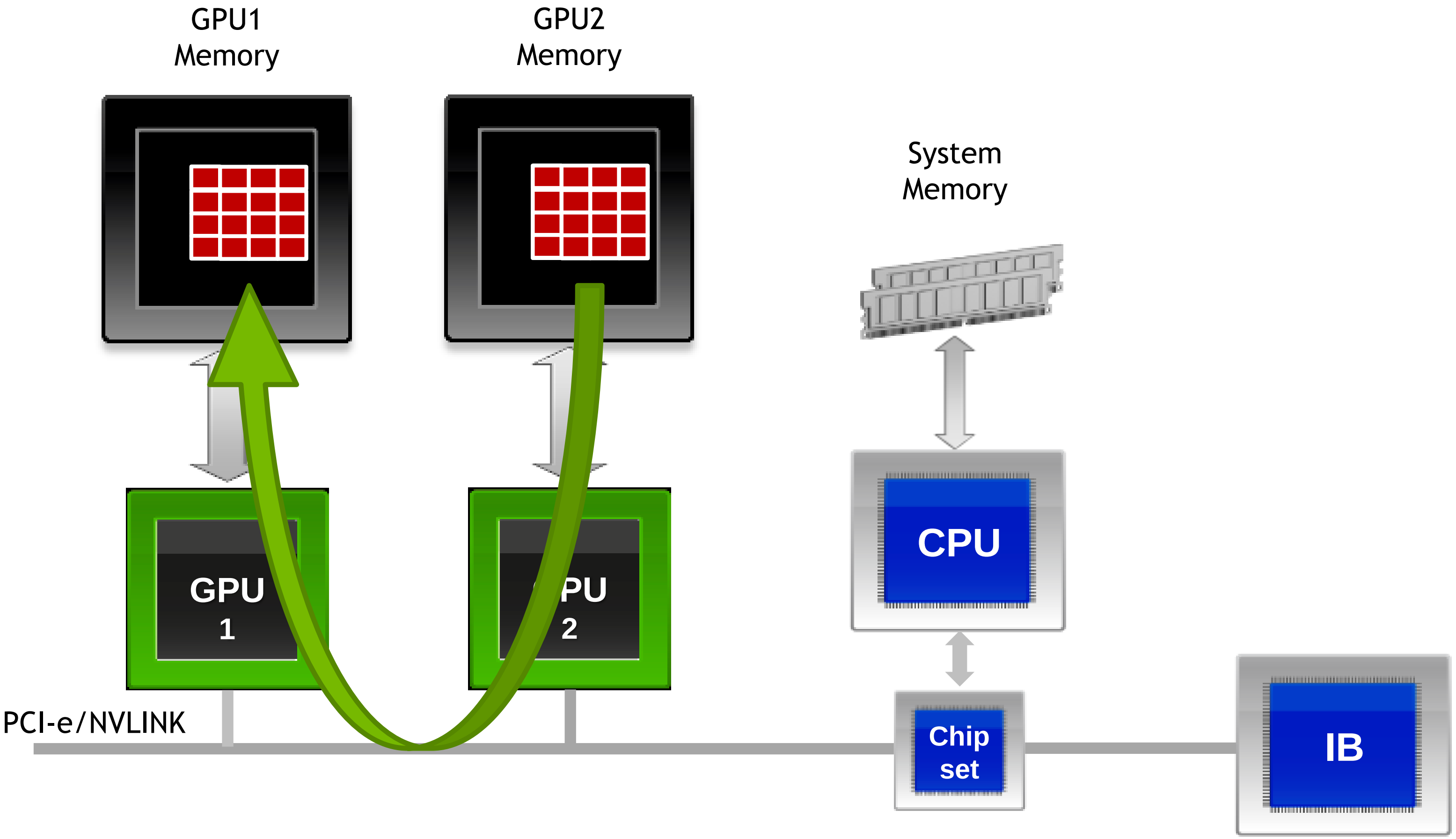
NVIDIA GPUDIRECT™

Accelerated Communication with Network & Storage Devices



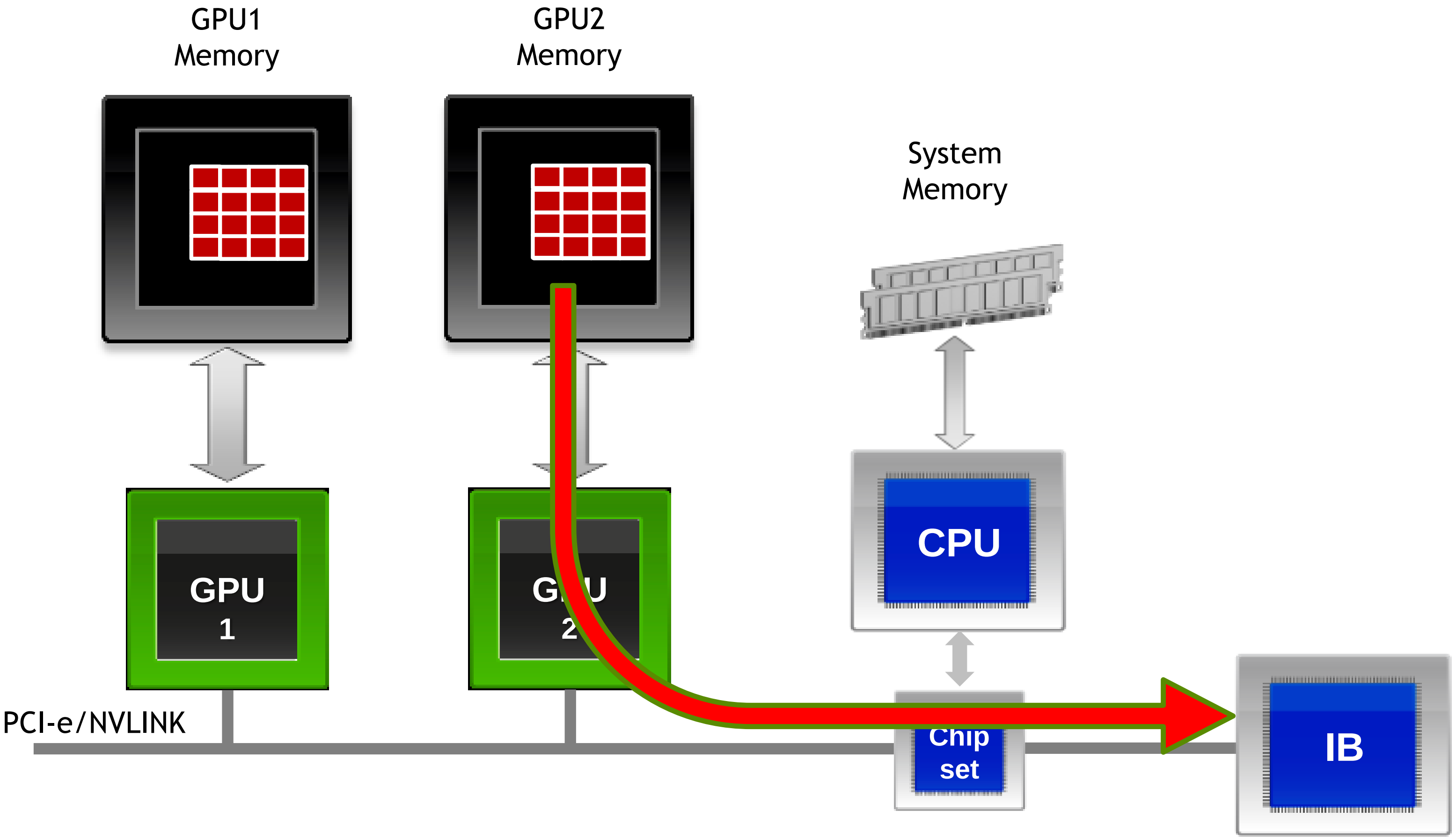
NVIDIA GPUDIRECT™

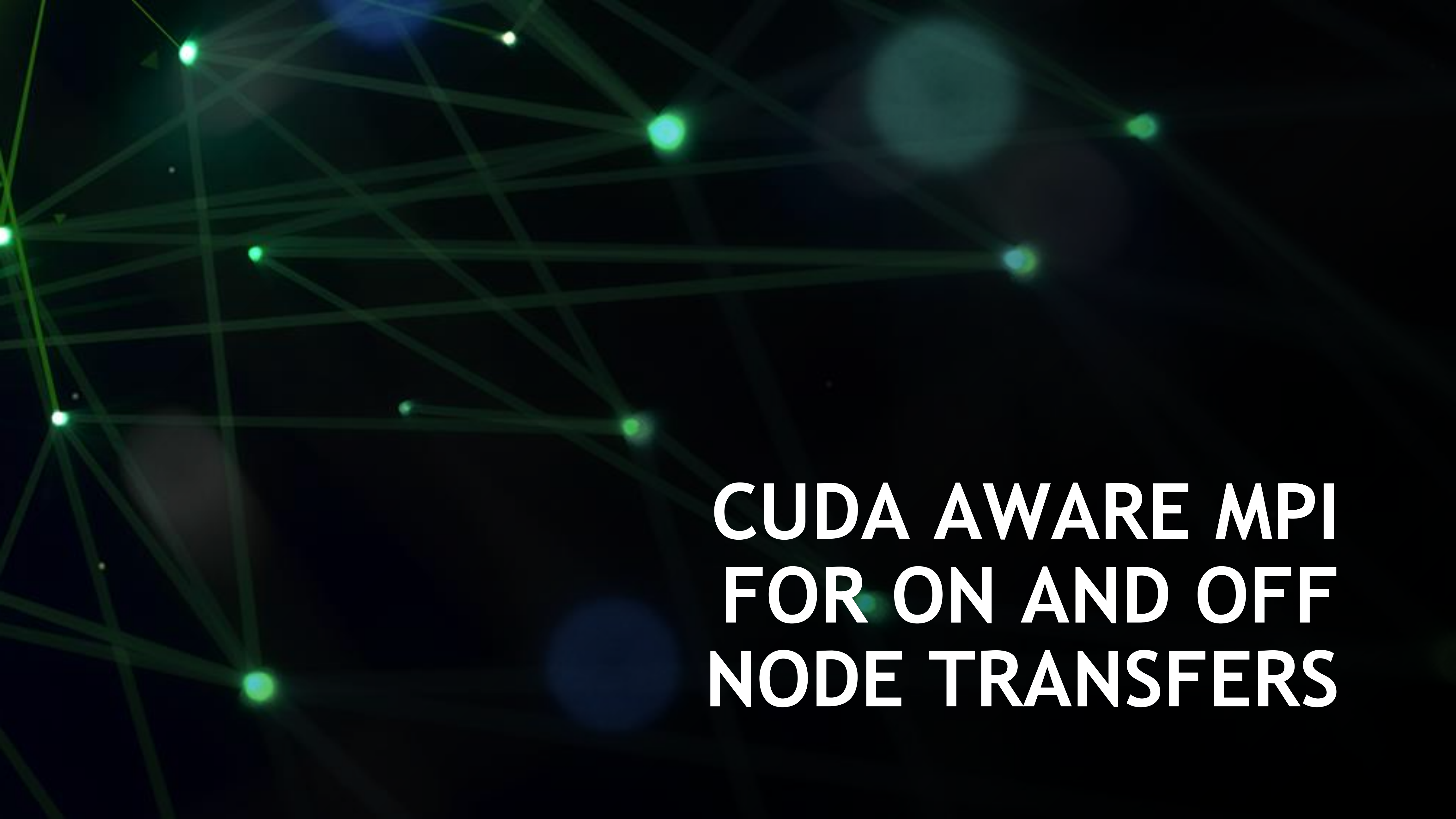
Peer to Peer Transfers



