CS/EE 217 GPU Architecture and Parallel Programming

Lecture 21: Joint CUDA-MPI Programming

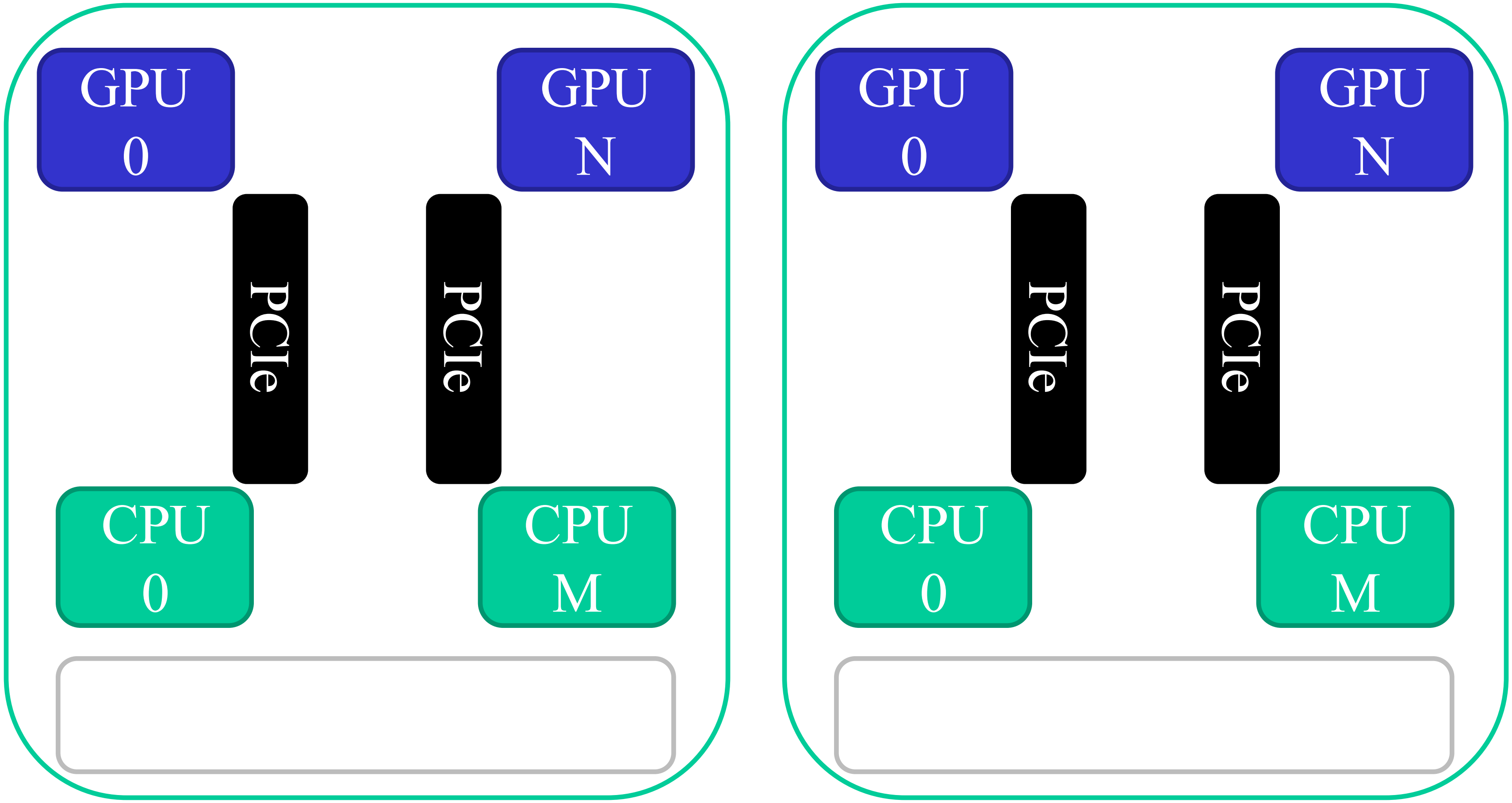
Objective

- To become proficient in writing simple joint MPI-CUDA heterogeneous applications
  - Understand the key sections of the application
  - Simplified code and efficient data movement using GMAC
  - One-way communication

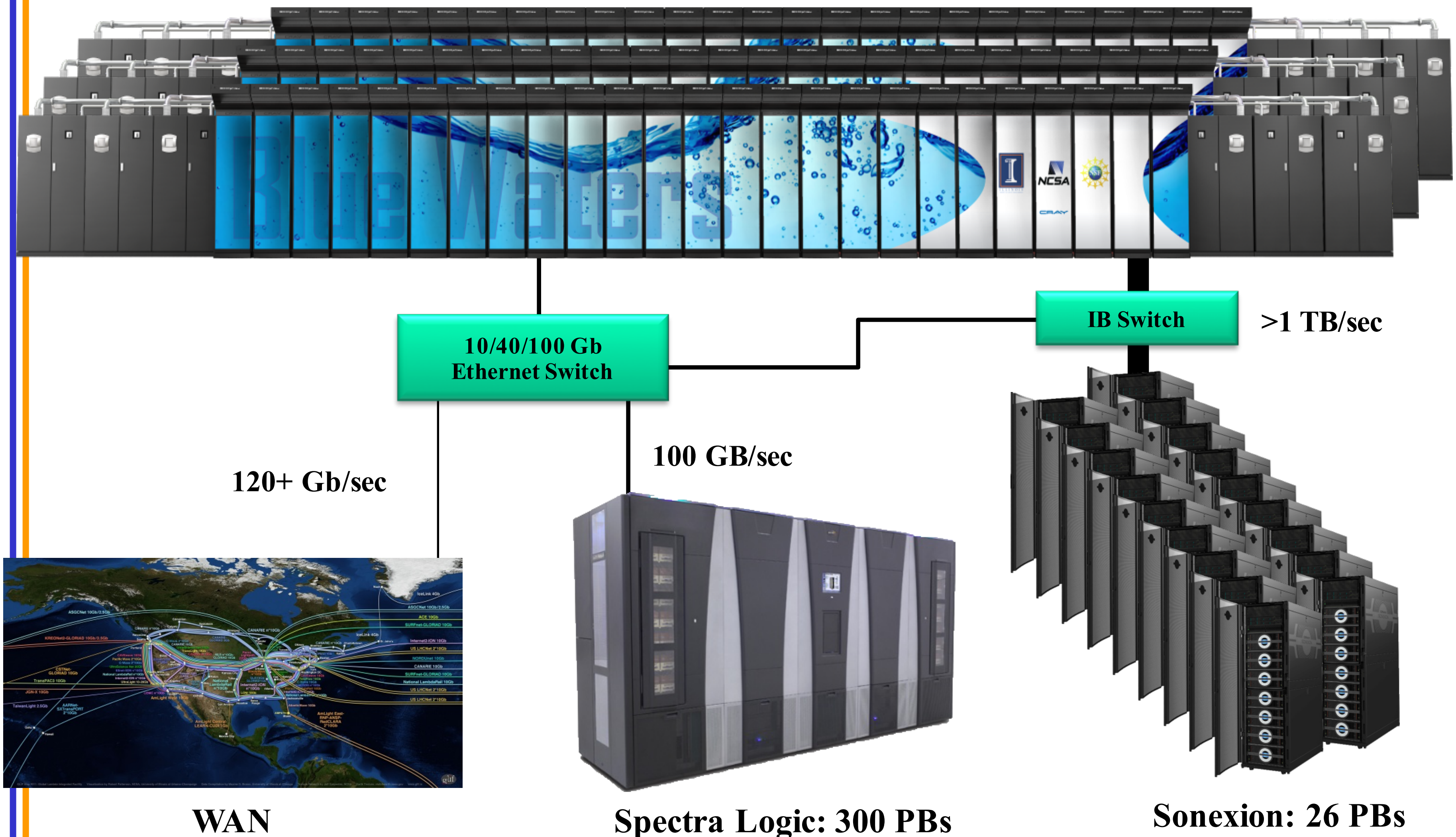
- To become familiar with a more sophisticated MPI application that requires two-way data-exchange

