NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Exercise Manual for SC3050 Advanced Computer Architecture

Practical Exercises on CUDA Programming for GPU

COLLEGE OF COMPUTING AND DATA SCIENCE
NANYANG TECHNOLOGICAL UNIVERSITY

CUDA Programming

Learning Objectives

Student will practice programming of GPU using CUDA C in order to appreciate the basic architecture of GPU.

Intended Learning Outcomes

At the end of this exercise, you should be able to

- write basic CUDA programs.
- make use of GPU to solve problem based on parallelism using multi-threading.

Equipment and accessories required

i) NVIDIA Jetson TX2 embedded board

ii) HDMI monitor, USB keyboard and USB mouse.

---

1. Introduction

NVIDIA Jetson TX2 is an embedded GPU-enable platform designed for AI edge computing. The GPU is based on NVIDIA Pascal architecture with 2 SMs and 256 CUDA cores, while the host is an ARM based multiprocessor (which consists of 2 Denver 64-bit CPUs + Quad-Core A57 Complex).

Fig.1 – Jetson TX2

It runs the Linux operating system, and can be set up as a standalone GPU enabled computing system by connecting a monitor, keyboard and mouse through its various interfaces.

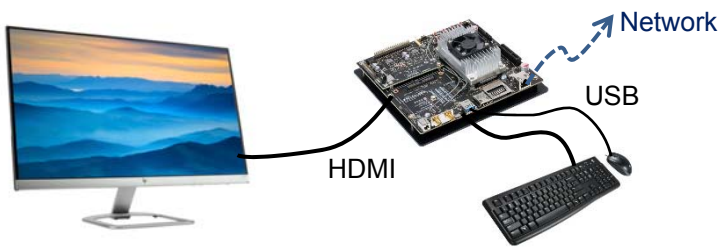

Fig. 2 - Standalone setup of Jetson TX2

As such, you should be familiar with some of the basic Linux commands and applications that are commonly used for developing programs in Linux environment. Some examples are as follows:

- know how to open a terminal to enter commonly used commands.
- ls command - for listing the (current) directory's contents.
- cd command - for changing to another directory.
- using gedit application to edit your source code.
- using the Arrow keys to repeat previously used commands.
- using Tab key to complete a command.
- know how to compile C program using gcc – but in this exercise, you will use the NVIDIA nvcc compiler instead.

Note: As this is a year 3 course, this exercise 'manual' is purposely written with brevity, such that students will have to learn how to look for relevant information, and more importantly learn how to approach a problem without step-by-step guiding instructions (as commonly found in lower year courses). Nevertheless, most of the exercises had been covered in the lecture (on CUDA programming), and students can refer to the lecture notes (or google) for the detail.

2 Basic CUDA programming

2.1 Hello world!

The following is the customary program when we first explore programming (in C).

```
int main(void){
  printf("Hello, World - from CPU!")
  return 0;
  }
```

Modify the code to include two CUDA kernels to produce the output similar to the following when the program is executed.

```
Hello from CPU!
Hello from GPU1[0]!
Hello from GPU1[1]!
Hello from GPU1[2]!
Hello from GPU1[3]!
Hello from GPU2[0]!
Hello from GPU2[1]!
Hello from GPU2[2]!
Hello from GPU2[3]!
Hello from GPU2[4]!
Hello from GPU2[5]!
```

Discussion:
Observe whether your program produces the output exactly as shown above. What could be the reasons if otherwise?

2.2 Vector Addition using parallel threads

$$\boldsymbol{A} = \begin{bmatrix} 22 \\ 13 \\ 16 \\ 5 \end{bmatrix} \qquad \boldsymbol{B} = \begin{bmatrix} 5 \\ 22 \\ 17 \\ 37 \end{bmatrix}$$

Given two vectors ***A*** and ***B*** shown above, code a CUDA program that adds the two vectors to produce the result and stored as vector ***C***. The output of the program should be as follows:

```
A    22 13 16  5
B     5 22 17 37
C    27 35 33 42
```

2.3 Dot Product

For the same vectors ***A*** and ***B*** in 2.2 above, code a CUDA program to compute its dot product. The output of the program should be as follows:

```
A    22 13 16  5
B     5 22 17 37
Answer = 853
```

3 An application

You are running a webstore that lists the following online items at the respective prices shown.