NVIDIA GPUDIRECT™

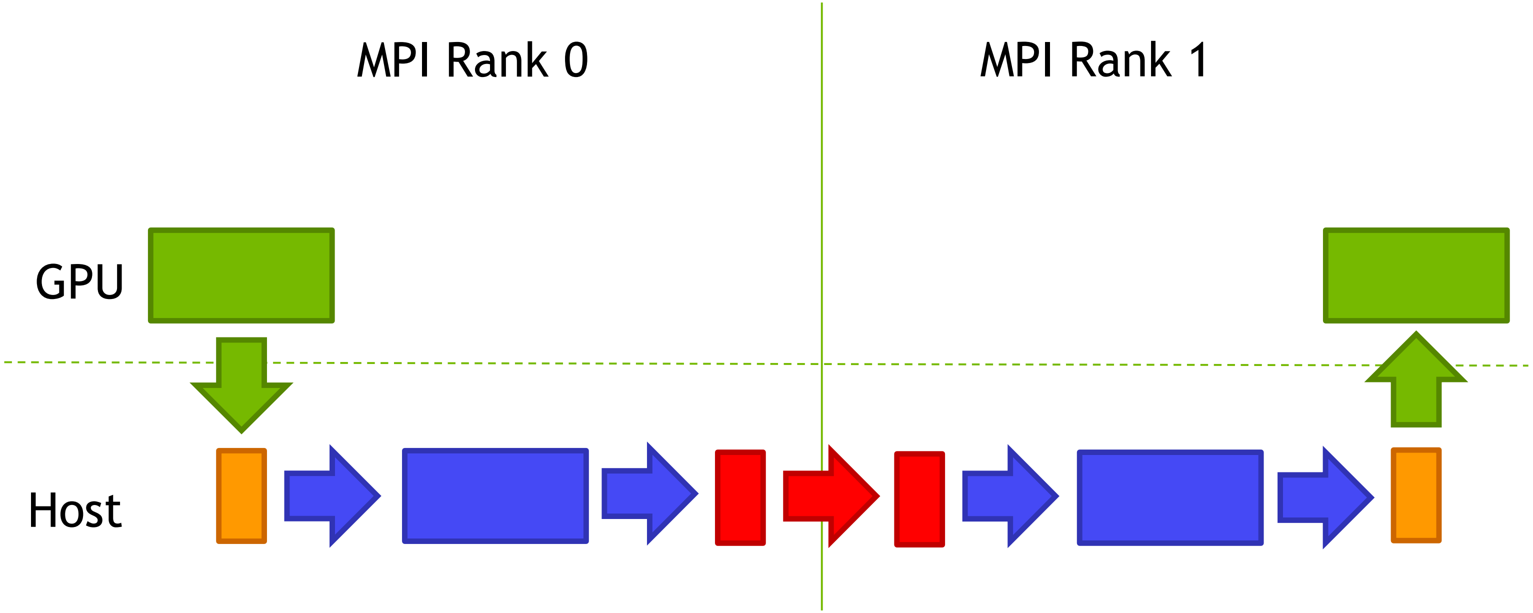
Support for RDMA





CUDA AWARE MPI FOR ON AND OFF NODE TRANSFERS

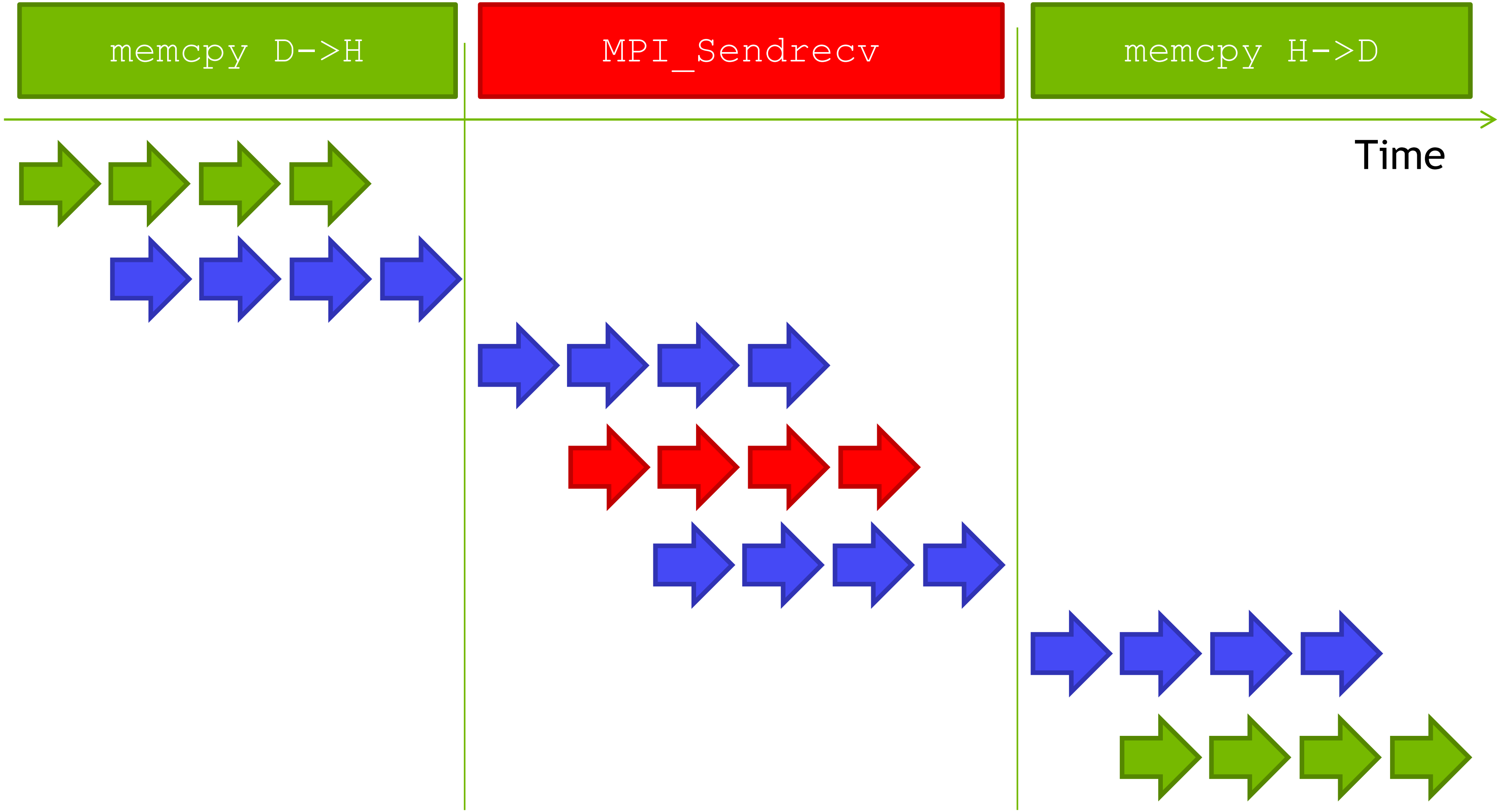
REGULAR MPI GPU TO REMOTE GPU



