



PROGRAMMING MULTI-GPU NODES

Steve Abbott & Jeff Larkin, November 2018

AGENDA

Summit Node Overview

Multi-GPU Programming Models

Multi-GPU Programming with OpenACC and CUDA

GPUDirect, CUDA Aware MPI, and CUDA IPC

SpectrumMPI & Jsruntime Tips and Tricks

The background of the slide features an abstract network diagram. It consists of numerous small, bright green circular nodes scattered across a dark, almost black, background. These nodes are interconnected by a dense web of thin, light green lines, creating a complex, web-like structure that suggests a global network or data flow. The lines vary in opacity and thickness, adding depth to the visual. The overall aesthetic is high-tech and digital.

SUMMIT NODE OVERVIEW

SUMMIT NODE

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



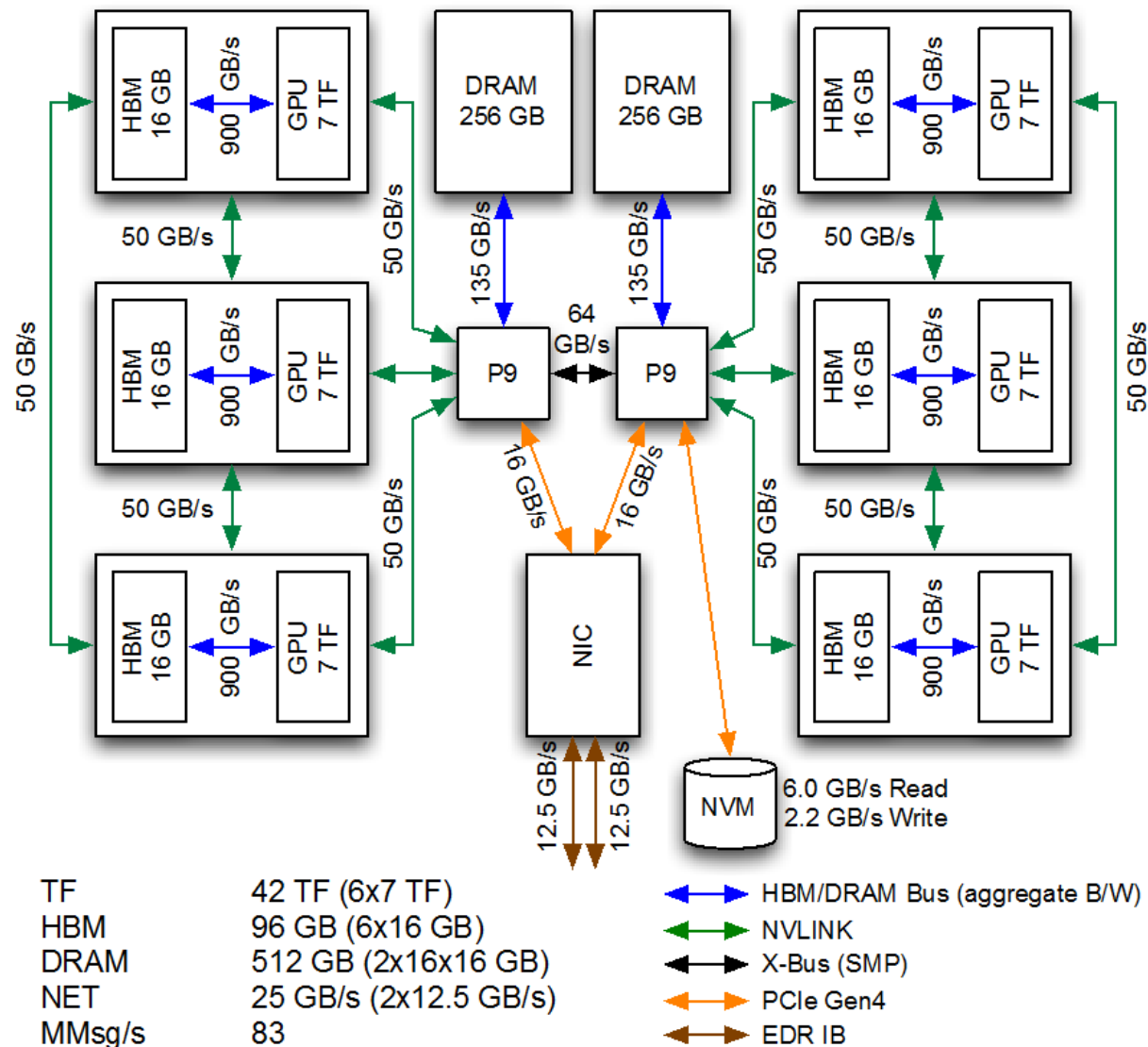
UNDER THE HOOD

Summit has fat nodes!

Many connections

Many devices

Many stacks



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

The background of the slide features an abstract network diagram. It consists of numerous small, bright green circular nodes scattered across a dark, almost black, background. These nodes are interconnected by a dense web of thin, light green lines, creating a complex, web-like structure that suggests a distributed system or a network topology. The lines vary in opacity and brightness, giving a sense of depth and connectivity. The overall aesthetic is high-tech and digital.

MULTI-GPU PROGRAMMING MODELS

MULTI-GPU PROGRAMMING MODELS

Single Thread, Multiple GPUs

- A single thread will change devices as-needed to send data and kernels to different GPUs

Multiple Threads, Multiple GPUs

- Using OpenMP, Pthreads, or similar, each thread can manage its own GPU

Multiple Ranks, Single GPU

- Each rank acts as-if there's just 1 GPU, but multiple ranks per node use all GPUs

Multiple Ranks, Multiple GPUs

- Each rank manages multiple GPUs, multiple ranks/node. Gets complicated quickly!

MULTI-GPU PROGRAMMING MODELS

Trade-offs Between Approaches

- Conceptually Simple
- Requires additional loops
- CPU can become a bottleneck
- Remaining CPU cores often underutilized

Single Thread, Multiple GPUs

- Conceptually Very Simple
- Set and forget the device numbers
- Relies on external Threading API
- Can see improved utilization
- Watch affinity

Multiple Threads, Multiple GPUs

- Little to no code changes required
- Re-uses existing domain decomposition
- Probably already using MPI
- Watch affinity

Multiple Ranks, Single GPU

- Easily share data between peer devices
- Coordinating between GPUs extremely tricky

Multiple Ranks, Multiple GPUs

MULTI-DEVICE CUDA

CUDA by default exposes all devices, numbered 0 - (N-1), if devices are not all the same, it will reorder the “best” to device 0.

Each device has its own pool of streams.

If you do nothing, *all* work will go to Device #0.

Developer must change the current device explicitly

MULTI-DEVICE OPENACC

OpenACC presents devices numbered 0 - (N-1) for each device type available.

The order of the devices comes from the runtime, almost certainly the same as CUDA

By default all data and work go to the *current device*

Developers must change the current device and maybe the current device type using an API

MULTI-DEVICE OPENMP

OpenMP devices numbered 0 - (N-1) for *ALL* devices on the machine, including the host.

The order is determined by the runtime, but devices of the same type are contiguous.

To change the device for data and compute a clause is added to directives.

Device API routines include a devicenum

The background of the slide features a complex, abstract network diagram. It consists of numerous small, bright green circular nodes scattered across a dark, almost black, background. These nodes are interconnected by a dense web of thin, light green lines, creating a sense of connectivity and data flow. The lines vary in length and orientation, some forming straight paths while others create more intricate, overlapping patterns. The overall effect is reminiscent of a digital network or a complex system architecture.

MULTI-GPU PROGRAMMING WITH OPENACC AND CUDA

MULTI-GPU W/ CUDA AND OPENACC

The CUDA and OpenACC approaches are sufficiently similar, that I will demonstrate using OpenACC.

Decoder Ring:

OpenACC	CUDA
acc_get_device_type()	N/A
acc_set_device_type()	N/A
acc_set_device_num()	cudaSetDevice()
acc_get_device_num()	cudaGetDevice()
acc_get_num_devices()	cudaGetDeviceCount()

Multi-Device Pipeline

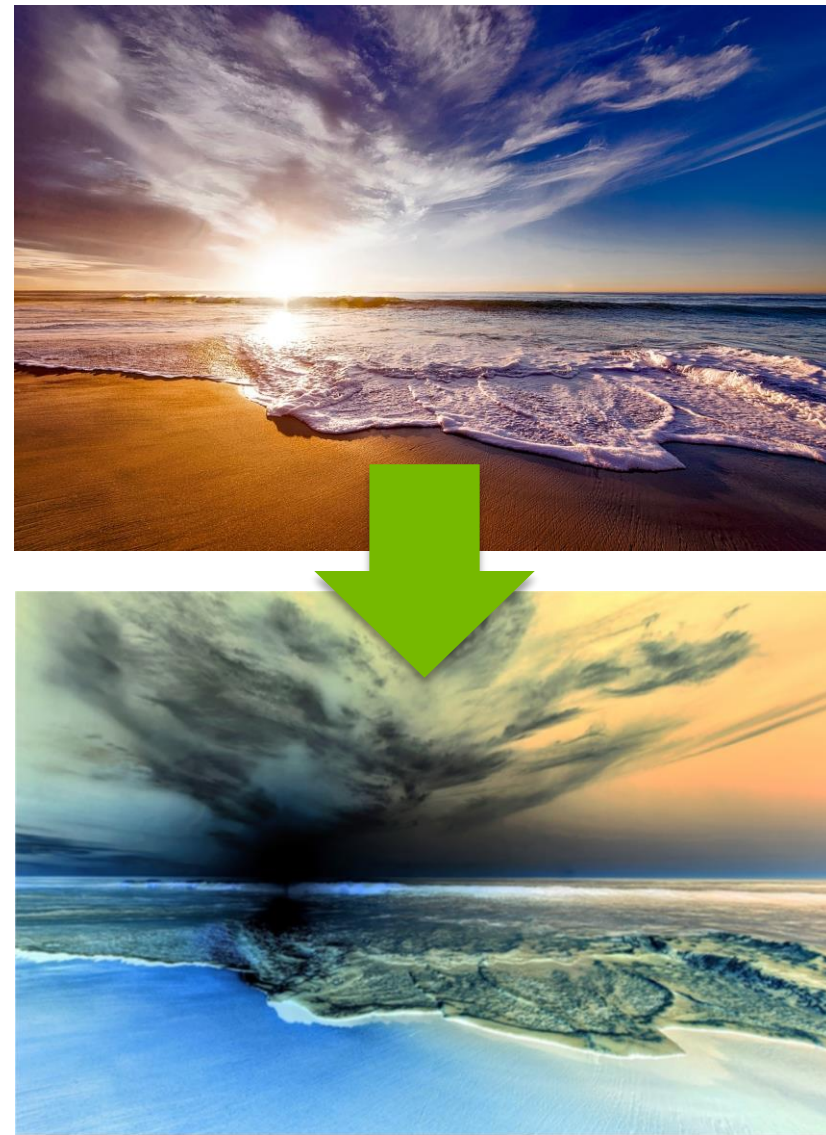
A Case Study

We'll use a simple image filter to demonstrate these techniques.

No inter-GPU communication required

Pipelining: Breaking a large operation into smaller parts so that independent operations can overlap.

Since each part is independent, they can easily be run on different devices. We will extend the filter to run on more than one device.

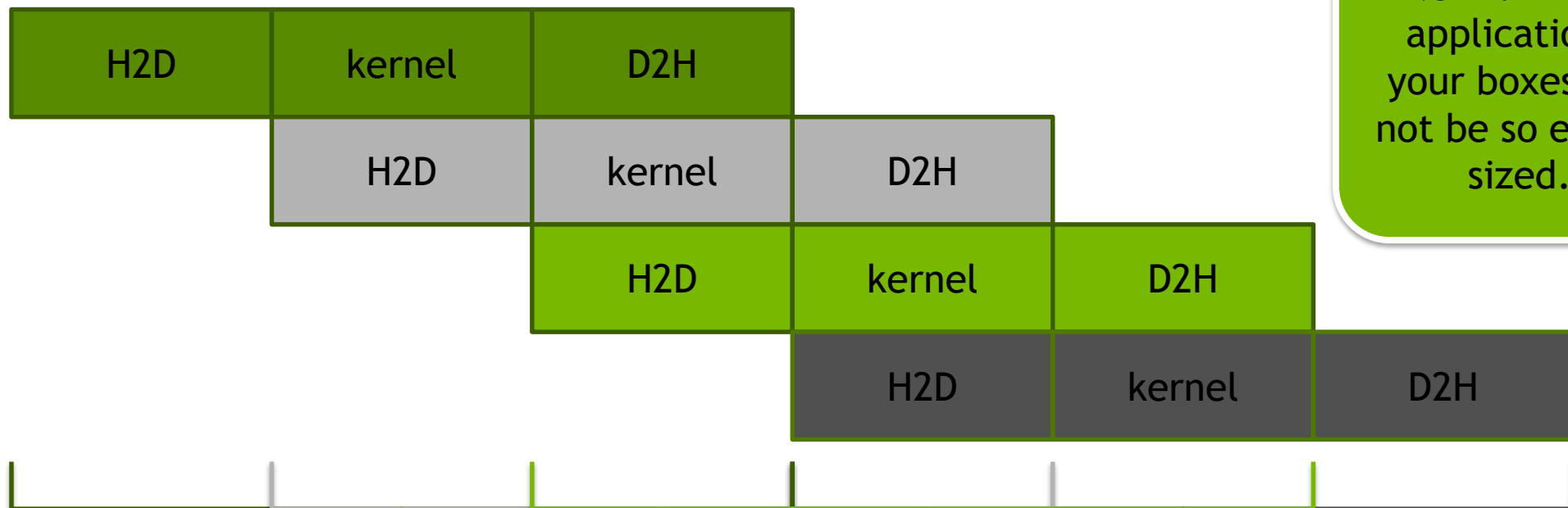


Pipelining in a Nutshell



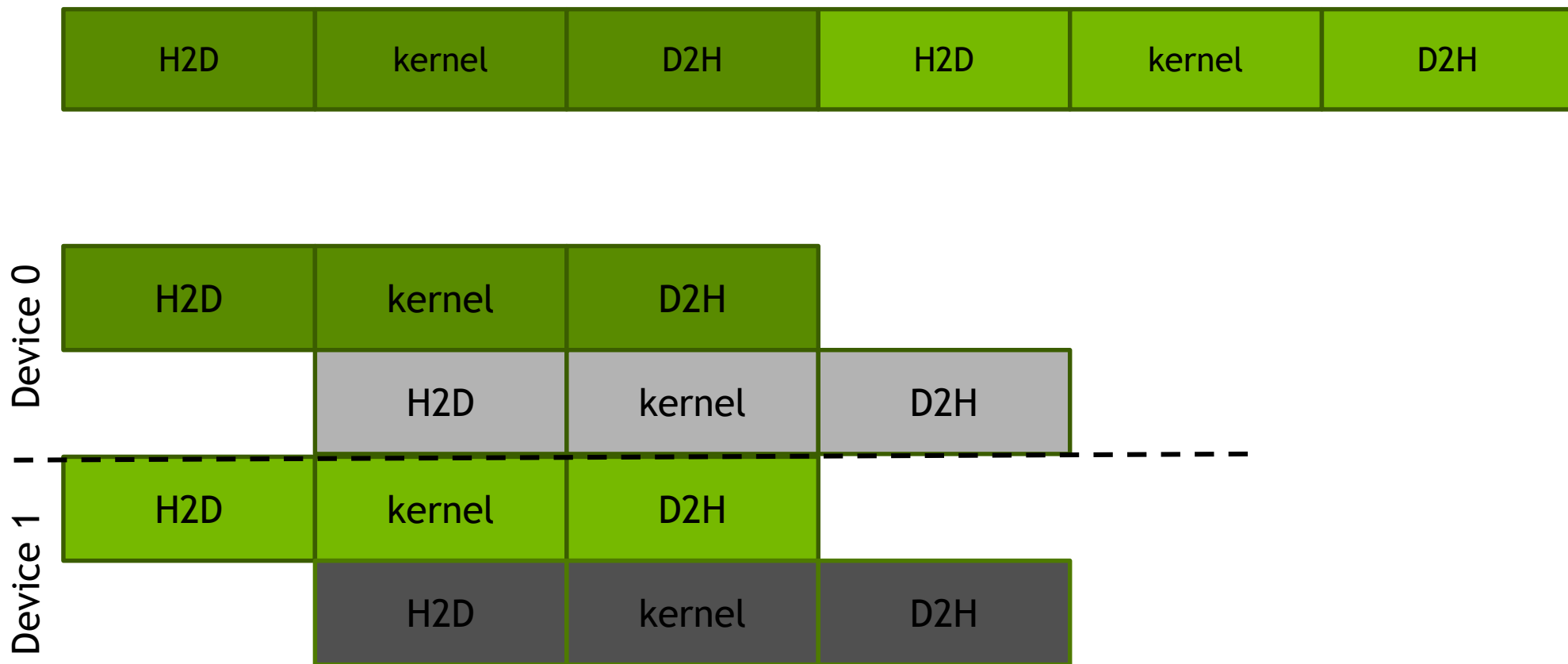
Two Independent Operations Serialized

NOTE: In real applications, your boxes will not be so evenly sized.



