



GPUDIRECT, CUDA AWARE MPI, & CUDA IPC

Steve Abbott, February 12, 2019

AGENDA

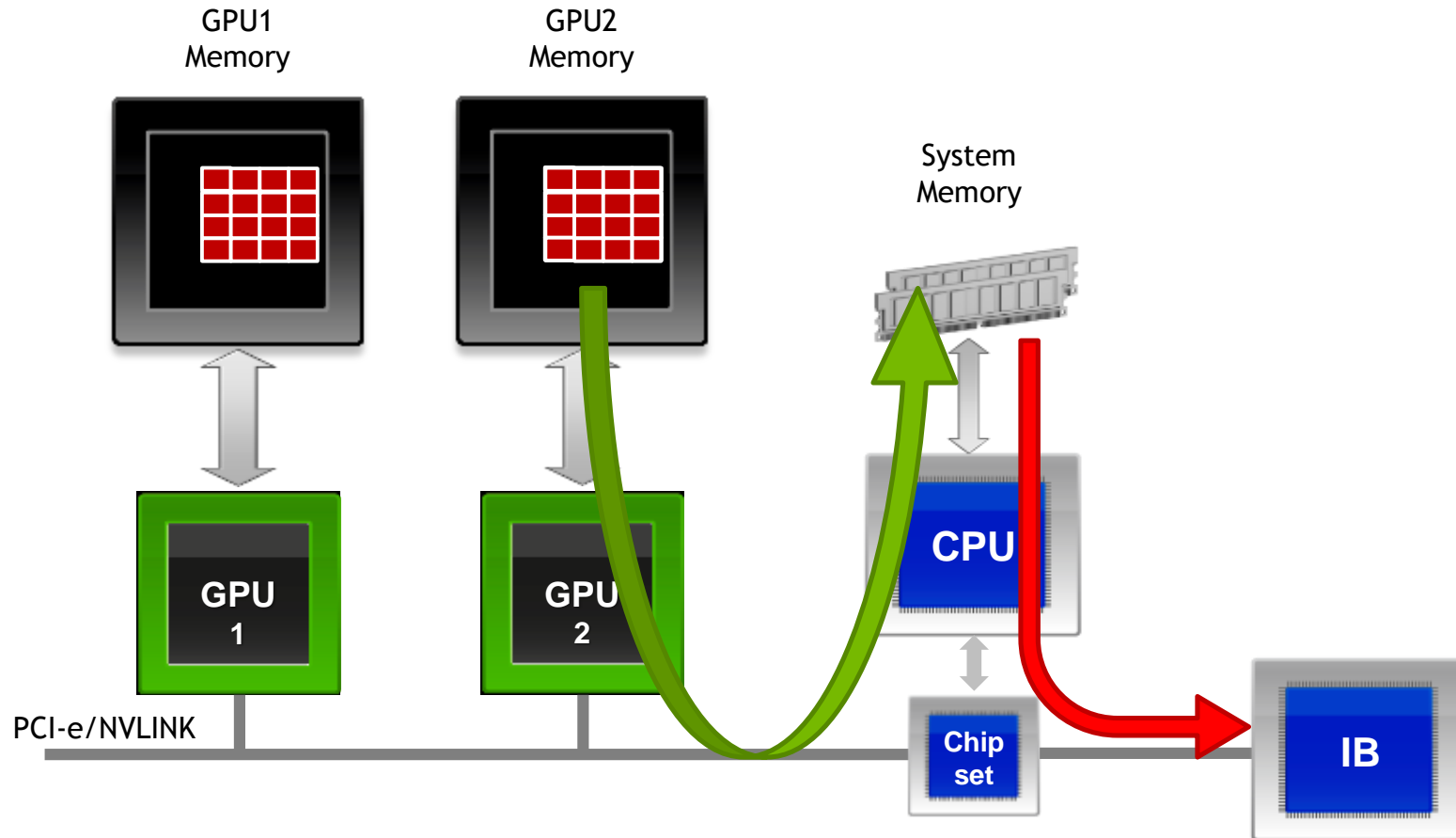
What is GPU Direct?
CUDA Aware MPI
Advanced On Node Communication



