CSE 599 l
Accelerated Computing - Programming GPUS
Multi-GPU Programming
Objective

● Look at patterns for multi-gpu computation, including:
  ○ Multiple GPUs hosted in the same PC
  ○ Multiple GPUs hosted in multiple PCs on the same network
When is it Time to Scale to Multiple GPUs?

- The data to be processed doesn’t fit in the global memory of a single GPU memory and is accessed too frequently and/or irregularly for zero-copy
- Using a single GPU results in many “waves” of blocks which are serialized
- The latencies involved in inter-GPU communication can be hidden by computation done by each GPU

Done properly (and applied to the right problem), bandwidth of multi-GPU programs can grow almost exactly linearly with the number of GPUs (i.e. approach 100% efficiency)
The Ideal Situation
Data-Parallel Multi-GPU Programming

For truly massive $N$ and fully-independent computation, simply distribute $N/M$ input elements to $M$ devices

In the simplest case, all devices are connected via PCIe to the same node
How Many Devices are Available?

```c
int numGpus;
cudaGetDeviceCount(&numGpus);

printf("There are \%d CUDA-capable devices available\n", numGpus);

for (int i = 0; i < numGpus; ++i) {
    cudaDeviceProp deviceProperties;
    cudaGetDeviceProperties(&deviceProperties, i);
    printf("Device \%d has compute capability \%d.%d\n",
        i, deviceProperties.major, deviceProperties.minor);
}
```
VectorAdd with Multiple GPUs

```c
void superMassiveVectorAdd(const float * h_A, const float * h_B, float * h_C, const int N) {

    const int nPerGpu = (N - 1) / NUM_GPUS + 1;

    float * d_As[NUM_GPUS];
    float * d_Bs[NUM_GPUS];
    float * d_Cs[NUM_GPUS];
    cudaStream_t streams[NUM_GPUS];

    for (int i = 0; i < NUM_GPUS; ++i) {
        // set current device
        cudaSetDevice(i);
        // allocate memory on current device
        cudaMalloc(&d_As[i], nPerGpu * sizeof(float));
        cudaMalloc(&d_Bs[i], nPerGpu * sizeof(float));
        cudaMalloc(&d_Cs[i], nPerGpu * sizeof(float));

        cudaStreamCreate(&streams[i]);
    }

    for (int i = 0; i < NUM_GPUS; ++i) {
        cudaSetDevice(i);
        cudaMemcpyAsync(d_As[i], h_A + i * nPerGpu, nPerGpu * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
        cudaMemcpyAsync(d_Bs[i], h_B + i * nPerGpu, nPerGpu * sizeof(float), cudaMemcpyHostToDevice, streams[i]);

        vectorAddKernel<<<(nPerGpu-1)/256+1,256,0,streams[i]>>>(d_As[i], d_Bs[i], d_Cs[i], nPerGpu);
        cudaMemcpyAsync(h_C + nPerGpu, d_Cs[i], nPerGpu * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
    }
}
```
Streams and Synchronization with Multiple GPUs

- Streams are specific to a device (the current device when it was created)
  - Kernels can only be launched in a stream if the associated device is current
  - Events can only be recorded in a stream if the associated device is current

- Any stream can query or synchronize with an event on any other stream, even if the stream is associated with another device
  - This allows for inter-device synchrony
UVA and Multiple GPUs

Uniform Virtual Addressing (UVA) maps all global device memory into the same address space.

This means any pointer to global memory on any device can be used to determine the location within the device memory, as well as **which device**

```c
int whichDevice(const void * ptr) {
    cudaPointerAttributes ptrAttributes;
    cudaPointerGetAttributes(&ptrAttributes, ptr);
    return ptrAttributes.device;
}
```

This also means cudaMemcpy can be used to copy to/from any device at any time, regardless of the current device.
Enabling Peer-to-Peer Access

```c
void enableAllPossibleP2PAccess(const int numGpus) {
    for (int device = 0; device < numGpus; ++device) {
        cudaSetDevice(device);
        for (int peerDevice = 0; peerDevice < numGpus; ++peerDevice) {
            int accessIsPossible = 0;
            cudaDeviceCanAccessPeer(&accessIsPossible, device, peerDevice);
            if (accessIsPossible) {
                cudaDeviceEnablePeerAccess(peerDevice);
                printf("Device %d now has access to device %d\n", device, peerDevice);
            }
        }
    }
}
```

This will allow to GPUs attached to the same PCI-e root node to communicate directly over PCI-e.

