WITS UNIVERSITY

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

COMS4040A & COMS7045A Exercise for Lec6 & Lec7

April 6, 2025

Problems

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 differences.
3. 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.
4. Based on **histo_template.cu**, implement the two kernels using interleaved sectioning, and shared memory respectively. Note the various helper functions given in the base program, such as timing and error checking, used in NVIDIA_CUDA_Samples. Verify all your results and compare the performance of your implementations.
5. Implement matrix transpose using global memory. Your program needs to work with square matrices. In the testing, use square matrices of size $2^9, 2^{10}, 2^{11}, 2^{12}$, and report the performance on these sizes. Verify and compare the results against a serial implementation, and compare the results among the various implementations. The performance should be measured using timing, speedup, and throughput.

6. Implement two kernels which compute the sum of all the elements in a vector (a reduction operation). You should implement the kernels in two different ways:
   (a) Using shared memory.
   (b) Using global memory.

   In this implementation, you should also include a CPU reduction in order to verify the results of the GPU version. Test with different large data sizes and compare the performances. Note that, for reduction, when the array size is large, summation of the values may cause overflow, hence wrong results. To verify your implementation, start with very simple array entries, such as all 1's.
7. Based on the matrix multiplication example in the lecture slide, implement the tiled matrix multiplication. You may start from the base code **matrix_multiplication.cu** given. Run the program using different data size, execution configuration and tile width to measure the performance.