263-0007-00: Advanced Systems Lab
Assignment 2: 80 points
Due Date: Th, March 12th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2020/
Questions: fastcode@lists.inf.ethz.ch

Exercises:

1. *Short project info (10 pts)*
   Go to the list of milestones for the projects. If you have not done that yet, please register your project there. Read through the different points and fill in the first two with the following about your project (be brief):

   Solution: This will be different for each student.

2. *Optimization Blockers (25 pts)*

   In this exercise, we consider the following short computation that is available in Code Expert:

```
1  void slowperformance1(double *w, double *x, double *y, double *z, int n) {
2    for(int i = 0; i < n; i++) {
3        for (int j = 0; j < n; j++) {
4            if ((i + j) % 2) {
5                z[i] += 1.0 / (x[i*n + j] * sqrt(w[i]));
6            }
7            else {
8                z[i] = compute(w[i], y[i*n + j], z[i]);
9            }
10           z[i] *= x[i*n + j];
11       }
12   }
13 }
```

   Discussion:

   (a) Create a table with the runtime numbers of each function that you created. Briefly discuss the table.

   Solution:

Implementation	Impl. 1	Impl. 2	Impl. 3	Impl. 4	Impl. 5	Impl. 6
Runtime (cycles)	24.9M	7.88M	5.27M	4.09M	2.71M	1.96M

   The table above reports runtime in cycles for six different implementations of the above code, with optimizations turned on (`-O3 -fno-tree-vectorize`). The M stands for millions. These numbers were recorded on a Intel(R) Xeon(R) Silver 4210 @ 2.20GHz Cascade Lake with hyperthreading disabled, and compiled with GCC 8.3.1.

   Implementation 1 is the original code. Implementation 2 unrolls two iterations in both loops to eliminate the if-condition branch. Implementation 3 inlines function `compute`, removes invariant code from the inner loop and applies strength reduction to transform divisions with constants to multiplications. At this step, all the operations in the inner loop are additions and multiplications. We can already observe a speedup of 4.7 over the original code. Implementation 4 performs scalar replacement, and Implementation 5 unrolls four times the outer loop to improve the ILP (with scalar replacement). Finally, Implementation 6 is the same as 5 but with eight iterations unrolled instead of four.

   (b) What is the speedup of function *maxperformance* compared to *slowperformance1*?

   Solution: The speedup of Implementation 6 is 12.7.

   (c) What is the performance in flops/cycle of your function *maxperformance*.

   Solution: Implementation 6 performs $3n^2 + 8n$ flops. The performance is 1.53 flops/cycle for $n = 1000$.

3. *Microbenchmarks(45 pts)*

In Code Expert, we provide three functions in file `sigmoid.h` that implement a floating point square root instruction (*sqrtsd*) and two commonly used activation functions in Neural Networks (*sigmoid1* and *sigmoid2*). Your task is to write a program (without vector instructions, i.e., standard C) in Code Expert that benchmarks the maximum and minimum latency and inverse throughput (also called "gap") of *sqrtsd* and *sigmoid1*. In addition, the latency and gap of *sigmoid2* for inputs 1.0 and 0.0. More specifically:

- Read and understand the code.
- Implement the functions provided in the skeleton in file `microbenchmark.cpp`:

```
void    initialize_microbenchmark_data (microbenchmark_mode_t mode);
double  microbenchmark_get_sqrt_latency();
double  microbenchmark_get_sqrt_gap();
double  microbenchmark_get_sigmoid1_latency();
double  microbenchmark_get_sigmoid1_gap();
double  microbenchmark_get_sigmoid2_latency();
double  microbenchmark_get_sigmoid2_gap();
```

- You can use the `initialize_microbenchmark_data` function for any kind of initialization that you may need (e.g. for initializing the input values for the sigmoid functions).
- Function *microbenchmark_get_sqrt_latency* should return the measured latency of the *sqrtsd* function. Analogously, the other functions should return the latency (or gap) of the function implied by its name. The gap is the inverse of the throughput. The latency and gap have to be measured in cycles.
- Note that the latency and gap of some instructions (e.g. square root) can vary depending on their input. Thus, your task is to find the minimum and maximum latency and gap of *sqrtsd* and *sigmoid1*. In addition, the latency and gap of *sigmoid2* for inputs 1.0 and 0.0.
- It is not allowed to manually inline the functions in `sigmoid.h` into the implementation of your microbenchmarks.

Discussion:

(a) Do the latency and gap of the square root instruction match what is in the Intel Optimization Manual?
Solution: Yes, the manual reports a latency and gap for *sqrtsd* (Skylake) of 18 and 6 cycles respectively. This is consistent with the microbenchmarks.

(b) Based on the dependency, latency and gap information of the operations used to implement function *sigmoid1*. Is the measured latency and gap of the *sigmoid1* close to what you would expect?
Solution: Yes, the *sigmoid1* function consists of a multiplication, an addition, a square root instruction and a division. Note that the addition and multiplication will be fused into an FMA instruction. According to the Intel's manual, division has a latency and gap of 14 and 4 cycles respectively and FMA has a latency and gap of 4 and 0.5 cycles. This gives a theoretical latency of 4+18+14 = 36 cycles which is consistent with the measurements. To determine the theoretical gap, note that division and square root share the same port and execution unit and they become the bottleneck. Thus, the gap is 6+4 = 10 cycles which is also close to the measurements.