CpSc 418 | **Homework 5** | Due: April 6, 2018, 11:59pm
55 points | | Early Bird: April 4, 2018, 11:59pm

Homework 5

Version: April 1, 2018

Template and support code for the assignment can be found at

`https://www.ugrad.cs.ubc.ca/~cs418/2017-2/hw/5/hw5-template.zip`.

Tests included with the template code are not complete, and you should expect that we will run additional tests against your code.

Please submit your solution using the `handin` program. Submit your solution as

`cs418 hw5`

Your submission should consist of the following files:

- `moments-kernel.cu`: Your solution to the moments problem.
- `conv-kernel.cu`: Your solution to the convolution problem.
- Files for the other questions to be announced.
- `hw5.txt` or `hw5.pdf`: Written response portion of the assignment.

These files must be at the top level of your submitted archive and **not** in a subdirectory. Any other files submitted will be ignored, including any other support code files from the assignment template—your submission files must compile and run with the unmodified support files.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on the corresponding programming problem. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully.

We will take off points for code that prints results unless we specifically asked for a print-out or the template code we provided generates it. Using `printf()` when debugging is great, but you need to delete or comment out such calls before submitting your solution. Printing an error to `stdout` when your function is called with invalid arguments is acceptable but not required.

1. **Are We Having Another Moment?** (25 points). In homework 2 you implemented the reduce pattern in Erlang to compute (an approximation of) the k^{th} central moment of a random variable X

   $$\mu_k = E\left[(X - E[X])^k\right]$$

   where $E[X]$ is the *expected value* (or *mean* or *average*) of X. The approximation was constructed from n samples of X given by the data set $\{x_i\}_{i=0}^{n-1}$ using the formulas

   $$E[X] \approx \frac{\sum_{i=0}^{n-1} x_i}{n} \qquad \text{and} \qquad \mu_k \approx \frac{\sum_{i=0}^{n-1} (x_i - E[X])^k}{n}. \tag{1}$$

   Note that the indexing has been shifted to be zero-based since we are working in C for this assignment.

   In this question you will implement the calculation of the expectation and central moment approximations in CUDA. To give the GPU a little more computational work, your code will return all moments $k = 2, 3, \ldots, k_{\max}$ where $k_{\max}$ is a (bounded) run-time parameter; in other words, your code will return approximations of $E[X]$ and $\mu_2, \mu_3, \ldots, \mu_{k_{\max}}$ (a total of $k_{\max}$ values), rather than just one μ_k.

   As discussed in class, reduce problems are generally not well suited to GPU architectures because the theoretical peak CGMA is typically low; however, we often perform reduce operations on the GPU as an intermediate step in a bigger algorithm when:

- Other steps in that algorithm can achieve significant speedup on the GPU.
- We do not want to incur the delay of transferring the data to/from the CPU in order to compute the reduce.

Consequently, there is an incentive to perform the reduce on the GPU and to make it as efficient as possible. For this question we will perform only a reduce operation, but to better simulate the typical use case we will not include the data transfer time in our timing calculations.

In the `Moments-Template/` subdirectory of the assignment archive you will find some template code. Note that this template is much more minimal than that provided in homework 4 or in other parts of this homework; for example, it does not include a CPU implementation, any error checking or any timing, and only provides one type of input set (random data). Your submitted code must build against these template files on the `linXX` machines using the provided `Makefile`; however, we **strongly** recommend that you implement additional features while developing your solution in order to build confidence that you have a correct and efficient implementation. You are welcome to use template code from other CPSC 418 homeworks or questions for this purpose.

Note that your implementation will be using *mixed precision* in the calculation: The input data set $\{x_i\}_{i=0}^{n-1}$ is `float` (single precision floating point), but all internal calculations and the returned data $E[X]$ and $\{\mu_k\}_{k=2}^{k_{\max}}$ is `double` (double precision floating point).

(a) (4 points) Why mixed precision: Give one reason why we do not use `float` for all our computations, and one reason why we do not use `double` for all our input data.[1] Answers which give more than one reason for each will receive zero. Briefly explain your answers.