GPUDIRECT

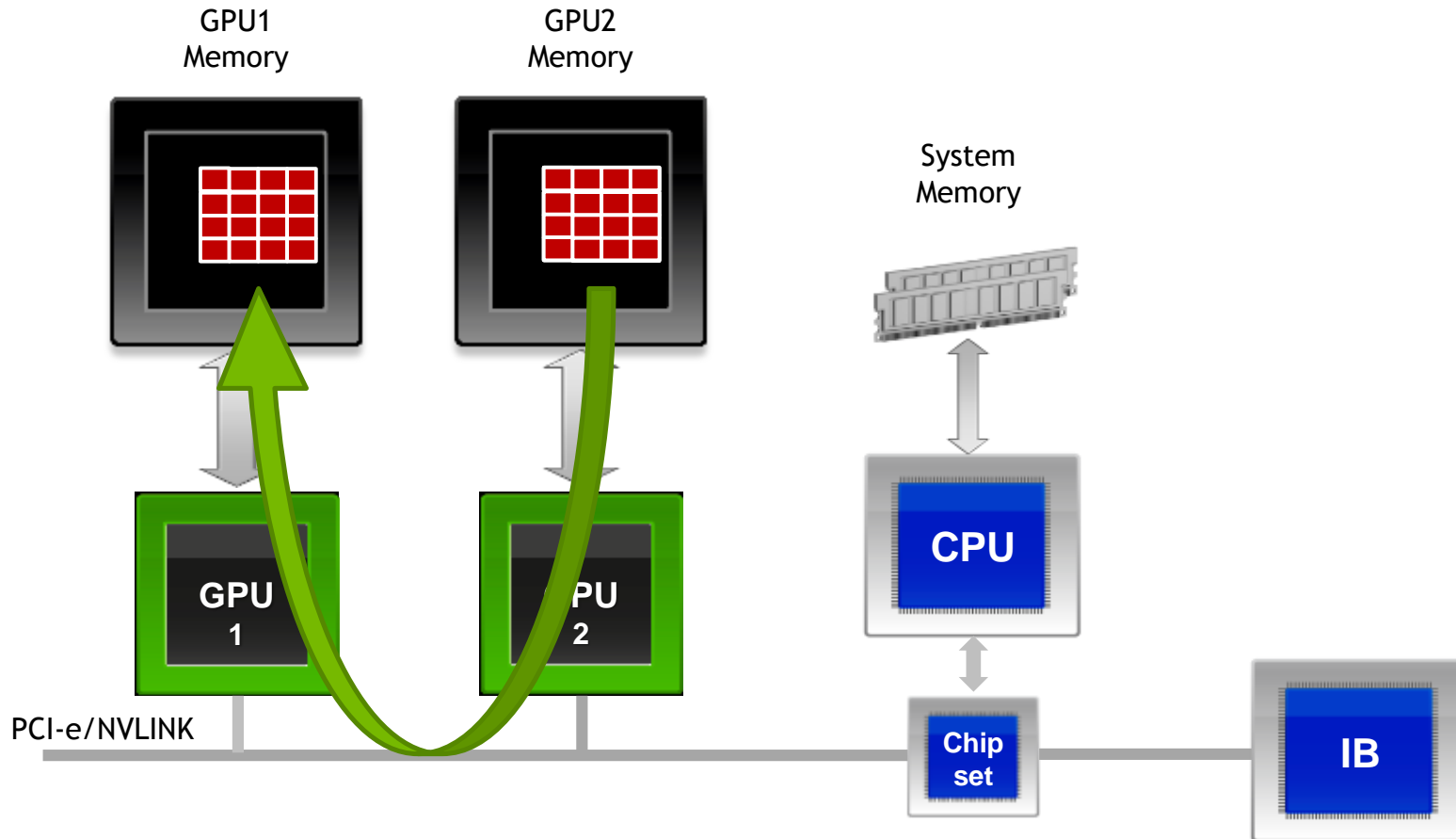
NVIDIA GPUDIRECT™

Accelerated Communication with Network & Storage Devices



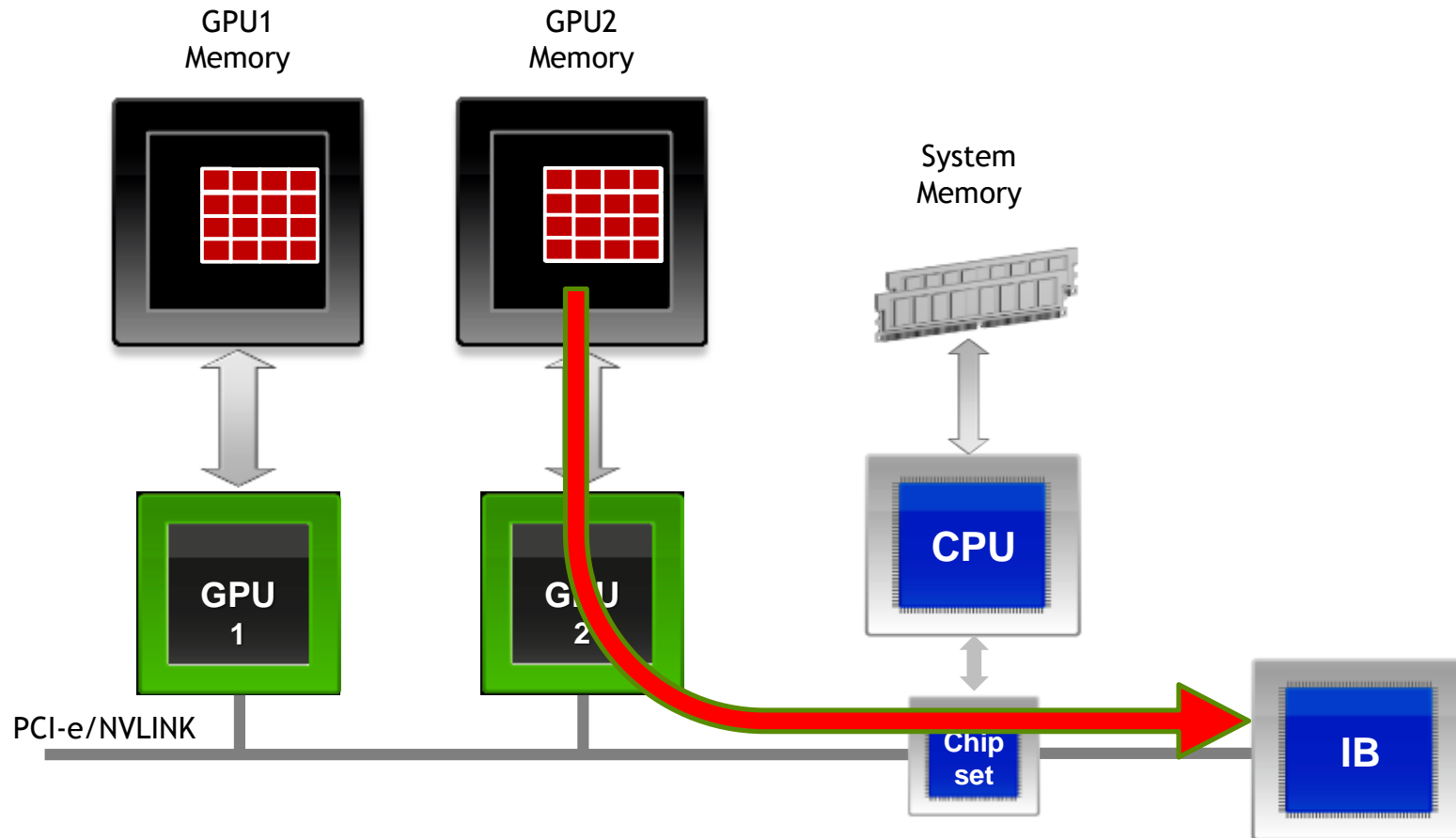
NVIDIA GPUDIRECT™

Peer to Peer Transfers



NVIDIA GPUDIRECT™

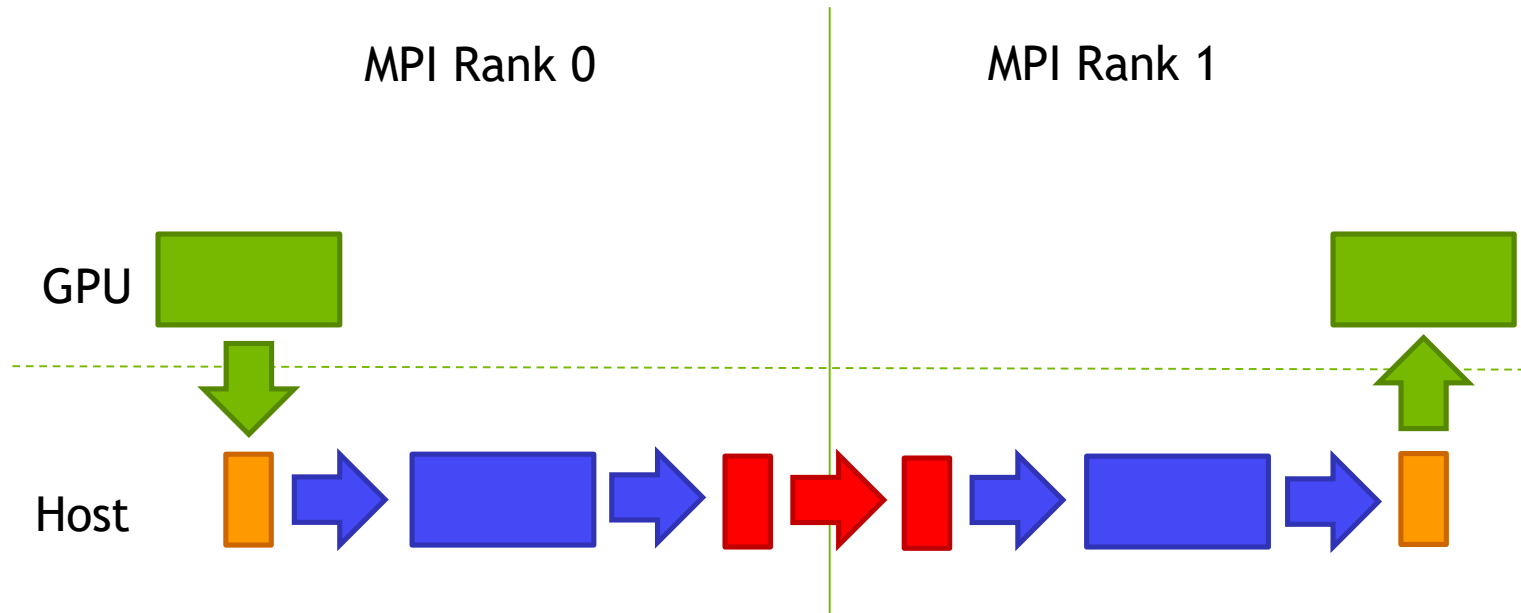
Support for RDMA



A network diagram with green nodes and lines on a dark background. The nodes are represented by small green circles, and the lines are thin green lines connecting the nodes. The background is dark blue/black with some faint, larger green circles.