CUDA-based cluster

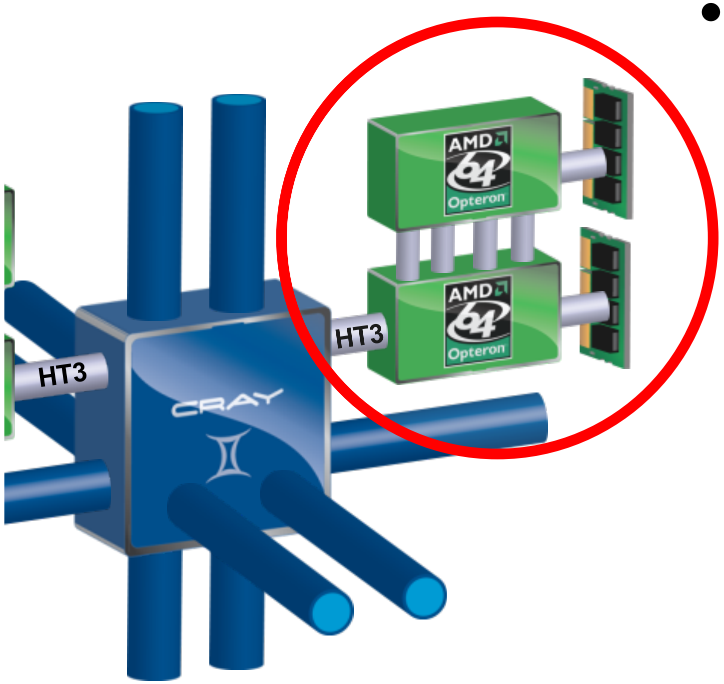
- Each node contains *N* GPUs



Blue Waters Computing System



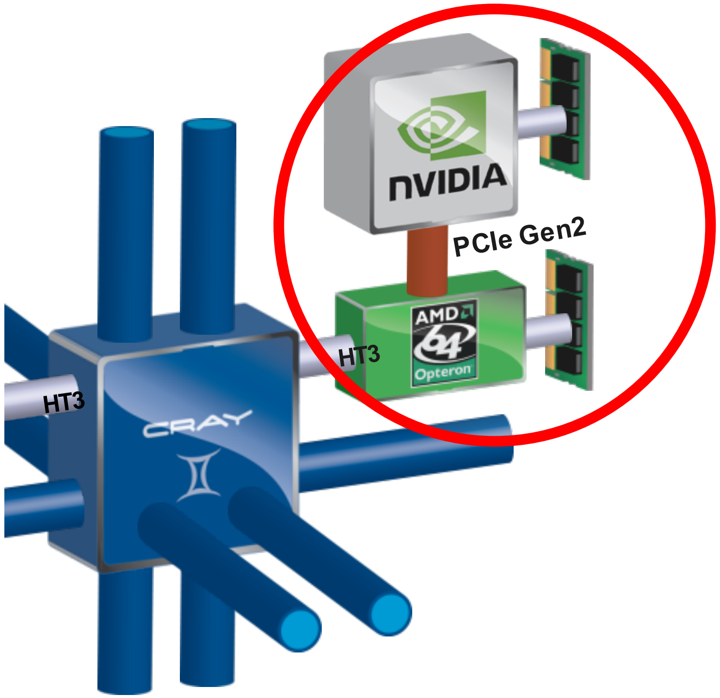
Cray XE6 Nodes



Blue Waters contains 22,640 Cray XE6 compute 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