(a) 2T harddisk (HD) - $29.99
(b) BT earpiece (EP) - $14.99
(c) Iphone screen protector (SP) - $9.99
(d) 10G USB C thumbdrive (TD) - $24.99

For one particular week in a particular month, the sale figures for these items are as follows:

Item	Mon	Tue	Wed	Thu	Fri	Sat	Sun
HD	3	2	0	3	4	10	8
BT	5	4	3	5	5	13	11
SP	2	5	3	4	5	21	15
TD	0	1	1	4	3	16	8

Code a CUDA based program that computes the sale amount received for each of the day.

Things to consider: How should you represent the data? How can the program be designed in such a way that it is easily scalable to include many more items as your business expanded?

Hint: The following shows an example of how a 2D array can be defined in C.

```
int 2Darray[2][4]= {
   { 5,  7,  2, 20},
   {15, 11, 22, 18}
};
```

Lab Report for Lab 5 (Name:_____, Matric:_____, Group:____)

EVALUATION

1) In Problem 2.2 of Lab 5, how many thread blocks did you create in your CUDA program to solve the Vector Addition problem, and how many threads per block did you create?

   Number of thread blocks: _____

   Number of threads per block: _____

2) In Problem 2.3 of Lab 5, how many threads in total did you create in your CUDA program to solve the Dot Product problem?

   Number of threads in total: _____

3) In Problem 3 of Lab 5, how many threads in total did you launch in your CUDA program to solve the problem in a way that the threads run in the maximum parallelism on the Jetson board? What is the total sales amount of the entire week for the webstore?

   Number of threads in total: _____

   Total sales amount: _____

```
__global__ void cuda_hello(int n) {
    printf("Hello from GPU%d[%d]!\n",
        n, threadIdx.x);
}

int main() {
    printf("Hello from CPU!\n");
    cuda_hello<<<1,4>>>(1);
    cudaDeviceSynchronize();
    cuda_hello<<<1,6>>>(2);
    return 0;
}
```

```
__global__ void vector_add(int *a, int *b, int *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];

}

int main() {
    int[4] a = {22, 13, 16, 5};
    int[4] b = {5, 22, 17, 37};
    int[4] c;
    int *d_a, *d_b, *d_c;
    int size = 4 * sizeof(int);
    // Allocate memory on the GPU
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);
    // Copy the input vectors from host memory to GPU buffers
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch the vector_add kernel on GPU
    vector_add<<<1, 4>>>(d_a, d_b, d_c);
    // Copy the result vector from GPU buffer to host memory
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Free GPU memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    // Print the result
    for (int i = 0; i < 4; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
    return 0;
}
```

```
__global__ void vector_dot_product(int *a, int *b, int *c) {
    __shared__ int temp[4];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    temp[i] = a[i] * b[i];
    __syncthreads(); // Synchronize threads within a block
    if (i == 0) {
        int sum = 0;
        for (int j = 0; j < blockDim.x; j++) {
            sum += temp[j];
        }
        *c = sum; // Store the result in c
    }
}

int main() {
    int[4] a = {22, 13, 16, 5};
    int[4] b = {5, 22, 17, 37};
    int c;
    int *d_a, *d_b, *d_c;
    int size = 4 * sizeof(int);
    // Allocate memory on the GPU
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, sizeof(int));
    // Copy the input vectors from host memory to GPU buffers
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch the vector_add kernel on GPU
    vector_dot_product<<<1, 4>>>(d_a, d_b, d_c);
    // Copy the result vector from GPU buffer to host memory
    cudaMemcpy(c, d_c, sizeof(int), cudaMemcpyDeviceToHost);
    // Free GPU memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    // Print the result
    printf("Dot product: %d\n", c);
    return 0;
}
```