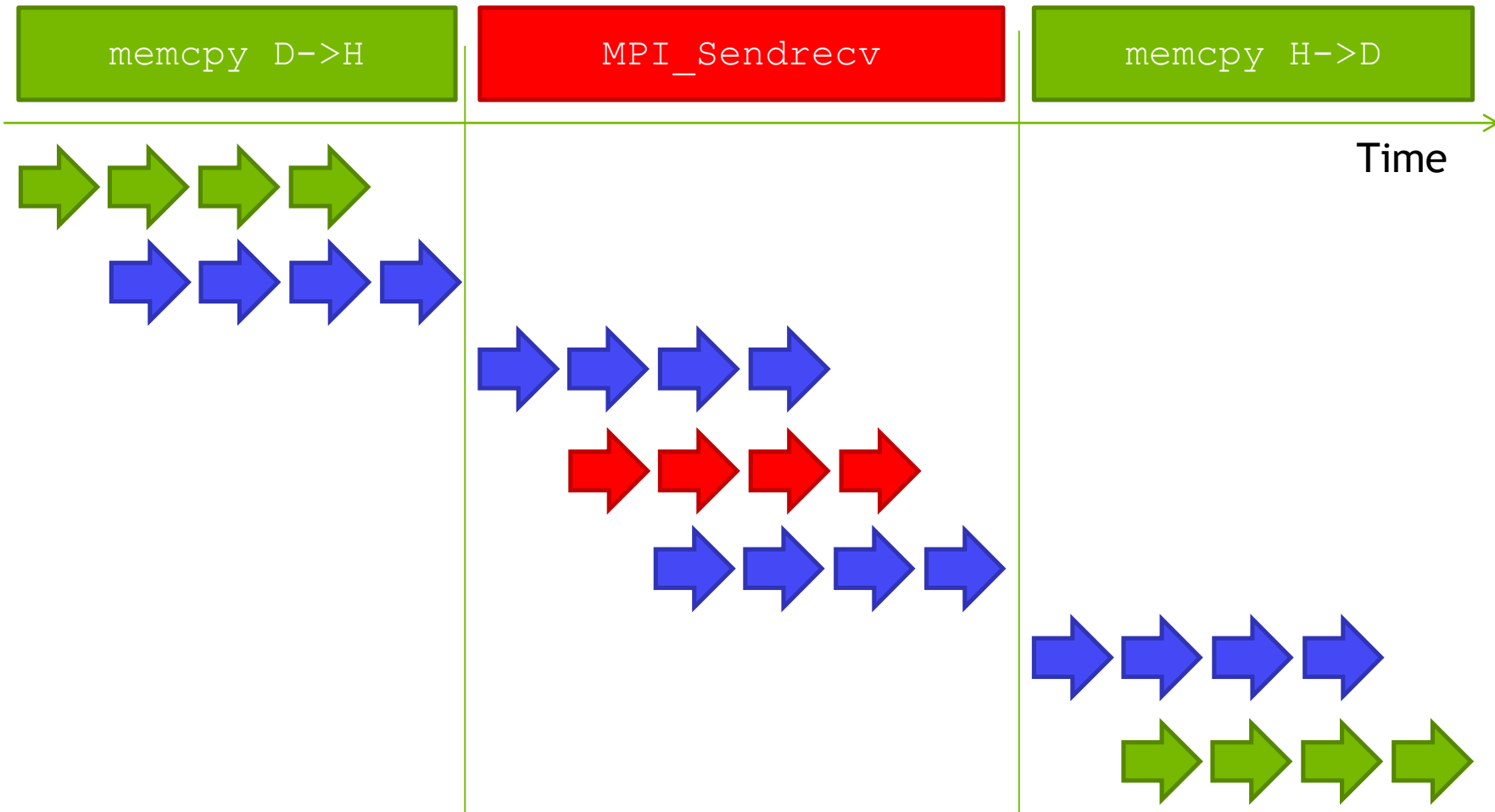
CUDA AWARE MPI FOR ON AND OFF NODE TRANSFERS