Other inter-device communications must go through the host.
Direct Peer-to-Peer Access

With UVA, devices for which peer access has been enabled can read and write directly to the global memory of a peer device.

These reads and writes happen directly over PCI-e, similarly to zero-copy memory reads and writes between device and host.
Peer-to-peer Memory Copy

cudaMemcpy can also be used to explicitly transfer data from one device to another.

This is done in the same way as any other data transfer, and uses the cudaMemcpyDeviceToDevice flag.
Five-Level Processing Hierarchy

**Node-level:**
- PCI-e Transfers
- Coordination through stream, event, and device synchronization

**Grid-level:**
- Global memory
- Atomic operations and flags for intra-grid, inter-block coordination

**Block-level:**
- Shared memory
- __sync-threads for intra-block, inter-thread coordination

**Warp-level:**
- Registers
- Shuffle instructions for intra-warp, inter-thread coordination

**Thread-level:**
- Local memory / registers
- No coordination needed
Six-Level Processing Hierarchy

Network-level:
- Network transfers
- Coordination must also be managed through network transfers

Node-level:
- PCI-e Transfers
- Coordination through stream, event, and device synchronization

Grid-level:
- Global memory
- Atomic operations and flags for intra-grid, inter-block coordination

Block-level:
- Shared memory
- __sync-threads for intra-block, inter-thread coordination

Warp-level:
- Registers
- Shuffle instructions for intra-warp, inter-thread coordination

Thread-level:
- Local memory / registers
- No coordination needed
Computation unit

~1 clock cycle

Memory unit
Warp

thread thread thread thread

register register register register
Warp

~5 clock cycles

Shared Memory
Grid

Block

~500 clock cycles

Global Memories
Module 18 – Related Programming Models: MPI
Lecture 18.1 - Introduction to Heterogeneous Supercomputing and MPI
Objective

- To learn the basics of an MPI application
  - Blue Waters, a supercomputer clusters with heterogeneous CPU-GPU nodes
  - MPI initialization, message passing, and barrier synchronization API functions
  - Vector addition example
Blue Waters - Operational at Illinois since 3/2013

12.5 PF
1.6 PB DRAM
$250M

120+ Gb/sec

10/40/100 Gb Ethernet Switch

100 GB/sec

WAN

Spectra Logic: 300 P Bs

1.6 PB DRAM

IB Switch

Sonexion: 26 PBs

23
Cray XE6 Dual Socket Nodes

- Two AMD Interlagos chips
  - 16 core modules, 64 threads
  - 313 GFs peak performance
  - 64 GBs memory
    - 102 GB/sec memory bandwidth

- Gemini Interconnect
  - Router chip & network interface
  - Injection Bandwidth (peak)
    - 9.6 GB/sec per direction

Blue Waters contains 22,640 Cray XE6 compute nodes.
Cray XK7 Dual Socket Nodes

- One AMD Interlagos chip
  - 8 core modules, 32 threads
  - 156.5 GFs peak performance
  - 32 GBs memory
    - 51 GB/s bandwidth
- One NVIDIA Kepler chip
  - 1.3 TFs peak performance
  - 6 GBs GDDR5 memory
    - 250 GB/sec bandwidth
- Gemini Interconnect
  - Same as XE6 nodes

Blue Waters contains 4,224 Cray XK7 compute nodes.
Many processes distributed in a cluster

Each process computes part of the output
Processes communicate with each other
Processes can synchronize
MPI Initialization, Info and Sync

