Joint CUDA-MPI Programming
Objective

➢ To become proficient in writing a simple joint MPI-CUDA heterogeneous application
  ➢ Understand the key sections of the application
  ➢ One-way communication

➢ To become familiar with a more sophisticated MPI application that requires two-way data exchange
Titan – the new ruler of TOP500

17.6 PF Sustained, 27.1 PF Peak

S3D – improve efficiency of biofuel combustion
WL-LSMS – interactions between electrons and atoms in magnetic materials
Denovo – improve efficiency and reduce waste in nuclear reactors
LAMMPS – improvements to semiconductors, biomolecules, polymers
CAM-SE – more accurate climate simulations
NRDF – laser fusion, fluid dynamics, medical imaging, nuclear reactors, …
Blue Waters Computing System

- **Spectra Logic**: 300 PBs
- **Sonexion**: 26 PBs
- **10/40/100 Gb Ethernet Switch**
- **IB Switch**: >1 TB/sec
- **WAN**: 120+ Gb/sec
- **100 GB/sec**
Cray XE6 Nodes

- Dual-socket Node
  - 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 Nodes

- **Dual-socket Node**
  - 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

- Gemmi Interconnect
  - Same as XE6 nodes

Blue Waters contains 3,072 Cray XK7 compute nodes.
Blue Waters
3D Torus Size
23 x 24 x 24
# Blue Waters and Titan Computing Systems

<table>
<thead>
<tr>
<th>System Attribute</th>
<th>NCSA Blue Waters</th>
<th>ORNL Titan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendors</td>
<td>Cray/AMD/NVIDIA</td>
<td>Cray/AMD/NVIDIA</td>
</tr>
<tr>
<td>Processors</td>
<td>Interlagos/Kepler</td>
<td>Interlagos/Kepler</td>
</tr>
<tr>
<td>Total Peak Performance (PF)</td>
<td>11.5</td>
<td>27.1</td>
</tr>
<tr>
<td>Total Peak Performance (CPU/GPU)</td>
<td>7.1/4</td>
<td>2.6/24.5</td>
</tr>
<tr>
<td>Number of CPU Chips</td>
<td>48,352</td>
<td>18,688</td>
</tr>
<tr>
<td>Number of GPU Chips</td>
<td>3,072</td>
<td>18,688</td>
</tr>
<tr>
<td>Amount of CPU Memory (TB)</td>
<td>1511</td>
<td>584</td>
</tr>
<tr>
<td>Interconnect</td>
<td>3D Torus</td>
<td>3D Torus</td>
</tr>
<tr>
<td>Amount of On-line Disk Storage (PB)</td>
<td>26</td>
<td>13.6</td>
</tr>
<tr>
<td>Sustained Disk Transfer (TB/sec)</td>
<td>&gt;1</td>
<td>0.4-0.7</td>
</tr>
<tr>
<td>Amount of Archival Storage</td>
<td>300</td>
<td>15-30</td>
</tr>
<tr>
<td>Sustained Tape Transfer (GB/sec)</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>
Disconnect between TOP500 and usable performance

LINPACK is a single test that solves $Ax=b$ with dense linear equations using Gaussian elimination with partial pivoting. For matrix $A$, that is size $M \times M$, LINPACK requires $\frac{2}{3} M^2 + 2M^2$ operations, $O(N^2)$ memory and $O(N^3)$ Floating Point operations.
TOP500 Issues