Overlapping Copying and Computation

Multi-device Pipelining in a Nutshell



Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
    copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
    long starty = MAX(0,blocky * blocksize - filtersize/2);
    long endy    = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3)
    starty = blocky * blocksize;
    endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3)
    for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
        <filter code ommitted>
        out[y * step + x * ch]      = 255 - (scale * blue);
        out[y * step + x * ch + 1 ] = 255 - (scale * green);
        out[y * step + x * ch + 2 ] = 255 - (scale * red);
    }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3)
}
#pragma acc wait
}
```

Cycle between 3 async
queues by blocks.

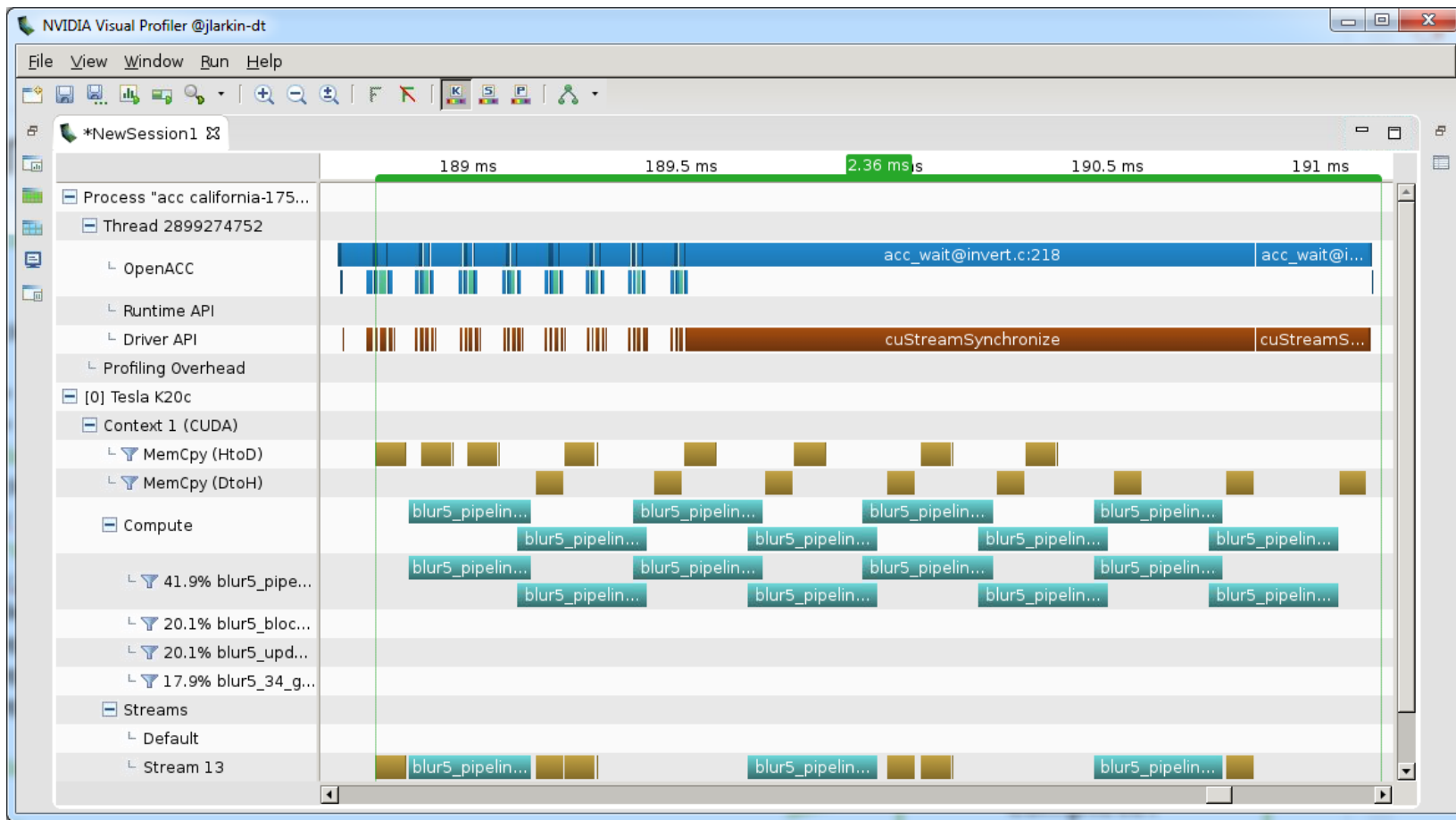
Pipelined Code

```
#pragma acc data create(imgData[w*h*ch],out[w*h*ch])
    copyin(filter)
{
for ( long blocky = 0; blocky < nblocks; blocky++)
{
    long starty = MAX(0,blocky * blocksize - filtersize/2);
    long endy    = MIN(h,starty + blocksize + filtersize/2);
#pragma acc update device(imgData[starty*step:blocksize*step]) async(block%3)
    starty = blocky * blocksize;
    endy = starty + blocksize;
#pragma acc parallel loop collapse(2) gang vector async(block%3)
    for (y=starty; y<endy; y++) for ( x=0; x<w; x++ ) {
        <filter code ommitted>
        out[y * step + x * ch]      = 255 - (scale * blue);
        out[y * step + x * ch + 1 ] = 255 - (scale * green);
        out[y * step + x * ch + 2 ] = 255 - (scale * red);
    }
#pragma acc update self(out[starty*step:blocksize*step]) async(block%3)
}
#pragma acc wait
}
```

Cycle between 3 async queues by blocks.

Wait for all blocks to complete.

NVPROF Timeline of Pipeline



Extending to multiple devices

Create 1 OpenMP thread on the CPU per-device. This is not strictly necessary, but simplifies the code.

Within each thread, set the device number.

Divide the blocks as evenly as possible among the CPU threads.

Multi-GPU Pipelined Code (OpenMP)

```
#pragma omp parallel num_threads(acc_get_num_devices(acc_device_default))
{
    acc_set_device_num(omp_get_thread_num(),acc_device_default);
    int queue = 1;
    #pragma acc data create(imgData[w*h*ch],out[w*h*ch])
    {
        #pragma omp for schedule(static)
        for ( long blocky = 0; blocky < nblocks; blocky++) {
            // For data copies we need to include the ghost zones for the filter
            long starty = MAX(0,blocky * blocksize - filtersize/2);
            long endy    = MIN(h,starty + blocksize + filtersize/2);
            #pragma acc update device(imgData[starty*step:(endy-starty)*step]) async(queue)
            starty = blocky * blocksize;
            endy = starty + blocksize;
            #pragma acc parallel loop collapse(2) gang vector async(queue)
            for ( long y = starty; y < endy; y++ ) { for ( long x = 0; x < w; x++ ) {
                <filter code removed for space>
            }}
            #pragma acc update self(out[starty*step:blocksize*step]) async(queue)
            queue = (queue%3)+1;
        }
        #pragma acc wait
    }
}
```

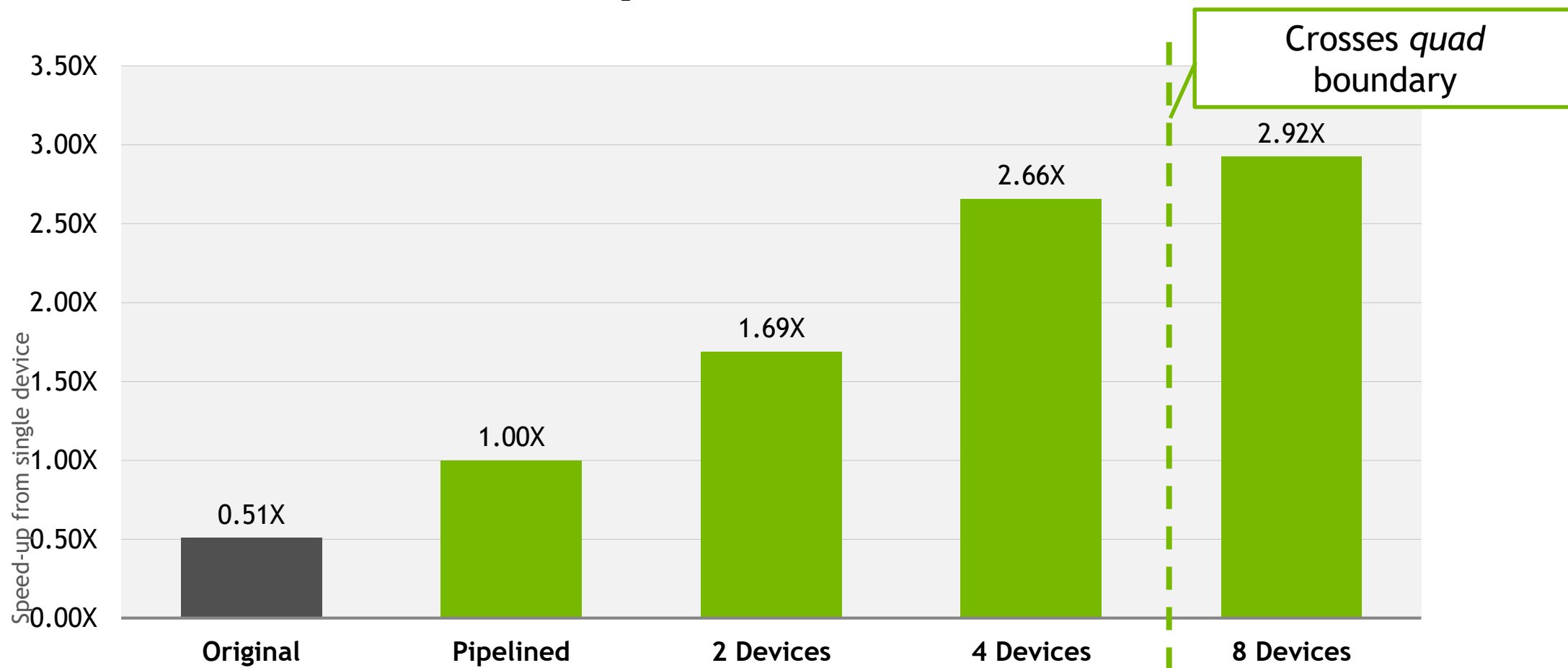
Spawn 1 thread per device.

Set the device number per-thread.

Divide the work among threads.

Wait for each device in its thread.

Multi-GPU Pipelined Performance



OpenACC with MPI

Domain decomposition is performed using MPI ranks

Each rank should set its own device

- Maybe `acc_set_device_num`
- Maybe handled by environment variable (`CUDA_VISIBLE_DEVICES`)

GPU affinity can be handled by standard MPI task placement

Multiple MPI Ranks/GPU (using MPS) can work in place of OpenACC work queues/CUDA Streams

Setting a device by local rank

```
// This is not portable to other MPI libraries
char *comm_local_rank = getenv("OMPI_COMM_WORLD_LOCAL_RANK");
int local_rank = atoi(comm_local_rank);
char *comm_local_size = getenv("OMPI_COMM_WORLD_LOCAL_SIZE");
int local_size = atoi(comm_local_size);
int num_devices = acc_get_num_devices(acc_device_nvidia);
#pragma acc set device_num(local_rank%num_devices) \
               device_type(acc_device_nvidia)
```

Determine a unique ID
for each rank on the
same node.

Use this unique ID to
select a device per
rank.

You may also try using `MPI_Comm_split_type()` using `MPI_COMM_TYPE_SHARED` or `OMPI_COMM_TYPE_SOCKET`.

In the end, you need to understand how `jsrun/mpiirun` is placing your ranks.

MPI Image Filter (pseudocode)

```
if (rank == 0 ) read_image();  
// Distribute the image to all ranks  
MPI_Scatterv(image);
```

Decompose image
across processes
(ranks)

```
MPI_Barrier(); // Ensures all ranks line up for timing  
omp_get_wtime();  
blur_filter(); // Contains OpenACC filter  
MPI_Barrier(); // Ensures all ranks complete before timing  
omp_get_wtime();
```

Receive final parts
from all ranks.

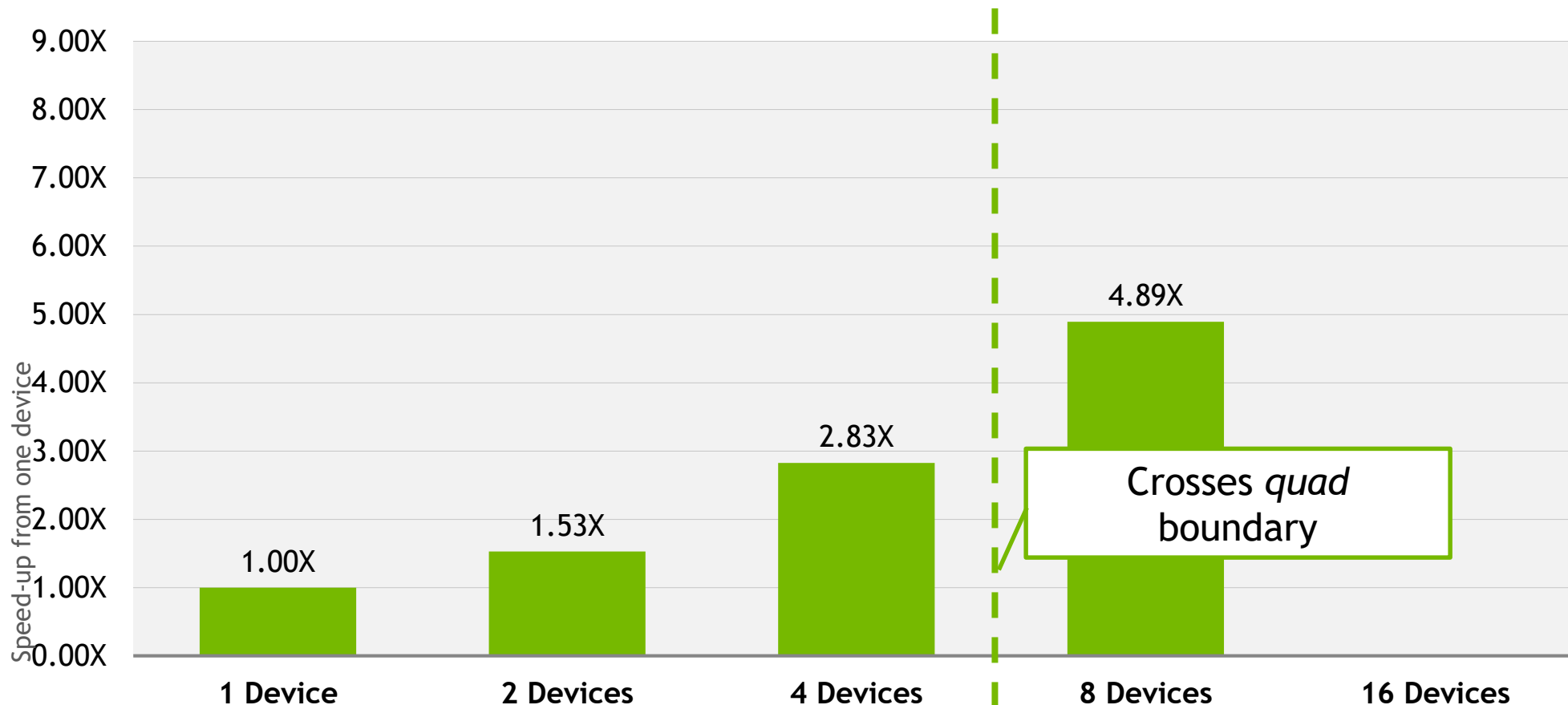
```
MPI_Gatherv(out);  
if (rank == 0 ) write_image();
```

