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 & Jsrun Tips and Tricks
SUMMIT NODE
OVERVIEW
SUMMIT NODE
(2) IBM POWER9 + (6) NVIDIA VOLTA V100

256 GB (DDR4)

135 GB/s

CPU 0

0 (0-3)
1 (4-7)
2 (8-11)
3 (12-15)
4 (16-19)
5 (20-23)
6 (24-27)

14 (56-59)
15 (60-63)
16 (64-67)
17 (68-71)
18 (72-75)
19 (76-79)
20 (80-83)

256 GB (DDR4)

135 GB/s

CPU 1

22 (88-91)
23 (92-95)
24 (96-99)
25 (100-103)
26 (104-107)
27 (108-111)
28 (112-115)

29 (116-119)
30 (120-123)
31 (124-127)
32 (128-131)
33 (132-135)
34 (136-139)
35 (140-143)

36 (144-147)
37 (148-151)
38 (152-155)
39 (156-159)
40 (160-163)
41 (164-167)
42 (168-171)

16 GB (HBM2)
16 GB (HBM2)
16 GB (HBM2)
16 GB (HBM2)
16 GB (HBM2)
16 GB (HBM2)

64 GB/s

NVLink2

(50 GB/s)

(900 GB/s)
UNDER THE HOOD

Summit has fat nodes!

Many connections
Many devices
Many stacks
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

- **Single Thread, Multiple GPUs**
  - Conceptually Simple
  - Requires additional loops
  - CPU can become a bottleneck
  - Remaining CPU cores often underutilized

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

- **Multiple Ranks, Single GPU**
  - Little to no code changes required
  - Re-uses existing domain decomposition
  - Can see improved utilization
  - Watch affinity

- **Multiple Ranks, Multiple GPUs**
  - Easily share data between peer devices
  - Coordinating between GPUs extremely tricky
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
MULTI-GPU PROGRAMMING WITH OPENACC AND CUDA
The CUDA and OpenACC approaches are sufficiently similar, that I will demonstrate using OpenACC.

Decoder Ring:

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc_get_device_type()</td>
<td>N/A</td>
</tr>
<tr>
<td>acc_set_device_type()</td>
<td>N/A</td>
</tr>
<tr>
<td>acc_set_device_num()</td>
<td>cudaSetDevice()</td>
</tr>
<tr>
<td>acc_get_device_num()</td>
<td>cudaGetDevice()</td>
</tr>
<tr>
<td>acc_get_num_devices()</td>
<td>cudaGetDeviceCount()</td>
</tr>
</tbody>
</table>
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

Device 0

H2D | kernel | D2H
--- | ------ | ---
H2D | kernel | D2H

Device 1

H2D | kernel | D2H
--- | ------ | ---
H2D | kernel | D2H
#pragma acc data create(imgData[w*h*ch], out[w*h*ch])
    copyin(filter)
{
    for ( long blocky = 0; blocky < nbblocks; 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
}
Pipelined Code

```c
#pragma acc data create(imgData[w*h*ch], out[w*h*ch])
   copyin(filter)
{
    for ( long blocky = 0; blocky < nbblocks; 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++) {
        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)

```c
#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++) {
            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

Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1)
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)

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/mpirun is placing your ranks.
MPI Image Filter (pseudocode)

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

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();

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

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

There’s a variety of ways to do MPI decomposition, this is what I used for this particular example.
Multi-GPU Pipelined Performance (MPI)

Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1), Communication Excluded
Multi-GPU Pipelined Performance (MPI)

Source: PGI 17.3, NVIDIA Tesla P100 (DGX-1), Communication Excluded
MULTI-DEVICE CUDA
Same Pattern, Different API

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

```c
MPI_Comm_rank(local_comm, &local_rank);

cudaSetDevice(local_rank);

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

```c
cudaDeviceSynchronize();
```

```c
MPI_Comm_rank(local_comm, &local_rank);

cudaSetDevice(local_rank);

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

```c
cudaDeviceSynchronize();
```
# Multi-Device OpenMP 4.5

Same Pattern, Different API

```c
#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
}
```

```c
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, he 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

GPU1 Memory

GPU2 Memory

System Memory

CPU

Chip set

IB

PCI-e/NVLINK
NVIDIA GPUDIRECT™
Support for RDMA

GPU1 Memory

GPU1

GPU2 Memory

System Memory

GPU2

PCI-e/NVLINK

CPU

Chip set

IB
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);

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 Rank 0       MPI Rank 1

GPU

Host

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

MPI_rank 0

GPU

Host

MPI_rank 1

GPU

Host

#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_Sendrecv

Time
MPI GPU TO REMOTE GPU

Support for RDMA

MPI Send:

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

MPIRecv:

```c
MPI_Recv(r_buf_d, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &stat);
```
MPI GPU TO REMOTE GPU

Support for RDMA

MPI Rank 0

MPI Rank 1

#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

MPI Rank 0

MPI Rank 1

GPU

Host

#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

MPI_Sendrecv

Time
ADVANCED ON-NODE COMMUNICATION
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 );
        cudaMemcpyAsync(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), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), cudaMemcpyDeviceToHost );
    }
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        cudaSetDevice( dev_id ); cudaDeviceSynchronize();
        12_norm += *(l2_norm_h[dev_id]);
    }
    12_norm = std::sqrt( 12_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

```c
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), ...);
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), ...);
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);
    cudaMemcpyAsync(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)!

cuSetDevice( 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);

myBuf

Process 1

Process 2

Process 3
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), ...);
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 straightforward 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
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`

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

```bash
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`)

**New since CUDA 9**

**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

```bash
jsrun <args> nvprof --output-profile profile.<metric>.%q{OMPI_COMM_WORLD_RANK}\n--aggregate-mode off --event-collection-mode continuous \n--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

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

Switch to unguided analysis
EXAMPLES
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 [--smipiargs="-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[--smipiargs="-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!
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 straightforward 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
- 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!