(b) (12 points) Correctness: Create a file `moments-kernel.cu` and implement the necessary functions listed in `moments-kernel.h`. Note that `moments-kernel.h` defines an upper bound $k_{\max} \leq$ `MAX_MOMENT`. The output array `moments` should contain the following elements: `moments[0]` $= E[X]$, `moments[1]` $= \mu_2$, `moments[2]` $= \mu_3$, $\ldots$ `moments[max_moment-1]` $= \mu_{k_{\max}}$. Ensure that your code works for the conditions listed in the comments of `moments-kernel.h`, and note that we will be testing your code against input vectors other than the one provided in the template.

(c) (2 points) Operations: Assume $n \gg 1$ and $k_{\max} \geq 2$, and consider the computational cost of (1) as a function of n and/or $k_{\max}$. How many additions are needed? How many multiplications are needed? Your answer must include any constants on the leading term(s) in n and/or $k_{\max}$, but you can use $\mathcal{O}()$ for the remaining term(s).

(d) (6 points) Throughput: Choose constants to return from `best_n()` and `best_moment()` in `moments-kernel.cu` to maximize the FLOPS achieved by your implementation while keeping run time under one second and respecting the constraints specified for `compute_all_gpu()`. Note that your code is launching the kernel(s), so you can choose the block and grid size yourself. As a reference point, Ian's implementation achieves a little over $5.5(10^9)$ (double precision) FLOPS. Solutions which achieve at least $4.0(10^9)$ FLOPS will receive full credit, while those that beat $6.0(10^9)$ FLOPS will receive a small bonus, and larger bonuses will be given for even higher throughputs. Ian's solution does use shared memory, but none of the fancy multi-kernel tricks from K&H chapter 8, so there are likely ways of beating

[1]The CUDA compiler `nvcc` does not follow the C standard requiring that all intermediate calculations be done in double precision: It does perform intermediate calculations in single precision if all input and output operands are single precision. Therefore, "the C standard requirement for double precision intermediate computations" is not the reason why we do not use `float` for all our computations.

it. (The extra credit is intended to make this problem *fun* for those who want to see how they can apply the things we have learned about parallel programming and performance analysis and measurement to get a really fast implementation. If you are having fun then go for it, but do not stress about getting the bonus points.)

(e) (1 point) Efficiency: What fraction of peak throughput did your solution achieve on the GTX 1060 3GB GPUs in the lab? If you did not get a FLOPS count for your own code, you can report the fraction of peak throughput achieved by Ian's $5.5(10^9)$ FLOPS solution.

2. **Convolution** (30 points). In this question you will implement 2D convolution. In the `Conv-Template/` subdirectory of the assignment archive you will find the following files:

- `conv-main.cu`: Driver file which
  - Parses the input arguments.
  - Loads the input data set, either from a PGM formatted image file or using one of the predefined array filling routines in `conv-helper.c`.
  - Loads a convolution mask using one of the predefined array filling routines.
  - Runs and times the 2D convolution on the CPU.
  - Allocates space and loads the data onto the GPU.
  - Runs and times the 2D convolution on the GPU.
  - Checks that the GPU results are close enough to the CPU results.
  - May print array data to the screen.
  - May save array data into PGM format image files.
- `conv-helper.c`: A collection of routines to:
  - Create input data for the convolution.
  - Read and write PGM formatted image files.

  You may find these routines useful for debugging.
- `timing418.c` and `cuda-helper.cu`: Same collections of auxiliary functions as in homework 4.
- `conv-kernel.h`: The function prototypes for what you must supply in `conv-kernel.cu`.
- Additional header files.

The code is arranged so that the timing loop does not include data transfer to and from the GPU in order to more accurately measure computational speed.

You must implement your code in `conv-kernel.cu`. This file will contain at least the following three functions:

- `set_up_mask()`: Copies the mask data onto the GPU and records the mask size for subsequent convolutions.
- `clean_up_mask()`: Performs any cleanup required when we are done working with a particular mask. After calling this function, it should be possible to call `set_up_mask()` with new mask data (which may be a different size).
- `conv_gpu()`: Performs a 2D convolution on the given input data using the mask which was previously set up.

You will need to implement additional functions, but they will not be visible outside the file. Remember that `conv-kernel.cu` is the only file which you should submit, so your entire implementation for the first two parts of this question must be contained within this file.