REGULAR MPI GPU TO REMOTE GPU



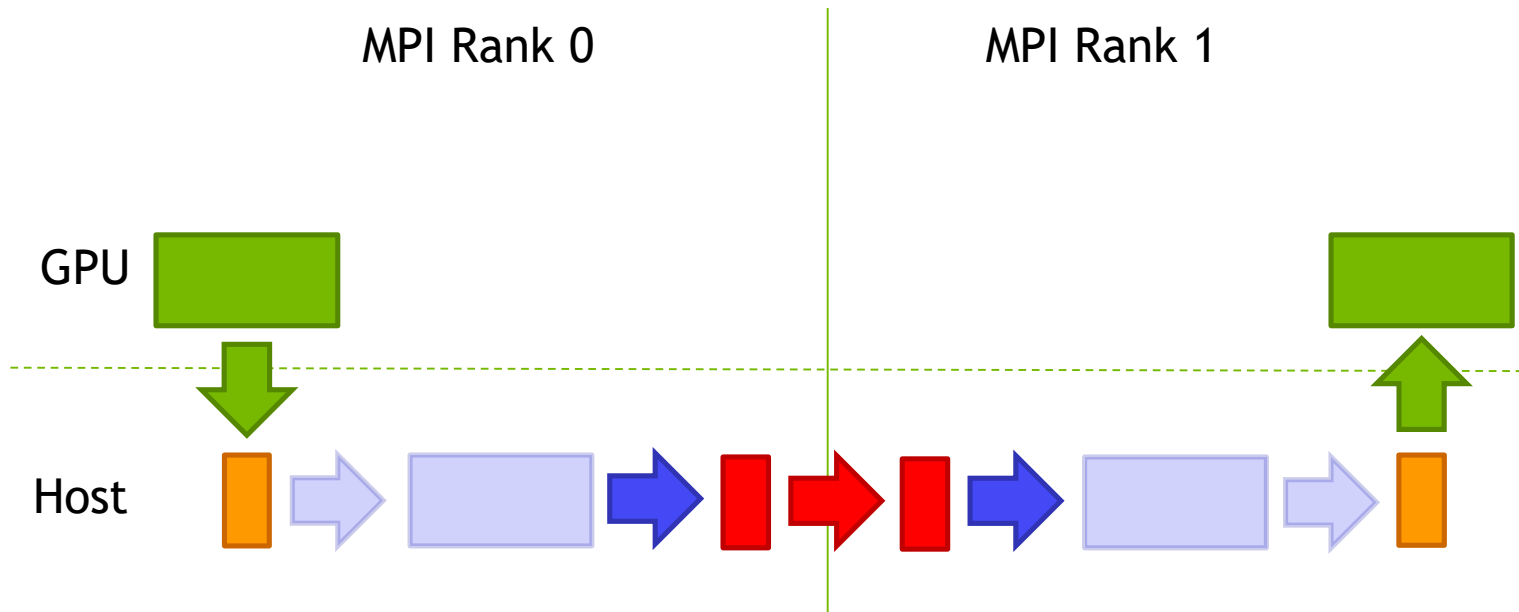
```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);  
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);  
  
MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);  
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```


REGULAR MPI GPU TO REMOTE GPU



MPI GPU TO REMOTE GPU

without GPUDirect

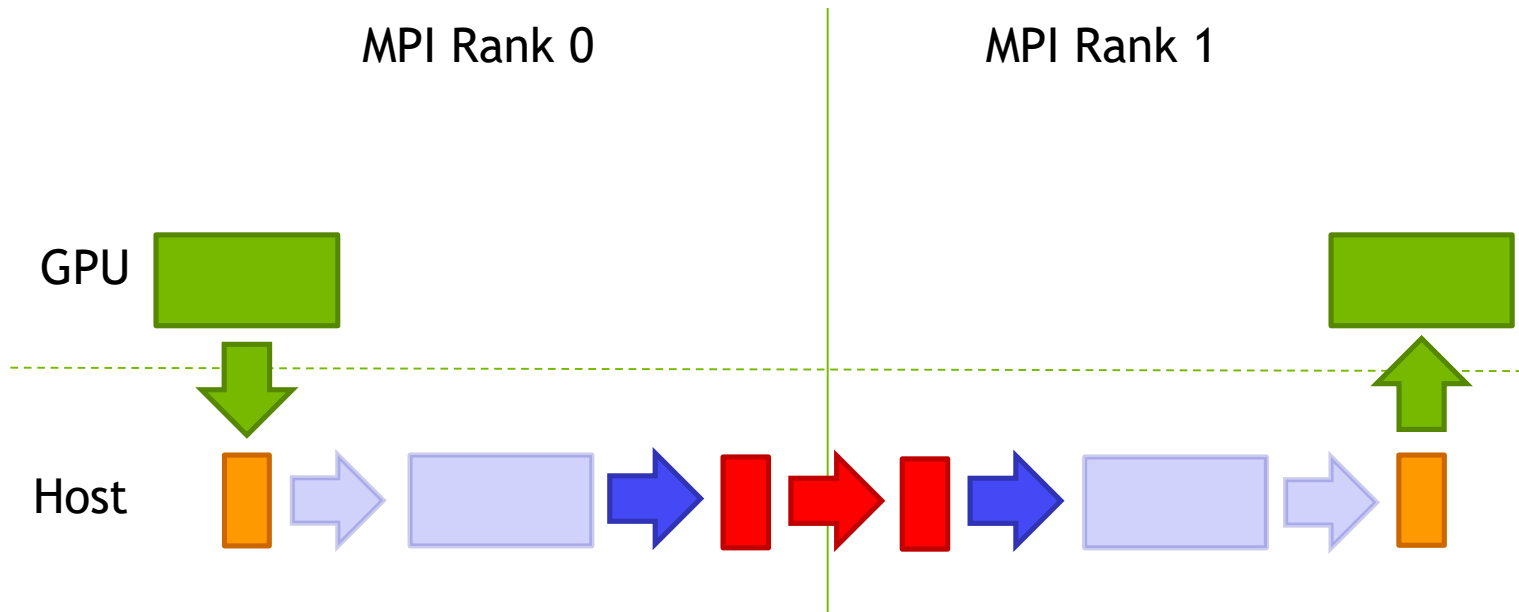


```
MPI_Send(s_buf_d, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
```

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

MPI GPU TO REMOTE GPU

without GPUDirect

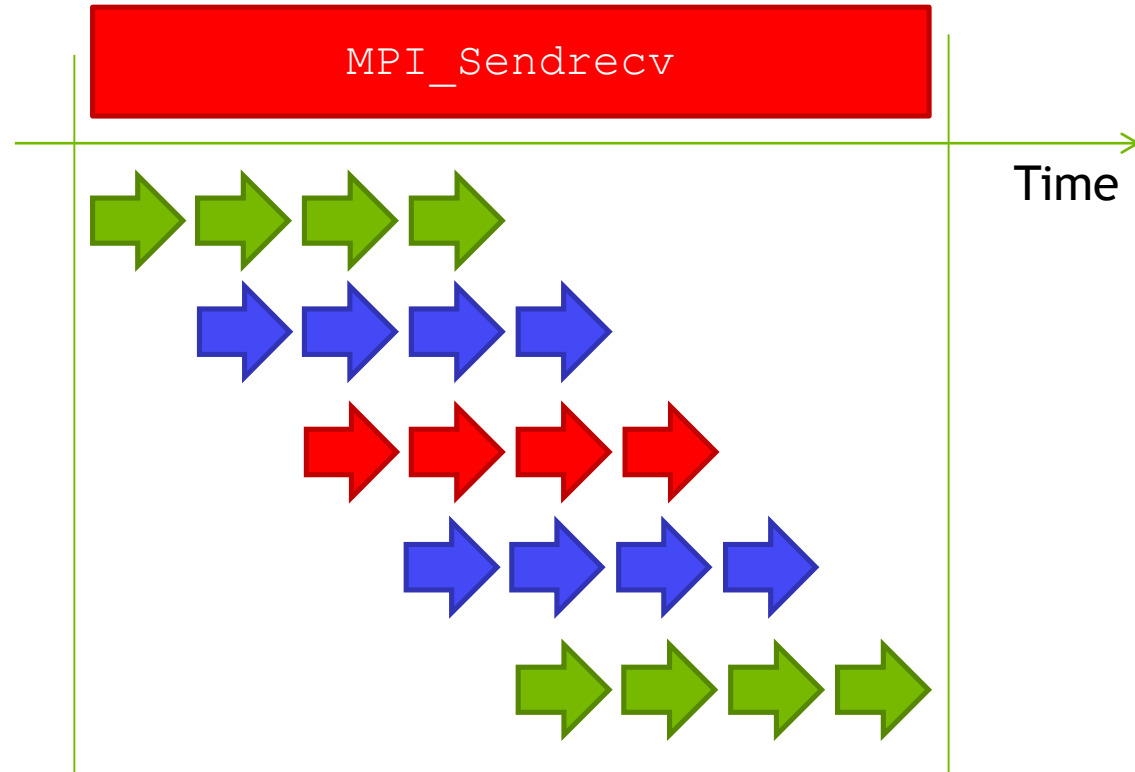


```
#pragma acc host_data use_device (s_buf, r_buf)
MPI_Send(s_buf, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

