Heterogeneous Computing Fundamentals

CSInParallel Project

July 26, 2012

CONTENTS

Contents:

CHAPTER

ONE

PREREQUISITES

Before getting started with this module, please keep in mind that we assume you have **sufficient knowledge with C programming with Message Passing Interface(MPI) and CUDA C.** If you are not familiar with MPI or CUDA C, please read the **GPU Computing module**, and **Distributed Memory Computing module**.

CHAPTER

TWO

INTRODUCTION TO HETEROGENEOUS COMPUTING

You have already seen that you can decompose the task and distribute each smaller task to each worker, and let each node work simultaneously in an MPI program. Thus, it enables a programmer to work on a larger problem size, and reduce the computational time. Similarly, CUDA C is a GPU programming language, and is another parallel programming model. Our goal of this module is to program in the hybrid environment CUDA and MPI.

It turns out that we can produce a new parallel programming model that combines MPI and CUDA together. The idea is that we use MPI to distribute work among nodes, each of which uses CUDA to work on its task. In order for this to work, we need to keep in mind that we need to have a cluster or something similar, where each node contains GPUs capable of doing CUDA computation.

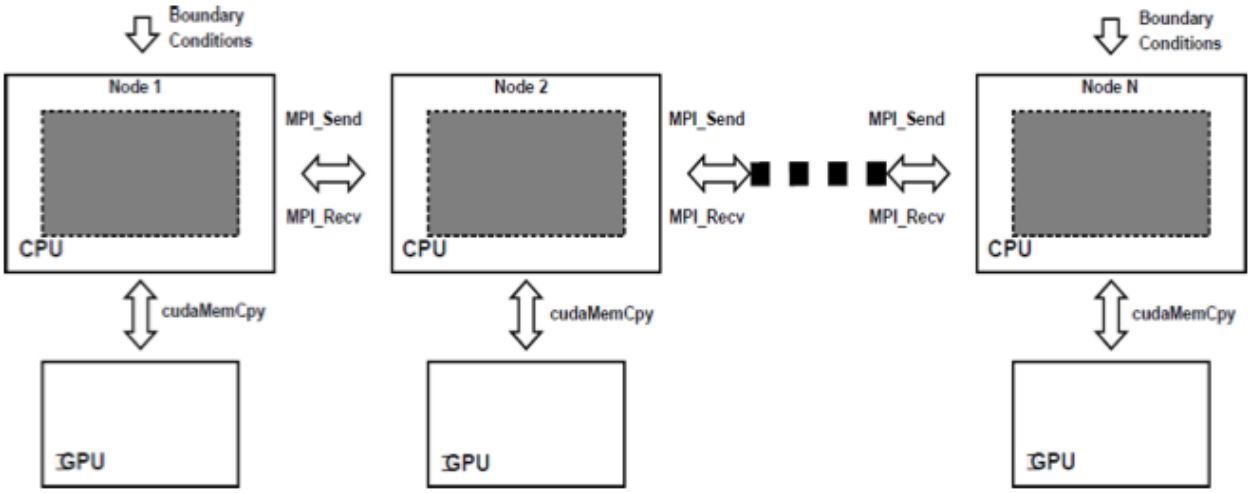

Figure 1: Heterogeneous Programming Model: CUDA and MPI from cacuda.googlecode.com [1]

Recommended Reading

- Please Do Your Reading: On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters

References

CHAPTER

THREE

CODING AND COMPILING A HETEROGENEOUS PROGRAM

3.1 Heterogeneous Program: Hello World

Distributed memory computing and GPU computing are two different parallel programming models. In this section, you will learn how to put these two parallel models together, and that will speed up your running time. As always we will look at the **Hello World** program using hybrid CUDA and MPI model. In order to combine CUDA and MPI, we need to get their codes to communicate to each other during the compilation. Let's look at the **Hello World** program below.