```
$ jsrun -n 6 -a 1 -c 1 -g 1 ...
```

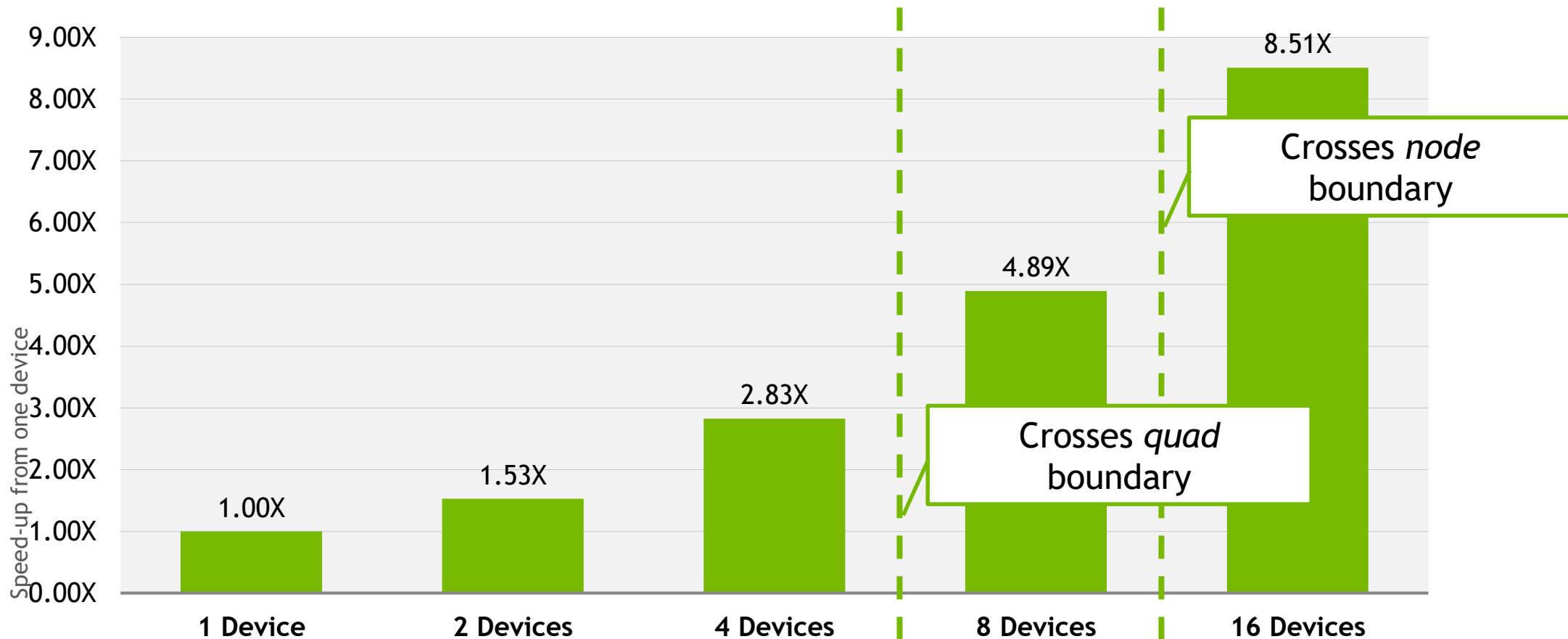
Launch with good
GPU/process affinity

There's a variety of ways to do MPI decomposition, this is what I used for this particular example.

Multi-GPU Pipelined Performance (MPI)



Multi-GPU Pipelined Performance (MPI)



MULTI-DEVICE CUDA

Same Pattern, Different API

```
#pragma omp parallel
{
    cudaSetDevice(idx);
    #pragma omp for
    for ( int b=0; b < nblocks; b++)
    {
        cudaMemcpyAsync(..., streams[b%3]);
        blur_kernel <<<griddim, blockdim,
                        0, streams[b%3]>>>();
        cudaMemcpyAsync(..., streams[b%3]);
    }

    cudaDeviceSynchronize();
}
```

```
MPI_Comm_rank(local_comm, &local_rank);

cudaSetDevice(local_rank);

for ( int b=0; b < nblocks; b++)
{
    cudaMemcpyAsync(..., streams[b%3]);
    blur_kernel <<<griddim, blockdim,
                0, streams[b%3]>>>();
    cudaMemcpyAsync(..., streams[b%3]);
}

cudaDeviceSynchronize();
```


MULTI-DEVICE OPENMP 4.5

Same Pattern, Different API

```
#pragma omp parallel num_threads(num_dev)
{
    #pragma omp for
    for ( int b=0; b < nblocks; b++)
    {
        #pragma omp target update map(to:...) \
            device(dev) depend(inout:A) \
            nowait
        #pragma omp target teams distribute \
            parallel for simd device(dev) \
            depend(inout:A)
        for(...) { ... }
        #pragma omp target update map(from:...) \
            device(dev) depend(inout:A) \
            nowait
    }
    #pragma omp taskwait
}
```

```
MPI_Comm_rank(local_comm, &local_rank);
int dev = local_rank;

