WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A & COMS7045A Exercise III: CUDA C Programming

Hairong Wang

March 12, 2020

1 Objectives

Learn to

- Write simple CUDA programs;
- Compile and run a CUDA program;
- Measure the performance of a CUDA program using CUDA event.

2 CUDA C Programming

1. The program **double.cu** doubles each element in an array. Currently, the program only doubles some part of the array, not the entire array. Why? Fix the program so that it gives the correct result.
2. The program **vector_addition.cu** adds two vectors. Each thread performs one pair-wise addition. Compile and run it. Note the use of `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()` functions, kernel launch, and timing using events —
   - `cudaEventCreate()`,
   - `cudaEventRecord()`,
   - `cudaEventSynchronize()`,
   - `cudaEventElapsedTime()`,
   - `cudaEventDestroy()` functions.

   (a) Add a sequential version of vector addition to compare the performance between sequential and parallel computations.
   (b) Experiment with different data sizes, different kernel execution configurations to see the effects.
3. Open **matrix_op1.cu** and do the following:
   (a) Implement the `multiplyGPU()` function (see `multiplyCPU()`). This function takes two device arrays `g_v1` and `g_v2` and multiplies each of their elements and stores the result at the corresponding index in the device array `g_out`. The main trick to this part is calculating the correct index for each thread âĂŞ remember `blockDim`, `blockIdx` and `threadIdx` to help you. The `multiplyCPU()` function is the CPU version of the kernel and can be used as a guide when implementing the GPU version.
   (b) Launch the `multiplyGPU()` kernel. Remember the `<<<grid, block>>>` notation for launching kernels. The trick here is to choose a good block size and then calculate the number of the blocks in the grid using that size. Try to find the most efficient combination.
   (c) Implement the `expensiveFunctionGPU()` function (see `expensiveFunctionCPU()`). This function takes two device arrays `g_v1` and `g_v2` and performs some complex operation on each of their

elements and stores the result at the corresponding index in the device array `g_out`. As before, the main trick to this part is calculating the correct index for each thread. The `expensiveFunctionCPU()` function is the CPU version of the kernel.

(d) Launch the `expensiveFunctionGPU()` kernel.

Once you finish the above tasks, add appropriate timing for all the functions and see the performance.

4. The program **julia_cpu.cu** is a CPU implementation of drawing slices of Julia Set (see Chapter 4, CUDA by Example). Read through the chapter to understand the code.
   (a) Based on this program, develop an OpenMP implementation of the problem.
   (b) Complete the CUDA C implementation by following the discussion on the book.
   (c) Add proper timing in all three, i.e. serial, OpenMP, CUDA, implementations to compare the performance.