CUDA program

```
#include <stdio.h>
#include <cuda.h>

/* kernel function for GPU */
__global__ void kernel(void) {
}

extern "C" void hello(void) {
    kernel<<<1, 1>>>();
    printf("Hello World !\n");
}
```

MPI program integrated with CUDA

```
#include <mpi.h>

#define MAX 80    /* maximum characters for naming the node */

/* Declaring the CUDA function */
void hello();

int main(int argc, char *argv[]) {

   int rank, nprocs, len;
   char name[MAX];      /* char array for storing the name of each node */

   /* Initializing the MPI execution environment */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
17      MPI_Get_processor_name(name, &len);
18
19      /* Call CUDA function */
20      hello();
21
22      /* Print the rank, size, and name of each node */
23      printf("I am %d of %d on %s\n", rank, size, name);
24
25      /*Terminating the MPI environment*/
26      MPI_Finalize();
27      return 0;
28  }
```

Comments

- From source codes above, CUDA program creates a grid consisting a block, which has a single thread. It will print "Hello World !". The **hello** function in CUDA program uses the keyword **extern "C"**, so the MPI program is able to link to use **hello** function using a 'C' compatible header file that contains just the declaration of **hello** function. In addition, MPI program only creates the MPI execution environment, defines the size of the MPI_COMM_WORLD, gives the unique rank to each process, calls **hello** function from CUDA program to print "Hello World !", and prints the rank, size, and name of the process. Finally, all processes terminate the MPI execution environment.

3.2 Compiling a Heterogeneous Program

The most common way of compiling a heterogeneous program MPI and Cuda is:

1. Make a CUDA object from the CUDA program. This can be done by using this command on the terminal:

   ```
   nvcc -c cuda.cu -o cuda.o
   ```

2. Make an MPI object from MPI program. This can be done by using this command on the terminal:

   ```
   mpicc -c mpi.c -o mpi.o
   ```

3. Make an executable file from both objects. This can be done by using this command on the terminal:

   ```
   mpicc -o cudampi mpi.o cuda.o -L/usr/local/cuda/lib64 -lcudart
   ```

To execute the executable file, **cudampi**, we can enter the following command on the terminal:

```
mpirun -machinefile machines -x LD_LIBRARY_PATH -np #processes ./cudampi
```

3.2.1 Timing a Heterogeneous CUDA and MPI

- In order to time your hybrid CUDA and MPI program, you just need to use MPI_Wtime() function as in an MPI program.
- We need to keep in mind that a heterogeneous CUDA and MPI program theoretically has lower running time than an MPI does; however, running time also depends on each node's properties such as memory. Copying data from a CPU to GPU may take a long period of time, which results in a much longer running time for a heterogeneous program. Therefore, you do not always get benefits from the heterogeneous programming model.

3.2.2 Activity 1: Vector Addition

In this activity, we are going to compute vector addition by using hybrid programming model, CUDA and MPI. Vector addition is very simple and easy. Suppose we have vector A and vector B, and both have the same length. To add vector A and B, we just add the corresponding elements of A and B. This results a new vector of the same length.

Comments on CUDA Program

- We will walk you through this first activity step by step. First, let's look at the CUDA program for vector addition. We need to have a kernel function for vector addition. This should be straight forward to you. Each thread computes an element of the result matrix, where thread index is the index of that element.

```
__global__ void kernel(int *a, int *b, int *c) {
        /* this thread index is the index of the vector */
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        // TO DO
        // add corresponding elements of a and b
        // end TO DO
}
```

- Another function in the CUDA program is **run_kernel**, which works on the host(CPU) and calls the kernel function on the device(GPU). This function allocates memory on the GPU for storing input vectors, copies input vectors onto the device, does the calculations on the device, copies output vector back to the host, and erases all those vectors on the device. This function will be called in the MPI program.

```
/*
* size is the number of elements in the vector
* nblocks is the number of blocks per grid
* nthreads is the number of threads per block
*/
extern "C" void run_kernel(int *a, int *b, int *c, int size, int nblocks, int nthreads)

        /* pointers for storing each vector on the device*/
        int *dev_a, *dev_b, *dev_c;

        /* Allocate memory on the device */
        cudaMalloc((void**)&dev_a, sizeof(int)*size);
        cudaMalloc((void**)&dev_b, sizeof(int)*size);
        cudaMalloc((void**)&dev_c, sizeof(int)*size);

        /* Copy vectors a and b from host to device */
        cudaMemcpy(dev_a, a, sizeof(int)*size, cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, sizeof(int)*size, cudaMemcpyHostToDevice);

        /* Calling the kernel function to do calculation */

        // TO DO
        // Call kernel function here
        // end TO DO

        /* Copy the result vector from device to host*/
        cudaMemcpy(c, dev_c, sizeof(int)*size, cudaMemcpyDeviceToHost);

        /* Free memory on the device */
        cudaFree(dev_a);
        cudaFree(dev_b);
```

```
        cudaFree(dev_c);
}
```