for ( int b=0; b < nblocks; b++)
{
    #pragma omp target update map(to:...) \
        device(dev) depend(inout:A) \
        nowait
    #pragma omp target teams distribute \
        parallel for simd device(dev) \
        depend(inout:A)
    for(...) { ... }
    #pragma omp target update map(from:...) \
        device(dev) depend(inout:A) \
        nowait
}
#pragma omp taskwait
```

Multi-GPU Approaches

Choosing an approach

Single-Threaded, Multiple-GPUs - Requires additional loops to manage devices, likely undesirable.

Multi-Threaded, Multiple-GPUs - Very convenient set-and-forget the device. Could possibly conflict with existing threading.

Multiple-Ranks, Single-GPU each - Probably the simplest if you already have MPI, the decomposition is done. Must get your MPI placement correct

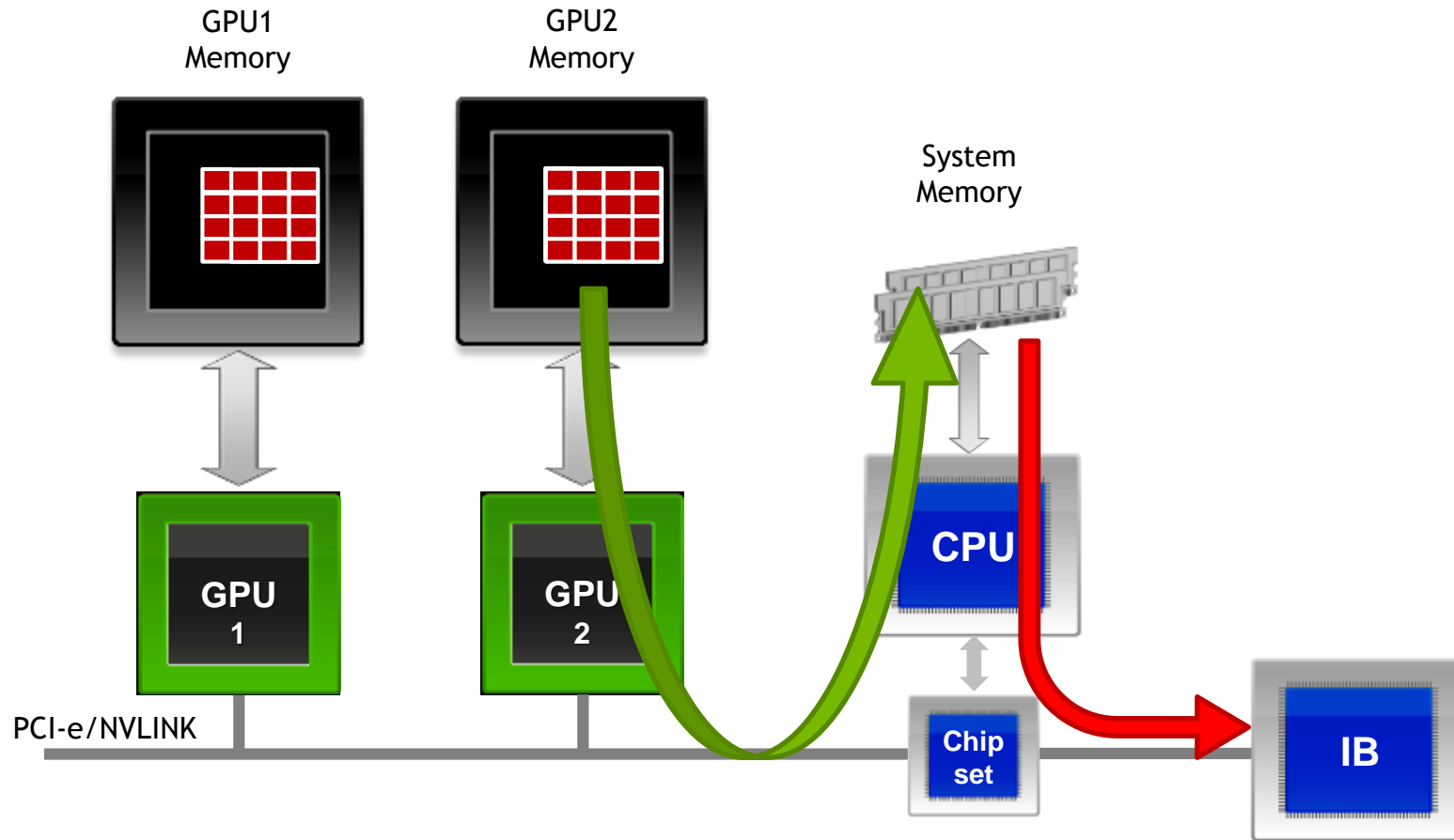
Multiple-Ranks, Multiple-GPUs - Can allow all GPUs to share common data structures. Only do this if you absolutely need to, difficult to get right.



**GPUDIRECT,
CUDA AWARE MPI,
& CUDA IPC**

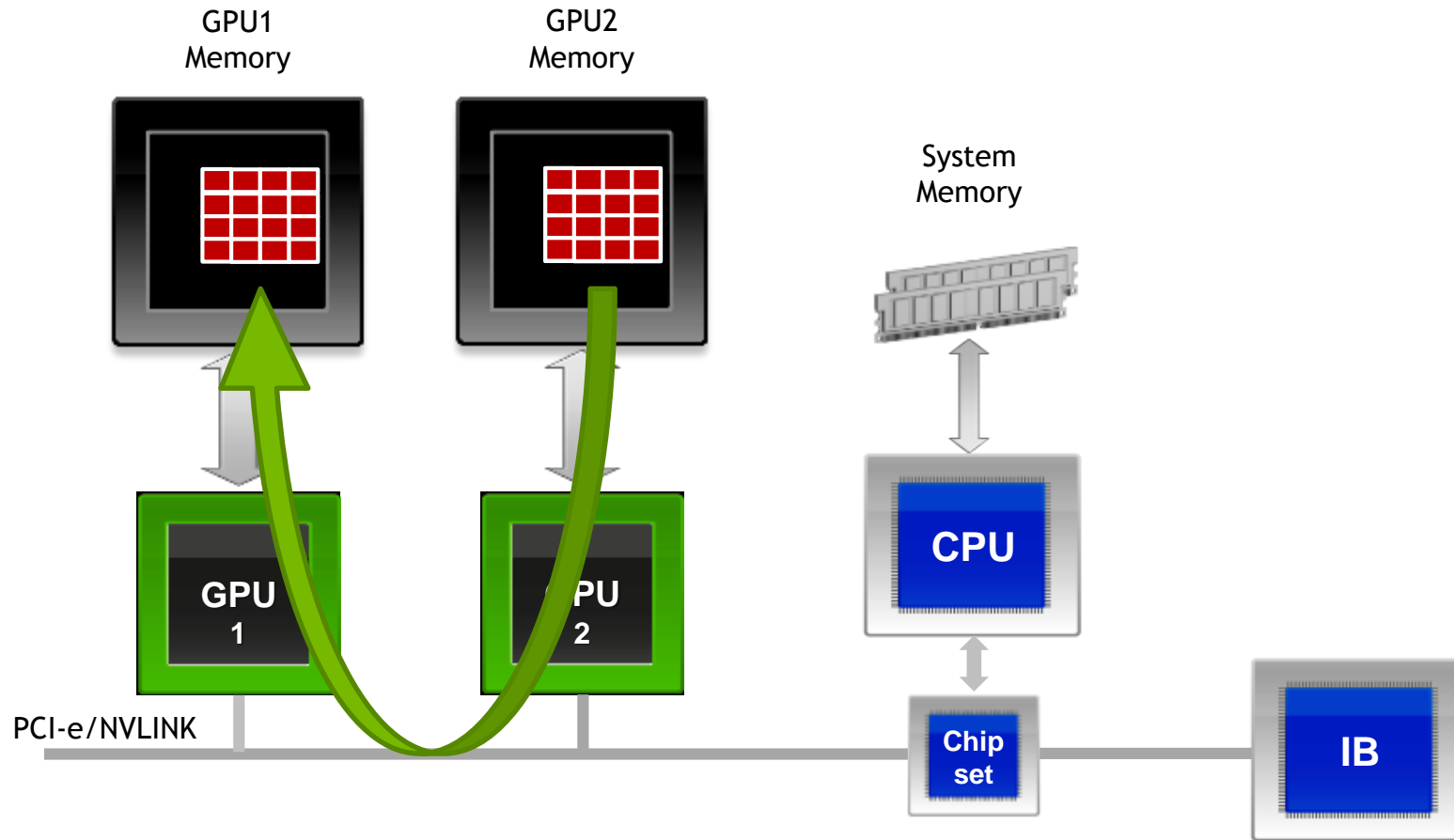
NVIDIA GPUDIRECT™

Accelerated Communication with Network & Storage Devices



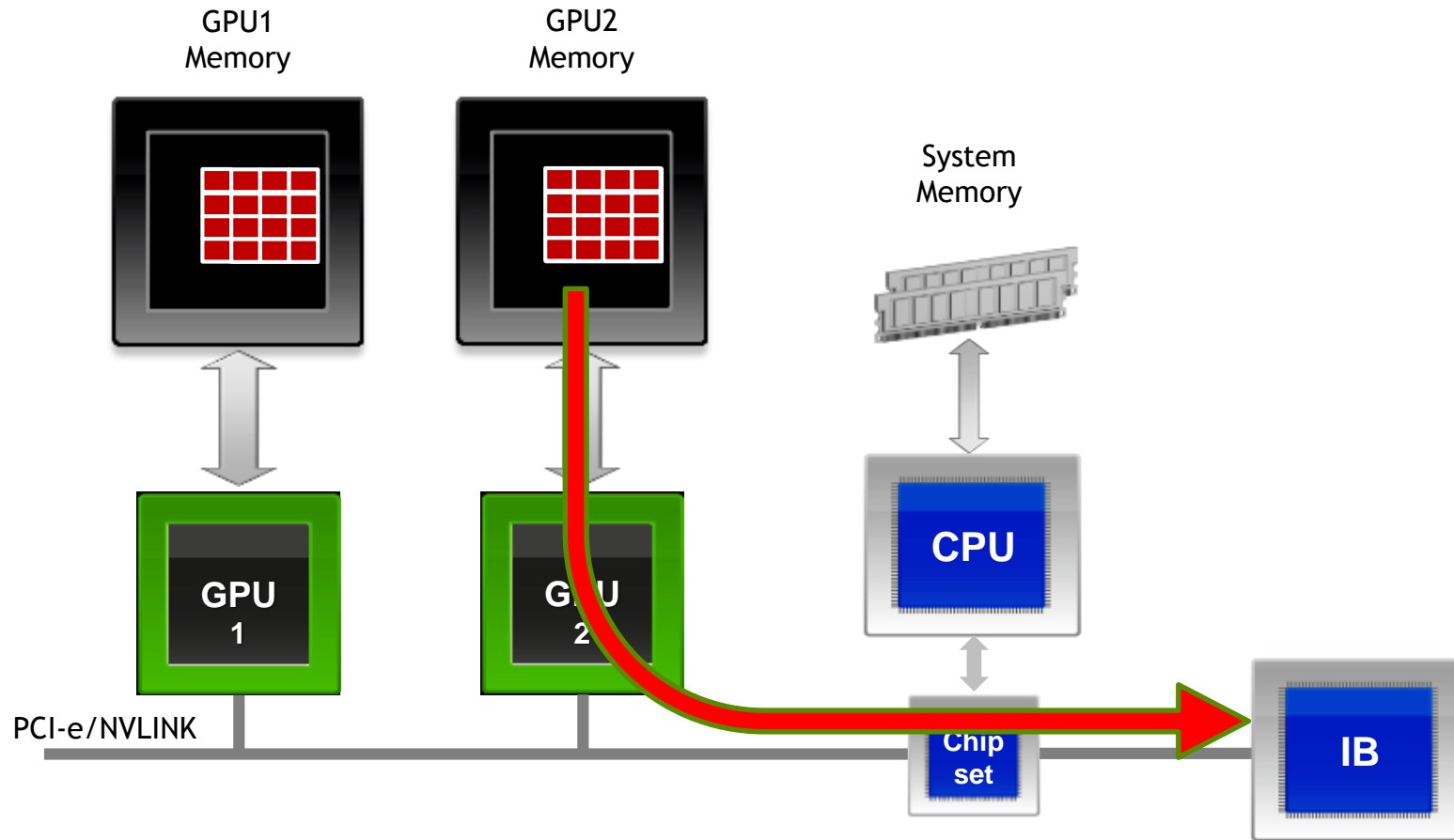
NVIDIA GPUDIRECT™

Peer to Peer Transfers



NVIDIA GPUDIRECT™

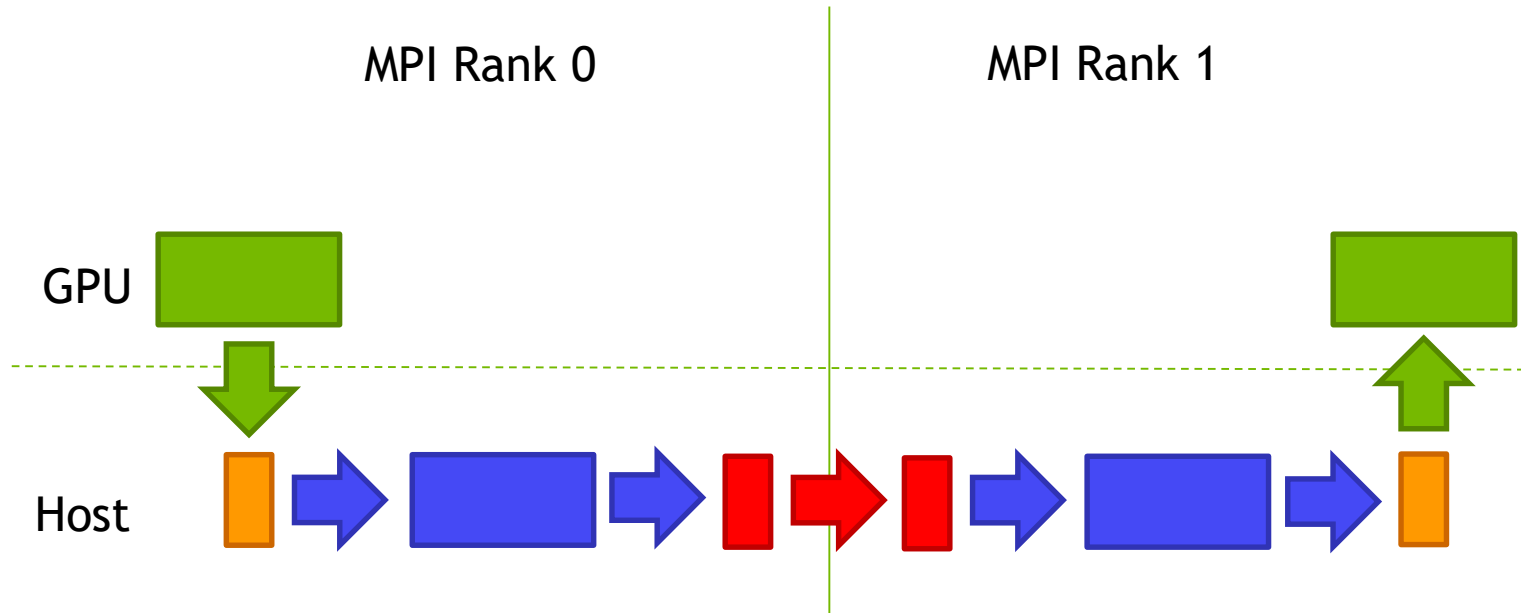
Support for RDMA



The background of the slide features an abstract network diagram. It consists of numerous small, glowing green circular nodes of varying sizes, interconnected by a dense web of thin, light green lines. The lines crisscross the dark, almost black, background, creating a complex, web-like structure that suggests a distributed system or a network topology. The overall aesthetic is high-tech and digital.

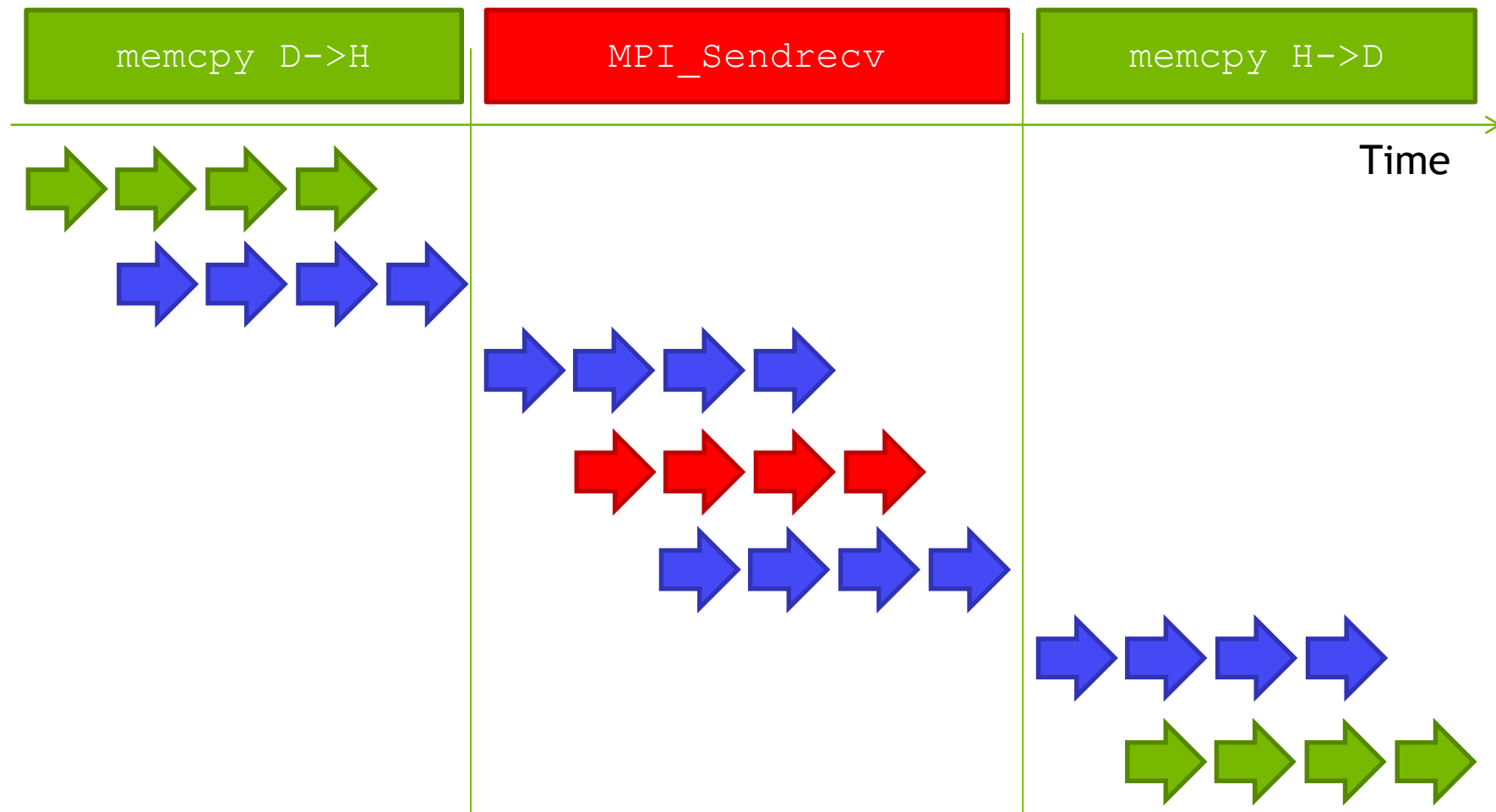
CUDA AWARE MPI FOR ON AND OFF NODE TRANSFERS

REGULAR MPI GPU TO REMOTE GPU



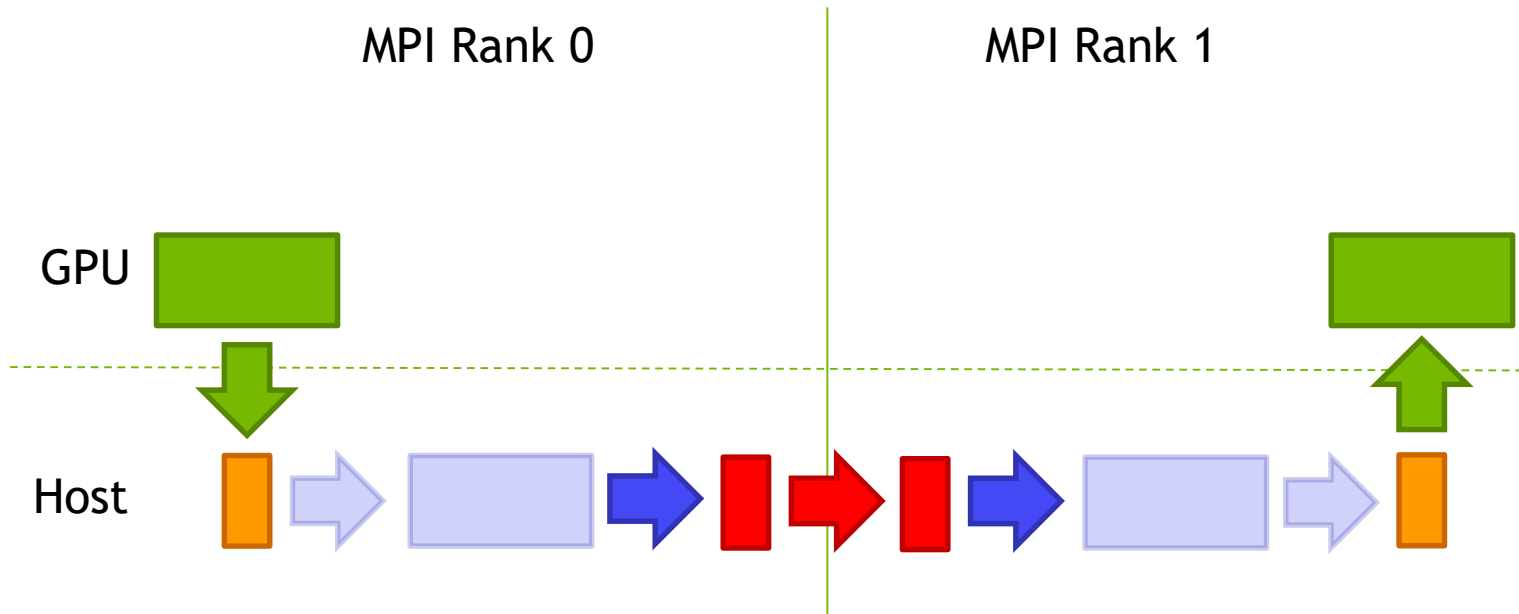
```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);  
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);  
  
MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);  
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```