- **int MPI_Init(int *argc, char ***argv)**
  - Initialize MPI

- **MPI_COMM_WORLD**
  - MPI group with all allocated nodes

- **int MPI_Comm_rank (MPI_Comm comm, int *rank)**
  - Rank of the calling process in group of comm

- **int MPI_Comm_size (MPI_Comm comm, int *size)**
  - Number of processes in the group of comm
Vector Addition: Main Process

```c
int main(int argc, char *argv[]) {
    int vector_size = 1024 * 1024 * 1024;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Need 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1); return 1;
    }
```
Vector Addition: Main Process

    if(pid < np - 1)
        compute_node(vector_size / (np - 1));
    else
        data_server(vector_size);

    MPI_Finalize();
    return 0;
}
MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - `Buf`: Initial address of send buffer (choice)
  - `Count`: Number of elements in send buffer (nonnegative integer)
  - `Datatype`: Datatype of each send buffer element (handle)
  - `Dest`: Rank of destination (integer)
  - `Tag`: Message tag (integer)
  - `Comm`: Communicator (handle)
MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - `Buf`: Initial address of send buffer (choice)
  - `Count`: Number of elements in send buffer (nonnegative integer)
  - `Datatype`: Datatype of each send buffer element (handle)
  - `Dest`: Rank of destination (integer)
  - `Tag`: Message tag (integer)
  - `Comm`: Communicator (handle)
MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - `Buf`: Initial address of receive buffer (choice)
  - `Count`: Maximum number of elements in receive buffer (integer)
  - `Datatype`: Datatype of each receive buffer element (handle)
  - `Source`: Rank of source (integer)
  - `Tag`: Message tag (integer)
  - `Comm`: Communicator (handle)
  - `Status`: Status object (Status)
MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - `Buf`: Initial address of receive buffer (choice)
  - `Count`: Maximum number of elements in receive buffer (integer)
  - `Datatype`: Datatype of each receive buffer element (handle)
  - `Source`: Rank of source (integer)
  - `Tag`: Message tag (integer)
  - `Comm`: Communicator (handle)
  - `Status`: Status object (Status)
Vector Addition: Server Process (I)

```c
void data_server(unsigned int vector_size) {
    int np, num_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_bytes  = vector_size * sizeof(float);
    float *input_a = 0, *input_b = 0, *output = 0;

    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    /* Allocate input data */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if (input_a == NULL || input_b == NULL || output == NULL) {
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    /* Initialize input data */
    random_data(input_a, vector_size, 1, 10);
    random_data(input_b, vector_size, 1, 10);
}
```
Vector Addition: Server Process (II)

/* Send data to compute nodes */
float *ptr_a = input_a;
float *ptr_b = input_b;

for(int process = 0; process < num_nodes; process++) {
    MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_a += vector_size / num_nodes;

    MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_b += vector_size / num_nodes;
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);
Vector Addition: Server Process (III)

/* Wait for previous communications */
MPI_BARRIER(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_nodes; process++) {
    MPI_Recv(output + process * vector_size / num_nodes,
              vector_size / num_nodes, MPI_REAL, process,
              DATA_COLLECT, MPI_COMM_WORLD, &status);
}

/* Store output data */
store_output(output, vector_size);

/* Release resources */
free(input_a);
free(input_b);
free(output);
}
Vector Addition: Compute Process (I)

```c
void compute_node(unsigned int vector_size) {
    int np;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a, *input_b, *output;
    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    /* Alloc host memory */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);

    /* Get the input data from server process */
    MPI_Recv(input_a, vector_size, MPI_FLOAT, server_process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
    MPI_Recv(input_b, vector_size, MPI_FLOAT, server_process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
```
MPI Barriers

- `int MPI_Barrier (MPI_Comm comm)`
  - `Comm`: Communicator (handle)

- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.
MPI Barriers

- Wait until all other processes in the MPI group reach the same barrier
  - All processes are executing Do_Stuff()
  - Some processes reach the barrier and the wait in the barrier until all reach the barrier