Comments on MPI Program

- This MPI program is basically the MPI program with an addition of a function from CUDA program. It splits both input vectors into smaller pieces, and sends each piece of each vector to each worker. Then we will call the **run_kernel** function from CUDA program to calculate additions of the two vectors on each node.
- First we need to initialize the MPI execution environment, define the size of all processes, and give a unique rank to each process. Then we will ask the master to initialize the input vectors, split the input vectors into smaller chunks, and send these chunks to each process. Your task is to send the pieces of input vectors to each worker.

```
/******************* Master **********************/
if (rank == MASTER) {
        /* Initializing both vectors in master */
        int sum = 0;
        for (i = 0; i < WIDTH; i++) {
                arr_a[i] = i;
                arr_b[i] = i*i;
        }
        /* Decomposing the problem into smaller problems, and send each task
        * to each worker. Master not taking part in any computation.
        */
        num_worker = size - 1;
        ave_size = WIDTH/num_worker;    /* finding the average size of task for a proces
        extra = WIDTH % num_worker;             /* finding extra task for some processes
        offset = 0;
        mtype = FROM_MASTER;                    /* message sends from master */

        /* Master sends each task to each worker */
        for (dest = 1; dest <= num_worker; dest++) {
                eles = (dest <= extra) ? ave_size + 1: ave_size;
                MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                MPI_Send(&eles, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);

                // TO DO
                // send a piece of each vector to each worker
                // end TO DO

                printf("Master sent elements %d to %d to rank %d\n", offset, offset + el
                offset += eles;
        }
}
```

- Then we want all workers to receive the messages sent from master, and we call the **run_kernel** function from CUDA program to compute the sum of both vectors on each worker. This function will call the kernel function and compute additions on the GPU of each worker. When they are done with computations, each worker needs to send its result vector to the master. Your task is to receive the vectors sent from master, and to call **run_kernel** function from the CUDA program.

```
/* The workers receive the task from master, and will each run run_kernel to
* compute the sum of each element from vector a and vector b. After computation
* each worker sends the result back to master node.
```

```
*/
/****************************** Workers ************************/
if (rank > MASTER) {
        mtype = FROM_MASTER;
        source = MASTER;
        /* Receive data from master */

        // TO DO
        // receive the vectors sent from master
        // end TO DO

        MPI_Get_processor_name(name, &len);

        /* Use kernel to compute the sum of element a and b */

        // TO DO
        // call run_kernel function here
        // end TO DO

        /* send result back to the master */
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&eles, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&arr_c, eles, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
}
```

- We need to ask the master to receive the result vector sent from each worker. We then can check to see if they are correct. Verification part should not be included in your timing.

```
/* Master receives the result from each worker */
mtype = FROM_WORKER;
for(i = 1; i <= num_worker; i++) {
        source = i;
        MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&eles, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&arr_c[offset], eles, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        printf("Received results from task %d\n", source);
}

/* checking the result on master */
for (i = 0; i < WIDTH; i ++) {
        if (arr_c[i] != arr_a[i] + arr_b[i]) {
                printf("Failure !");
                return 0;
        }
}
printf("Successful !\n");
```

Download the source code to do your activity: `download CUDA program`

`download MPI program`

Download the entire source code: `download CUDA program`

`download MPI program`

CHAPTER

FOUR

ACTIVITIES

In this chapter, we are going to look at two problems, one is vector-matrix multiplication, and the other is matrix-matrix multiplication. We chose these problems because they are relatively easy to decompose, and not complicated.