REGULAR MPI GPU TO REMOTE GPU



MPI GPU TO REMOTE GPU

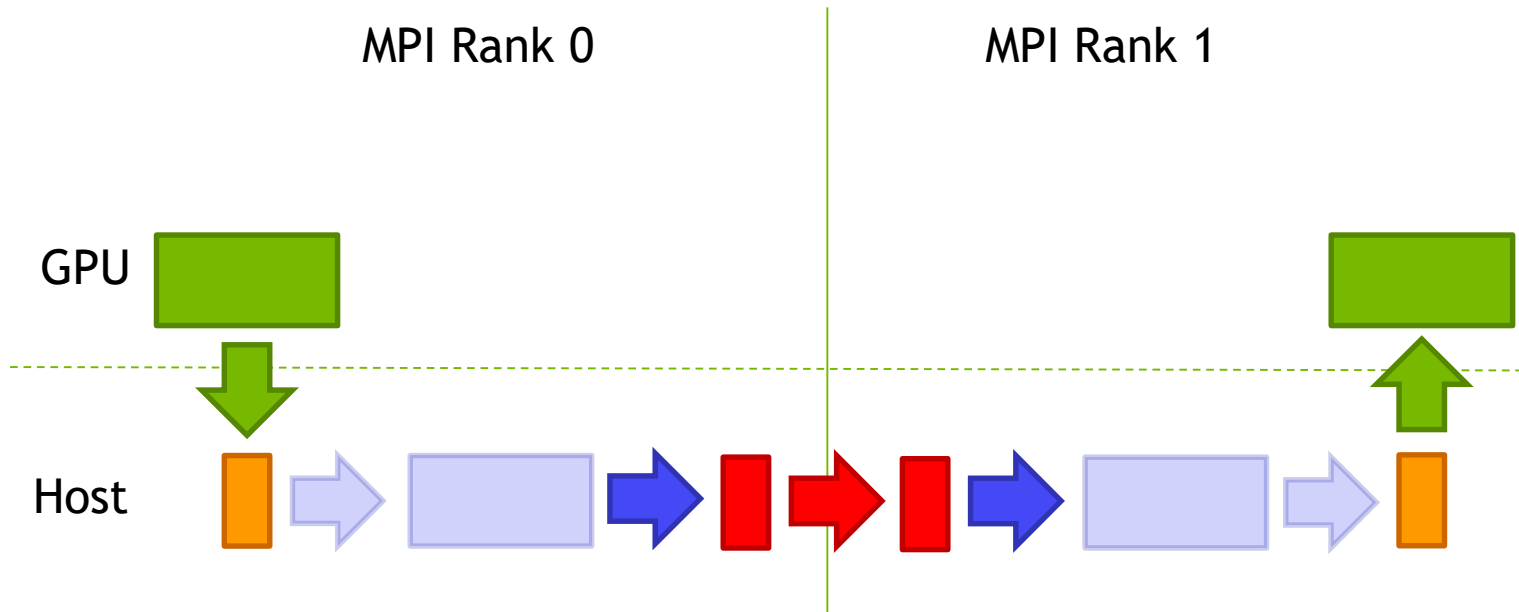
without GPUDirect



```
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

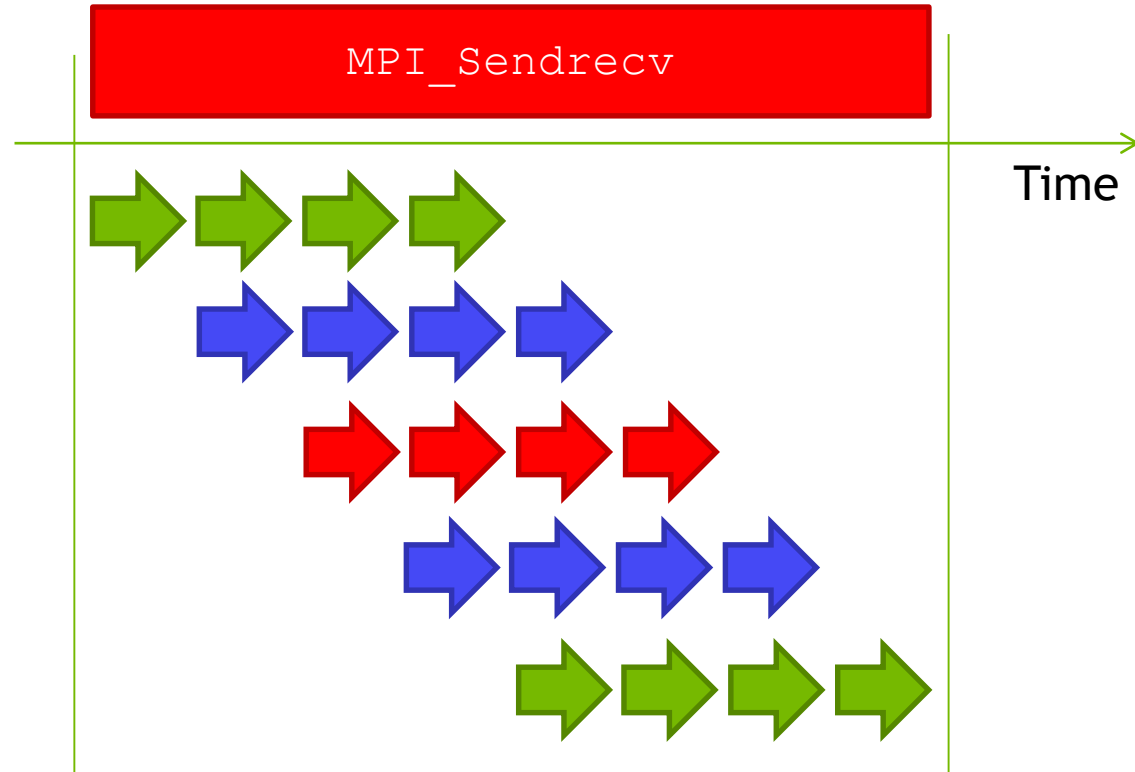


```
#pragma acc host_data use_device (s_buf, r_buf)
MPI_Send(s_buf, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

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

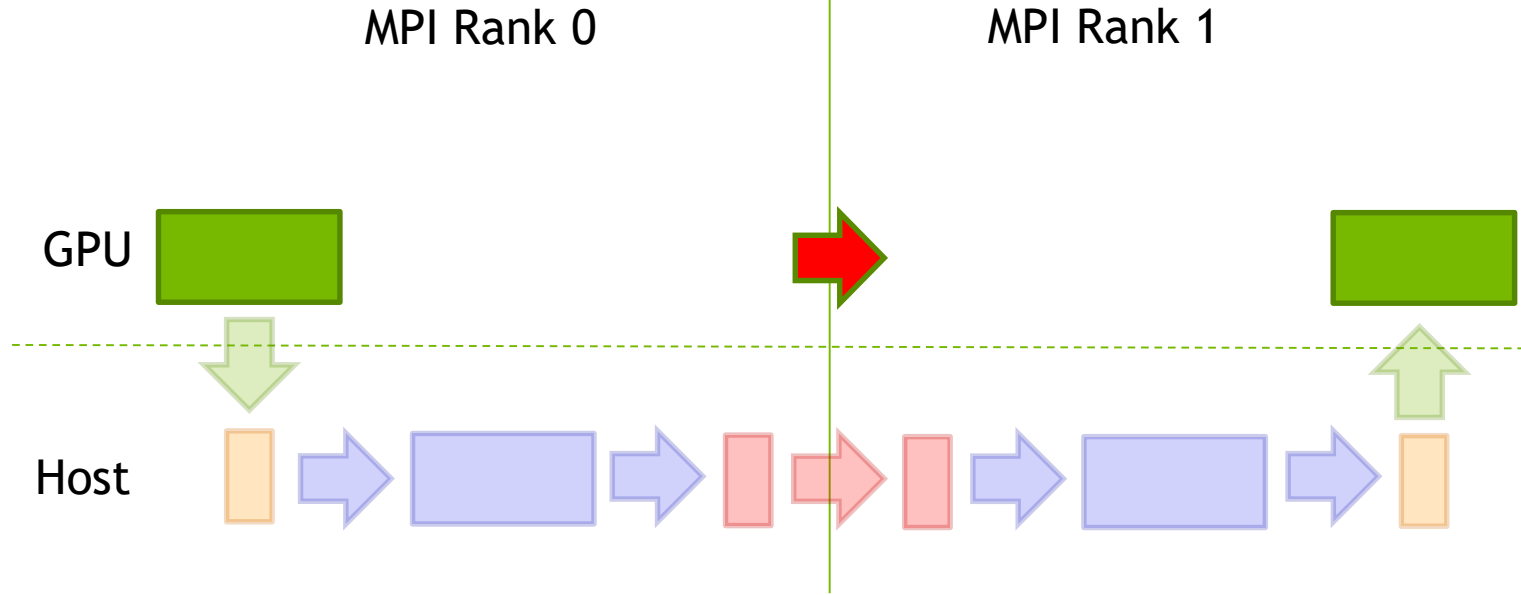
MPI GPU TO REMOTE GPU

without GPUDirect



MPI GPU TO REMOTE GPU

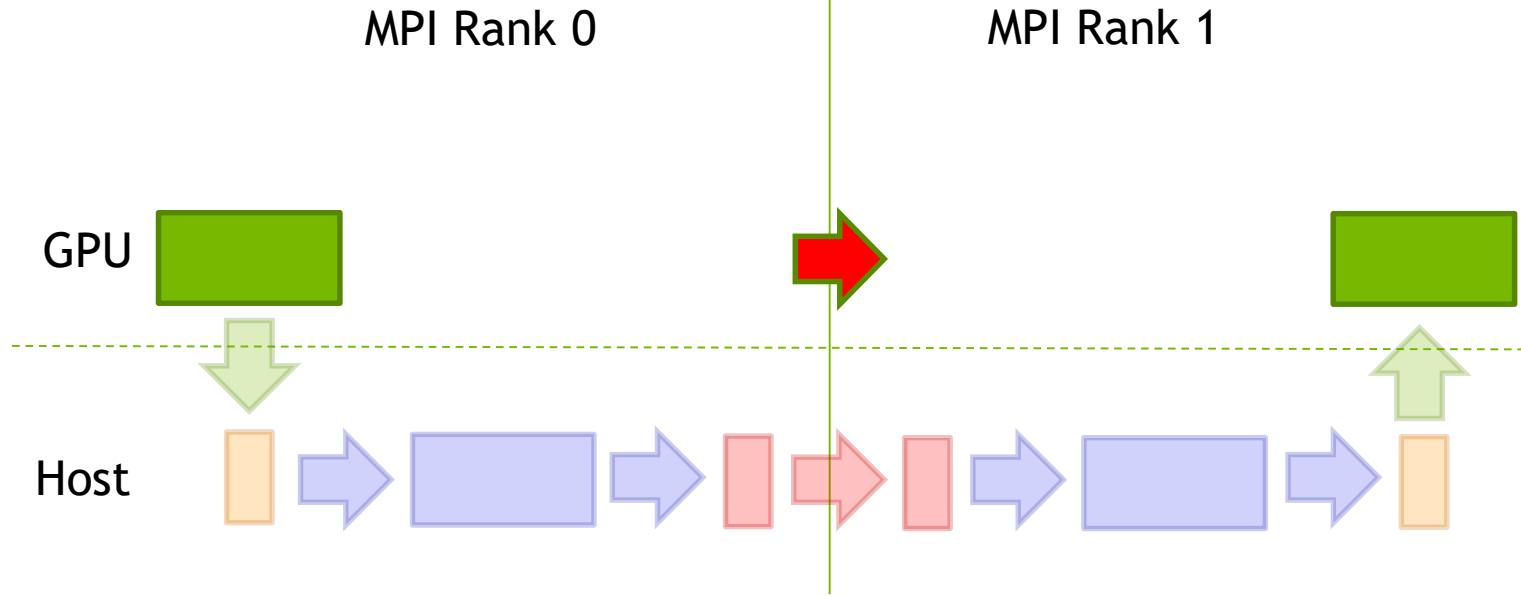
Support for RDMA



```
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

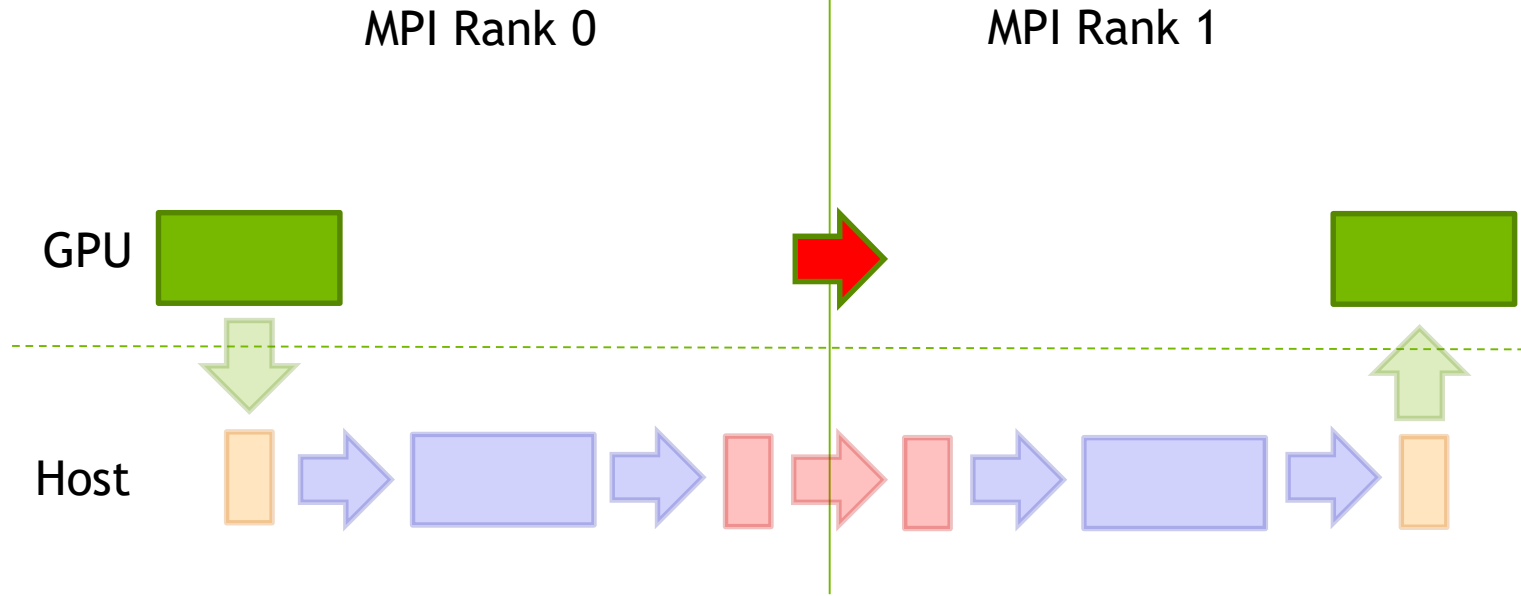


```
#pragma acc host_data use_device (s_buf, r_buf)
MPI_Send(s_buf, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

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

MPI GPU TO REMOTE GPU

Support for RDMA



```
#pragma omp data use_device_ptr(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



The background of the slide features a complex network of thin, light green lines connecting various glowing green nodes of different sizes. The nodes are scattered across the dark blue and black background, creating a sense of a global or digital network. The lines are semi-transparent, allowing overlapping connections to appear darker.

ADVANCED ON-NODE COMMUNICATION

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

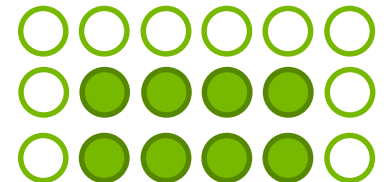
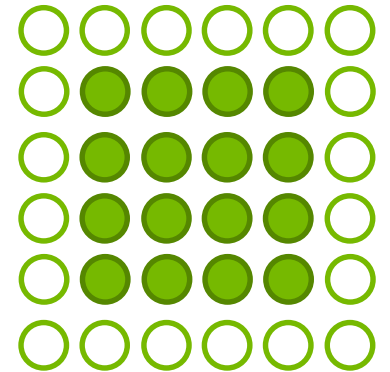
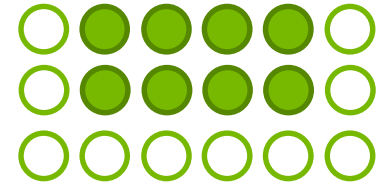
Enable P2P

```
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

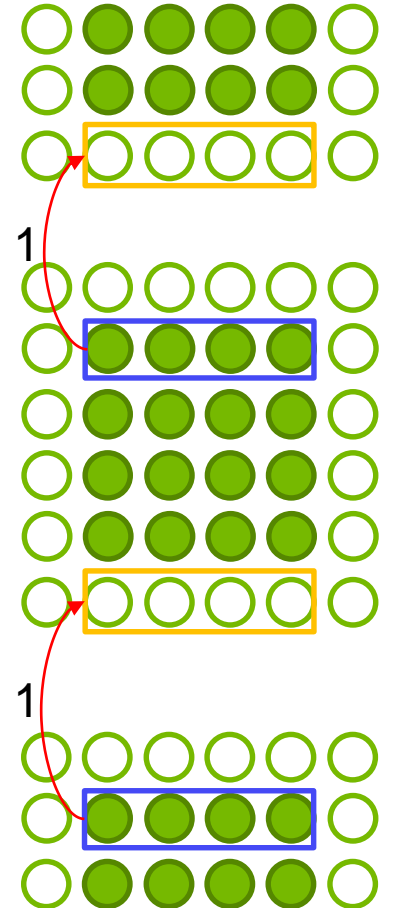
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