Example Code

```c
Do_stuff();
MPI_Barrier();
Do_more_stuff();
```
Vector Addition: Compute Process (II)

/* Compute the partial vector addition */
for(int i = 0; i < vector_size; ++i) {
    output[i] = input_a[i] + input_b[i];
}

/* Report to barrier after computation is done*/
MPI_Barrier(MPI_COMM_WORLD);

/* Send the output */
MPI_Send(output, vector_size, MPI_FLOAT,
              server_process, DATA_COLLECT, MPI_COMM_WORLD);

/* Release memory */
free(input_a);
free(input_b);
free(output);
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
Module 18 – Related Programming Models: MPI
Lecture 18.2 – Introduction to MPI-CUDA Programming
A Typical Wave Propagation Application

- **Do** $T = 0$, $T_{max}$
- **Insert Source** (e.g. acoustic wave)
- **Stencil Computation to compute Laplacian**
- **Time Integration**
- **Absorbing Boundary Conditions**
- $T == T_{max}$
Review of Stencil Computations

- Example: wave propagation modeling

\[ \nabla^2 U - \frac{1}{v^2} \frac{\partial U}{\partial t} = 0 \]

- Approximate Laplacian using finite differences

Boundary Conditions

Laplacian and Time Integration
Wave Propagation: Kernel Code

/* Coefficients used to calculate the laplacian */
__constant__ float coeff[5];

__global__ void wave_propagation(float *next, float *in,
                                 float *prev, float *velocity, dim3 dim)
{
    unsigned x = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned y = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned z = threadIdx.z + blockIdx.z * blockDim.z;

    /* Point index in the input and output matrixes */
    unsigned n = x + y * dim.z + z * dim.x * dim.y;

    /* Only compute for points within the matrixes */
    if (x < dim.x && y < dim.y && z < dim.z) {

        /* Calculate the contribution of each point to the laplacian */
        float laplacian = coeff[0] + in[n];
    }
}
Wave Propagation: Kernel Code

```c
for(int i = 1; i < 5; ++i) {
    laplacian += coeff[i] *
        (in[n - i] + /* Left */
         in[n + i] + /* Right */
         in[n - i * dim.x] + /* Top */
         in[n + i * dim.x] + /* Bottom */
         in[n - i * dim.x * dim.y] + /* Behind */
         in[n + i * dim.x * dim.y]); /* Front */
}

/* Time integration */
next[n] = velocity[n] * laplacian + 2 * in[n] - prev[n];
```
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies
int main(int argc, char *argv[]) {
    int pad = 0, dimx = 480+pad, dimy = 480, dimz = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np, num_comp_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes = num_points * sizeof(float);
    float *input = 0, *output = NULL, *velocity = NULL;
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    velocity = (float *)malloc(num_bytes);
    if (input == NULL || output == NULL || velocity == NULL) {
        printf("Server couldn't allocate memory\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    /* Initialize input data and velocity */
    random_data(input, dimx, dimy, dimz, 1, 10);
    random_data(velocity, dimx, dimy, dimz, 1, 10);
/* Calculate number of shared points */
int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
float *input_send_address = input;

/* Send input data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
         DATA_DISTRIBUTE, MPI_COMM_WORLD);
send_address += dimx * dimy * (dimz / num_comp_nodes - 4);

/* Send input data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
    MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
             DATA_DISTRIBUTE, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send input data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
         DATA_DISTRIBUTE, MPI_COMM_WORLD);
Stencil Code: Server Process (III)

```c
float *velocity_send_address = velocity;

/* Send velocity data to compute nodes */
for (int process = 0; process < num_comp_nodes; process++) {
    MPI_Send(velocity + process * num_points / num_comp_nodes,
             num_points / num_comp_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for (int process = 0; process < num_comp_nodes; process++)
    MPI_Recv(output + process * num_points / num_comp_nodes,
              num_points / num_comp_nodes, MPI_FLOAT, process,
              DATA_COLLECT, MPI_COMM_WORLD, &status);
```
Stencil Code: Server Process (IV)