4.1 Activity 2: Vector Matrix Multiplication

In this activity, we are going to compute vector-matrix multiplication in hybrid environment MPI and CUDA. You might already see this problem before in the MPI module. If so, it will not be much different. The basic idea is we want to split the rows of the matrix, and ask the master to send some rows of the matrix and entire input vector to each worker. We then ask each worker to receive messages from the master, and each worker will call the CUDA function to do computation on their own GPU. Basically, on the GPU each thread will compute a new element of the vector.

Comments on CUDA program

- First let's look at the kernel function in the CUDA program. We use one thread to compute multiplications of a row of the matrix with the vector, and this produces a new element of the result vector. We also use our two-dimensional array, the matrix, as a one-dimensional array to ease the problem. We are using two-dimensional block and thread. We use *threadIdx.y* because we want each thread to work on the multiplications of each row of the matrix with the vector.

```
/* kernel function for computation on the GPU */
__global__ void kernel(int *A, int *x, int *y, int width, int block_size) {

        int tid = blockIdx.y * block_size + threadIdx.y;
        int entry = 0;
        for (int i = 0; i < width; i++) {
                entry += A[tid * width + i] * x[i];
        }
        y[tid] = entry;
}
```

- Then we need to have a function that calls the kernel function on the host. This function is to allocate memory for the matrix and vector on the device, copy matrix and vector from host to device, compute the vector matrix multiplication, and copy the result vector from device to the host. This function will be called in the MPI program. Your task is to call kernel function at **TO DO**.

```
/* function on the host, CPU */
extern "C" void run_kernel(int *A, int *x, int *y, int width, int block_size) {

        /* the size of the matrix and the vector */
```

```
        int matrix_size = sizeof(int) * width * width;
        int vector_size = sizeof(int) * width;

        /* Pointes array on GPU */
        int *dev_A, *dev_x, *dev_y;

        /* Allocate memory on GPU */
        cudaMalloc((void**)&dev_A, matrix_size);
        cudaMalloc((void**)&dev_x, vector_size);
        cudaMalloc((void**)&dev_y, vector_size);

        /* Copy matrix and vector from CPU to GPU */
        cudaMemcpy(dev_A, A, matrix_size, cudaMemcpyHostToDevice);
        cudaMemcpy(dev_x, x, vector_size, cudaMemcpyHostToDevice);

        /* Initializing the grid size and block size */
        dim3 dimGrid(width/block_size, width/block_size);
        dim3 dimBlock(block_size, block_size);

        /* Running the kernel function */
        // TO DO
        // call the kernel function
        // end TO DO

        /* Copy the output vector from GPU to CPU */
        cudaMemcpy(y, dev_y, vector_size, cudaMemcpyDeviceToHost);

        /* Free memory on GPU */
        cudaFree(dev_A);
        cudaFree(dev_x);
        cudaFree(dev_y);
}
```

Comments on MPI program

- Now we can look at our MPI program with an addition of a CUDA function. First we need to initialize the MPI execution environment, define the size of MPI_COMM_WORLD, and give a unique rank to each process. Then we ask the master to initialize the input matrix and vector, divide the matrix by row, and send their pieces and the entire vector to each worker. Your task is to complete code at **TO DO**.

```
/* Initialize MPI execution environment */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

MPI_Get_processor_name(name, &len);

/****************************** Master ***************************/
if (rank == 0) {
        /* Initialize Matrix and Vector */
        for(i = 0; i < ROWS; i++) {
                // change here to use random integer
                vector[i] = 1;
                for(j = 0; j < COLS; j++) {
                        // change here to use random integer
                        matrix[i][j] = 1;
                }
```

```
        }

        numworkers = nprocs - 1;

        /* divide the number of rows for each worker */
        averow = ROWS/numworkers;
        extra = ROWS%numworkers;
        offset = 0;
        mtype = FROM_MASTER;

        /* Master sends smaller task to each worker */
        for(dest = 1; dest <= numworkers; dest++) {
                rows = (dest <= extra) ? averow + 1 : averow;

                // TO DO
                // send each piece of matrix and entire vector to each worker
                // end TO DO

                printf("Master sent elements %d to %d to rank %d\n", offset, offset + ro
                offset += rows;
        }
}
```