Cray XK7 Nodes

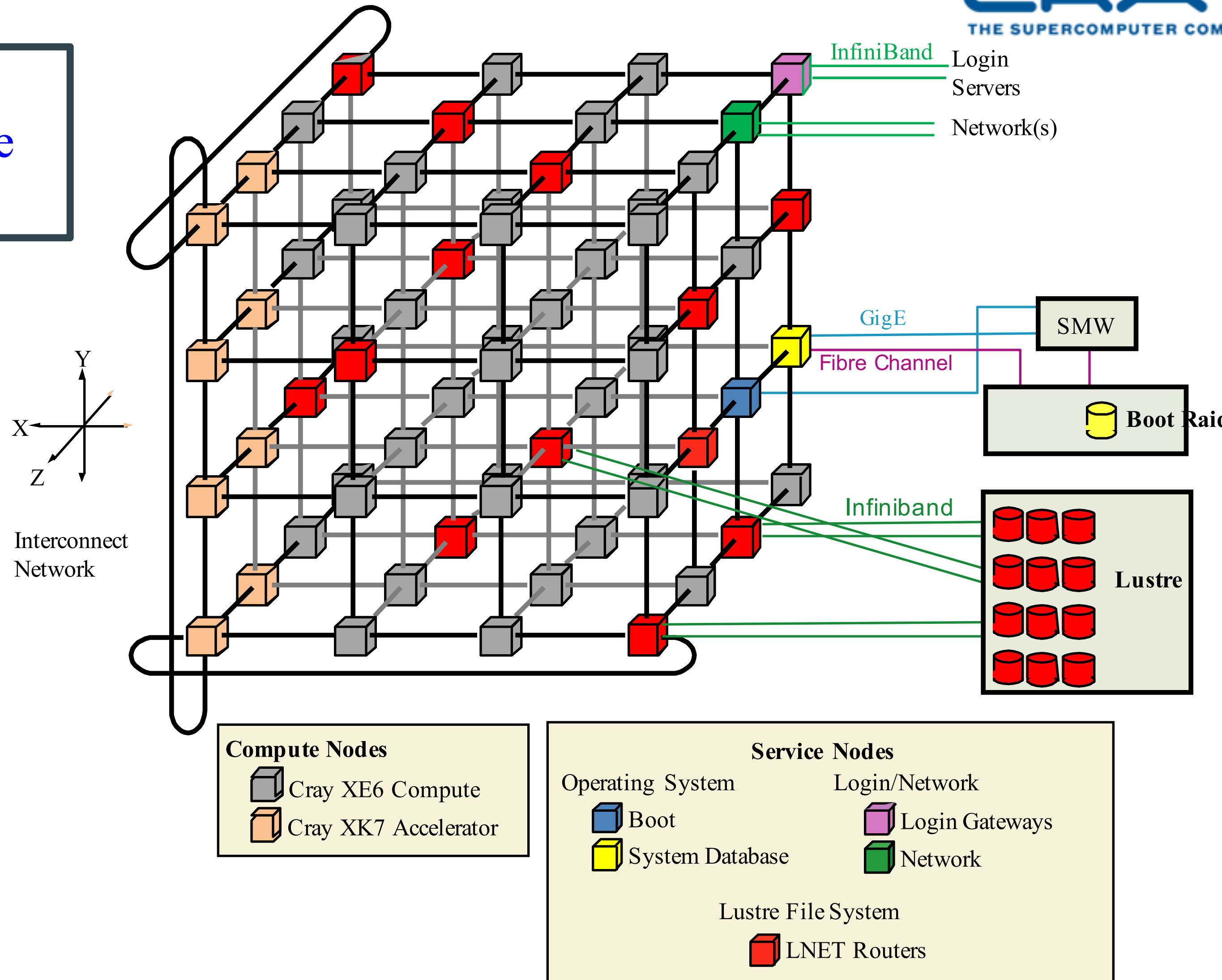


- Dual-socket Node
  - One AMD Interlagos chip
    - 8 core modules, 32 threads
    - 156.5 GFs peak performance
    - 32 GBs memory
      - 51 GB/s bandwidth
  - One NVIDIA Kepler chip
    - 1.3 TFs peak performance
    - 6 GBs GDDR5 memory
      - 250 GB/sec bandwidth
  - Gemini Interconnect
    -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

Blue Waters contains 3,072 Cray XK7 compute nodes.

Gemini Interconnect Network

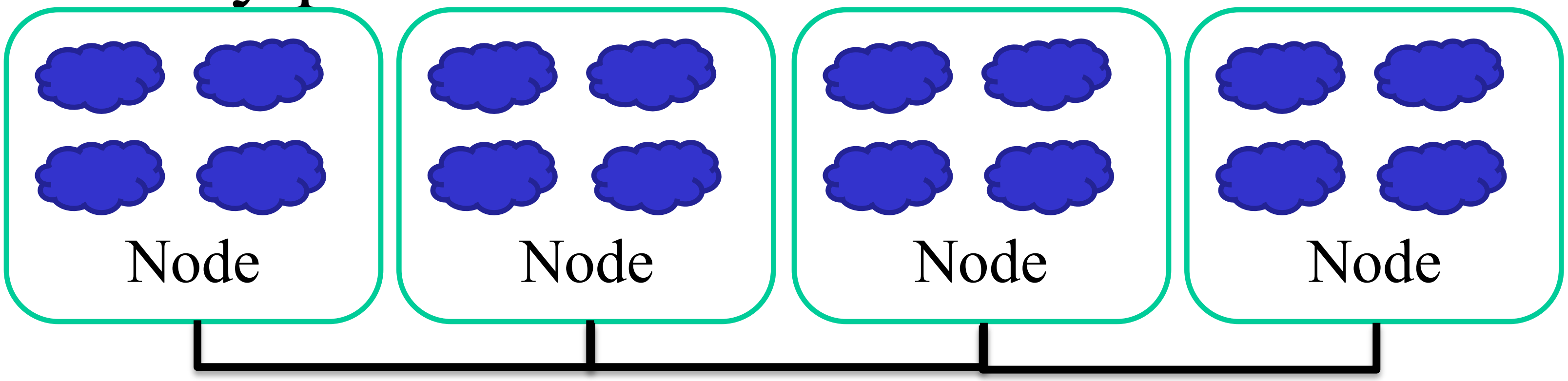
Blue Waters
3D Torus Size
23 x 24 x 24



Science Area	Number of Teams	Codes	Struct Grids	Unstruct Grids	Dense Matrix	Sparse Matrix	N-Body	Monte Carlo	FFT	PIC	Significant I/O
Climate and Weather	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
Plasmas/Magnetosphere	2	H3D(M),VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
Stellar Atmospheres and Supernovae	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
Cosmology	2	Enzo, pGADGET	X			X	X				
Combustion/Turbulence	2	PSDNS, DISTUF	X						X		
General Relativity	2	Cactus, Harm3D, LazEV	X			X					
Molecular Dynamics	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
Quantum Chemistry	2	SIAL, GAMESS, NWChem			X	X	X	X			X
Material Science	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
Earthquakes/Seismology	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
Quantum Chromo Dynamics	1	Chroma, MILC, USQCD	X		X	X					
Social Networks	1	EPISIMDEMICS									
Evolution	1	Eve									
Engineering/System of Systems	1	GRIPS,Revisit						X			
Computer Science	1			X	X	X			X		X

MPI Model

- 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

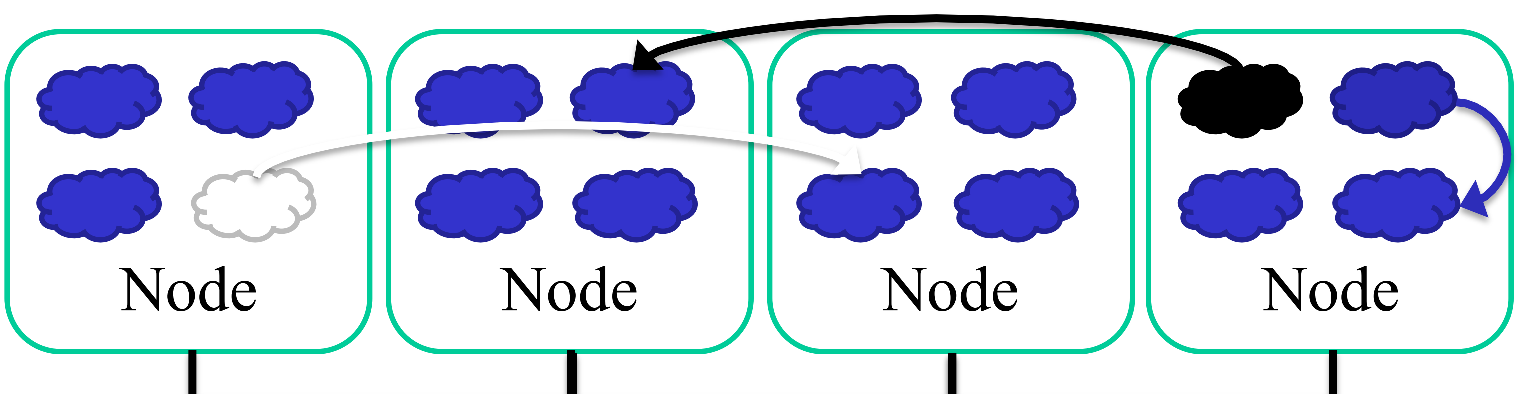
```
int main(int argc, char *argv[]) {
    int vector_size = 1024 * 1024 * 1024;
    int pid=-1, np=-1;

    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;
    }
    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)

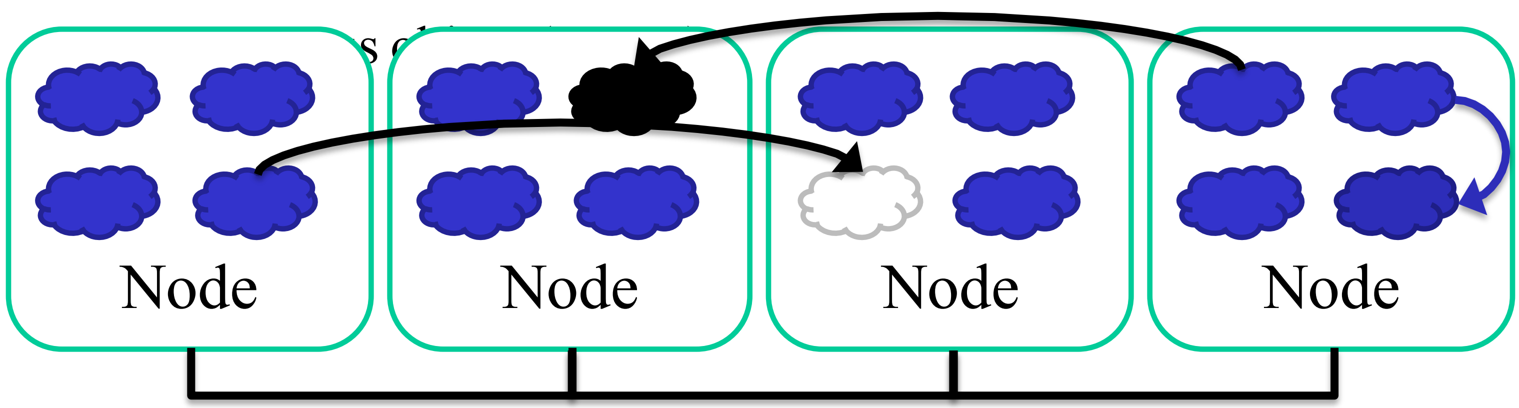


MPI Receiving Data

- `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)



Vector Addition: Server Process (I)