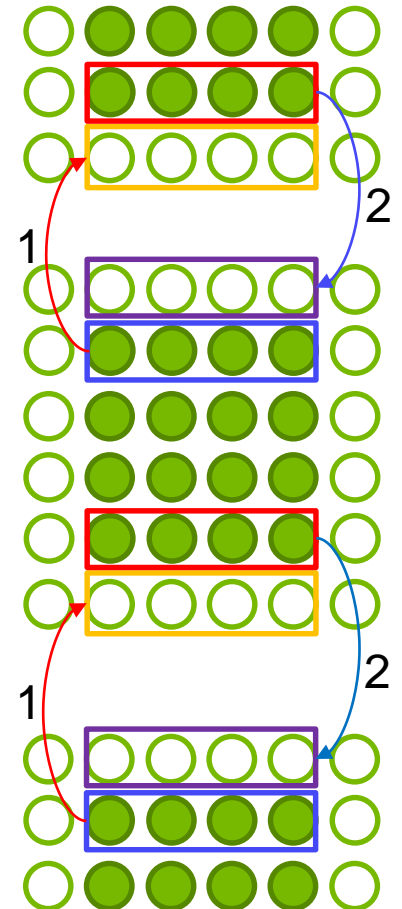
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



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

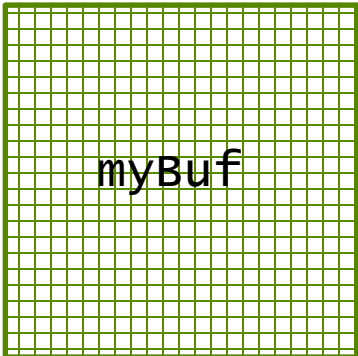
Enable CUDA Intra-Process Communication (IPC)!

```
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)!

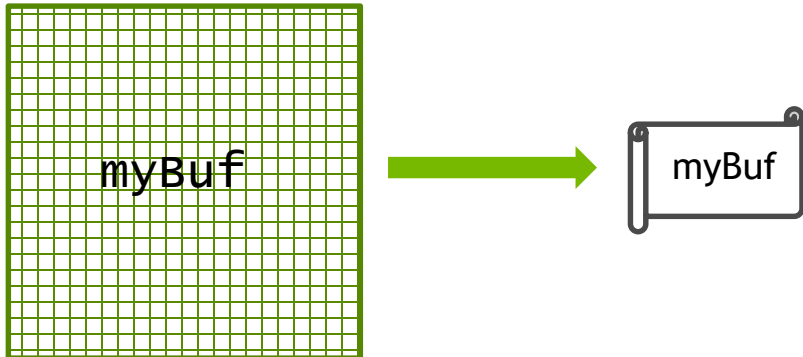
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Intra-Process Communication (IPC)!

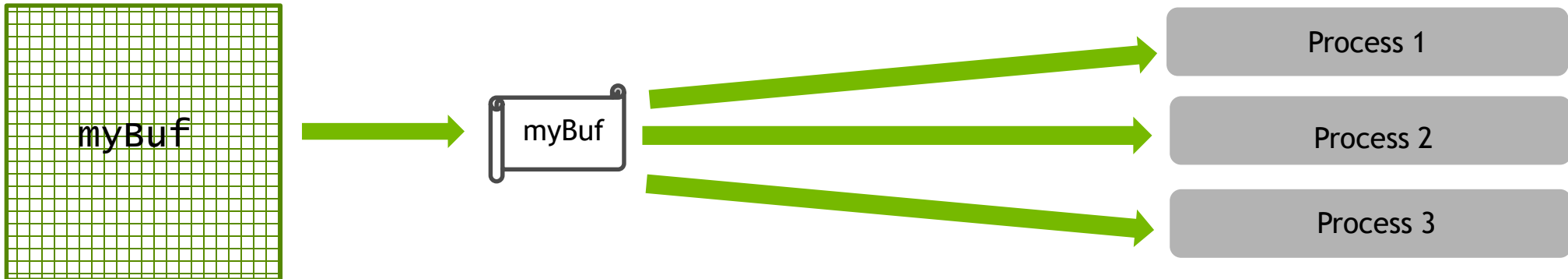
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



GPUDIRECT P2P

Enable CUDA Intra-Process Communication (IPC)!

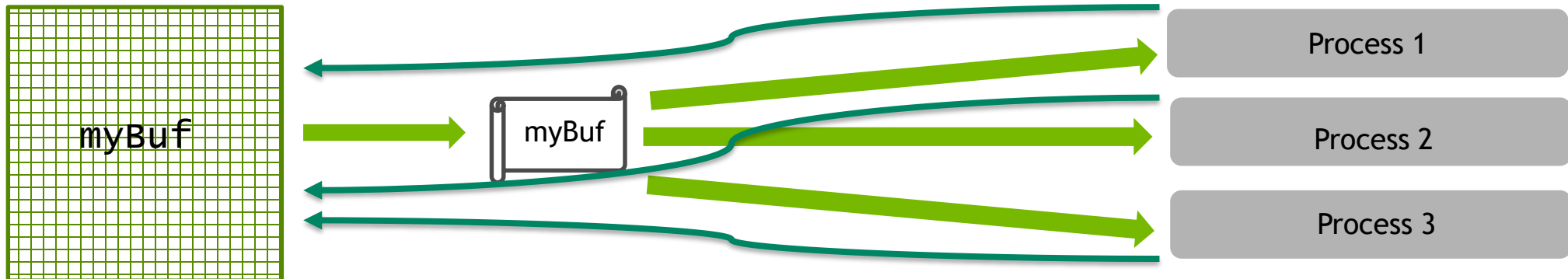
```
cudaSetDevice( dev_id );  
// Allocate and fill my device buffer  
cudaMalloc((void **) &myBuf, nbytes);  
cudaMemcpy((void *) myBuf, (void*) buf, nbytes, cudaMemcpyHostToDevice);  
// Get my IPC handle  
cudaIpcMemHandle_t myIpc;  
cudaIpcGetMemHandle(&myIpc, myBuf);
```



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);
```

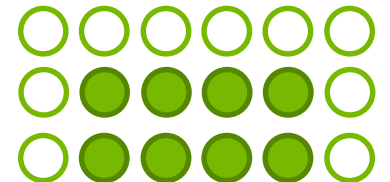
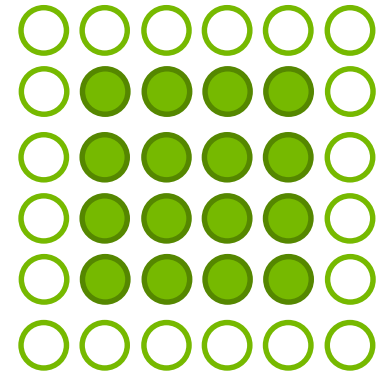
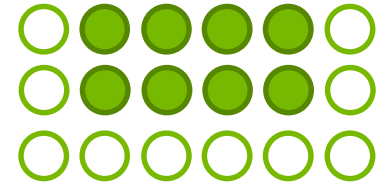


EXAMPLE JACOBI

Top/Bottom Halo

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

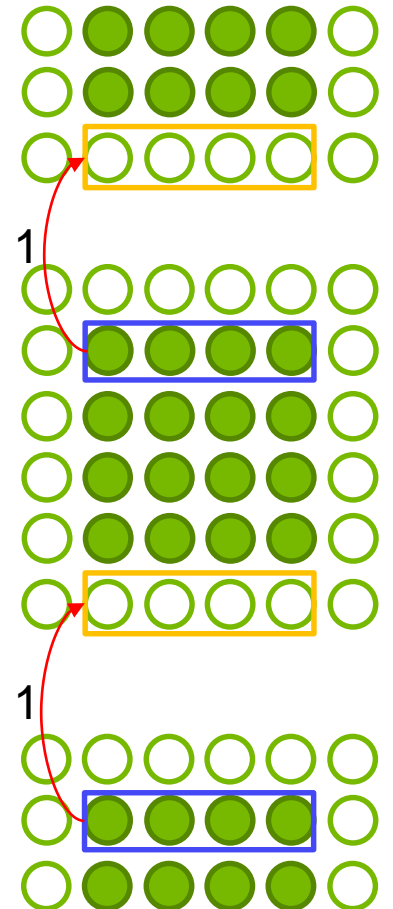
cudaMemcpyAsync(
    a_new[top]+(iy_end[top]*nx),
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



EXAMPLE JACOBI

Top/Bottom Halo

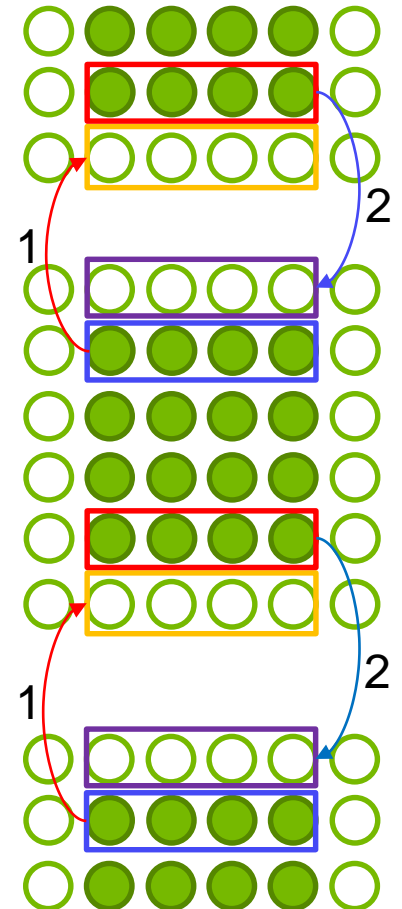
```
cudaIpcOpenMemHandle((void **) &a_new[top], topIpc,  
    cudaIpcMemLazyEnablePeerAccess); cudaCheckError();  
  
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



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
 - ▶ 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!



SPECTRUMMPI & JSRUN TIPS AND TRICKS

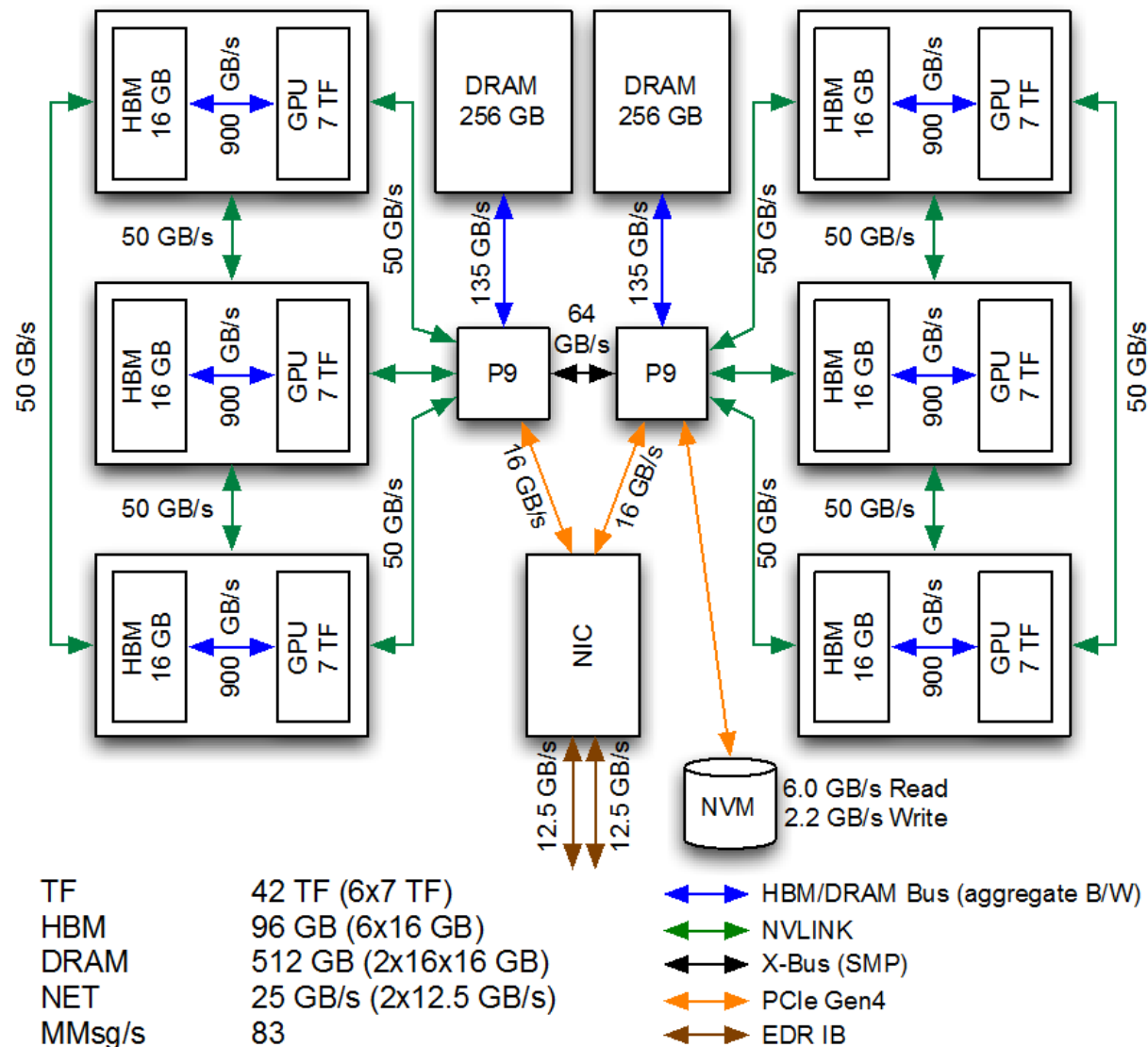
UNDER THE HOOD

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.

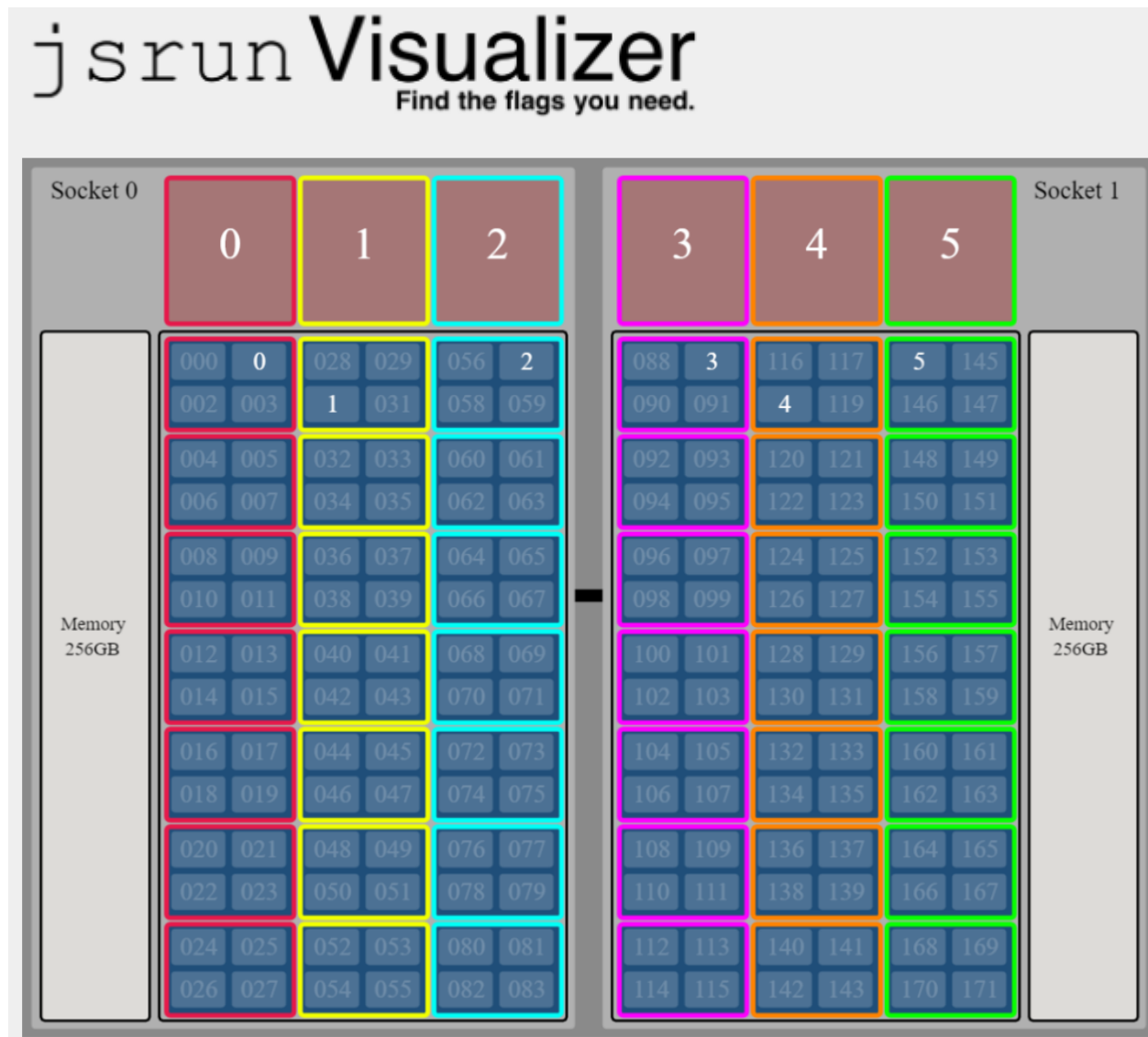
The background is a dark blue gradient. It features a network of thin, light green lines that crisscross the frame. At various points where these lines intersect or terminate, there are small, bright green circular dots. Some of these dots have a soft, out-of-focus glow around them. The overall effect is that of a digital or scientific network visualization.

ESSENTIAL TOOLS

OLCF JSRUN VISUALIZER

For (most of) your layout needs!

<https://jsrunvisualizer.olcf.ornl.gov/index.html>



JSRUN/SMPI GPU OPTIONS

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

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

PROFILING MPI+CUDA APPLICATIONS

Using `nvprof`+`NVVP`

New since CUDA 9

Embed MPI rank in output filename, process name, and context name (OpenMPI)

