CMPT 295 Assignment 8 (2%)

Submit your solutions by Friday, November 22, 2019 1pm.
Remember, when appropriate, to justify your answers.

1. [6 marks] *Latency and Throughput Bounds*

   A superscalar with six function units can perform integer multiplication and floating point multiplication with computation times (measured in clock cycles) as follows:

	Latency	Issue	Capacity
`mul`	2	1	3
`fmul`	5	5	3

   For the purposes of the calculations, assume that the cycle time is 450 ps, i.e., an equivalent scalar machine would run at a peak rate of 2.22 GIPS.

   (a) [2 marks] In the case of `mul`, calculations can be issued every clock cycle ($I = 1$), but they take 2 clock cycles to complete ($L = 2$). Does it make sense that the issue time is strictly less than the latency? Why or why not?

   Would it make sense if the issue time was strictly greater than the latency? Why or why not?

   (b) [2 marks] Compute the latency bound and throughput bound for `mul` and `fmul`. Express your answers in cycles per instruction (CPI) and GIPS.

   (c) [1 mark] Consider a program that does nothing but a long sequence of multiplication instructions. If there are no data dependencies among the multiplications, i.e., they could be run in any order with the maximum degree of parallelism, how quickly could multiplications be performed? Express your answer using GIPS. Give one answer for integer `mul` and one for floating-point `fmul`.

   (d) [1 mark] If the program instead contained one long critical path, i.e., one linear data dependent sequence of multiplications like:

   ```
   x <- x * a
   x <- x * b
   x <- x * c
   x <- x * d
   x <- x * e
   x <- x * f
   . . .
   ```

   how quickly could multiplications be performed? Again, give two answers: one for `mul`, and one for `fmul`.

2. [14 marks] *Code Optimization*

   There are many transformations you can do to your code to achieve a faster executable. The one that is stressed the most throughout your Computing Science degree is to choose/design the best algorithm. Here "best" refers to the *asymptotic* behaviour of its running time as the input size N gets large.

   The big-O notation is employed to compare algorithms in a way that is independent of the characteristics of both the machine and the programming language. E.g., A $O(N)$ algorithm scales *linearly*, i.e., doubling the input size will result in doubling in the running time, but a $O(N^2)$ algorithm scales *quadratically*, i.e., doubling the input size will result in quadrupling the running time.

The virtue of big-O notation is that any lower ordered effects or leading constants are ignored. But that is also the shortcoming of the big-O. As a programmer, by paying careful attention to architecture, memory references, critical paths, and so forth, you can often transform your code and reduce leading constants by 20% or more. These sorts of optimizations can often make the difference between a completely usable product and something that, well, ..., isn't.

You are going to try your hand at optimization. The base code contains a function `min_max_avg()` that does some data analysis on an $N \times M$ grid that represents a *topographic map*. If you've never heard of such a thing, picture a 2-dimensional array `grid[N][M]` keyed by latitude and longitude. Each entry in the grid represents that location's height above sea level.

The function computes the min, max and average height of the entire map, but also of each row (latitude) and each column (longitude). With such information in hand, it will be easier to locate any strategically-important high ground, but also any wind-sheltered valleys.

The function takes 12 arguments:

- `float grid[N][M]` - the 2-dimensional array of heights (input);
- `int N` - the number of rows (input);
- `int M` - the number of columns (input);
- `float *min` - a pointer to the minimum height of the entire map (output);
- `float *max` - a pointer to the maximum height of the entire map (output);
- `float *avg` - a pointer to the average height of the entire map (output);
- `float col_min[M]` - an array of minimums, one for each column of the map (output);
- `float col_max[M]` - an array of maximums, one for each column of the map (output);
- `float col_avg[M]` - an array of averages, one for each column of the map (output);
- `float row_min[N]` - an array of minimums, one for each row of the map (output);
- `float row_max[N]` - an array of maximums, one for each row of the map (output); and
- `float row_avg[N]` - an array of averages, one for each row of the map (output).

Your code will be judged on the following:

(a) [−14 marks] *Correctness:* If you introduce a bug, you will receive a mark of 0 for this question.

(b) [3 marks] *Reducing Memory Aliasing:* There are several examples of memory aliasing in the function. Some of them may cause unintentional bugs. You will reduce the occurrences of memory aliasing to zero. This *should* — but might not — decrease the overall running time.

(c) [6 marks] *Loop Efficiency:* The current layout of the loops is inefficient. Use some of the techniques suggested in class and in your textbook to reduce the work.

(d) [5 marks + 2 BONUS] *Running Time:* There are more optimizations possible than just these. Your code will be benchmarked by the marker. If your code achieves a cycle time that is less than X, then you will receive 5/5. What is X, you might ask? Since optimization is usually an open ended problem, X is going to be kept secret.

To properly reward the most motivated among you, the 25 best solutions below X will be awarded 2 BONUS marks.

For your reference, the benchmark of the base code on `cpu-csil10.csil.sfu.ca` is 7500 cycles; on the computers in ASB 9840, it is 5150 cycles. Use "`./x < smallgrid`" to test. A basic set of optimizations yields benchmarking numbers of 5000 and 3700 respectively. You should consider X to be strictly less than these numbers.

Submit your code to CourSys for testing and benchmarking (part (d)). Print a hardcopy of your code (handwritten code will receive a grade of 0) and annotate your code with an explanation of the reduction in memory aliasing (part (b)) and gains in loop efficiency (part (c)).