GPU Teaching Kit

Accelerated Computing

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Module 18 – Related Programming Models: MPI

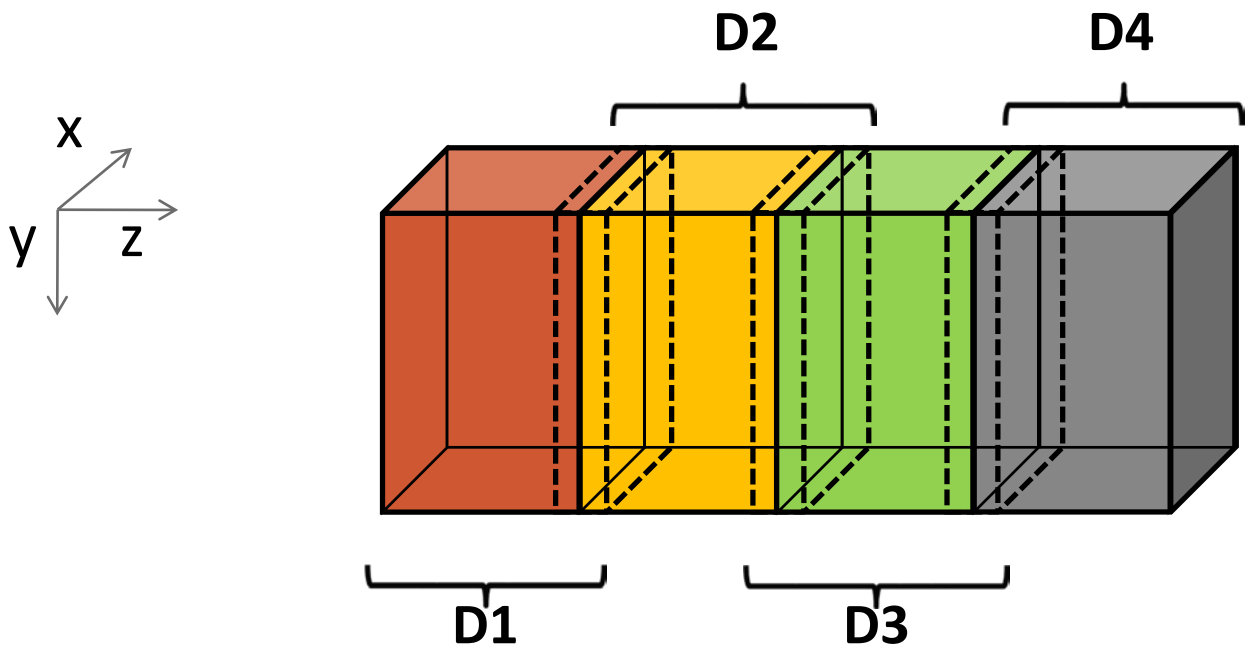
Lecture 18.3 – Overlapping Computation with Communication

