Parallel Computing Laboratory
IT-300
Fall-2019

By
Dr. B. Neelima
National Institute of Technology Karnataka
(NITK), Surathkal

Week: 16th and 23rd October-2019

The students are given logins in group of 5 on DGX station server. They are expected to work in a team or time-share the logins. The login details are at: https://docs.google.com/spreadsheets/d/1jgYu_Kg-1ozYAUBQ_YZaKOX6lJ2MSMziCkrcmPqt9xs/edit#gid=0

The following are the instructions to work on the remote server:

1. First check whether you are connected to internet. Internet connection is must as you are working on a remote system.
2. Note down your username and password.
3. Open a terminal and login and work as follows:

>ssh -X username@10.100.52.96
username@10.100.52.96's password☹enter the password)
you will see the following prompt, indicating that you are on DGX station:
username@nvidia-DGX-Station:~$

$cd ~ //see that you are on top directory
$ mkdir reg.no. // create a directory with your register number

- Go to the folder

$ cp -R /usr/local/cuda/samples //copies the samples (cuda 9.0) into your folder, creates a 'samples' folder inside your named folder

$ cd samples
$ make -k (creates object files of all the sample programs

// before going to execution step, you may cfreate your own programs in excercises folder as follows:

On your login page…you can create your programs inside the same folder and execute them as above
$ cd ../../.. (go to home)
$ mkdir excercises
$ cd excercises
$ vi vecAdd.cu

Execution Step:
$ nvidia-docker run -it --net="host" -v /home/"path to cuda folder" /:/workspace/ nvcr.io/nvidia/cuda:9.0-devel-ubuntu16.04
// Opens root@nvidia-DGX-Station:

$ cd workspace
$ls
// you can see the folder that you created in your login as cuda or other files

$ cd cuda/smaples
$cd 0_Simple/vectorAdd
$./vectorAdd
//Test Passed message is success ful execution of your program

$. cd 1_Utilities
$ cd deviceQuery
$ ./deviceQuery

//gives you the details of all. The device on this remote server and features of the GPUs

// you may walk through the code to learn how to obtain such details
//similarly you can walk through other samples as per your interest

To execute your excercises… go to cuda fodler

$ cd excercises

$ nvcc -o vecAdd vecAdd.cu
$./vecAdd

To exit execution stage, type
$exit. //you will be on login node

// similar create your own programs in login page before execution stage and work on them

EXECUTING CUDA PROGRAMS:

After you finish installation and testing the samples, write your own codes in cuda and run them as per the following instructions:

Write the cuda program in c-style. The file is saved as: filename.cu

Compilation using nvidia: nvcc filename.cu. (use the required flags as per the requirement)
executing: ./a.out

Exercise 1: deviceQuery

1. Run the deviceQuery program from samples and send the screenshot of details of your device.

Exercise 2: Simple vector addition using CUDA

1. Write the following CUDA program and change the blocks and threads as per your device and see the variations in execution time. Report the same. (the following is not a complete program). Report the performance graph for block size variations and/or/grid variations.

```
#define N 256
#include <stdio.h>

__global void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

__global void vecAdd (int *a, int *b, int *c) {
 int i = threadIdx.x;
 c[i] = a[i] + b[i];
}
```

2. Change the number of elements and/or number of threads. IF they are not exactly divisible, design your grid to pad the last block.

```
#define N 1618
#define T 1024 // max threads per block
#include <stdio.h>

__global void vecAdd (int *a, int *b, int *c);

int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;

 // initialize a and b with real values (NOT SHOWN)

 size = N * sizeof(int);

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);

 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<(int)ceil(N/T),T>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);

 exit (0);
}

__global void vecAdd (int *a, int *b, int *c) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < N) {
  c[i] = a[i] + b[i];
 }
}
```

3. Run matrix addition on GPU and observe the effect of block variations. (the following is not a complete program). Report the performance graph for block size variations and/or/grid variations.

```
#define N 512
#define BLOCK_DIM 512

__global void matrixAdd (int *a, int *b, int *c);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize a and b with real values (NOT SHOWN)

 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
 dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));

 matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixAdd (int *a, int *b, int *c) {
 int col = blockIdx.x * blockDim.x + threadIdx.x;
 int row = blockIdx.y * blockDim.y + threadIdx.y;

 int index = col + row * N;

 if (col < N && row < N) {
  c[index] = a[index] + b[index];
 }
}
```

4. Run matrix multiplication on GPU. (the following is not a complete program). Report the performance graph for block size variations and/or/grid variations.

```
#define N 16
#include <stdio.h>

__global void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 // initialize matrices a and b with appropriate values

 int size = N * N * sizeof(int);
 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

__global void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;

 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }
}
```

5. Run matrix multiplication using tiling and/or shared memory. (the following is not a complete program). Report the performance graph for block size variations and/or/grid variations.

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.     Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.    __syncthreads();

11.   for (int k = 0; k < TILE_WIDTH; ++k)
12.   Pvalue += Mds[ty][k] * Nds[k][tx];
13.   Synchthreads();
14.  }
13.   Pd[Row*Width+Col] = Pvalue;
}
```