**Homework 2 — Due Wednesday May 2, 11 PM**

In this programming assignment you will use NVIDIA's Compute Unified Device Architecture (CUDA) language to implement a basic string shift algorithm and the pagerank algorithm. In the process you will learn how to write general purpose GPU programming applications and consider some optimization techniques. You must turn in your own copy of the assignment as described below. You may discuss the assignment with your peers, but you may not share answers. Please direct your questions about the assignment to the forum.

# NOTE

This programming assignment will require NVIDIA GPU hardware and software support for CUDA. You are required to use the `cme213-cluster` for this assignment. Please read **Section B** first to understand how to compile and run your CUDA program on `cme213-cluster`.

# CUDA

"C for CUDA" is a programming language subset and extension of the C programming language, and is commonly referenced as simply CUDA. Many languages support wrappers for CUDA, but in this class we will develop in C for CUDA and compile with `nvcc`.

The programmer creates a general purpose kernel to be run on a GPU, analogous to a function or method on a CPU. The compiler allows you to run C++ code on the CPU and the CUDA code on the device (GPU). Functions which run on the host are prefaced with `__host__` in the function declaration. Kernels run on the device are prefaced with `__global__`. Kernels that are run on the device and that are only called from the device are prefaced with `__device__`.

The first step you should take in any CUDA program is to move the data from the host memory to device memory. The function calls `cudaMalloc` and `cudaMemcpy` allocate and copy data, respectively. `cudaMalloc` will allocate a specified number of bytes in the device main memory and return a pointer to the memory block, similar to malloc in C. You should not try to dereference a pointer allocated with cudaMalloc from a host function.

The second step is to use `cudaMemcpy` from the CUDA API to transfer a block of memory from the host to the device. You can also use this function to copy memory from the device to the host. It takes four parameters, a pointer to the device memory, a pointer to the host memory, a size, and the direction to move data (`cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`). We have already provided the code to copy the string from the host memory to the device memory space, and to copy it back after calling your shift kernel.

Kernels are launched in CUDA using the syntax `kernelName<<<...>>>(...)`. The arguments inside of the chevrons (`<<<blocks, threads>>>`) specify the number of thread blocks and thread per block to be launched for the kernel. The arguments to the kernel are passed by value like in normal C/C++ functions.

There are some read-only variables that all threads running on the device possess. The three most valuable to you for this assignment are `blockIdx`, `blockDim`, and `threadIdx`. Each of these variables contains fields x, y, and z. `blockIdx` contains the x, y, and z coordinates of the thread block where this thread is located. `blockDim` contains the dimensions of thread block where the thread resides. `threadIdx` contains the indices of this thread within the thread block.

We encourage you to consult the development materials available from NVIDIA, particularly the CUDA Programming Guide and the Best Practices Guide available at
`http://docs.nvidia.com/cuda/index.html`
Additional documentation can be downloaded from canvas.

## Problem 1 String Shift

The purpose of this problem is to give you experience writing your first simple CUDA program. This program will help us examine how various factors can affect the achieved memory bandwidth. The program will take an input string and shift each character by constant amount. You will do this in three different ways:

- by shifting one byte at a time,
- by shifting 4 bytes at a time,
- and finally by shifting 8 bytes at a time.

You should be able to take the files we give you and type `make main_q1` to build the executable. The executable takes 1 argument — the number of times to double the input file in size. For debugging it is recommended to use a value of 0. The executable will run, but since the CUDA code hasn't been written yet (that's your job), it will report errors and quit.

For this problem we provide the following starter code (* means you should *not* modify the file): :

- `*main_q1.cu` — This is the main file. We have already written most of the code for this assignment so you can concentrate on the CUDA code. We take care of loading the input file, computing the host solution and checking your results against the host reference. There is also a loop setup that will generate a table of results for the three kernels for a variety of sizes.
- `shift.cu` — This is the file you will need to modify and submit. It already contains the necessary function headers — do not change these. You should fill in the body of each function.
- `*Makefile` — `make main_q1` will build the binary. `make clean` will remove the executables. You should be able to build and run the program when you first download it, however only the host code will run.
- `hw2.sh` — This script is used to submit jobs to the queue (see Appendix B). You need to comment out the other lines in the file if you only want to run `./main_q1`.