• The TOP500 gives no indication of the cost or value of a system
• The TOP500 encourages organizations to make poor choices
• The TOP500 provides little historical value
• The TOP500 is dominated by who has the most money to spend – not what system is the best.
• The Linpack TOP500 measure takes too long to run and does not represent strong scaling
• The TOP500 metric has not kept up with changing algorithmic methods.
• The TOP500 Linpack performance test is dominated by single-core, dense linear algebra peak performance
• There is no relationship between the TOP500 ranking and real work potential, user productivity, system usability for real applications. The TOP500 list disenfranchises many important application areas.
• The Linpack benchmark serves only one or two of the four purposes of a good benchmark.
<table>
<thead>
<tr>
<th>Science Area</th>
<th>Number of Teams</th>
<th>Codes</th>
<th>Struct Grids</th>
<th>Unstruct Grids</th>
<th>Dense Matrix</th>
<th>Sparse Matrix</th>
<th>N-Body</th>
<th>Monte Carlo</th>
<th>FFT</th>
<th>PIC</th>
<th>Significant I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Climate and Weather</td>
<td>3</td>
<td>CESM, GCRM, CM1/WRF, HOMME</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Plasmas/Magnetosphere</td>
<td>2</td>
<td>H3D(M), VPIC, OSIRIS, Magtail/UPIC</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Stellar Atmospheres and Supernovae</td>
<td>5</td>
<td>PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Cosmology</td>
<td>2</td>
<td>Enzo, pGADGET</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combustion/Turbulence</td>
<td>2</td>
<td>PSDNS, DISTUF</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Relativity</td>
<td>2</td>
<td>Cactus, Harm3D, LazEv</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td>4</td>
<td>AMBER, Gromacs, NAMD, LAMMPS</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantum Chemistry</td>
<td>2</td>
<td>SIAL, GAMESS, NWChem</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Material Science</td>
<td>3</td>
<td>NEMOS, OMEN, GW, QMCPACK</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Earthquakes/Seismology</td>
<td>2</td>
<td>AWP-ODC, HERCULES, PLSQR, SPECFEM3D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Quantum Chromo Dynamics</td>
<td>1</td>
<td>Chroma, MILC, USQCD</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Social Networks</td>
<td>1</td>
<td>EPISIMDEMICS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evolution</td>
<td>1</td>
<td>Eve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Engineering/System of Systems</td>
<td>1</td>
<td>GRIPS, Revisit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>Computer Science</td>
<td>1</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
CUDA-based cluster

- Each node contains $N$ GPUs
Message Passing Interface

- MPI is a standard message passing API

- Oriented to cluster machines
  - Distributed memory
  - Hides underlying interconnection network

- Processes execute on different nodes of a network
MPI Model

- Many processes distributed in a cluster
  - Node
  - Node
  - Node
  - Node

- Each process computes part of the output
- Processes communicate with each other
- Processes can synchronize
MPI Message Types

- **Point-to-point communication**
  - Send and Receive

- **Collective communication**
  - Barrier
  - Broadcast
  - Reduce
  - Gather and Scatter
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

- **int MPI_Barrier (MPI_Comm comm)**
  - Blocks the caller until all group members have called it; returns at any process only after all group members have entered the call
Vector Addition: Main Process

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

```c
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)
MPI Send and Receive Data

```c
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)
```

- **send/recvbuf**: Initial address of send/receive buffer (choice)
- **send/recvcount**: Number of elements in send/receive buffer (integer)
- **send/recvtype**: Datatype of each send/receive buffer element (handle)
- **dest**: Rank of destination (integer)
- **source**: Rank of source (integer)
- **send/recvtag**: Send/receive tag (integer)
- **comm**: Communicator (handle)
- **status**: Status object (Status). This refers to the receive operation.
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 and output 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)

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

for(int process = 1; process < last_node; 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;
}
```
// Collect output data
MPI_Status status;
for(int process = 0; process < num_nodes; process++) {
    MPI_Recv(output + process * num_points / num_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);
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);
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];
}

// 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 main challenge of MPI + CUDA is not integrating the two but overlapping communication and computation.
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_stencil(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
- 3D-Stencil introduces data dependencies
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 = 0;