/* Store output data */
store_output(output, dimx, dimy, dimz);

/* Release resources */
free(input);
free(velocity);
free(output);
Stencil Code: Compute Process (I)

```c
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points * sizeof(float);

    int left_ghost_offset   = 0;
    int right_ghost_offset  = dimx * dimy * (4 + dimz);

    float *input = NULL, *output = NULL, *prev = NULL, *v = NULL;

    /* Allocate device memory for input and output data */
    gmacMalloc((void **) &input, num_bytes);
    gmacMalloc((void **) &output, num_bytes);
    gmacMalloc((void **) &prev, num_bytes);
    gmacMalloc((void **) &v, num_bytes);
```
MPI_Status status;
int left_neighbor = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;
int server_process = np - 1;

/* Get the input data from server process */
float *rcv_address = input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
         DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);

/* Get the velocity data from server process */
rcv_address = h_v + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
         DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
Module 18 – Related Programming Models: MPI
Lecture 18.3 – Overlapping Computation with Communication
Objective

- To learn how to overlap computation with communication in a MPI+CUDA application
  - Stencil example
  - CUDA Stream as an enabler of overlap
  - MPI_SendRecv() function
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies
CUDA and MPI Communication

- Source MPI process:
  - cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost)
  - MPI_Send()
- Destination MPI process:
  - MPI_Recv()
  - cudaMemcpy(dst, src, cudaMemcpyDeviceToDevice)
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np,
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    num_comp_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output=0;
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL) {
        printf("server couldn't allocate memory
");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    /* Calculate number of shared points */
    int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
    int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
    float *send_address = input;
Data Server Process Code (II)

/* Send data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
         0, MPI_COMM_WORLD);

send_address += dimx * dimy * (dimz / num_comp_nodes - 4);
/* Send data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
    MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
             0, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
         0, MPI_COMM_WORLD);
Compute Process Code (I).

```c
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_halo_points  = 4 * dimx * dimy;
    unsigned int num_halo_bytes   = num_halo_points * sizeof(float);

    /* Alloc host memory */
    float *h_input  = (float *)malloc(num_bytes);
    /* Allocate device memory for input and output data */
    float *d_input = NULL;
    cudaMalloc((void **)&d_input, num_bytes);
    float *rcv_address = h_input + num_halo_points * (0 == pid);
    MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```
void launch_kernel(float *next, float *in, float *prev, float *velocity,
               int dimx, int dimy, int dimz)
{
    dim3 Gd, Bd, Vd;

    Vd.x = dimx; Vd.y = dimy; Vd.z = dimz;

    Bd.x = BLOCK_DIM_X; Bd.y = BLOCK_DIM_Y; Bd.z = BLOCK_DIM_Z;

    Gd.x = (dimx + Bd.x - 1) / Bd.x;
    Gd.y = (dimy + Bd.y - 1) / Bd.y;
    Gd.z = (dimz + Bd.z - 1) / Bd.z;

    wave_propagation<<<Gd, Bd>>>(next, in, prev, velocity, Vd);
}
MPI Sending and Receiving Data

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  -Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.
Compute Process Code (II)

```c
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_output = (float *)malloc(num_bytes);
cudaMalloc((void **) &d_output, num_bytes);

float *h_left_boundary = NULL, *h_right_boundary = NULL;
float *h_left_halo = NULL, *h_right_halo = NULL;

/* Alloc host memory for halo data */
cudaHostAlloc((void **) &h_left_boundary, num_halo_bytes);
cudaHostAlloc((void **) &h_right_boundary, num_halo_bytes);
cudaHostAlloc((void **) &h_left_halo, num_halo_bytes);
cudaHostAlloc((void **) &h_right_halo, num_halo_bytes);

/* Create streams used for stencil computation */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```
Device Memory Offsets Used for Data Exchange with Neighbors

