CpSc 418 **Homework 5** Due: Nov. 27, 2018, 11:59pm
Early Bird: Nov. 26, 2018, 11:59pm

55 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw5
Your submission should consist of two files:

- `hw5.cu`: CUDA source code for the coding parts your solution.
- `hw5.pdf` PDF for the written response parts of your solution and the plots.

A templates for hw5.cu is available at
http://www.ugrad.cs.ubc.ca/~cs418/2018-1/hw/5/src/hw5.cu.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on all of the programming problems. 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 print-out. For this assignment, the functions you write should return the specified values, but they should not print anything to `stdout`. Using io:format when debugging is great, but you need to delete or comment-out such calls before submitting your solution. Printing an error message to `stdout` when your function is called with invalid arguments is acceptable but not required. Your code must fail with some kind of error when called with invalid arguments.

1. Hello CUDA **(25 points)** The template file, hw5.cu provides CPU and GPU implementations of `saxpy` along with a function to compare the CPU and GPU results, and functions for timing measurements. Log on to one of the `linXX.ugrad.cs.ubc.ca` machines where `01` $\leq$ `XX` $\leq$ `25` – these have CUDA capable GPUs. Copy hw5.cu, Makefile, and hw5.sh to a folder under your home directory. Make sure that hw5.sh is executable: chmod +x hw5.sh You should be able to compile hw5.cu by giving the command (at the linux shell prompt): make You can then run the `saxpy` test with the command: ./hw5.sh saxpy test 1000000 1.2 This generates vectors for `x` and `y` of length `1000000` and sets `a` to `1.2`.

   The provided Makefile is really simple. Likewise, hw5.sh just runs `make` to ensure that `hw5.cu` has been compiled. If the `make` succeeds, `hw5.sh` runs `./hw5` with any arguments that you provided on the command line.

   - Why do we provide these files?
     So you people won't get stuck looking for `nvcc` or having problems with getting the right library modules loaded.
   - Should you try this at home?
     Of course! You will probably need to modify the definition of `NVCC` in Makefile and the `LD_LIBRARY_PATH` in hw5.sh to match your installation. Note that timing measurements should be performed on one of the `linXX.ugrad.cs.ubc.ca` machines.

   Congratulations! You have compiled and executed a CUDA program. you're nearly done with this first question.

   (a) **(6 points)** You can measure the time to execute `saxpy` with lists of length N by giving the commands:

```
$ ./hw5.sh saxpy time_kernel N      % time to execute the kernel
$ ./hw5.sh saxpy time_gpu N         % total time for GPU including memory copies between CPU and GPU
$ ./hw5.sh saxpy time_cpu N         % total time when computing saxpy on the CPU
```

Make a table showing these three timing measurements for $N = 10,000$, $N = 100,000$, $N = 1,000,000$, $N = 10,000,000$, and $N = 100,000,000$. Make your timing measurements on one of the `linXX.ugrad.cs.ubc.ca` machines and state which machine you used in your `hw5.pdf`.

(b) **(2 points)** Under what conditions is the GPU (i.e. `time_kernel`) faster than the CPU (i.e. `time_CPU`)? Give a short explanation for the conditions under which `time_kernel` < `time_cpu`.

(c) **(2 points)** Under what conditions is the GPU including the memory transfers faster than the CPU? Give a short explanation for the conditions under which `time_gpu` < `time_cpu`.

(d) **(3 points)**

- How many floating point operations does `saxpy` perform per array element?
- Give a one or two sentence justification for your answer.
- What choice of N from question 1a gives the highest number of floating point operations per second?

(e) **(3 points)**

- How many **bytes** of global memory data (loads and stores) are transferred between the GPU and its global memory per array element.
- Give a one or two sentence justification for your answer.
- What choice of N from question gives the highest memory bandwidth?

(f) **(3 points)** The GTX 1060 GPU (the GPUs in the `linXX` machines) is specified as having a peak floating point performance of 3470 GFlops (base clock) or 3935 GFlops (boosted clock). What percentage of the peak GFlops for the base-clock rate is achieved by `saxpy`? Include a short explanation of your answer.

(g) **(3 points)** The GTX 1060 GPU (the GPUs in the `linXX` machines) is specified as having a peak memory bandwidth of 192 Gbytes/sec. What percentage of the peak memory bandwidth is achieved by `saxpy`? Include a short explanation of your answer.