```
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 */
      *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;
}

/* 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 * 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);
}
```

Vector Addition: Compute Process (I)

```
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);
}
```

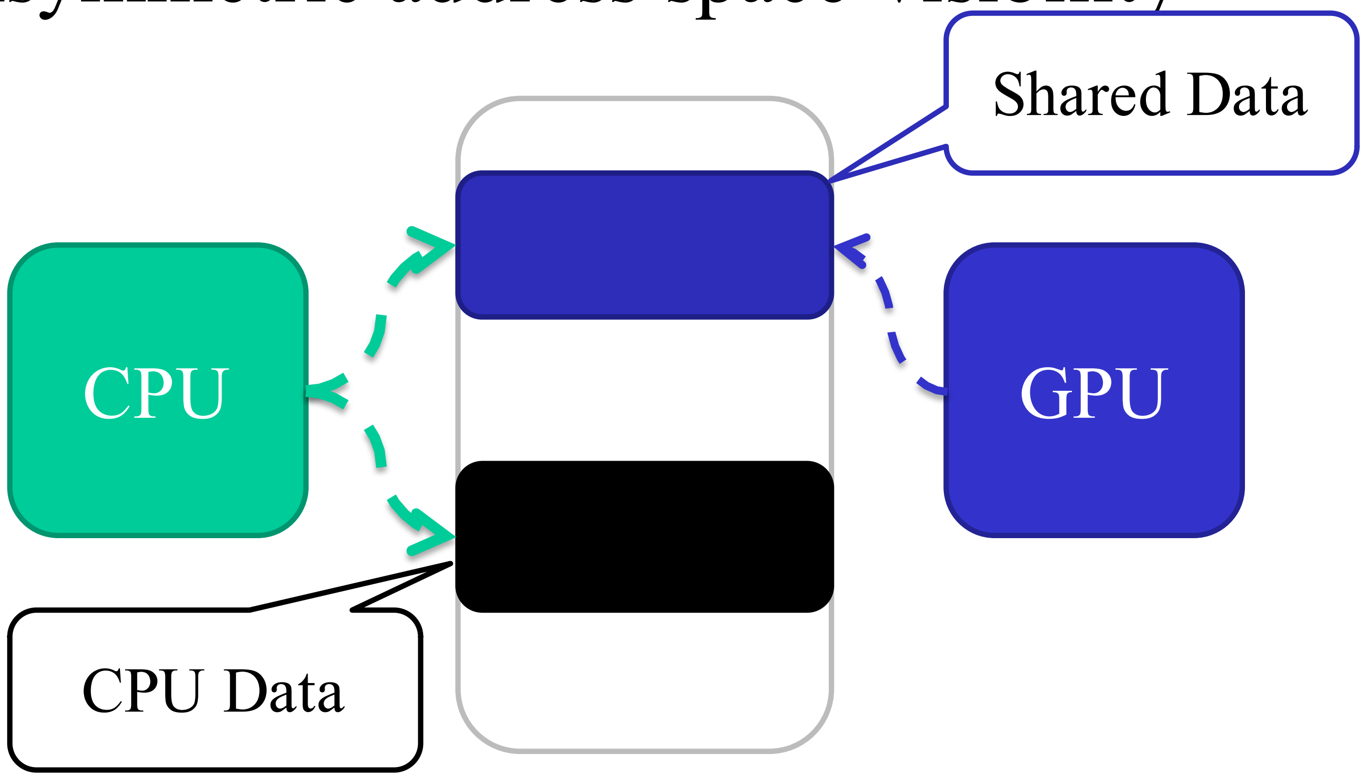
DETOUR: GLOBAL MEMORY FOR ACCELERATORS

GMAC

- User level library to support shared address space between CPU and accelerator
  - Hides the details of data transfer to and from the accelerator from the programmer
- Designed to integrate accelerators with MPI

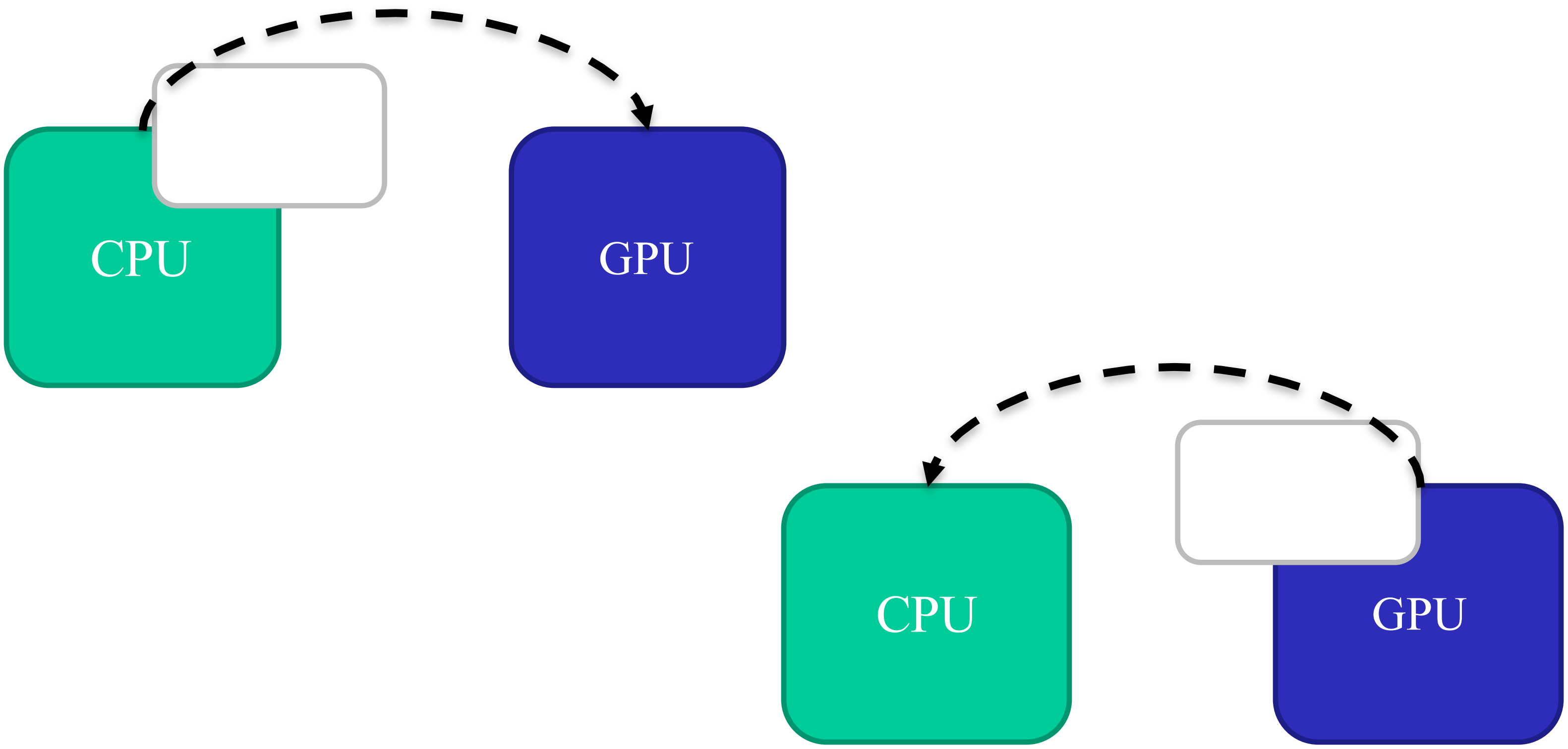
GMAC Memory Model

- Unified CPU / GPU virtual address space
- Asymmetric address space visibility



ADSM Consistency Model

- Implicit acquire / release primitives at accelerator call / return boundaries



GMAC Memory API

- Memory allocation