Ojective

- 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

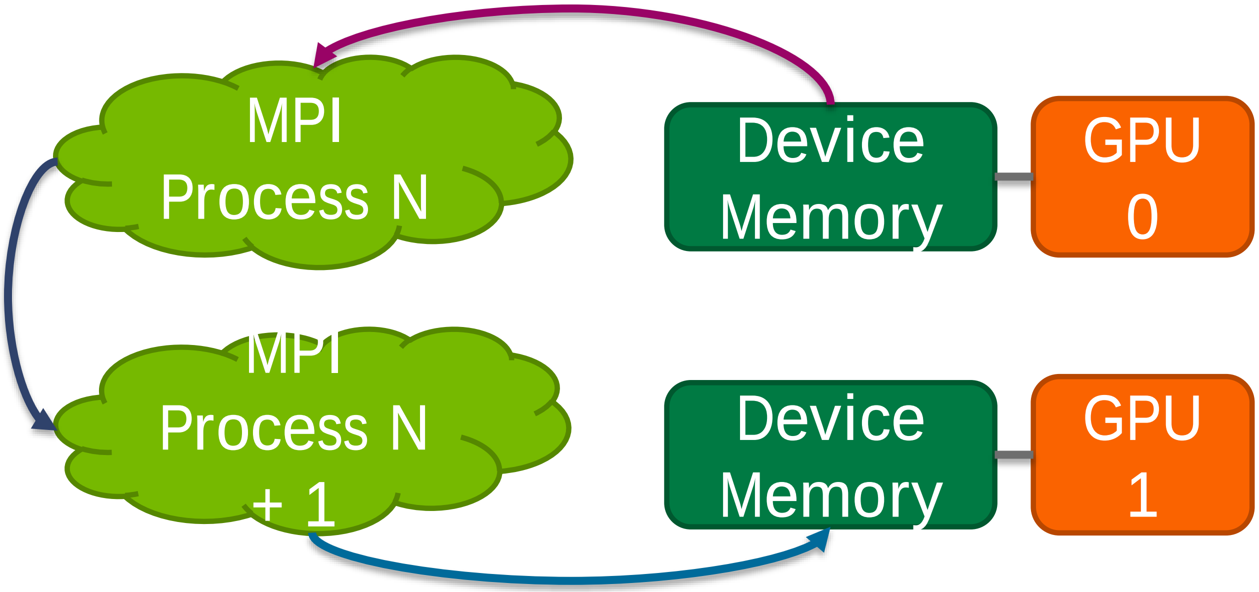
Stencil Domain Decomposition

- 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)



Data Server Process Code (I)

```
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\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;
```

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).

```
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);
        /* Alloca 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);
```

Stencil Code: Kernel Launch

```
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)

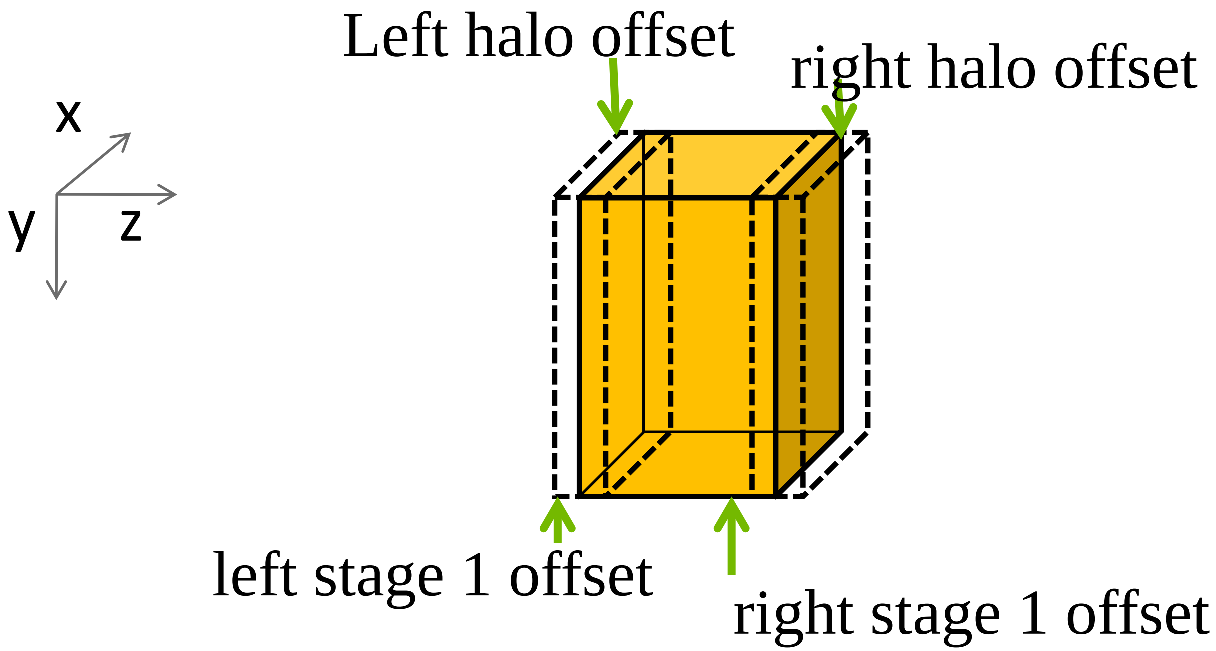
```
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_output = (float *)malloc(num_bytes);
cudaMalloc((void **)&d_output, num_bytes );

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, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes, cudaHostAllocDefault);

/* Create streams used for stencil computation */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```

Device Memory Offsets Used for Data Exchange with Neighbors



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 cofficients */
upload_coefficients(coeff, 5);

int left_halo_offset   = 0;
int right_halo_offset  = dimx * dimy * (4 + dimz);
int left_stage1_offset  = 0;
int right_stage1_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_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream0);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_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);
```

Syntax for MPI_Sendrecv()

– 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 (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

GPU Teaching Kit

Accelerated Computing