```
jsrun <args> nvprof --output-profile profile.%q{OMPI_COMM_WORLD_RANK} \
               --process-name "rank %q{OMPI_COMM_WORLD_RANK}" \
               --context-name "rank %q{OMPI_COMM_WORLD_RANK}" \
               --annotate-mpi openmpi
```

Alternatives:

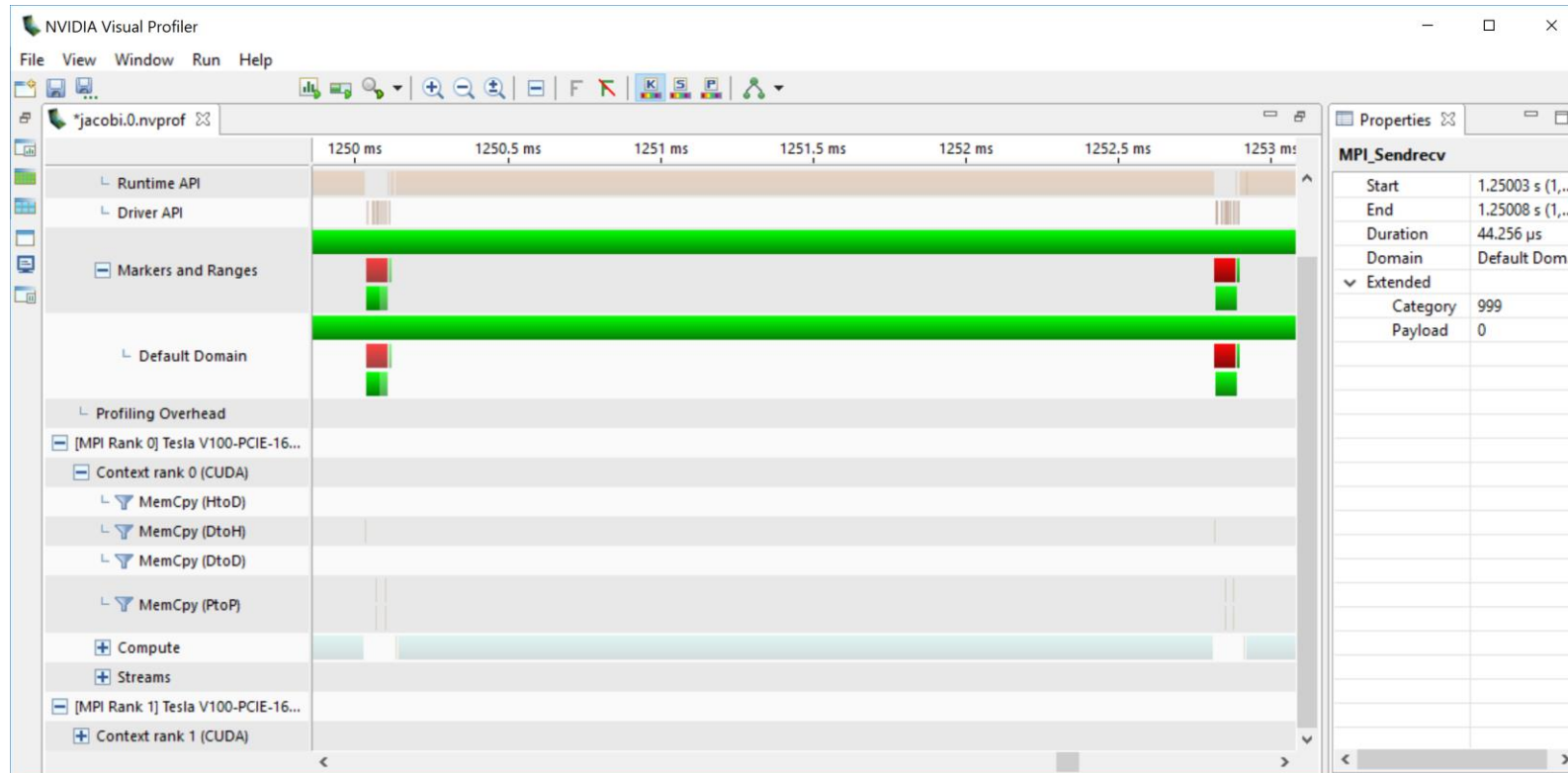
Only save the textual output (`--log-file`)

Collect data from all processes that run on a node (`--profile-all-processes`)

```
MVAPICH2: MV2_COMM_WORLD_RANK
--annotate-mpi mpich
```

PROFILING MPI+CUDA APPLICATIONS

Using nvprof+NVVP



PROFILING NVLINK USAGE

Using nvprof+NVVP

Run nvprof multiple times to collect metrics

```
jsrun <args> nvprof --output-profile profile.<metric>.%q{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

Import Nvprof Data

Import Profile Data for Single Process

Select one nvprof profile file containing timeline data and zero or more additional nvprof profile files containing event and metric values.

Profile Files | Timeline Options

Connection: Local Manage connections...

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

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_nratom_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_nratom_data_transmitted.0.nvprof
- C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_ratom_data_transmitted.0.nvprof

Kernel scopes:

Add the nvprof scope used for analysis in the format <context id/name>:<stream id/name>:<kernel name>:<unions>

☒ 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 Cancel

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 additional nvprof profile files containing event and metric values.

Profile Files | Timeline Options

Connection: Local Manage connections...

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

Event/Metric data files:

- C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.mpi_profile.0.nvprof Browse...
- C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_overhead_data_received.0.nvprof Remove
- 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_nratom_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_nratom_data_transmitted.0.nvprof
- C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_ratom_data_transmitted.0.nvprof

Kernel scopes:


Add the nvprof scope used for analysis in the format <context id/name> <stream id/name> <kernel name> <run 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 Cancel

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

 Switch to unguided analysis

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 additional nvprof profile files containing event and metric values.

Profile Files | Timeline Options

Connection: Local Manage connections...

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

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_nratom_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_nratom_data_transmitted.0.nvprof
C:\Users\sabbott\Documents\Multi_gpu_tutorial\onsocket.fullipc.nvlink_user_ratom_data_transmitted.0.nvprof

Kernel scopes:

Add the nvprof scope used for analysis in the format <context id/name>:<stream id/name>:<kernel name>:<run iterations>

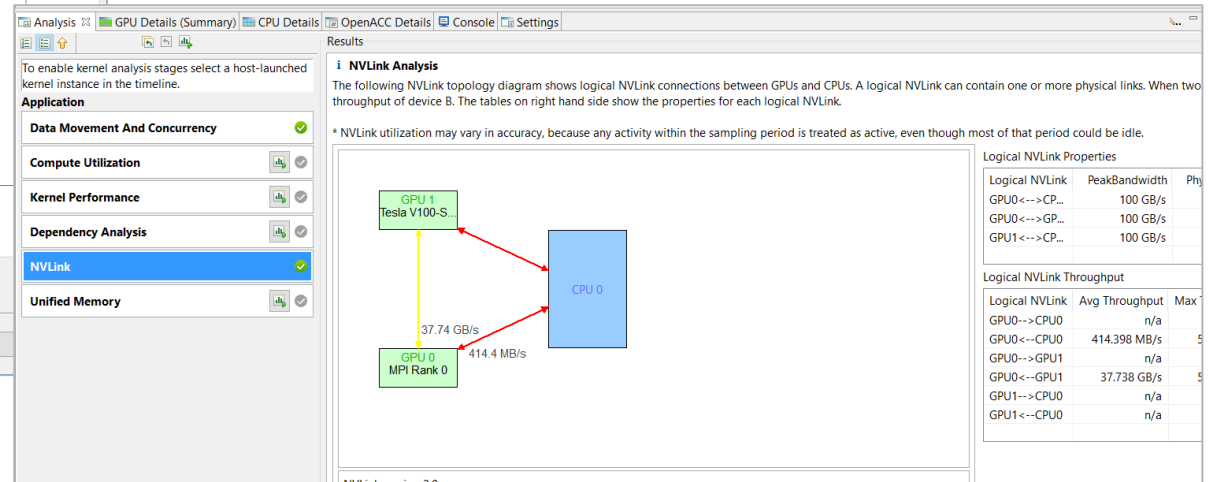
☒ Use fixed width segments for Unified memory timeline

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





EXAMPLES

SIMPLE MPI PING-PONG CODE

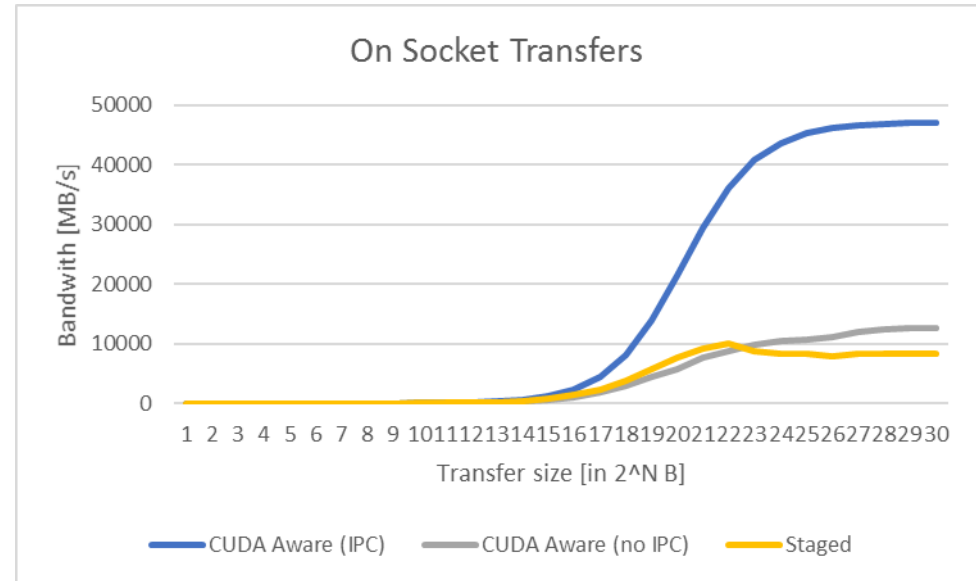
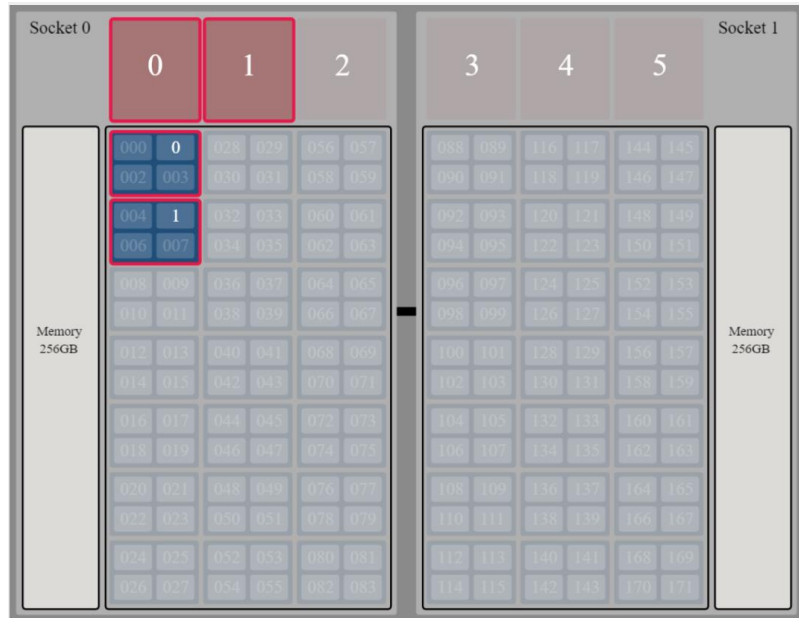
```
start = MPI_Wtime();
for (i = 0; i < NLOOPS; i++) {
    send_func(cubuf, buf, nbytes, 1, 1000 + i);
    recv_func(cubuf, buf, nbytes, 1, 2000 + i);
}
stop = MPI_Wtime();
```

```
void
stagedSend(void *cubuf, void *hostbuf, size_t nbytes, int dest, int tag)
{
    cudaMemcpy(hostbuf, cubuf, nbytes, cudaMemcpyDeviceToHost); cudaCheckError();
    MPI_Send(hostbuf, nbytes, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
}

void
nakedSend(void *cubuf, void *hostbuf, size_t nbytes, int dest, int tag)
{
    MPI_Send(cubuf, nbytes, MPI_BYTE, dest, tag, MPI_COMM_WORLD);
}
```

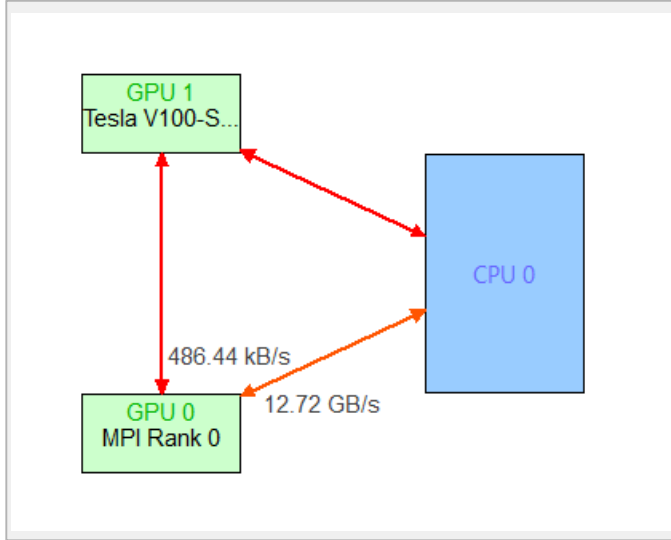
ON SOCKET TRANSFERS

`jsrun -n 1 -c 2 -g 2 -a 2 -d packed -b packed:1 [--smpiargs="-gpu"]`

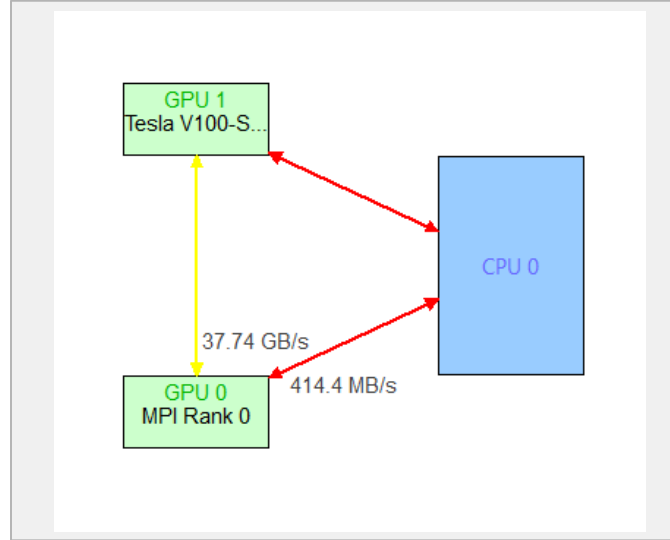


WHAT DOES DATA MOVEMENT LOOK LIKE?