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

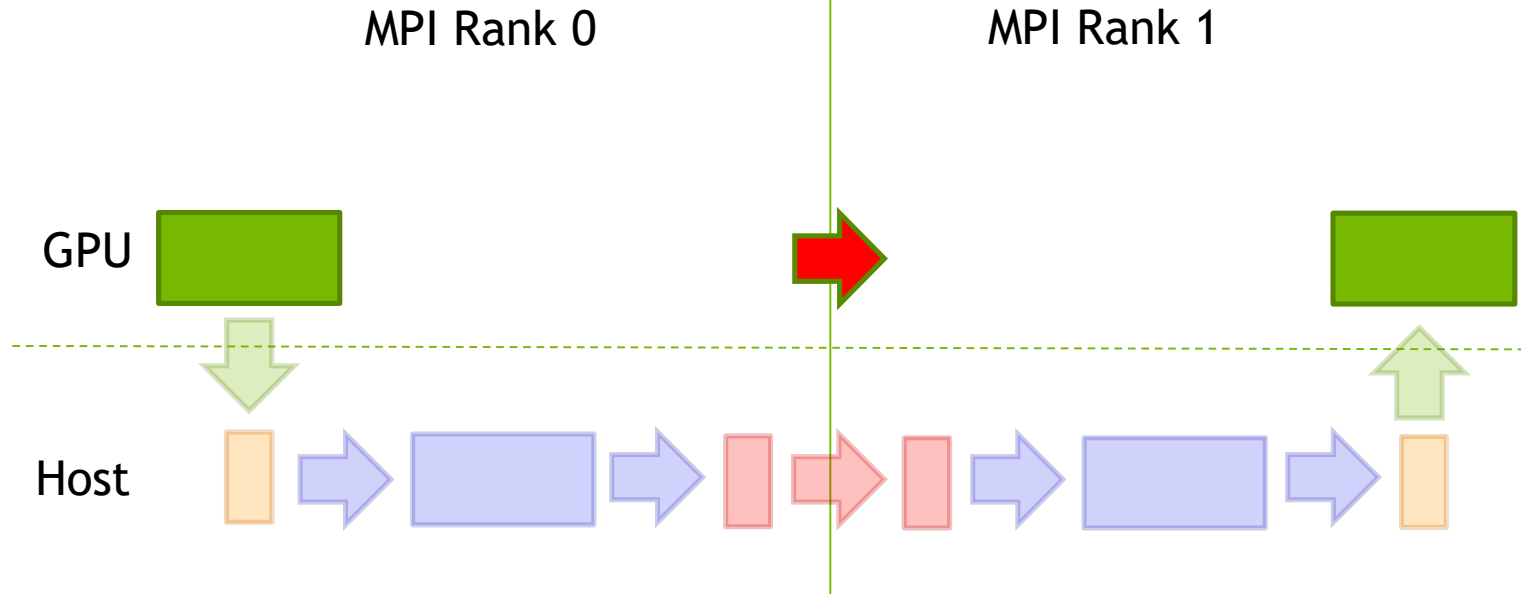
MPI GPU TO REMOTE GPU

without GPUDirect



MPI GPU TO REMOTE GPU

Support for RDMA

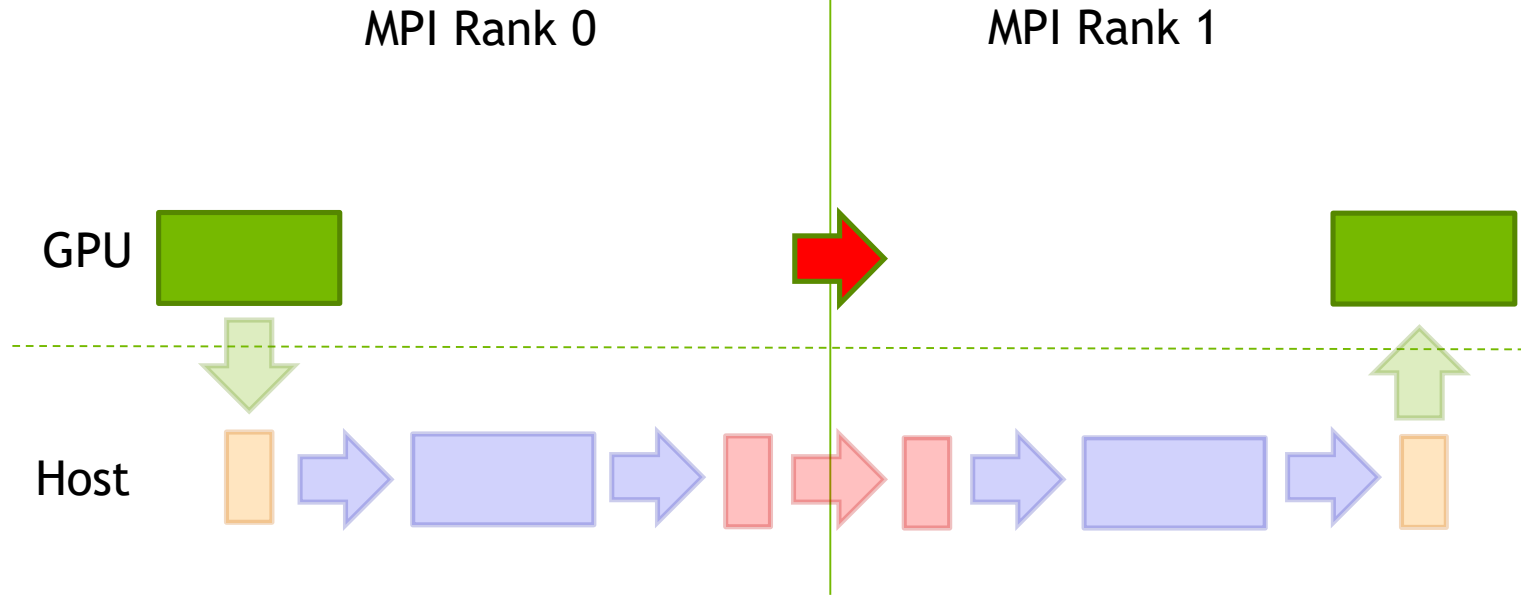


```
MPI_Send(s_buf_d, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
```

```
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**

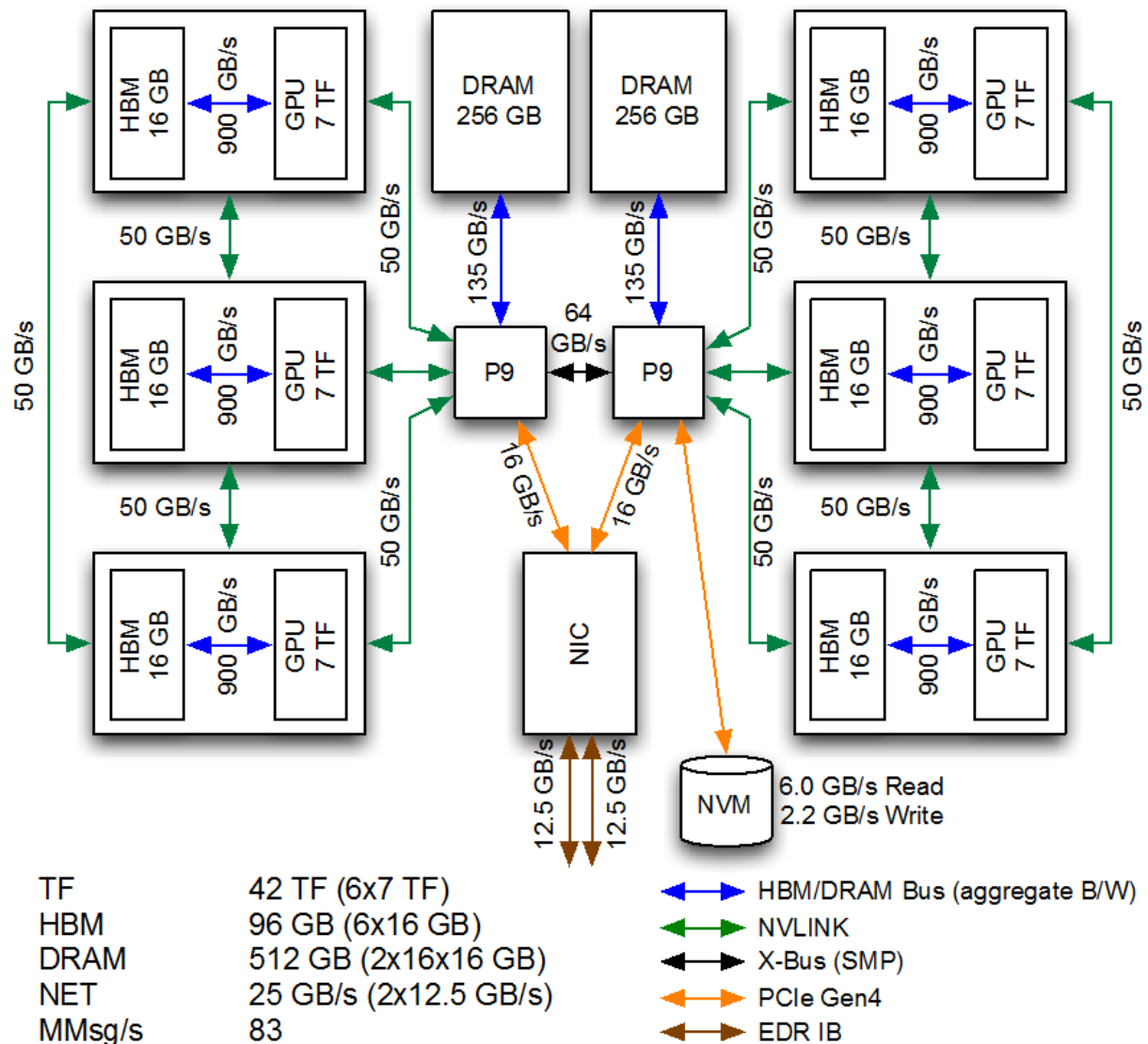
UNDER THE HOOD

Summit has fat nodes!

Many connections

Many devices

Many stacks



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

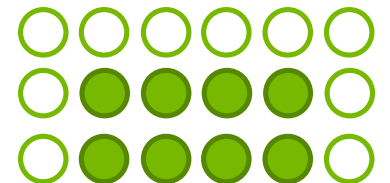
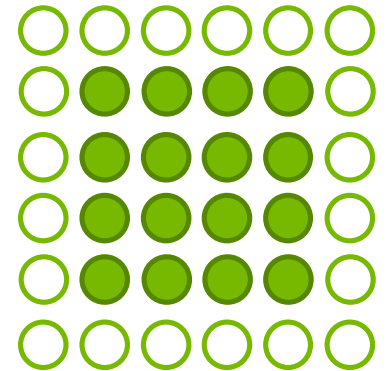
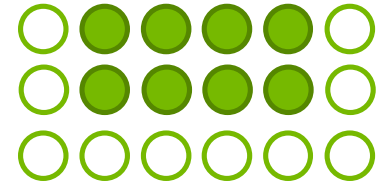
Enable P2P