```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

REGULAR MPI GPU TO REMOTE GPU



memcpy D->H

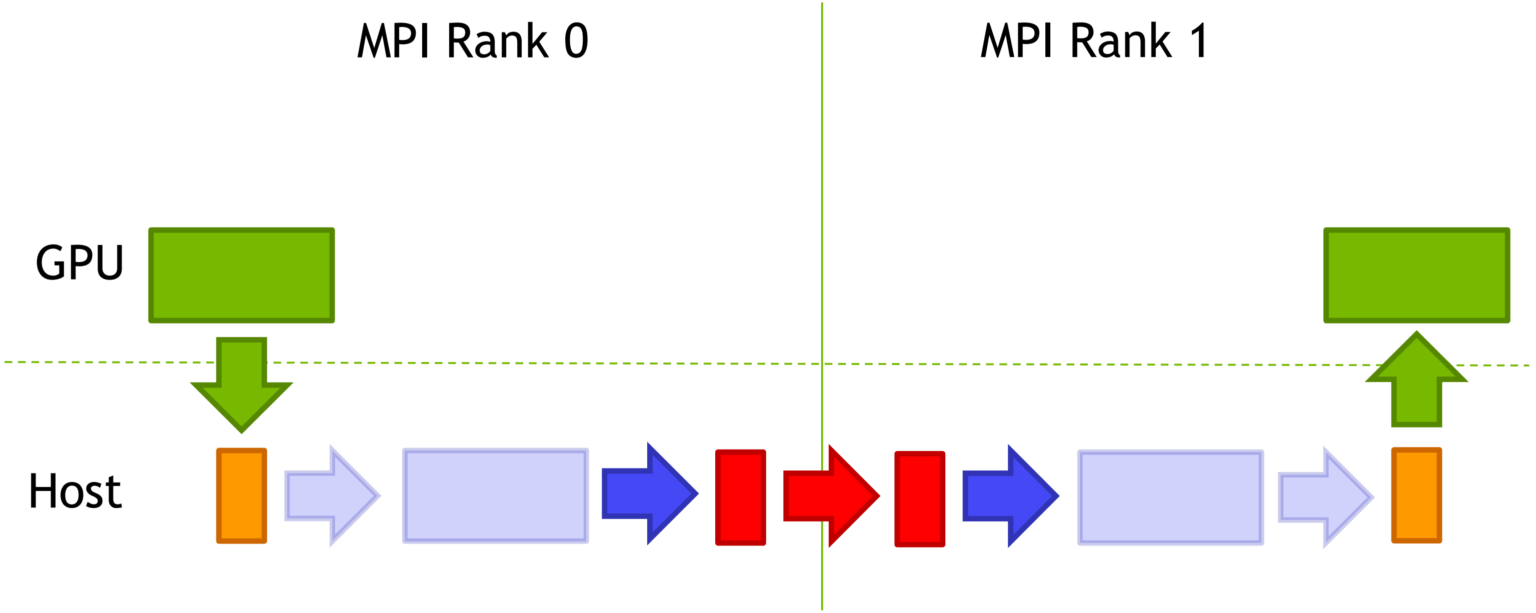
MPI_Sendrecv

memcpy H->D

Time

MPI GPU TO REMOTE GPU

without GPUDirect

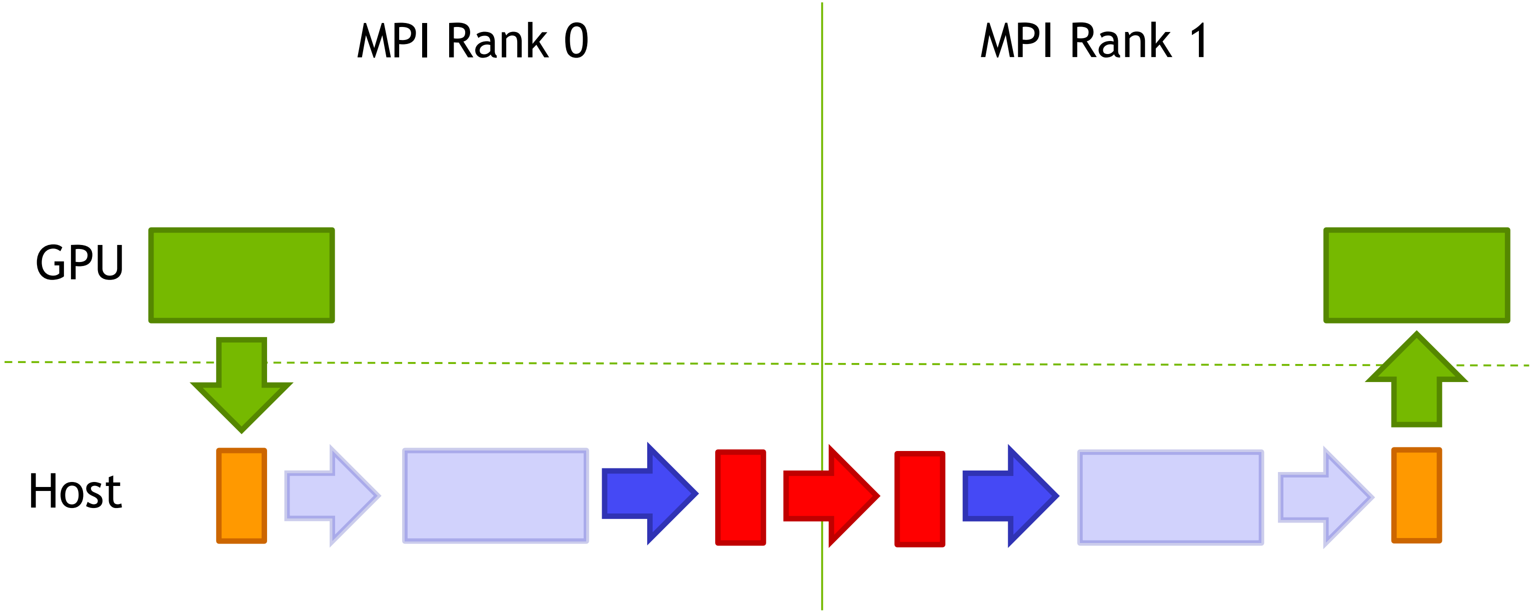