```
gmacError_t gmacMalloc(void **ptr, size_t size)
```

  – Allocated memory address (returned by reference)
  – Gets the size of the data to be allocated
  – Error code, `gmacSuccess` if no error

- Example usage

```
#include <gmac/cuda.h>

int main(int argc, char *argv[]) {
   float *foo = NULL;
   gmacError_t error;
   if((error = gmacMalloc((void **)&foo, FOO_SIZE)) != gmacSuccess)
      FATAL("Error allocating memory %s", gmacErrorString(error));
   . . .
}
```

GMAC Memory API

- Memory release

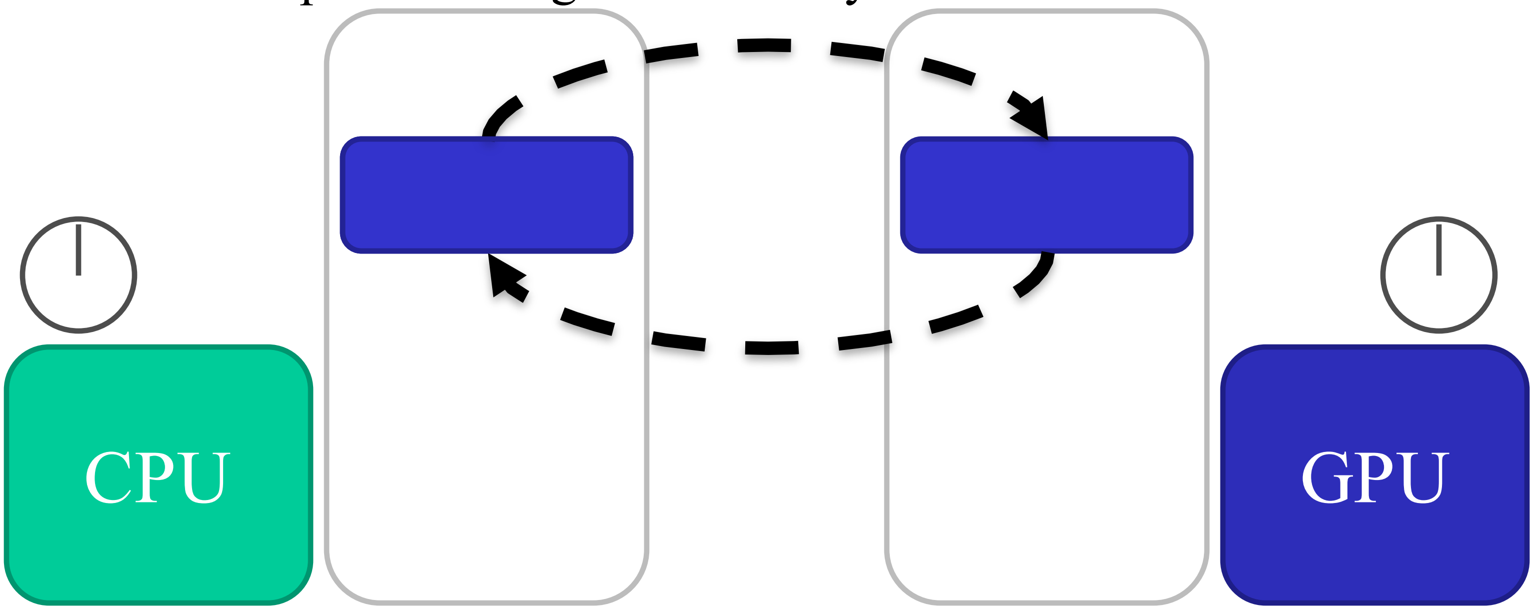
```
gmacError_t gmacFree(void *ptr)
```

  - Memory address to be release
  - Error code, `gmacSuccess` if no error
- Example usage