```
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

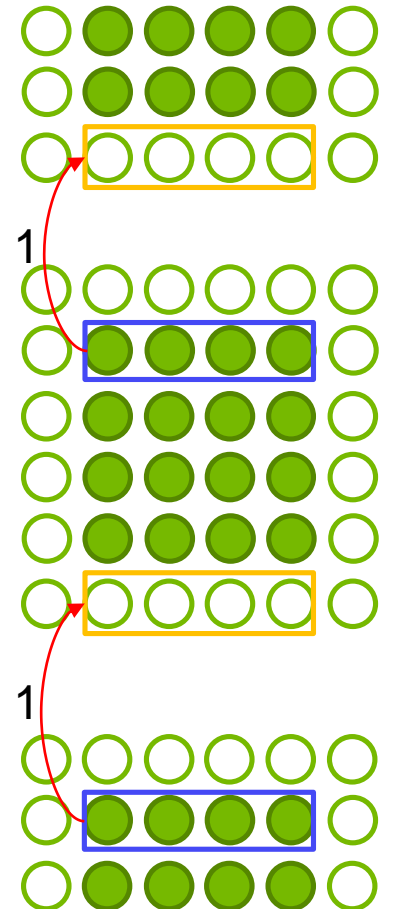
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



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), ...);
```

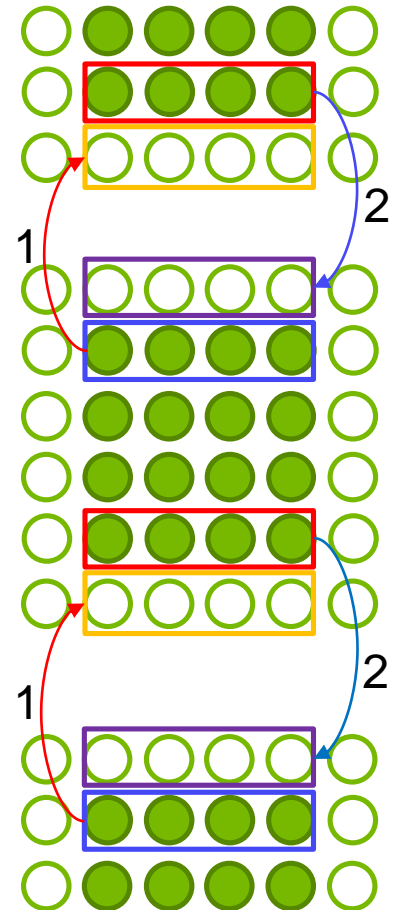


EXAMPLE JACOBI

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

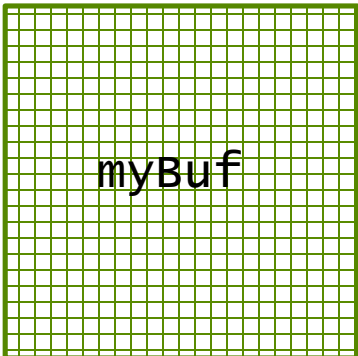
Enable CUDA Inter-Process Communication (IPC)!

```
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)!

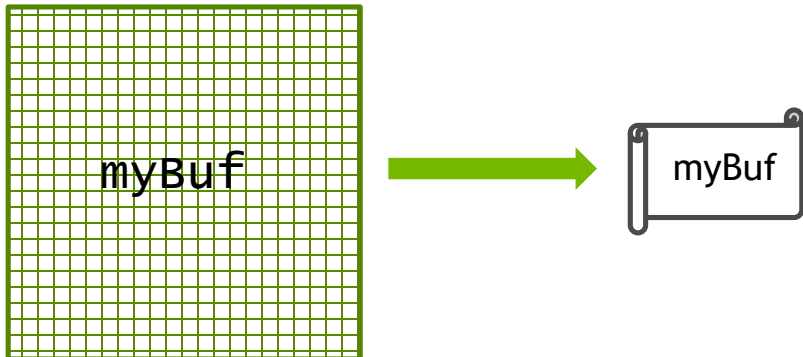
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

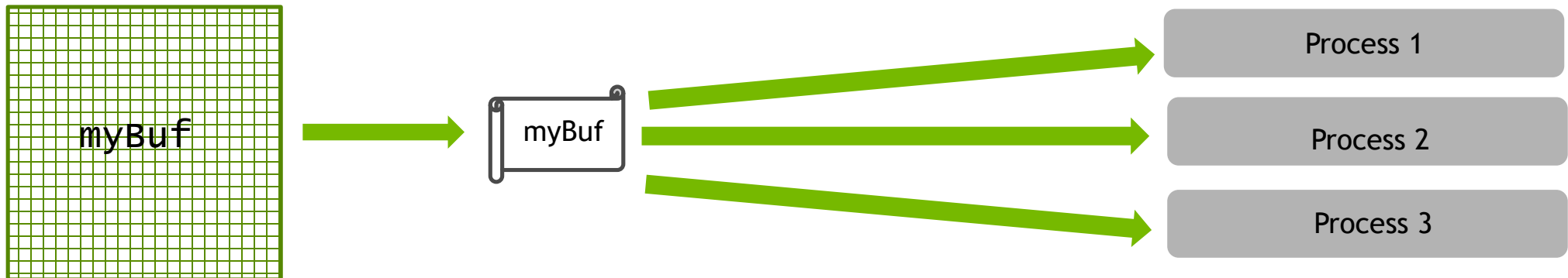
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Inter-Process Communication (IPC)!