```
MPI_Send(s_buf_d,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI GPU TO REMOTE GPU

without GPUDirect

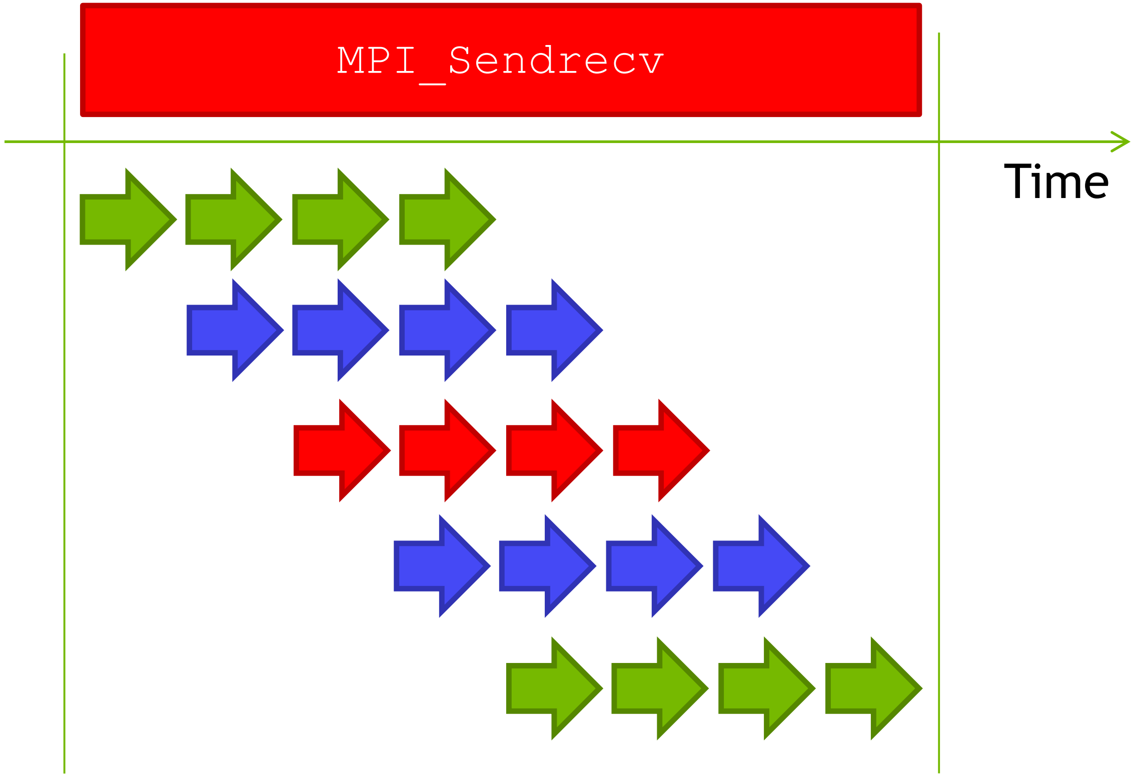


```
#pragma acc host_data use_device (s_buf, r_buf)
MPI_Send(s_buf,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

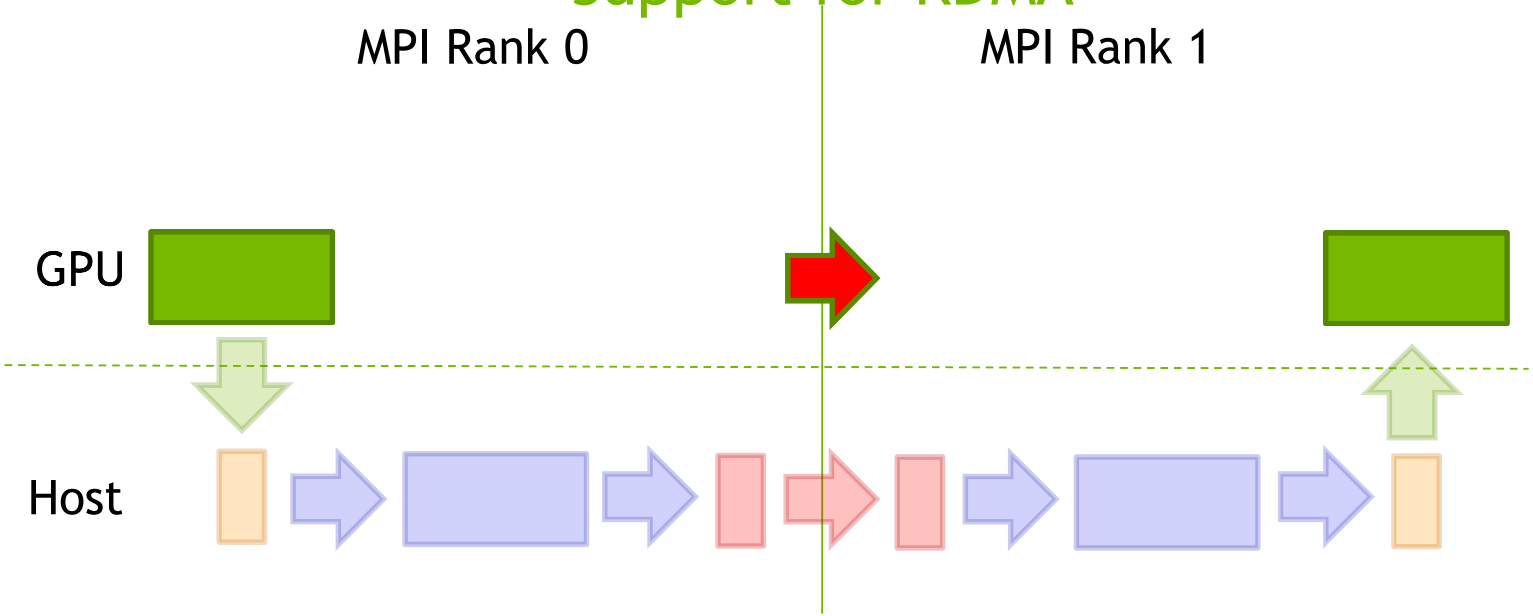
MPI GPU TO REMOTE GPU

without GPUDirect



MPI GPU TO REMOTE GPU

Support for RDMA

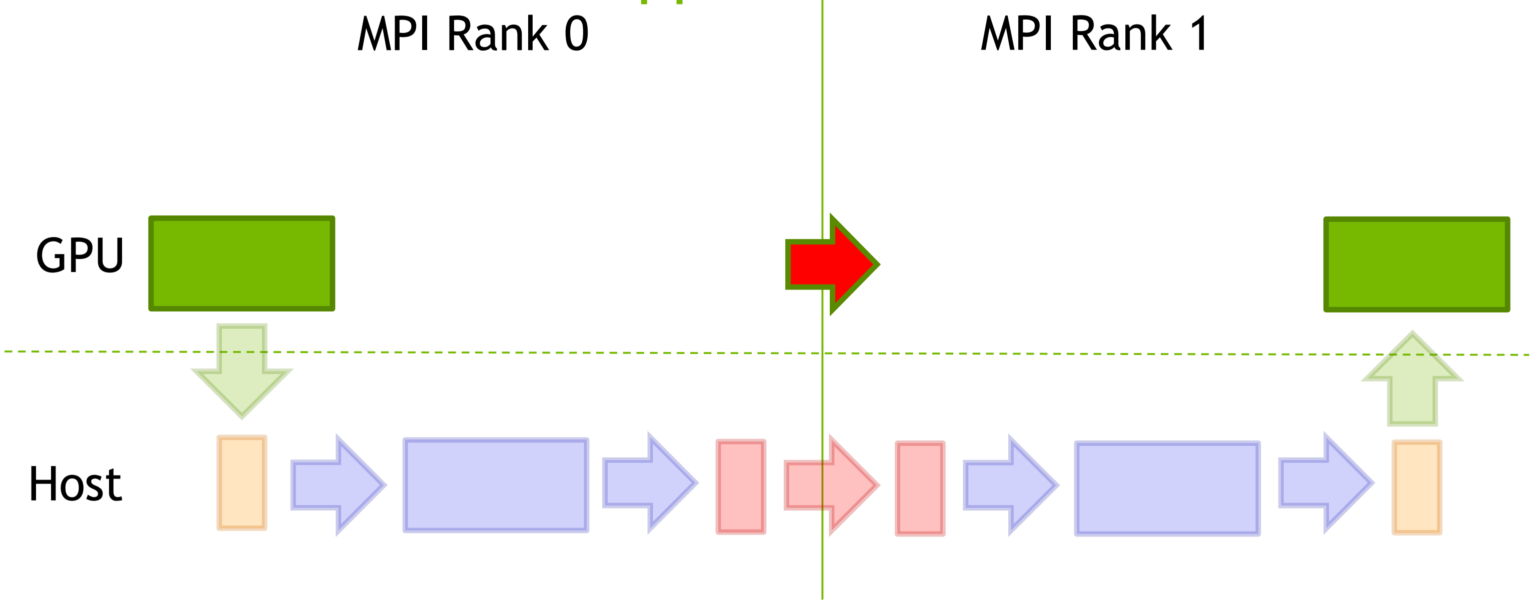