(a) (8 points): Implement a basic 2D convolution kernel on the GPU. This kernel will be called through `conv_gpu()` with argument `tiled = false`. Your code should definitely pass the tests included in `conv-main.cu`, but you may want to write additional tests. It should be capable of correctly handling input parameters of at least $m = n = 1024$ and $p = q = 41$ (and it should be able to handle input parameters with $m \neq n$ and $p \neq q$ as well). A portion of points for this part will be based on correctness and a portion on achieving a minimum throughput. To receive full points for the latter, for $m = n = 1024$ and $p = q = 41$ you should achieve at least 100 GFLOPS (10^{11} FLOPS). There are no bonus marks for faster versions of this kernel—once you get it running sufficiently fast, move on to the next part.

(b) (12 points): Implement a tiled 2D convolution kernel on the GPU. This kernel will be called through `conv_gpu()` with argument `tiled = true`. Your code should definitely pass the tests included in `conv-main.cu`, but you may want to write additional tests. It should be capable of correctly handling input parameters of at least $m = n = 8192$ and $p = q = 21$ (and it should be able to handle input parameters with $m \neq n$ and $p \neq q$ as well). A portion of points for this part will be based on correctness of your tiling and a portion on achieving a minimum throughput. Target throughput for this kernel will be announced on the class Piazza. Remember to set the functions `best_mn()` and `best_pq()` so that we know what parameters to use when confirming your best throughput.

For larger values of m and n we recommend that you:

- Turn off printing and saving of the input and output data, because the data sets get very large. For your convenience, disabling these features can be accomplished by modifying the constants `DEBUG_PRINT` and `DEBUG_SAVE` at the top of `conv_main.cu`.
- Compare your tiled GPU results against your basic GPU results rather than the CPU results, because the CPU version may take several seconds to run. For your convenience, this swap can be accomplished by modifying the constant `COMPARE_WITH_CPU` at the top of `conv_main.cu`.

Please note: The tiled implementation is tricky! Even if you cannot get it to work, you can still answer the next part.

(c) (10 points): Explore the effect of one or two of the following parameters on GPU throughput by holding the other parameters constant and varying your chosen parameter(s):

- (Up to 2 points) Input data size $m = n$.
- (Up to 3 points) Mask size $p = q$.
- (Up to 4 points) Block size.
- (Up to 5 points) Tile size.
- (Up to 6 points) Number of output elements assigned to each thread.

The parameters are listed in order of how much additional code will be needed to explore them; consequently, the latter ones are potentially worth more points. Note that only the "tile size" parameter requires a tiled implementation, so you can explore any of the other parameters even if you did not get the tiled implementation working. If you did get both basic and tiled versions working, be sure to specify from which version(s) your data are taken.

Your exploration should include at least 4 different parameter values (and more is better). Use plots and/or tables to visualize your results. Based on what you have learned about GPU architecture, include a brief hypothesis (a few sentences) as to why you are seeing the throughput effects that you see.

Answers which explore more than two parameters will be scored on the *simplest* two; in other words, *do not explore more than two parameters in your answer.*

Supplementary Material: Making the CUDA Template Code Work

We have provided `Makefile`s in the template code, so you should be able to simply call `make` in the appropriate directory. If the build fails mysteriously it is sometimes useful to `make clean` to clear out any stale object files and then `make` again. If the build is successful, you should get an executable binary file whose name is specified by the second line of the `Makefile`.

Executing the binary requires that your `LD_LIBRARY_PATH` environment variable in your shell is set correctly; for example, on the `linXX` machines it should include `/cs/local/lib/pkg/cudatoolkit/lib64`). You can see the value `LD_LIBRARY_PATH` by typing `echo $LD_LIBRARY_PATH` at the prompt. If it does not exist or has no value, you can set it (again on the `linXX` machines assuming that your account is using the default bash shell) with the command:

```
export LD_LIBRARY_PATH=/cs/local/lib/pkg/cudatoolkit/lib64
```

If it has a value which does not include the CUDA path, you can add the CUDA path (again on the `linXX` machines assuming that your account is using the default bash shell) with the command:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/cs/local/lib/pkg/cudatoolkit/lib64
```

You can either run the appropriate command manually every time you log in, or you can add it to your `~/.bashrc` file so it gets run automatically every time you log in.