**Question 1.1**

25 points. Fill in the functions in `shift.cu` so that the program no longer reports any errors. Here is a description of how the individual functions should behave:

- `shift_char(...)` should shift each `unsigned char` in `input_array` by `shift_amount` and write it back to `output_array`. For example, if `input_array[0]` = 49 and `shift_amount` = 5, then `output_array[0]` should equal 54.
- `shift_int(...)` should shift each *byte* in `input_array` by `shift_amount` and write it back to `output_array`. Since an `unsigned int` contains 32 bits (or 4 bytes), it means that bits 0–7, 8–15, 16–23, and 24–31 *each* have to be shifted by `shift_amount`. For example if `input_array[0]` contains the elements 73, 67, 77 and 69 and `shift_amount` = 32, then `output_array[0]` should look like:

$$\begin{array}{rl} \texttt{input\_array[0]} = & \boxed{73}\boxed{67}\boxed{77}\boxed{69} \\ + & \boxed{32}\boxed{32}\boxed{32}\boxed{32} \\ \texttt{output\_array[0]} = & \boxed{105}\boxed{99}\boxed{109}\boxed{101} \end{array} \quad (1)$$

  *Note*: In this case `input_array[0]` has the binary representation 01001001 01000011 01001101 1000101, or decimal representation 614'573'765, so do not expect to see the number 73677769 if you print `input_array[0]`.

  *Hint 1*: Since `shift_amount` is an `unsigned int` you can significantly speed up your computation if you represent `shift_amount` the way it's depicted in line (1). You can compute this representation in `doGPUShiftUInt(...)` and pass it as a modified `shift_amount` to `shift_int(...)`.

  *Hint 2*: You don't have to worry about overflow. In other words you can assume that each byte in `input_array[0]` is less than 255 − `shift_amount`.

- `shift_int2(...)` should shift each *byte* in `input_array` by `shift_amount` and write it back to `output_array`.

  *Note*: `uint2` is a struct with members `x` and `y`.

**Question 1.2**

5 points. Take the table that is generated once you've correctly implemented everything and generate a plot of bandwidth in GB/sec vs. problem size in MB. For these tables, pass the argument 8 to the executable so that it doubles the input 8 times for the maximum size.

**Question 1.3**

10 points. Performance should increase significantly from the `char` to `uint` versions of the kernel. Why? Why does the performance not change much between the `uint` and `uint2` versions of the kernel?

## Problem 2 PageRank

PageRank was the link analysis algorithm responsible (in part) for the success of Google. It generates a score for every node in a graph by considering the number of in links and out links of a node. We are going to compute a simplified model of pagerank, which, in every iteration computes the pagerank score as a vector $\pi$ and updates $\pi$ as

$$\pi(t+1) = \frac{1}{2}\, A\pi(t) + \frac{1}{2N}\, \mathbf{1}$$

where $A$ is a normalized adjacency matrix (so that each column sums to 1), $N$ is the number of nodes in the graph and $\mathbf{1}$ is a vector with all 1's. Each entry in the vector $\pi$ corresponds to the score for one node. The matrix $A$ is sparse and each row $i$ corresponds to the node $n_i$, the non-zero entries correspond to the nodes $n_j$ that have a directed edge to $n_i$ (i.e., $A_{ij} > 0 \Leftrightarrow (n_j, n_i) \in E$, where $E$ is the set of directed edges). Since we normalize the columns of $A$, the entries in the $j$'th column are all proportional to $1/\text{outDegree}(n_j)$. We will choose the *average* number of connections for a node to be $\mu \in \mathbb{N}_+$ and then have the actual number of connections per node vary from 1 to $2\mu - 1$. The total number of edges is given by $|E| = \mu N$.

In the actual algorithm this operation is performed until the change between successive $\pi$ vectors is sufficiently small. In our case we will choose a fixed number of iterations to more easily compare performance across various numbers of nodes and edges. If you wish to learn more about the algorithm itself, check `http://en.wikipedia.org/wiki/PageRank`

For this problem, we provide the following starter code (* means you should *not* modify the file):

- `main_q2.cu` — contains the code that sets up the problem and generates the reference solution. It also has a result generating loop that will generate a table of timing results for various numbers of edges and nodes. Other than filling in the bandwidth calculation and a tiny required change to answer one of the questions, you should not modify this file.
- `pagerank.cu` — this is the file you will need to modify and submit. Do not change the function headers but fill in the bodies and follow the hints/requirements in the comments.
- `*Makefile` — `$ make main_q2` will build the pagerank binary. `make clean` will remove the executables. You should be able to build and run the program when you first download it, however only the host code will run.
- `hw2.sh` — This script is used to submit jobs to the queue (see Appendix B). You need to comment out the other lines in the file if you only want to run `./main_q2`.

**Question 2.1**

35 points. Fill in the functions so that the program no longer reports any errors.