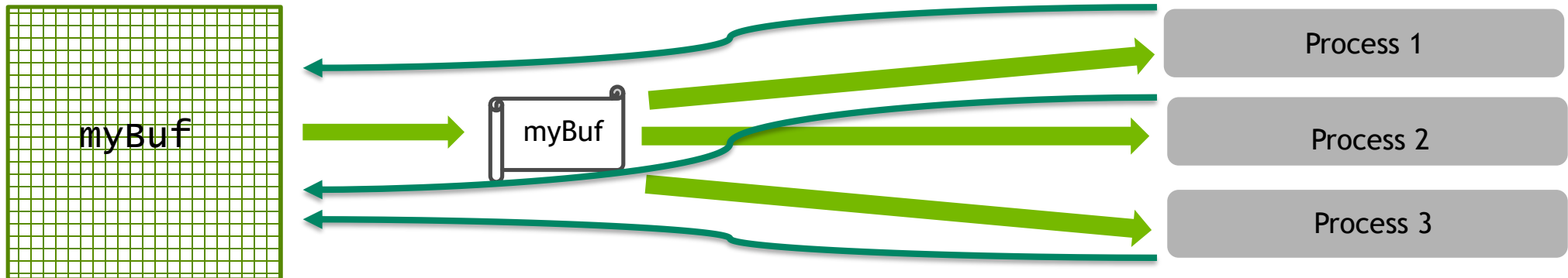
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



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);
```

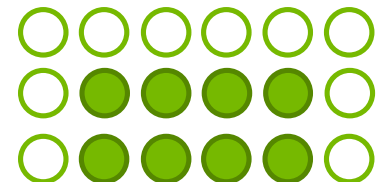
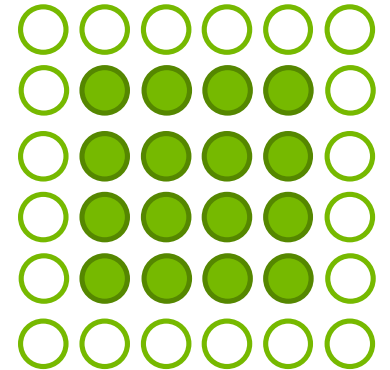
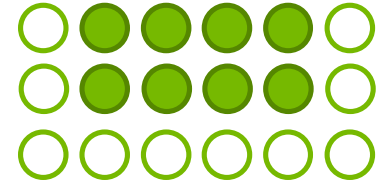


EXAMPLE JACOBI

Top/Bottom Halo

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

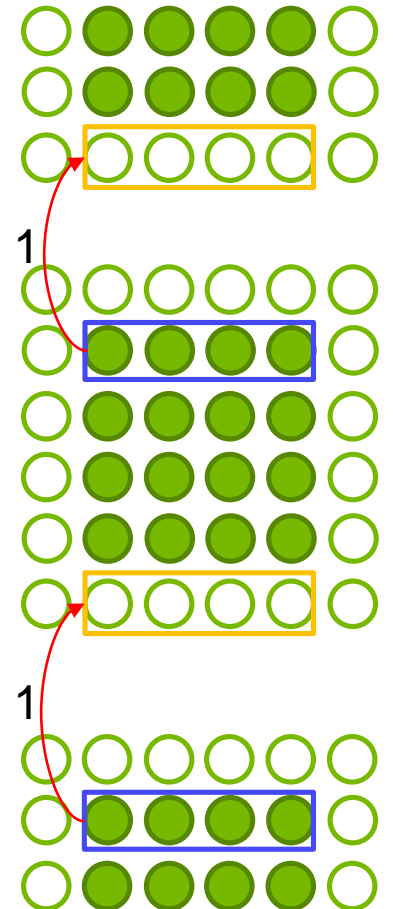
cudaMemcpyAsync(
    a_new[top]+(iy_end[top]*nx),
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

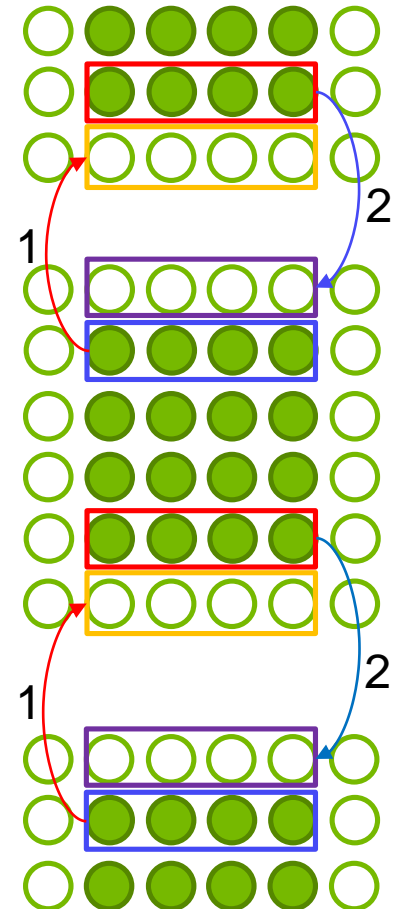
```
cudaIpcOpenMemHandle((void **) &a_new[top], topIpc,  
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();  
  
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



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), ...);
```



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!

An abstract graphic featuring a network of glowing green and blue nodes connected by thin, semi-transparent lines. The nodes are scattered across the frame, with some appearing as bright points and others as larger, softer glows. The background is a deep, dark blue/black, creating a high-contrast, futuristic aesthetic.

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]
```

The background features a complex network of thin, glowing green and blue lines that intersect to form various geometric shapes. Scattered throughout this network are several bright, circular nodes in shades of green and blue. The overall aesthetic is futuristic and digital, set against a solid black background.

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!