- Left halo offset
- Right halo offset
- Left stage 1 offset
- Right stage 1 offset
Compute Process Code (III)

MPI_Status status;
int left_neighbor = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

/* Upload stencil coefficients */
upload_coefficients(coeff, 5);

int left_halo_offset = 0;
int right_halo_offset = dimx * dimy * (4 + dimz);
int left_stagel_offset = 0;
int right_stagel_offset = dimx * dimy * (dimz - 4);
int stage2_offset = num_halo_points;

MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; i < nreps; i++) {
    /* Compute boundary values needed by other nodes first */
    launch_kernel(d_output + left_stagel_offset,
                  d_input + left_stagel_offset, dimx, dimy, 12, stream0);
    launch_kernel(d_output + right_stagel_offset,
                  d_input + right_stagel_offset, dimx, dimy, 12, stream0);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                  dimx, dimy, dimz, stream1);
Compute Process Code (IV)

/* Copy the data needed by other nodes to the host */
cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
        num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
cudaMemcpyAsync(h_right_boundary,
        d_output + right_stage1_offset + num_halo_points,
        num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
cudaStreamSynchronize(stream0);
Compute Process Code (V)

/* Send data to left, get data from right */
MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
    left_neighbor, i, h_right_halo,
    num_halo_points, MPI_FLOAT, right_neighbor, i,
    MPI_COMM_WORLD, &status );

/* Send data to right, get data from left */
MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
    right_neighbor, i, h_left_halo,
    num_halo_points, MPI_FLOAT, left_neighbor, i,
    MPI_COMM_WORLD, &status );

cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
    num_halo_bytes, cudaMemcpyHostToDevice, stream0);

cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
    num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
cudaDeviceSynchronize();

float *temp = d_output;
d_output = d_input; d_input = temp;
}
Compute Process Code (VI)

/* Wait for previous communications */
MPI_Barrier(MPI_COMM_WORLD);

float *temp = d_output;
d_output = d_input;
d_input = temp;

/* Send the output, skipping halo points */
cudaMemcpy(h_output, d_output, num_bytes,
           cudaMemcpyDeviceToHost);
float *send_address = h_output + num_ghost_points;
MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
         server_process, DATA_COLLECT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

/* Release resources */
free(h_input); free(h_output);
cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
cudaFree( d_input ); cudaFree( d_output );
Data Server Code (III)

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_comp_nodes; process++)
    MPI_Recv(output + process * num_points / num_comp_nodes,
              num_points / num_comp_nodes, MPI_REAL, process,
              DATA_COLLECT, MPI_COMM_WORLD, &status );

/* Store output data */
store_output(output, dimx, dimy, dimz);

/* Release resources */
free(input);
free(output);
More on MPI Message Types

- Point-to-point communication
  - Send and Receive
- Collective communication
  - Barrier
  - Broadcast
  - Reduce
  - Gather and Scatter
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
MPI Communication Groups

Using `MPI_COMM_WORLD` all the time can be expensive.

Smaller communication groups can be defined.

Synchronization and message passing within the smaller groups is faster.

This is analogous to the division of a CUDA thread grid into blocks.
MPI Communication Groups

```c
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

All nodes that pass the same color value to `MPI_Comm_split` will get back the same `newcomm` communicator.

The rank of nodes in the new communicators will be determined by `key`.

All nodes still participate in communications using the original `comm`.
MPI Communication Groups

MPI_Comm localCommunicator;

MPI_Comm_split(MPI_COMM_WORLD, nodeID / 3, nodeID, &localCommunicator);
Conclusion / Takeaways

- Programming for multiple GPUs on the same node is a relatively straightforward extension of single-GPU programming, where the host manages memory transfers between host and device and device and device.
- Programming for multiple nodes involves network communication, potentially introducing large latencies to be hidden.
- MPI (Message Passing Interface) is a useful API for managing memory transfer and synchronization between multiple nodes.
Sources