```
#include <gmac/cuda.h>

int main(int argc, char *argv[]) {
   float *foo = NULL;
   gmacError_t error;
   if((error = gmacMalloc((void **)&foo, FOO_SIZE)) != gmacSuccess)
      FATAL("Error allocating memory %s", gmacErrorString(error));
   . . .
   gmacFree(foo);
}
```

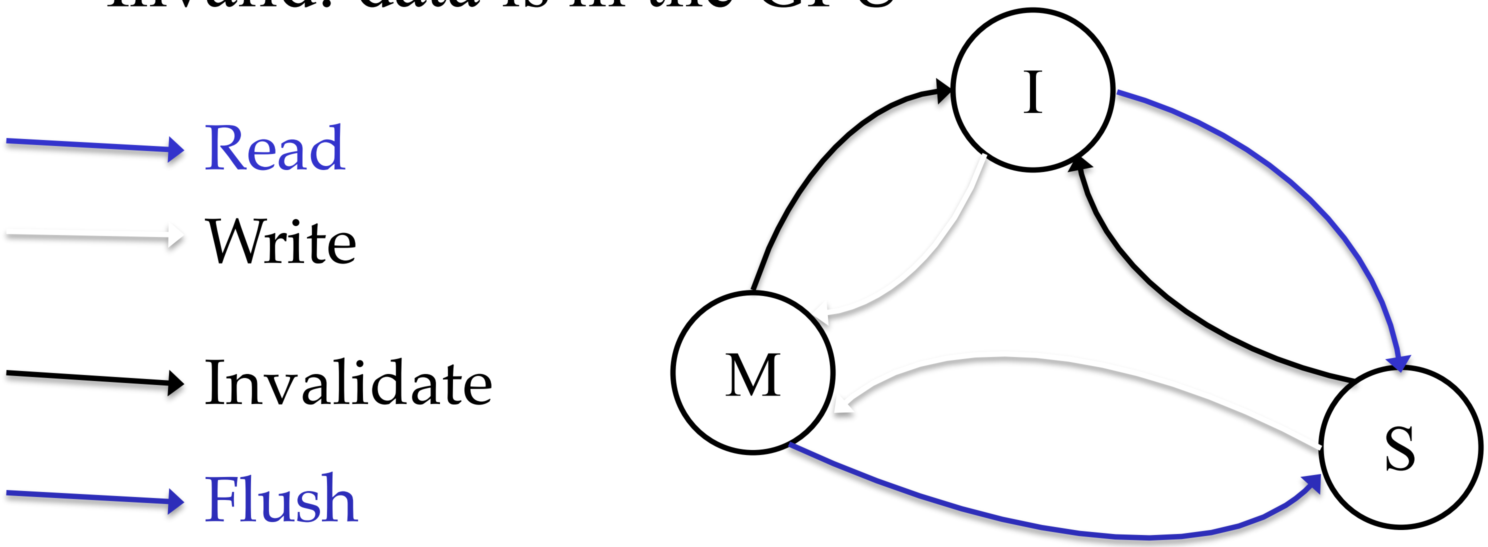
ADSM Eager Update

- Asynchronous data transfers while the CPU computes
- Optimized data transfers:
  - Memory block granularity
  - Avoid copies on Integrated GPU Systems



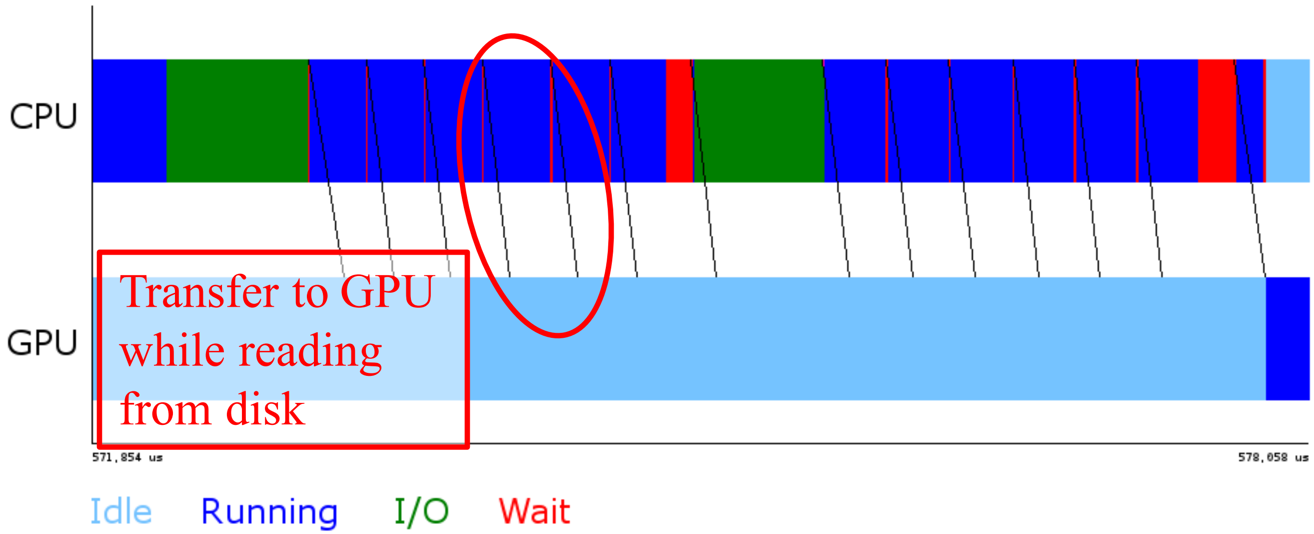
ADSM Coherence Protocol

- Only meaningful for the CPU
- Three state protocol:
  - Modified: data in the in the CPU
  - Shared: data is in both, CPU and GPU
  - Invalid: data is in the GPU



GMAC Optimizations

- Global Memory for Accelerators : user-level ADSM run-time
- Optimized library calls: memset, memcpy, fread, fwrite, MPI_send, MPI_receive
- Double buffering of data transfers



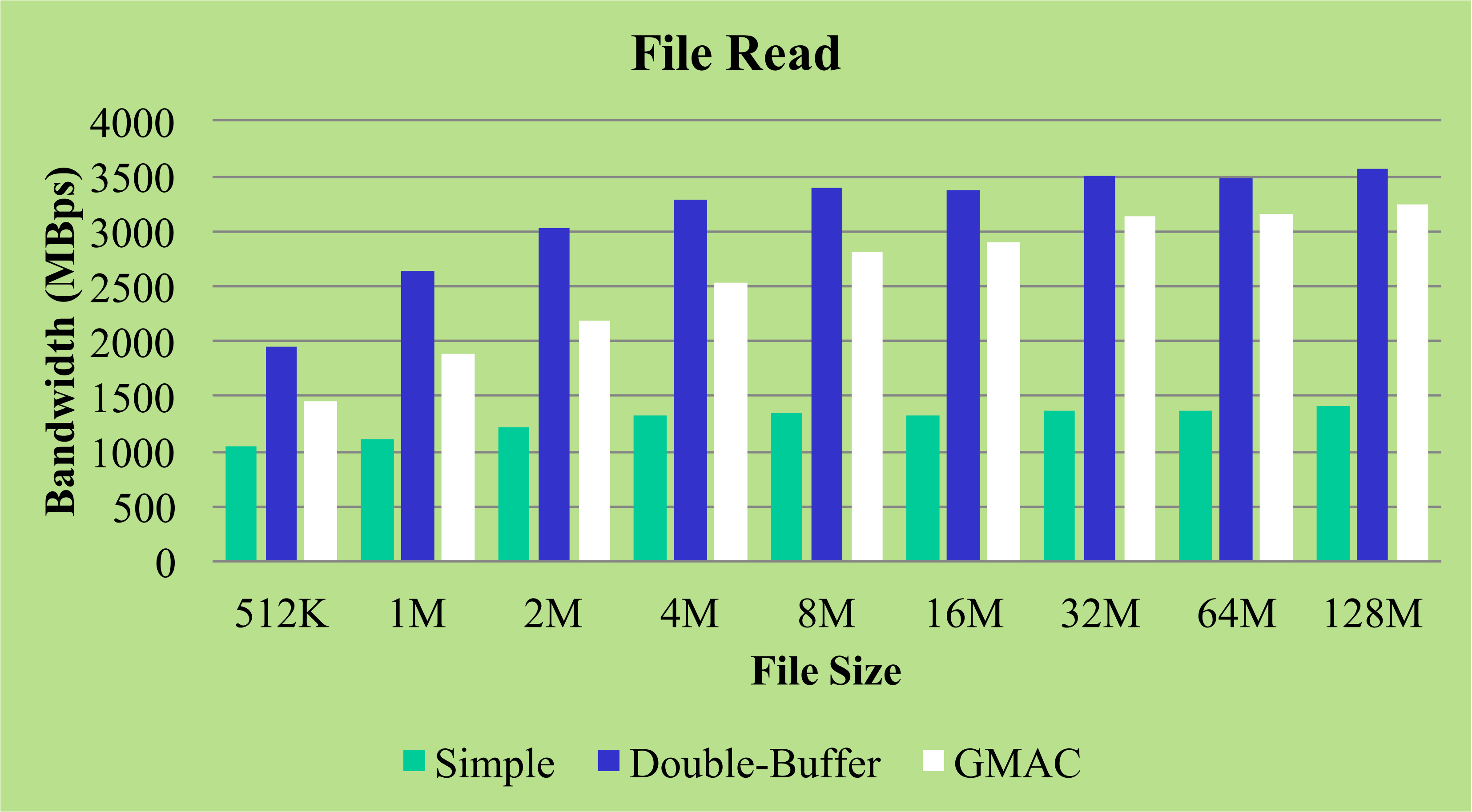
GMAC in Code (I)