```
MPI_Send(s_buf_d,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI GPU TO REMOTE GPU

Support for RDMA

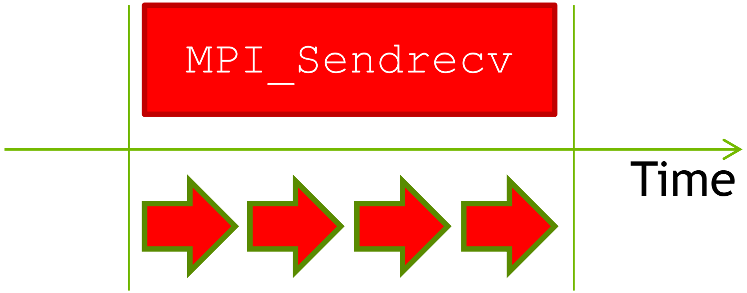


```
#pragma acc host_data use_device (s_buf, r_buf)
MPI_Send(s_buf,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI GPU TO REMOTE GPU

Support for RDMA



ADVANCED ON-NODE COMMUNICATION

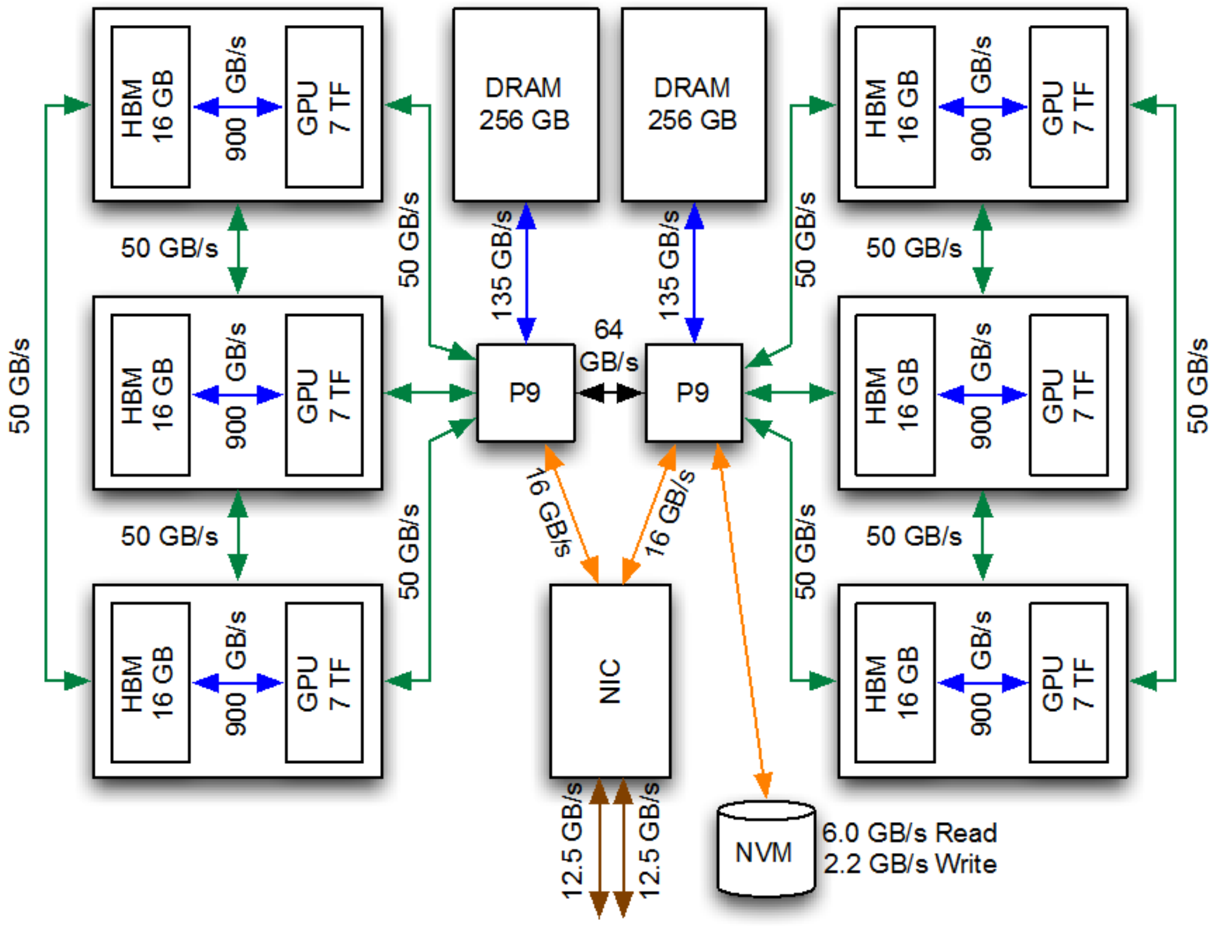
UNDER THE HOOD

Summit has fat nodes!

Many connections

Many devices

Many stacks



TF	42 TF (6x7 TF)
HBM	96 GB (6x16 GB)
DRAM	512 GB (2x16x16 GB)
NET	25 GB/s (2x12.5 GB/s)
MMsg/s	83

HBM/DRAM Bus (aggregate B/W)

NVLINK

X-Bus (SMP)

PCIe Gen4

EDR IB

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

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

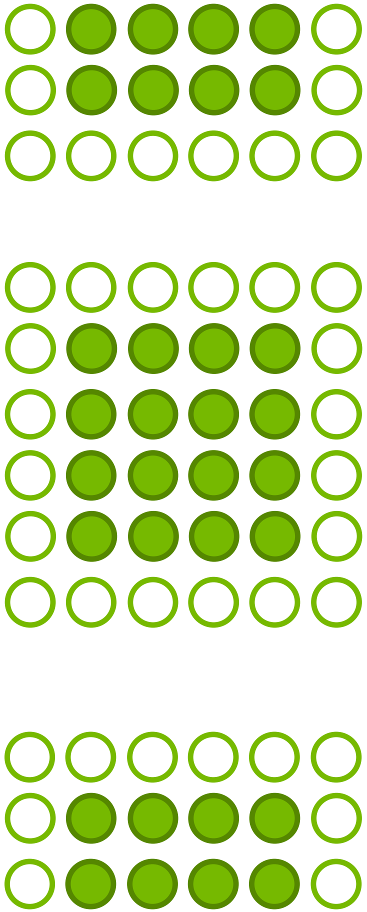
Enable P2P