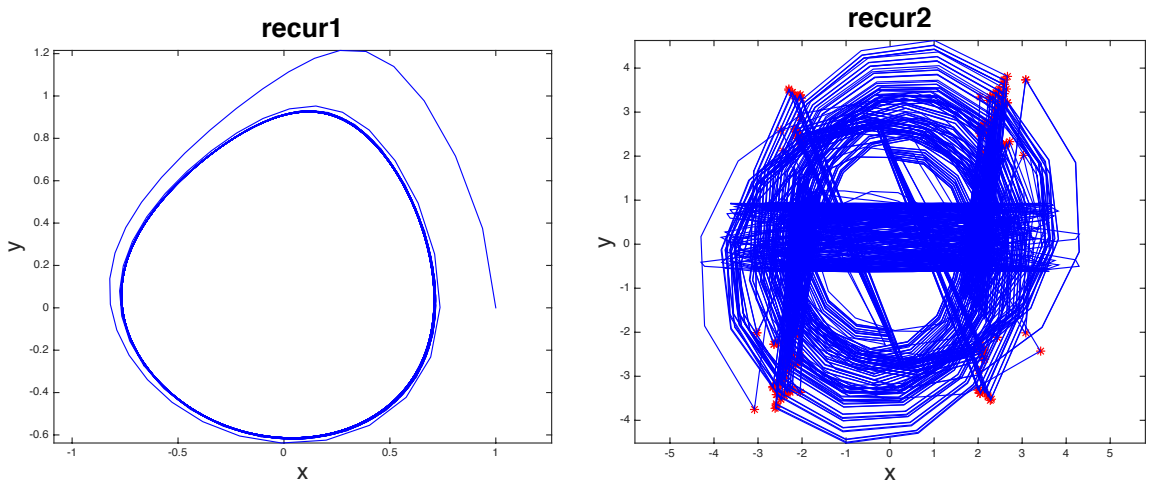
(h) **(3 points)** Based on your timing measurements from question 1a, what is the time for a kernel launch? Explain how you got your answer. Feel free to make more timing measurements and include that data in your answer.

2. Recurrences (**30 points**)
Each year, I come up with a recurrence to compute. This gives you a chance to push for peak GFlops without fighting a memory bandwidth bottleneck. Make your timing measurements on one of the `linXX.ugrad.cs.ubc.ca` machines and state which machine you used in your `hw5.pdf`.

(a) **(2 points)** `recur1` computes the recurrence

$$\begin{aligned} \texttt{x[j+1]} &= \texttt{x[j]} - \tfrac{1}{4}\texttt{y[j]} + \tfrac{1}{8}(\texttt{y[j]})^2 - (1/16)(\texttt{x[j]})^3 \\ \texttt{y[j+1]} &= \texttt{y[j]} + \tfrac{1}{4}\texttt{x[j]} + \tfrac{1}{8}(\texttt{x[j]})^2 - (1/16)(\texttt{y[j]})^3 \end{aligned}$$

This recurrence is implemented by the function `recur1_kernel` and `recur1_cpu` in `hw5.cu` How many floating point operations does `recur1_kernel` perform for each value of `j`?
Note 1: `recur1_cpu` performs the same number of floating point operations per value of `j`.
Note 2: if a multiplication and add can be performed as a fused multiply-add, count them as two operations.



`recur1` `recur2` – * indicates point where `flag` changes

Figure 1: 2000 steps of `recur1` and `recur2` starting at $x = 1$, $y = 0$.

(b) **(3 points)** Use

`hw5.sh recur1 time_kernel n=`N `m=`M

to measure the *throughput* of `recur1_kernel` for various choices of N and M. Find choices of N and M that maximize this throughput. Each choice should have a total run time of less than one second. Report your data and the peak throughput.
Note: to maximize throughput, `recur1_kernel` calculates the sequence for multiple starting points. N is the number of starting points. The test code uses the function `init_data3` to generate such data. The recurrence `recur1` is "stable" (i.e. doesn't diverge to $\pm\infty$) as long as neither `x` nor `y` grows too large. I'm not sure how large is "too large", but the values generated by `init_data3` are safe.

(c) **(2 points)** How many GFlops does `recur1_kernel` achieve with your peak-throughput choices for N and M? What percentage of the peak floating-point rate of the GTX 1060 does the kernel achieve?