- We need to ask all workers to receive the messages sent from the master. Then we want each worker to call the CUDA function to compute their vector matrix multiplication. After having computed their multiplications, each worker needs to send their result back to the master. Please complete the following code at **TO DO**.

```
/************************************ Workers ************************************/
if (rank > 0) {
        mtype = FROM_MASTER;
        /* Each worker receives messages sent from the master*/

        // TO DO
        // receive each piece of the matrix and vector sent from master
        // end TO DO

        printf("Worker rank %d, %s receives the messages\n", rank, name);

        /* use CUDA function to compute the the vector-matrix multiplication for each wo
        // TO DO
        // call a function from CUDA program
        // end TO DO

        /* Each worker sends the result back to the master */
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&result, rows, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        printf("Worker rank %d, %s sends the result to master \n", rank, name);
}
```

- Finally, we need to ask the master to receive the result vector sent from each worker, and prints the result vector.

```
/* Master receives the output from each worker*/
mtype = FROM_WORKER;
```

```
for (i = 1; i <= numworkers; i++) {
        source = i;
        MPI_Recv(&offset, 1, MPI_INT, source,mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
        MPI_Recv(&result[offset], rows, MPI_INT, source, mtype, MPI_COMM_WORLD, &status)
        printf("Received results from task %d\n", source);
}

/* Master prints results */
for (i = 0; i < ROWS; i++) {
        printf("The element of output vector is: %d\n", result[i]);
}
```

Download the source code to do your activity: `download CUDA program`

`download MPI program`

Download the entire source code: `download CUDA program`

`download MPI program`

4.2 Activity 3: Matrix Multiplication

In this activity, we are going to compute matrix-matrix multiplication in hybrid environment MPI and CUDA. The basic idea is we want to split the rows of the first matrix, and ask the master to send some rows of the first matrix and the entire second matrix to each worker. We then ask each worker to receive messages sent from the master, and each worker will call the CUDA function to do computation on their own GPU.

Note: This is not the most efficient method of computing matrix-matrix multiplication by using hybrid environment CUDA and MPI because when the second matrix gets too large, a progammer may not be able to send it to each worker. Furthermore, a programmer can improve the kernel function to be more effecient by using the shared memory archeticture in the GPU.

Comments on CUDA Program

- First let's look at the kernel function in the CUDA program. In this kernel function, we are using two different threads. One thread is to calculate the row index of the first matrix, and the other is to calculate the column index the second matrix. To calculate the row index, we use *threadIdx.y*, and to calculate the column index, we use *threadIdx.x*. Then we need to iterate over the width of the matrix, and multiply each corresponding elements of the two input matrices, and sum all of them to produce a new element of the result matrix. Your task to complete the code at **TO DO**.

```
/* kernel function */
__global__ void MatrixKernel(float *dM, float *dN, float *dP, int width) {

        /* calculate the row index of the dP element and M */
        // TO DO
        // int row = .........
        // end TO DO

        /* calculate the column index of dP element and N */
        // TO DO
        // int col = .........
        // end TO DO
```

```
        float pvalue = 0.0f;
        for (int k = 0; k < width; k++) {
                float M_elem = dM[row * width + k];
                float N_elem = dN[k * width + col];
                pvalue += M_elem * N_elem;
        }
        dP[row * width + col] = pvalue;
}
```

- Then we need to have a function that calls the kernel function on the host. This function is to allocate memory for the matrices on the device, copy matrices from host to device, compute the matrix multiplication, and copy the result matrix from device to the host. This function will be called in the MPI program. Your task is to call kernel function at **TO DO**.

```
/* function that you will call in mpi code */
extern "C" void MatrixMul(float* M, float* N, float* P, int width, int block_size) {

        int matrix_size = width * width * sizeof(float);
        float *dM, *dN, *dP;

        // Allocate and Load M and N to device memory
        cudaMalloc(&dM, matrix_size);
        cudaMemcpy(dM, M, matrix_size, cudaMemcpyHostToDevice);

        cudaMalloc(&dN, matrix_size);
        cudaMemcpy(dN, N, matrix_size, cudaMemcpyHostToDevice);

        // Allocate P on device
        cudaMalloc(&dP, matrix_size);

        dim3 dimGrid(width/block_size, width/block_size);
        dim3 dimBlock(block_size, block_size);

        // TO DO
        // call the kernel function
        // end TO DO

        cudaMemcpy(P, dP, matrix_size, cudaMemcpyDeviceToHost);

        cudaFree(dP);
        cudaFree(dM);
        cudaFree(dN);
}
```