```
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

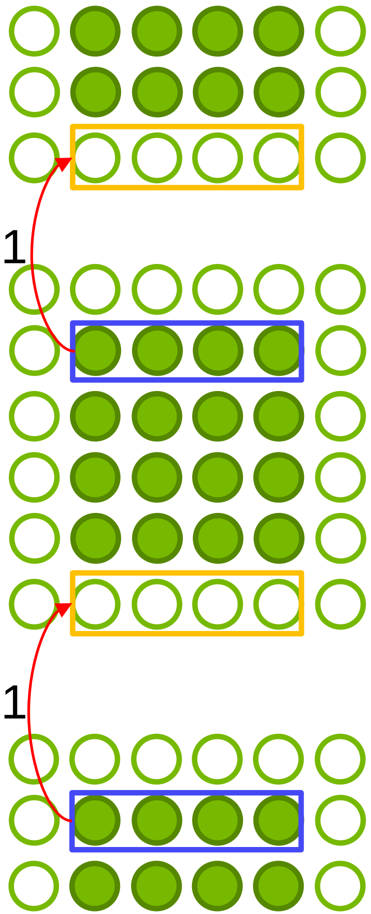
```
cudaMemcpyAsync(
  a_new[top]+(iy_end[top]*nx),
  a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



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), ...);
```

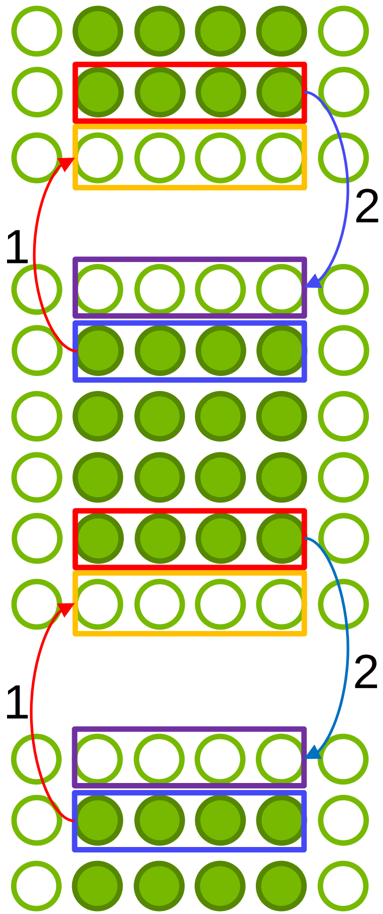


EXAMPLE JACOBI

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), ...);
```



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

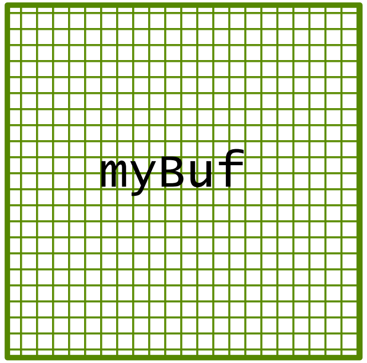
Enable CUDA Intra-Process Communication (IPC)!

```
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 Intra-Process Communication (IPC)!

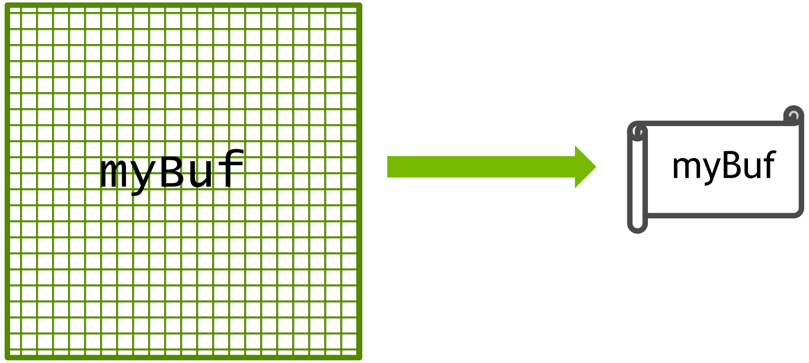
```
cudaSetDevice( dev_id );
// Allocate and fill my device buffer
cudaMalloc((void **) &myBuf, nbytes);
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);
// Get my IPC handle
cudaIpcMemHandle_t myIpc;
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Intra-Process Communication (IPC)!

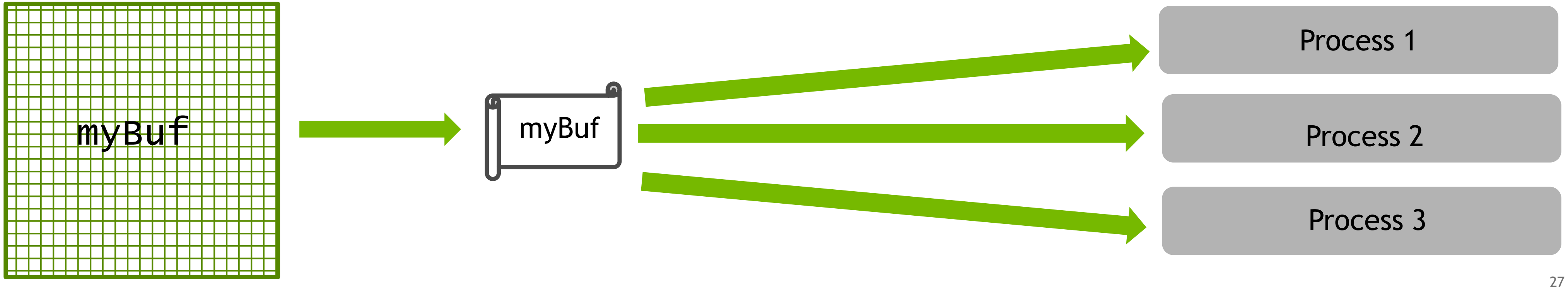
```
cudaSetDevice( dev_id );
// Allocate and fill my device buffer
cudaMalloc((void **) &myBuf, nbytes);
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);
// Get my IPC handle
cudaIpcMemHandle_t myIpc;
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Intra-Process Communication (IPC)!

