# COMS4040A/COMS7045A Exercise: CUDA C Programming

April 2020

## Programming

1. Querying device properties. When a CUDA application runs on a system, how does it find out the amount of resources available? In CUDA C, a built-in mechanism exists for a host code to query the properties of the devices available in the system.

   - `cudaGetDeviceCount` API function returns the number of CUDA devices in the system. The host code can determine the number of CUDA devices using:

     ```
     int dev_count;
     cudaGetDeviceCount(&dev_count);
     ```

     The CUDA runtime numbers all available devices in the system from 0 to `dev_count -1`.

   - `cudaDeviceProperties` API function returns the properties of the device whose number is given as an argument.

     ```
     cudaDeviceProp dev_prop;
     for(int i = 0; i<dev_count; i++)
       cudaDeviceProperties(&dev_prop, i);
       //decide if device has sufficient resources and
          capabilities.
     ```

     The built-in type `cudaDeviceProp` is a C struct type with fields representing the properties of a CUDA device.

   - In `cuda_device_query.cu`, a simple usage of `cudaGetDeviceProperties()` is given. You can use this function to query the size of shared memory, the number of registers etc. available on your device. Compile and run `cuda_device_query.cu` to check the properties of the device you are using.

   - For more detailed device query, see `deviceQuery.cpp` in NVIDIA_CUDA_Samples.

2. In CUDA, the host and device share separate memory address space. Complete the following in `data_movement.cu`.

   (a) Allocate memory to store `N` integers on the host for the pointers `h_a` and `h_b`.
   (b) Allocate memory to store `N` integers on the device for the pointer `d_a`.
   (c) Transfer the contents of `h_a` to `d_a`.
   (d) Transfer the contents of `d_a` to `h_b`.
   (e) Free the memory allocated on the host for `h_a` and `h_b`.

(f) Free the memory allocated on the device for `d_a`.

You need to insert the relevant code at the locations indicated by the `//YOUR CODE GOES HERE` comments in the file. When you execute the binary "Test Passed!" will be displayed if your code is correct. Study the simple utility function `checkCUDAError` in the code for CUDA runtime error checking.

3. Consider the vector addition problem in `vector_addition_v3.cu`, where an implementation using global memory is given. Add an implementation of the same problem using shared memory. Compare the throughput between using global memory and shared memory implementations. Can you reduce the global memory bandwidth consumption by using shared memory? Why?
4. 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 your 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.
5. Based on the matrix multiplication example in the lecture slide, implement the tiled matrix multiplication. A base code `matrix_ multiplication.cu` is given. Run the program using different data size, execution configuration and tile width to see the performance.
6. Program `reverse_array.cu` constructs a reversal of an input array. Write two CUDA kernels for the reversal using global memory and shared memory, respectively. Verify the results and compare the performance of your implementations.
7. 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.