Comments on MPI Program

- Now we can look at the MPI program. This MPI program is a revised version of the Matrix Mutiplication MPI program that was written by Blaise Barney. First we need to initialize the MPI execution environment, define the size of MPI_COMM_WORLD, and give a unique rank to each process. Then we ask the master to initialize the input matrices, divide these matrices, and send their pieces to each worker.

```
/************************** master task ************************************/
if (taskid == MASTER) {

        /* Initializing both matrices on master node */
        for (i = 0; i < ROW_A; i++)
```

```
                for (j = 0; j < COL_A; j++)
                        // change here to use random integer
                        a[i][j]= 1;
        for (i = 0; i < COL_A; i++)
                for (j = 0; j < COL_B; j++)
                        // change here to use random integer
                        b[i][j]= 1;

        /* Computing the average row and extra row for each process */
        averow = ROW_A/numworkers;
        extra = ROW_A%numworkers;
        offset = 0;
        mtype = FROM_MASTER;

        /* Distributing the task to each worker */
        for (dest = 1; dest <= numworkers; dest++) {
                rows = (dest <= extra) ? averow+1 : averow;
                printf("Sending %d rows to task %d offset = %d\n", rows, dest, offset);

                // TO DO
                // send some rows of first matrix and entire second matrix to each worke
                // end TO DO

                offset = offset + rows;
        }
}
```

- We need to ask all workers to receive the messages sent from the master. Then we want each worker to call the CUDA function to compute their matrix multiplication. After having computed their multiplications, each worker needs to send their result back to the master. Please complete the following code at **TO DO**.

```
/**************************** worker task ************************************/
if (taskid > MASTER) {

        /* Each receives task from master*/
        mtype = FROM_MASTER;

        // TO DO
        // receive the matrices sent from master
        // end TO DO

        /* Calling function from CUDA. Each worker computes on their GPU */

        // TO DO
        // call CUDA function
        // end TO DO

        /* Each worker sends result back to the master */
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
        MPI_Send(&c, rows*COL_B, MPI_FLOAT, MASTER, mtype, MPI_COMM_WORLD);
}
```

- Finally, we need to ask the master to receive the result matrix sent from each worker, and prints the result matrix.

```
        /* Receive results from worker tasks */
        mtype = FROM_WORKER;
        for (i = 1; i <= numworkers; i++) {
                source = i;
                MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
                MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
                MPI_Recv(&c[offset][0], rows*COL_B, MPI_FLOAT, source, mtype, MPI_COMM_WORLD, &s
                printf("Received results from task %d\n",source);
        }

        /* Master prints results */
        printf("******************************************************\n");
        printf("Result Matrix:\n");
        for (i = 0; i < ROW_A; i++) {
                printf("\n");
                for (j = 0; j < COL_B; j++)
                        printf("%6.2f   ", c[i][j]);
        }
```

Download the source code to do your activity: `download CUDA program`

`download MPI program`

Download the entire source code: `download CUDA program`

`download MPI program`

CHAPTER

FIVE

THE THREE MUSKETEERS

Another Heterogeneous programming model that we can look at is the parallel programming technique that combines MPI, CUDA, and OpenMP. Each of the parallel programming model uses different resource for their computation. OpenMP works on Multicore CPUs, CUDA works on GPUs, and MPI works on distributed memory cluster. Therefore, if you have a cluster, whose nodes contain more than one core, and at least a GPU, you can try this parallel programming method. However, in this last section of the module we just want introduce the idea of having this hardcore parallelism.

Recommended Reading

- Please read Scalability of Incompressible Flow Computations on Multi-GPU Clusters Using Dual-Level and TriLevel Parallelism.