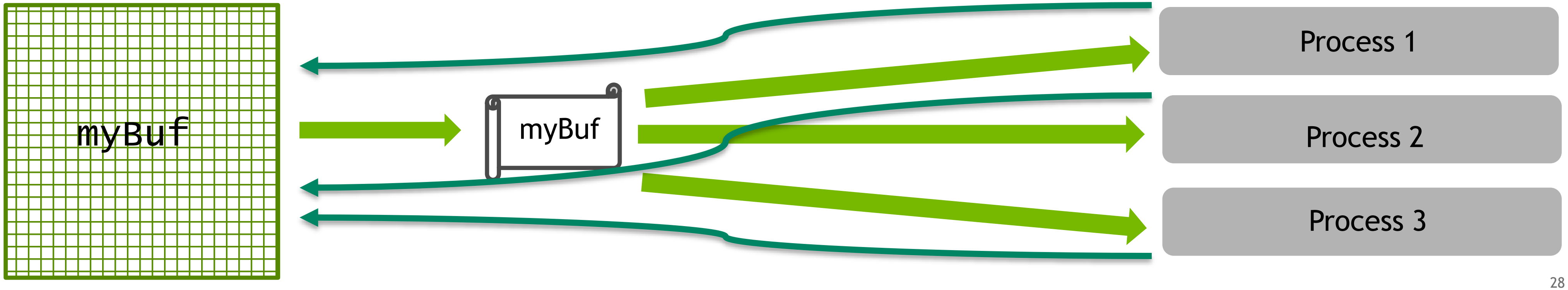
```
cudaSetDevice( dev_id );
// Allocate and fill my device buffer
cudaMalloc((void **) &myBuf, nbytes);
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);
// Get my IPC handle
cudaIpcMemHandle_t myIpc;
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Intra-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);
```

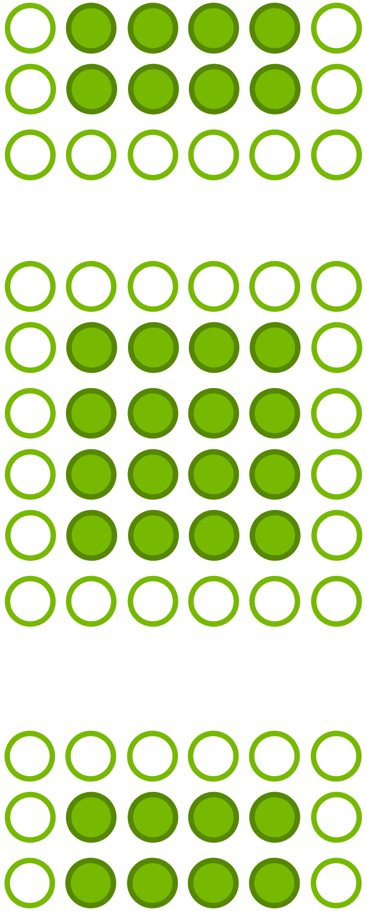


EXAMPLE JACOBI

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), ...);
```

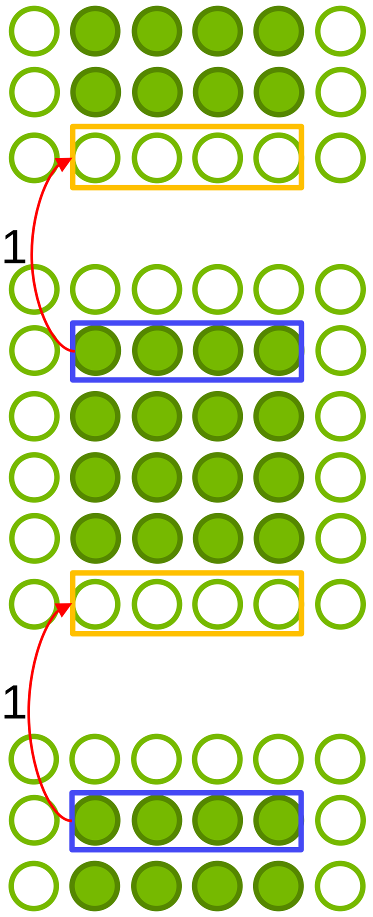


EXAMPLE JACOBI

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), ...);
```