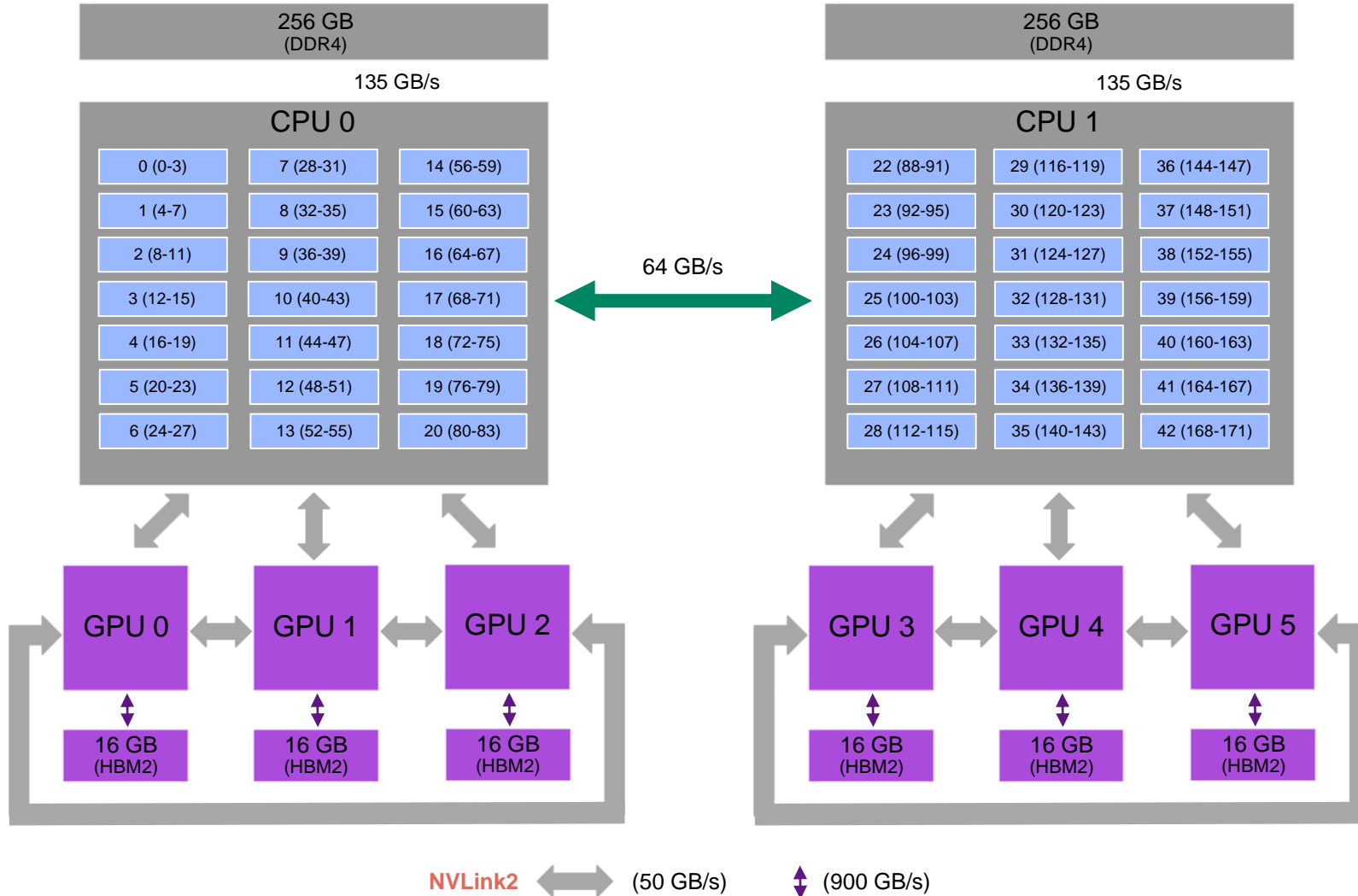
nVIDIA®

The background of the slide features a complex network of glowing green lines and nodes. The nodes are small, bright green circles of varying sizes, and the lines are thin, light green strands that crisscross the dark blue and black background, creating a sense of interconnectedness and data flow. The overall aesthetic is futuristic and technological.

SUMMIT NODE OVERVIEW

SUMMIT NODE

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



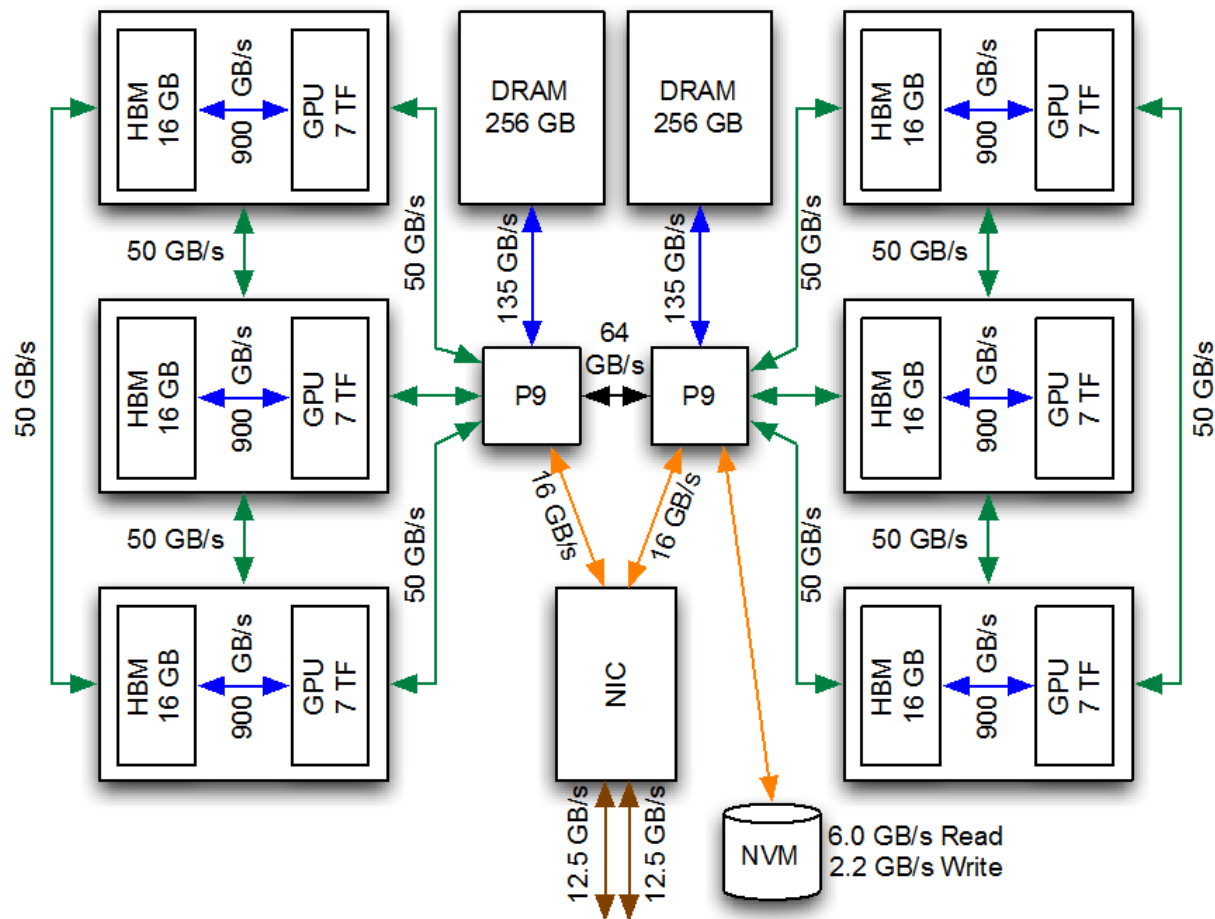
UNDER THE HOOD

Summit has fat nodes!

Many connections

Many devices

Many stacks



TF	42 TF (6x7 TF)	↔	HBM/DRAM Bus (aggregate B/W)
HBM	96 GB (6x16 GB)	↔	NVLINK
DRAM	512 GB (2x16x16 GB)	↔	X-Bus (SMP)
NET	25 GB/s (2x12.5 GB/s)	↔	PCIe Gen4
MMsg/s	83	↔	EDR IB

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