```
int main(int argc, char *argv[])
{
    FILE *fp;
    struct stat file_stat;
    gmacError_t gmac_ret;
    void *buffer;
    timestamp_t start_time, end_time;
    float bw;

    if(argc < LAST_PARAM) FATAL("Bad argument count");

    if((fp = fopen(argv[FILE_NAME], "r")) < 0)
        FATAL("Unable to open %s", argv[FILE_NAME]);

    if(fstat(fileno(fp), &file_stat) < 0)
        FATAL("Unable to read meta data for %s", argv[FILE_NAME]);
```

GMAC in Code (I)

```
    gmac_ret = gmacMalloc(&buffer, file_stat.st_size);
    if(gmac_ret != gmacSuccess) FATAL("Unable to allocate memory");

    start_time = get_timestamp();
    if(fread(buffer, 1, file_stat.st_size, fp) < file_stat.st_size)
        FATAL("Unable to read data from %s", argv[FILE_NAME]);
    gmac_ret = gmacThreadSynchronize();
    if(gmac_ret != gmacSuccess) FATAL("Unable to wait for device");
    end_time = get_timestamp();
    bw = 1.0f * file_stat.st_size / (end_time - start_time);
    fprintf(stdout, "%d bytes in %f msec : %f MBps\n", file_stat.st_size,
            1e-3f * (end_time - start_time), bw);
    gmac_ret = gmacFree(buffer);
    if(gmac_ret != gmacSuccess) FATAL("Unable to free device memory");
    fclose(fp);
    return 0;
}
```

Performance of GMAC



ADDING CUDA TO MPI

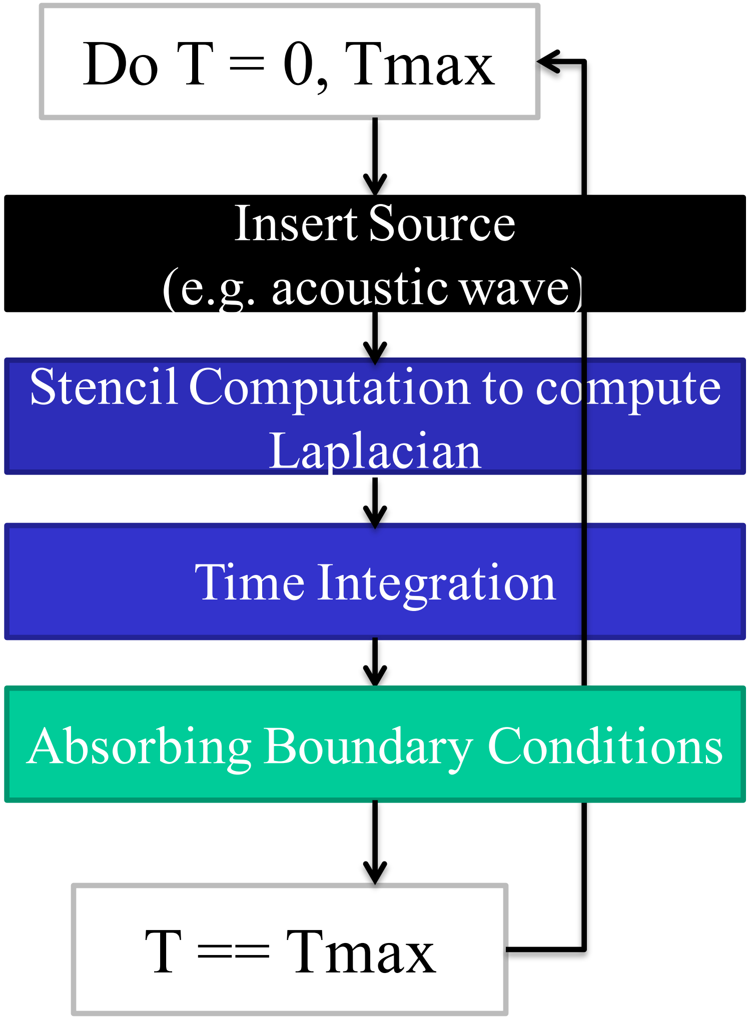
Vector Addition: CUDA Process (I)

```
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;

    /* Allocate memory */
    gmacMalloc((void **)&input_a, num_bytes);
    gmacMalloc((void **)&input_b, num_bytes);
    gmacMalloc((void **)&output, 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: CUDA Process (II)

```
    /* Compute the partial vector addition */
    dim3 Db(BLOCK_SIZE);
    dim3 Dg((vector_size + BLOCK_SIZE - 1) / BLOCK_SIZE);
    vector_add_kernel<<<Dg, Db>>>(gmacPtr(output), gmacPtr(input_a),
                                  gmacPtr(input_b), vector_size);
    gmacThreadSynchronize();

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

    /* Release device memory */
    gmacFree(d_input_a);
    gmacFree(d_input_b);
    gmacFree(d_output);
}
```