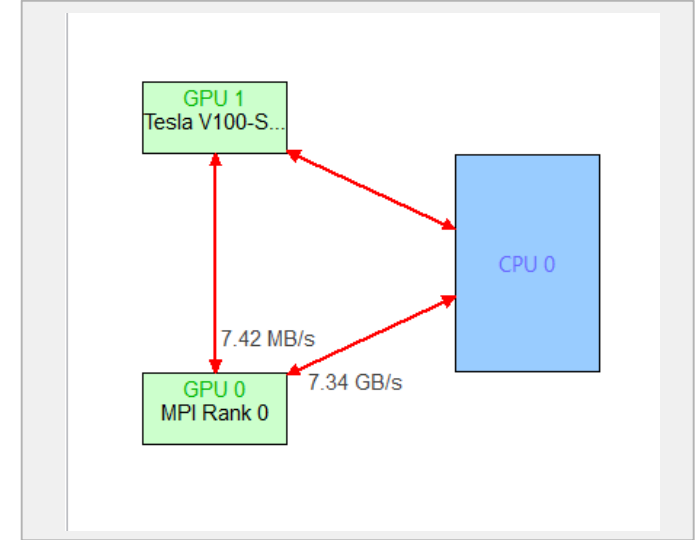
NVLinks provide alternate paths



Staged through the host



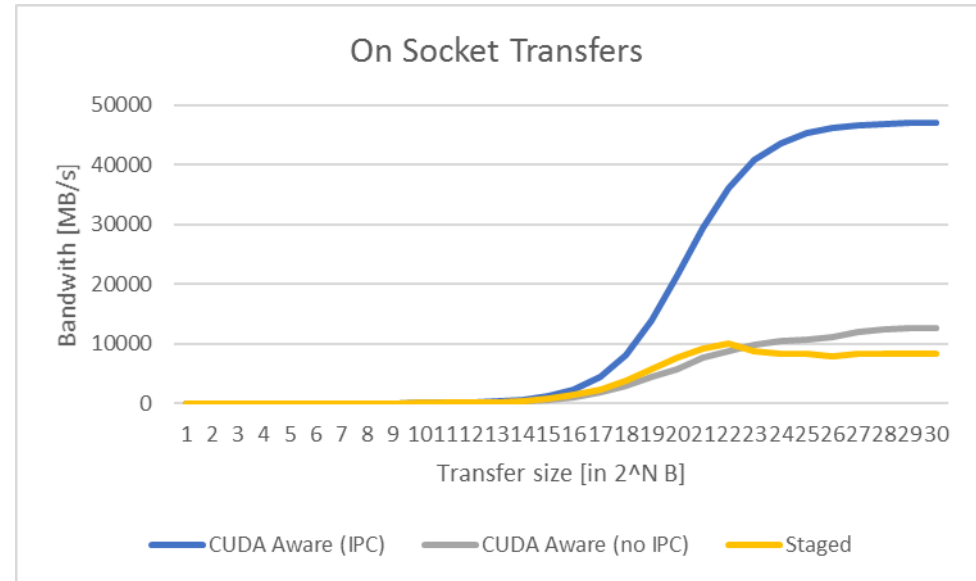
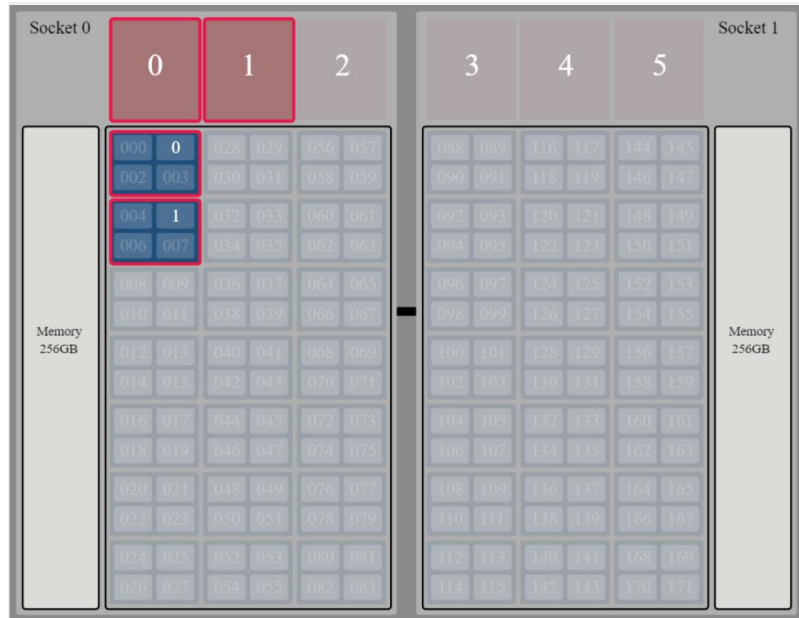
CUDA Aware MPI
With CUDA IPC



CUDA Aware MPI
Without CUDA IPC

ON SOCKET TRANSFERS

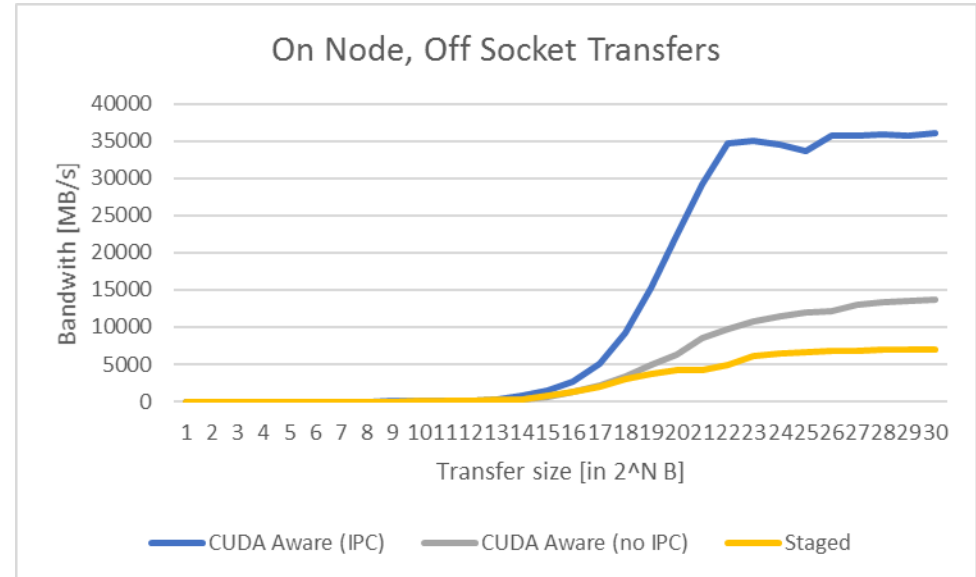
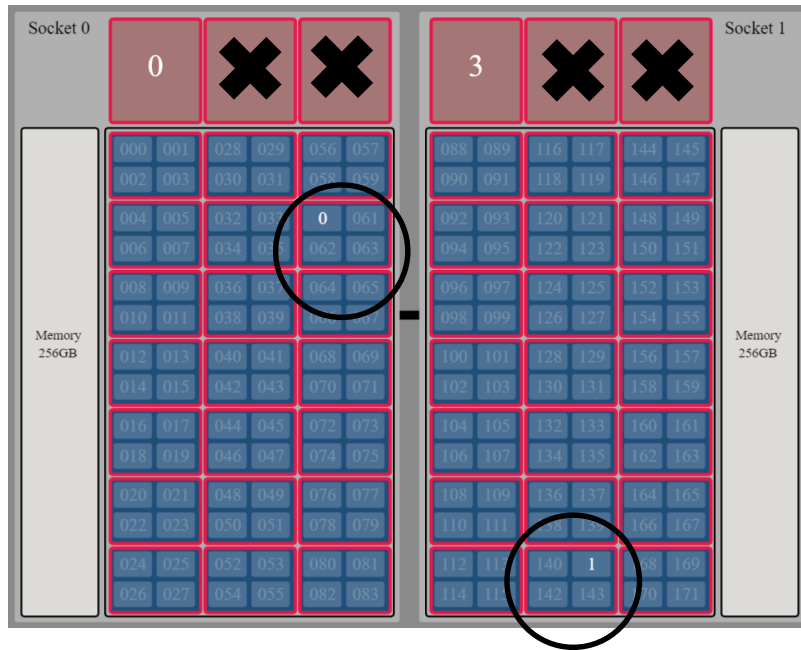
`jsrun -n 1 -c 2 -g 2 -a 2 -d packed -b packed:1 [--smpiargs="-gpu"]`



OFF SOCKET, ON NODE TRANSFERS

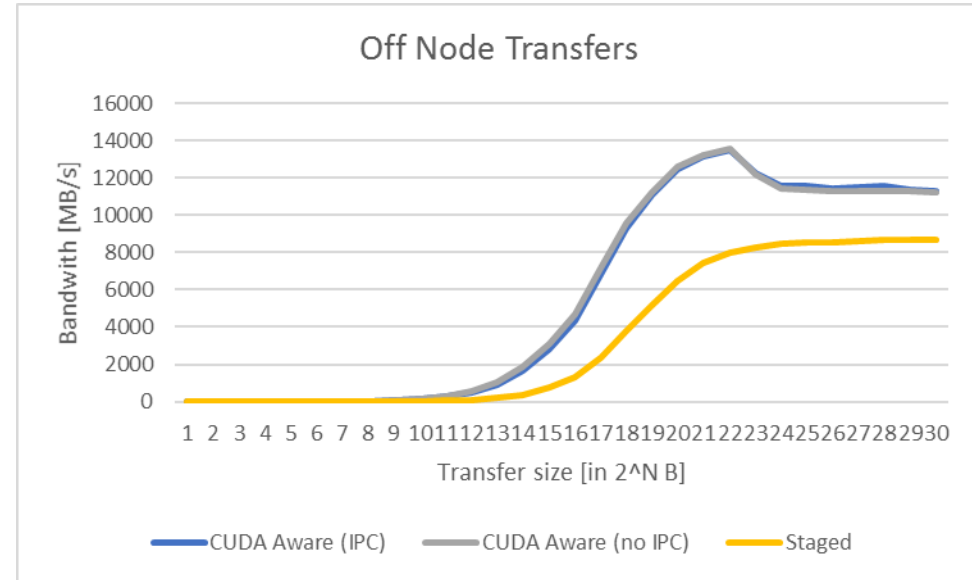
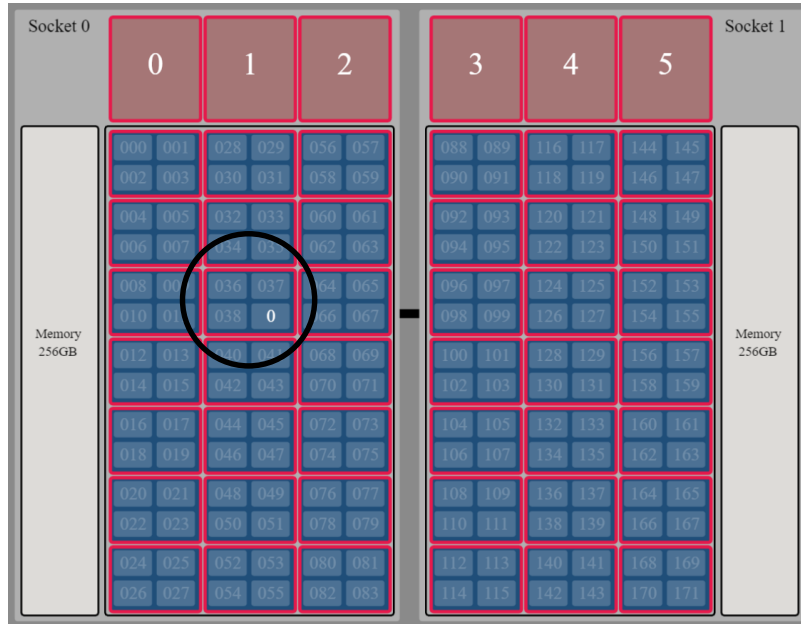
export CUDA_VISIBLE_DEVICES=0,3

jsrun -n 1 -c 42 -g 6 -a 2 -d packed -b packed:21 [--smpiargs="-gpu"]



OFF NODE TRANSFERS

`jsrun -n 2 -c 42 -g 6 -a 1 -d packed -b packed:42[--smpiargs="-gpu"]`



KNOWN ISSUES

Things to watch out for

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]

Avoid CUDA Managed Memory or MPI Derived Types in GPU sends!

The background features a dark blue field with a network of thin, light green lines connecting various points. These points are represented by small, glowing green dots of varying sizes. The lines and dots create a sense of a complex, interconnected system or network, possibly representing data flow or a global communication network. The overall aesthetic is modern and technological.

CLOSING SUMMARY

MULTI-GPU APPROACHES

Choosing an approach

Single-Threaded, Multiple-GPUs - Requires additional loops to manage devices, likely undesirable.

Multi-Threaded, Multiple-GPUs - Very convenient set-and-forget the device. Could possibly conflict with existing threading.

Multiple-Ranks, Single-GPU each - Probably the simplest if you already have MPI, the decomposition is done. Must get your MPI placement correct

Multiple-Ranks, Multiple-GPUs - Can allow all GPUs to share common data structures. Only do this if you absolutely need to, difficult to get right.

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 TRICK

- ▶ 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!