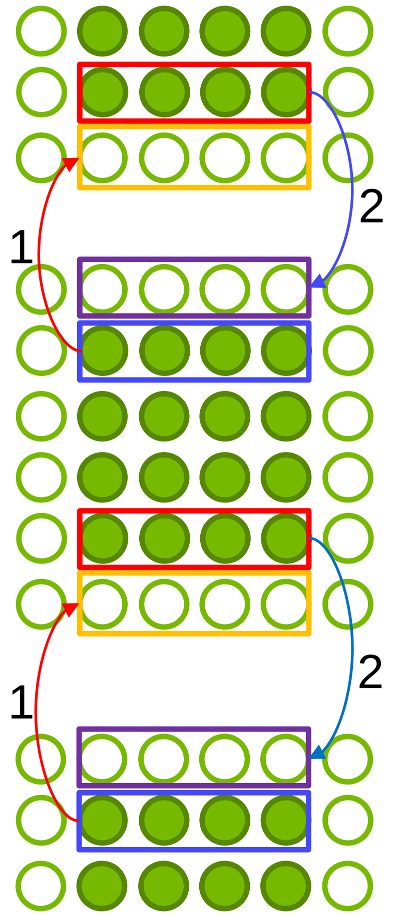
EXAMPLE JACOBI

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), ...);
```



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!

ESSENTIAL TOOLS

JSRUN/SMPI GPU OPTIONS

To enable CUDA aware MPI, use **jsrun --smpiargs="-gpu"**

To run GPU code without MPI, use **jsrun --smpiargs="off"**

KNOWN ISSUES

Things to watch out for (as of December)

No CUDA IPC across resource sets:

[1]Error opening IPC Memhandle from peer:0, invalid argument

One WAR: set PAMI_DISABLE_IPC=1

One (more complicated) WAR: bsub –step_cgroup n and

`swizzle`CUDA_VISIBLE_DEVICES [0,1,2] & [1,0,2] & [2,1,0]

CLOSING 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
  - WARNING: Unified memory support varies wildly between implementations!
- 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!

ESSENTIAL TOOLS AND TRICKS

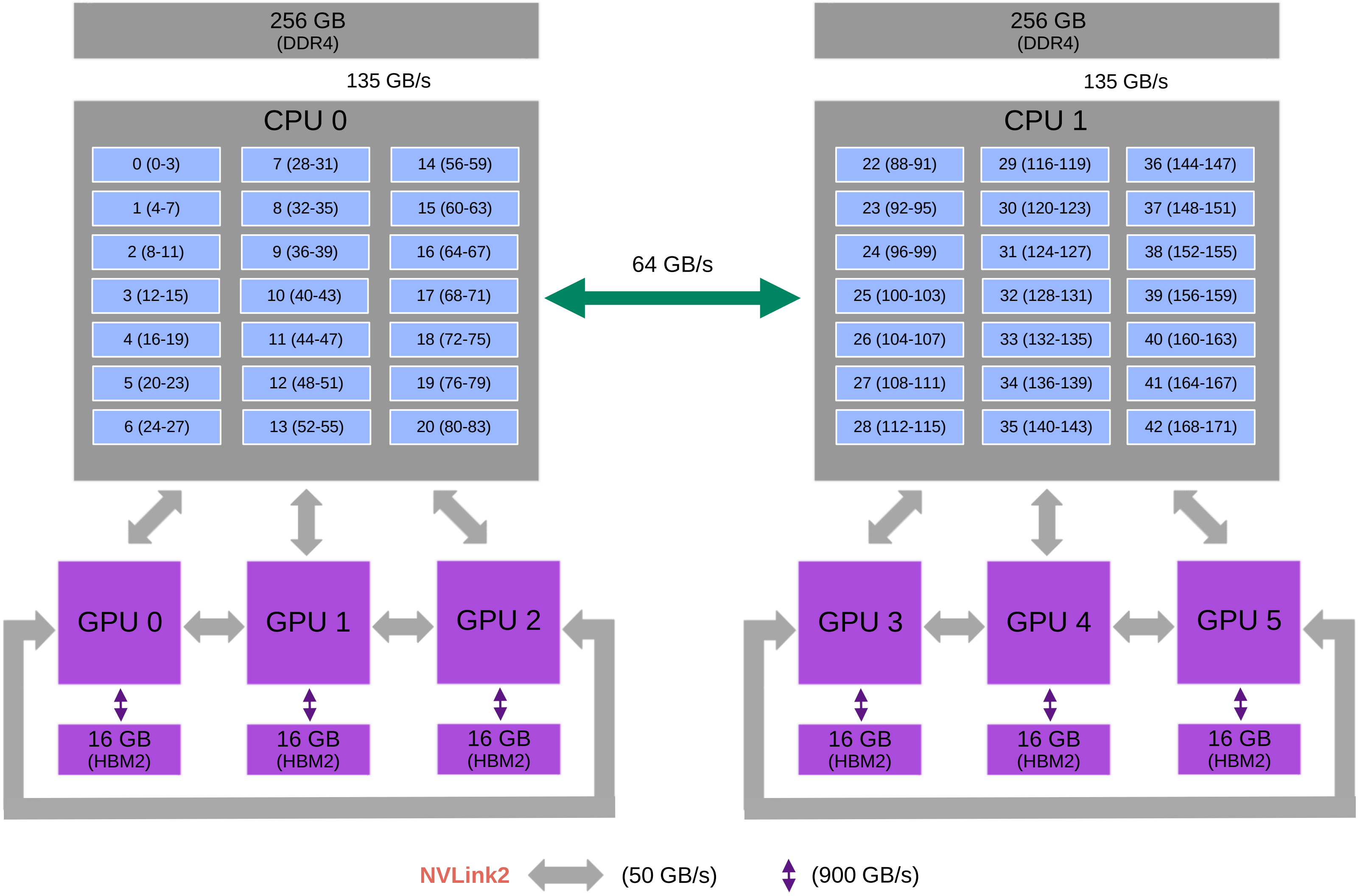
- Pick on-node layout with OLCF jsrun visualizer
  - https://jsrunvisualizer.olcf.ornl.gov/index.html
- Select MPI/GPU interaction with jsrun --smpiargs
  - “-gpu” for CUDA aware, “off” for pure GPU without MPI
- Profile MPI and NVLinks with nvprof
- Good performance will require experimentation!

NVIDIA®

SUMMIT NODE OVERVIEW

SUMMIT NODE

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



256 GB
(DDR4)

135 GB/s

CPU 0

0 (0-3)	7 (28-31)	14 (56-59)
1 (4-7)	8 (32-35)	15 (60-63)
2 (8-11)	9 (36-39)	16 (64-67)
3 (12-15)	10 (40-43)	17 (68-71)
4 (16-19)	11 (44-47)	18 (72-75)
5 (20-23)	12 (48-51)	19 (76-79)
6 (24-27)	13 (52-55)	20 (80-83)

64 GB/s

256 GB
(DDR4)

135 GB/s

CPU 1

22 (88-91)	29 (116-119)	36 (144-147)
23 (92-95)	30 (120-123)	37 (148-151)
24 (96-99)	31 (124-127)	38 (152-155)
25 (100-103)	32 (128-131)	39 (156-159)
26 (104-107)	33 (132-135)	40 (160-163)
27 (108-111)	34 (136-139)	41 (164-167)
28 (112-115)	35 (140-143)	42 (168-171)

NVLink2 ⟷ (50 GB/s) ↕ (900 GB/s)

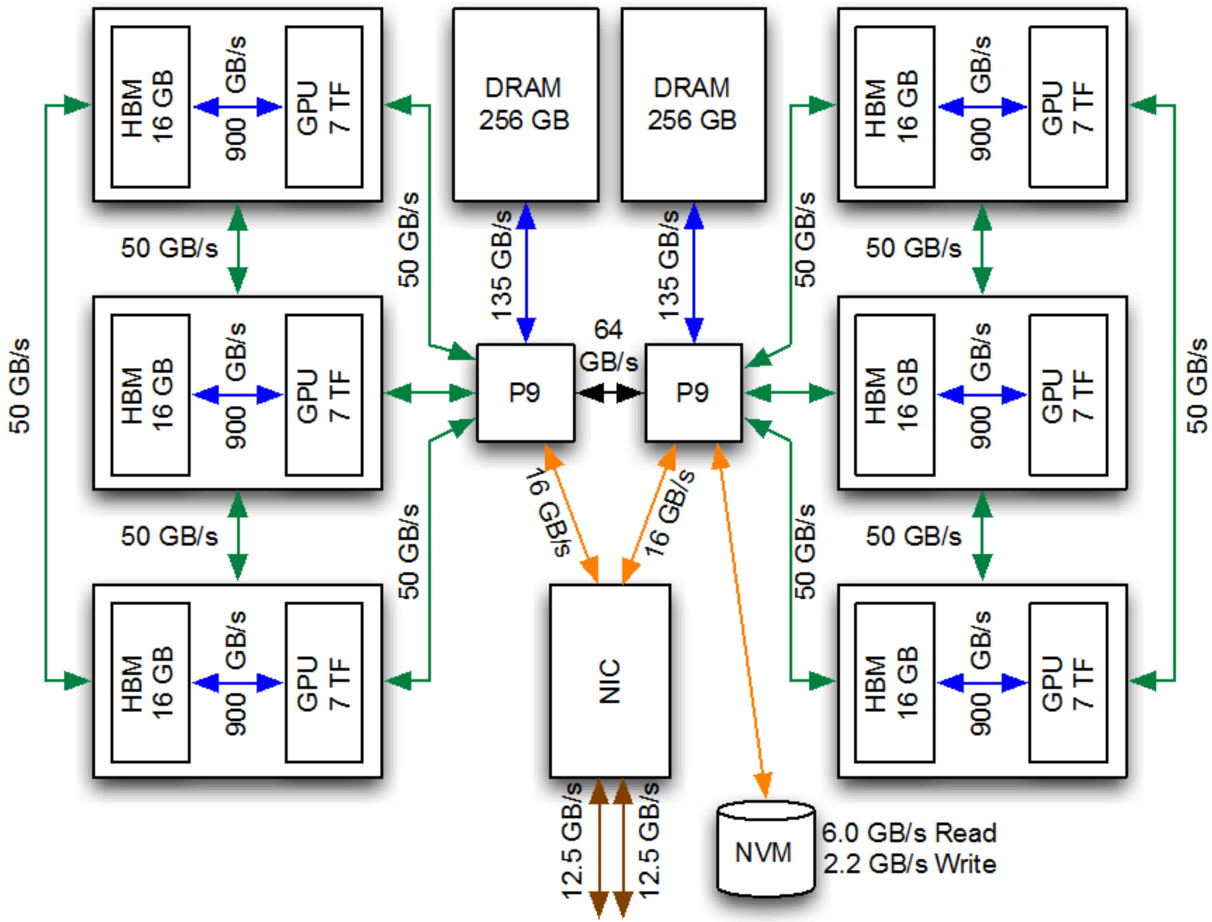
UNDER THE HOOD

Summit has fat nodes!

Many connections

Many devices

Many stacks



TF	42 TF (6x7 TF)
HBM	96 GB (6x16 GB)
DRAM	512 GB (2x16x16 GB)
NET	25 GB/s (2x12.5 GB/s)
MMsg/s	83

↔ HBM/DRAM Bus (aggregate B/W)

↔ NVLINK

↔ X-Bus (SMP)

↔ PCIe Gen4

↔ EDR IB

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.