(d) **(5 points)** `recur2` computes the recurrence

$$\begin{aligned}
\texttt{x[j+1]} &= 0.38 * \texttt{x[j]} + 0.32 * \texttt{y[j]}, && \text{if } (|x[i]| > 2) \wedge (|y[i] > 2)\\
&= 0.96 * \texttt{x[j]} - 0.52 * \texttt{y[j]}, && \text{if } \texttt{flag} \wedge ((|x[i]| \le 2) \vee (|y[i] \le 2))\\
&= 0.94 * \texttt{x[j]} + 0.37 * \texttt{y[j]}, && \text{if } \neg\texttt{flag} \wedge ((|x[i]| \le 2) \vee (|y[i] \le 2))\\
\texttt{y[j+1]} &= 0.25 * \texttt{x[j]} - 0.64 * \texttt{y[j]}, && \text{if } (|x[i]| > 2) \wedge (|y[i] > 2)\\
&= 0.47 * \texttt{x[j]} - 0.95 * \texttt{y[j]}, && \text{if } \texttt{flag} \wedge ((|x[i]| \le 2) \vee (|y[i] \le 2))\\
&= -0.62 * \texttt{x[j]} + 0.96 * \texttt{y[j]}, && \text{if } \neg\texttt{flag} \wedge ((|x[i]| \le 2) \vee (|y[i] \le 2))\\
\texttt{flag[j+1]} &= \neg\texttt{flag[j]}, && \text{if } (|x[i]| > 2) \wedge (|y[i] > 2)\\
\texttt{flag[j+1]} &= \texttt{flag[j]}, && \text{if } (|x[i]| \le 2) \vee (|y[i] \le 2)
\end{aligned}$$

This recurrence is implemented by the function `recur2_cpu` in `hw5.cu`. Complete the implementation of `recur2_kernel`. This GPU version should compute the same result as `recur2_cpu`, for example, as tested by

```
hw5.sh recur2 test
```

Note: this recurrence is numerically unstable. It won't blow up to $\pm\infty$, but if you start with two, nearby initial points, the resulting sequences will diverge from each other. This means that if you change any details to something that is mathematically equivalent but performs a different sequence of floating point operations, the simple test in `hw5.cu` is likely to fail for everything except for small values of `m`.

(e) **(2 points)** How many floating point operations does `recur2_cpu` perform for each value of `j`? Presumably, your `recur2_kernel` will perform the same number of floating point operations per value of `j`.

(f) **(3 points)** Use

```
hw5.sh recur2 time_kernel n=N m=M
```

to measure the *throughput* of `recur2_kernel` for various choices of N and M. Find choices of N and M that maximize this throughput. Each choice should have a total run time of less than one second. Report your data and the peak throughput.

(g) **(3 points)** Which kernel, `recur1_kernel` or `recur2_kernel` achieves the larger number of GFlops. Briefly explain the key issue that accounts for the performance difference.

(h) **(10 points)** Revise `recur1_kernel` – call it `recur1a_kernel` to maximize its throughput. Any optimizations are allowed, and if your code is faster than my solution, you will get extra credit points. Here are some places you can start:

- C specifies that *all* floating point operations must be done using double precision arithmetic, and rounded to single precision if the result is stored in a `float`. In general, GPUs have **much** better single-precision performance than double-precision performance. The `nvcc` compiler will generate single-precision operations *if* both operands are single precision floating point numbers. Here's the subtle catch: constants, such as `1.234` are double-precision. If you write it with a trailing `f`, e.g. `1.234f`, the constant is single precision. Try changing the floating-point constants in `recur1_kernel` to single-precision.
- Experiment with the total number of blocks, the block size, and the number of recurrence steps.
- Try loop unrolling. You could perform multiple steps of the recurrence for each execution of the loop body, or you could modify a thread to handle multiple values of `i` (i.e. multiple sequences of the recurrence) in a single thread.
- All optimizations are allowed – we will be tolerant of variations in the result due to round-off errors, but your solution needs to be mathematically equivalent to the original problem.

Now that you know the rules:

i. Write your optimized `recur1a_kernel`.
ii. Give a brief explanation in your `hw5.pdf` of each optimization that you performed.
iii. What is GFlops achieved by your optimized kernel? What percentage of the peak GFlops for a GTX 1060 does your kernel achieve?
iv. What is the speed-up of your kernel compared with `recur1_cpu`?

Note: you can go wild exploring the performance trade-offs of the GPU. If that's fun, do it. You can get full credit by doing a few simple optimizations and reporting the result.


Unless otherwise noted or cited, the questions and other material in this homework problem set is copyright 2018 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license `http://creativecommons.org/licenses/by/4.0/`