A Typical Wave Propagation Application

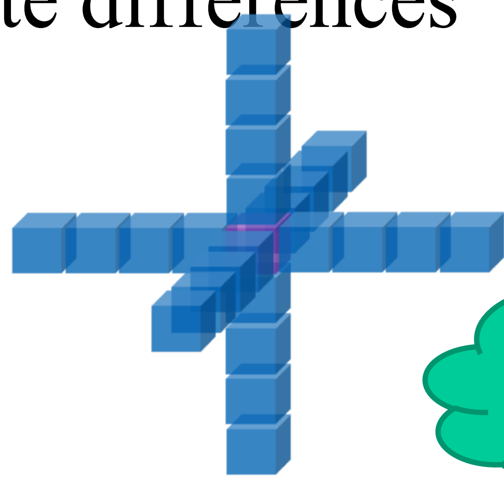


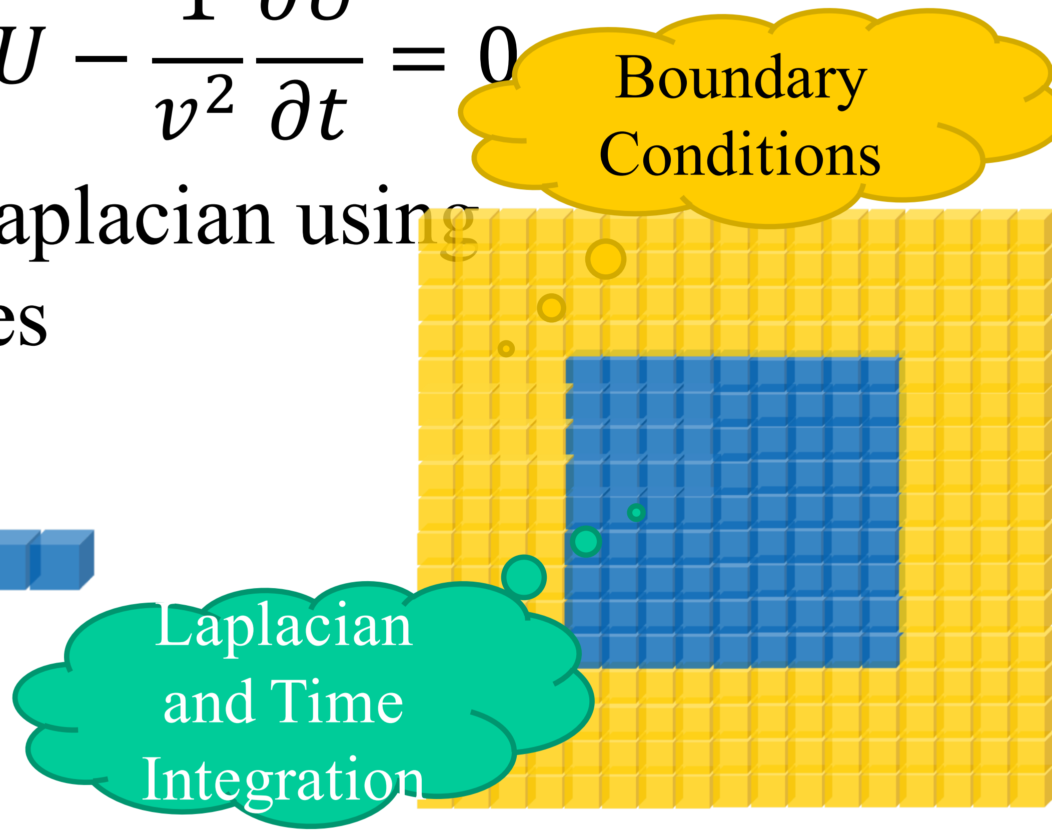
Review of Stencil Computations

- Example: wave propagation modeling

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

- Approximate Laplacian using finite differences





Wave Propagation: Kernel Code

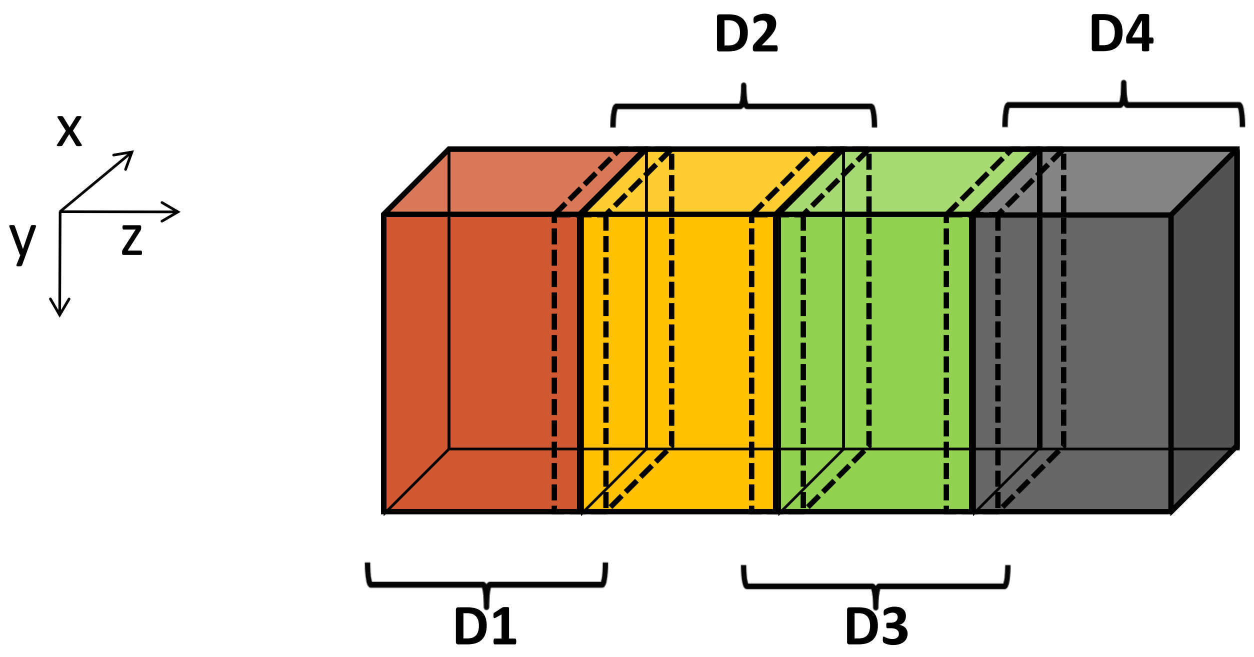
```
/* Coefficients used to calculate the laplacian */
__constant__ float coeff[5];

__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

```
        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



Wave Propagation: Main Process

```
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("Nedded 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;
}
```

Stencil Code: Server Process (I)

```
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(     );
          *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);
```

Stencil Code: Server Process (II)

```
/* 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);
      *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 (II)

```
        *velocity_send_address = velocity;

    /* Send velocity data to compute nodes */
    for(int process = 0; process < last_node + 1; process++) {
        MPI_Send(send_address, edge_num_points, MPI_FLOAT, process,
                DATA_DISTRIBUTE, MPI_COMM_WORLD);
        send_address += dimx * dimy * (dimz / num_comp_nodes);
    }

    /* 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 (III)

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

    /* Release resources */
    free(input);
    free(velocity);
    free(output);
}
```

Stencil Code: Compute Process (I)

```
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);

          *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);
```

Stencil Code: Compute Process (II)

```
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 */
      *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 );
```

QUESTIONS?