**Question 2.2**

10 points. What is the formula for the total number of bytes **read from and written to** the global memory by the algorithm? Analyze the code you've written and do the calculation "on paper" instead of running actual code. *Hint: your answer should base on the number of nodes, the average number of edges and the number of iterations. Don't include any data transfer between CPU and GPU in this calculation*

Edit the bandwidth calculation in the driver file `main_q2.cu` (search for TODO) to reflect your answer to Question 2.2. Use variables `*nodes`, `*edges` and `iterations`.

**Question 2.3**

5 points. From the table of results, plot the memory bandwidth (GB/sec) vs. problem size for an average number of edges equal to 10. Make sure the plot is readable. You do not have to comment the plot.

**Question 2.4**

10 points. What does the memory access pattern look like for this problem? Using your answer to this question, explain the difference in bandwidth between Problem 2 and Problem 3.

Total number of points: 100

# A Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file. The name of the file should be: `hw2.pdf`. Please upload the PDF file to Gradescope, **not** `cardinal.stanford.edu`.

2. The other files should be submitted using a submission script on `cardinal`. The submission script must be run on `cardinal.stanford.edu`.

3. Copy your submission files to `cardinal.stanford.edu`. You can use the following command in your terminal:
   `scp <your submission file(s)> <your SUNetID>@cardinal.stanford.edu:`
   The script will copy the files below to a directory accessible to the CME 213 staff. Only these files will be copied. Make sure these files exist and that no other files are required to compile and run your code. In particular, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. Here is the list of files we are expecting and that will be copied:

   ```
   main_q1.cu
   shift.cu
   main_q2.cu
   pagerank.cu
   ```

   The script will fail if one of these files does not exist. Make sure to upload `hw2.pdf` to Gradescope that is **not** the original pdf that comes with the homework!

4. Make sure your code compiles on `cme213-cluster` and runs. To check your code, we will run:
   `$ make`
   This should produce 2 executables: `main_q1`, `main_q2`.

5. Type:
   `/usr/bin/python /usr/class/cme213/WWW/script/submit.py hw2 <directory with your submission files>`

6. You can submit at most 10 times before the deadline; each submission will replace the previous one.

7. There will be a 10% penalty per 24 hours for late submission. We will not accept submissions that are submitted past two days.

# B Running on the cme213-cluster

You can find a simple introduction to the cluster on `https://hpcc-intranet.stanford.edu/how-do-i/`. Remember to use VPN before attempting to log in.

### B.1 Compiling

To use nvcc on the cme213-cluster, you must have the correct modules loaded. If you type the command

```
$ module list
```

the output should be:

```
Currently Loaded Modules:
  1) gnu/5.4.0   2) cuda/8.0   3) openmpi/1.10.6
```

If not, unload and load the appropriate modules using for example

```
$ module unload gnu7/7.3.0
$ module load gnu/5.4.0
```

You can copy and paste these two lines to your `~/.bashrc` file so they will be loaded automatically when you connect on the cluster.

Other useful commands include:

```
$ module avail
$ module -h
```

You should then be able to run `make` or `nvcc` to build your CUDA binary. You can even run your CUDA program at this time, but only the host code can be executed and you'll get runtime error with any GPU access. You'll have to request the GPU resource.

### B.2 Running CUDA program on cluster

For our purposes, it will be sufficient to only use two commands: `sbatch` to submit jobs, and `scancel` to delete jobs.

To run a job with `sbatch`, you must provide a script containing the commands you wish to run. We have provided the script `hw2.sh` for you to run your code on `cme213-cluster`. Note that this script runs all the 3 executables. You can change the script if you would like to run only one. You should change the `WORKDIR` in the script. The command to submit the job script is

```
$ sbatch hw2.sh
```

You can delete a job with

```
$ scancel jobID
```

You can check the status of the queue with either one of these commands:

```
$ squeue
$ squeue -p gpu
$ squeue -u <user name>
```

## C Advice and Hints

- You will need to use the bit shift `<<` and bitwise OR operators `|` to generate a suitable `uint` shift for shifting 4 bytes with one addition.
- Remember that the largest dimension of the grid in one dimension is 65,535.
- In order to perform a batch update, we use two pagerank vectors in our algorithm and switch their roles on every iteration (reading from one and writing to the other).
- For debugging it will be helpful to limit the number of cases being run to 1. In the shift problem do this by passing 0 as the parameter. In the pagerank problem change the values of `num_nodes` and `num_edges`.

- If you need some documentation on CUDA, you can look at the documents uploaded on canvas or visit the CUDA website at `http://docs.nvidia.com/cuda/index.html`.
- For Problem 2, make sure you understand how the sparse matrix is encoded in memory. This will greatly help you figure out the code to write.