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

  /* Allocate input data */
  input = (float *)malloc(num_bytes);
  output = (float *)malloc(num_bytes);
  if(input == NULL || output == NULL) {
    printf("server couldn't allocate memory\n");
    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;
}
/* Send data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
         DATA_DISTRIBUTED, 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_REAL, process,
             DATA_DISTRIBUTED, 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_REAL, last_node,
         DATA_DISTRIBUTED, 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_REAL, process,
              DATA_COLLECT, MPI_COMM_WORLD, &status);

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

/* Release resources */
free(input);
free(output);
Boundary Exchange Example (I)

- **Approach**: two-stage execution
  - **Stage 1**: compute the field points to be exchanged
Boundary Exchange Example (II)

- **Approach:** two-stage execution
  - Stage 2: Compute the remaining points *while* exchanging the boundaries
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);
    int left_stage1_offset = 0;
    int right_stage1_offset = dimx * dimy * (dimz - 4);
    int stage2_offset = num_ghost_points;
Stencil Code: Compute Process (II)

```c
float *h_input = NULL, *h_output = NULL;
float *d_input = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_left_ghost_own = NULL, *h_right_ghost_own = NULL;
float *h_left_ghost = NULL, *h_right_ghost = NULL;

/* Alloc host memory */
h_input = (float *)malloc(num_bytes);
h_output = (float *)malloc(num_bytes);

/* Alloc pinned host memory for ghost data */
cudaMallocHost((void**)&h_left_ghost_own, num_ghost_bytes);
cudaMallocHost((void**)&h_right_ghost_own, num_ghost_bytes);
cudaMallocHost((void**)&h_left_ghost, num_ghost_bytes);
cudaMallocHost((void**)&h_right_ghost, num_ghost_bytes);

/* Alloca device memory for input and output data */
cudaMalloc((void**)&d_input, num_bytes);
cudaMalloc((void**)&d_output, 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 main process */
float *rcv_address = h_input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_REAL, server_process,
          DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);

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

/* Create streams used for stencil computation */
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
// MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; i < nreps; i++) {
    /* Compute values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset, 
            d_input + left_stage1_offset, dimx, dimy, 12, stream1);
    launch_kernel(d_output + right_stage1_offset, 
            d_input + right_stage1_offset, dimx, dimy, 12, stream1);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset, 
            dimx, dimy, dimz, stream2);

    /* Copy the data needed by other nodes to the host */
    cudaMemcpyAsync(h_left_ghost_own, 
                        d_output + num_ghost_points, 
                        num_ghost_bytes, cudaMemcpyDeviceToHost, stream1 );
    cudaMemcpyAsync(h_right_ghost_own, 
                        d_output + right_stage1_offset + num_ghost_points, 
                        num_ghost_bytes, cudaMemcpyDeviceToHost, stream1 );
    cudaStreamSynchronize(stream1);
Stencil Code: Compute Process (V)

/* Send data to left, get data from right */
MPI_Sendrecv(h_left_ghost_own, num_ghost_points, MPI_REAL,
             left_neighbor, i, h_right_ghost,
             num_ghost_points, MPI_REAL, right_neighbor, i,
             MPI_COMM_WORLD, &status);

/* Send data to right, get data from left */
MPI_Sendrecv(h_right_ghost_own, num_ghost_points, MPI_REAL,
             right_neighbor, i, h_left_ghost,
             num_ghost_points, MPI_REAL, left_neighbor, i,
             MPI_COMM_WORLD, &status);

cudaMemcpyAsync(d_output+left_ghost_offset,  h_left_ghost,
                num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
                num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
cudaDeviceSynchronize();

float *temp = d_output;
d_output = d_input; d_input = temp;
}
/* Wait for previous communications */
// MPI_BARRIER(MPI_COMM_WORLD);

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

/* Send the output, skipping ghost 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);
}
Without GPU Direct

- There is an internal copy (not seen by the user) between CUDA buffers and Infiniband buffers.
With GPU Direct

- There is no internal copy, increasing performance
- The program code remains unchanged