263-0007-00: Advanced Systems Lab
Assignment 2: 80 points
Due Date: Th, March 16th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2023/
Questions: fastcode@lists.inf.ethz.ch

Exercises:

1. *Short project info (5 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 together with your team. It is enough if only one member of the team submits this in the project system. Consider the following about your project while filling the points (be brief):

   Point 1) An exact (as much as possible) but also short, problem specification.
   For example for MMM, it could be like this:
   Our goal is to implement matrix-matrix multiplication specified as follows:
   Input: Two real matrices A, B of compatible size, $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$. We may impose divisibility conditions on n, k, m depending on the actual implementation.
   Output: The matrix product $C = AB \in \mathbb{R}^{n \times m}$.
   Give the name of the algorithm you plan to consider for the problem and a precise reference (e.g., a link to a publication plus the page number) that explains it.

   Point 2) A very short explanation of what kind of code already exists and in which language it is written.

   Solution: This will be different for each student.

2. *Optimization Blockers (30 pts)*

   In this exercise, we consider the following short computation that is part of the supplied code in Code Expert:

```
1  void slow_performance1(mat* x, mat* y, mat*z) {
2    double t1;
3    for (int i = 0; i < z->n1; i++)
4      for (int j = 1; j < z->n2 - 1; j++) {
5        if (i % 2) {
6          t1 = mat_get(z,i,j)/sqrt(mat_get(y,0,i%2)) + (mat_get(x,i,j) + C1) *
7                                                       (mat_get(x,i,j) - C1);
8        } else {
9          t1 = mat_get(z,i,j)/sqrt(mat_get(y,0,i%2)) + C1*mat_get(x,i,j);
10       }
11       mat_set(z,i,j-1, mat_get(z,i,j-1)*cos(C2*M_PI*j));
12       mat_set(z,i,j,t1);
13       mat_set(z,i,j+1,fmax(mat_get(z,i,j+1), mat_get(x,i,j+1)));
14     }
15 }
```

   Do the following:

   - Read and understand the code. It enables you to register functions with the same signature, which will be timed in a microbenchmark fashion.
   - Create new functions where you perform optimizations to improve the runtime. For example, strength reduction, inlining, removing function calls, and others.
   - You may apply any optimization that produces the same result in exact arithmetic.
   - For every optimization you perform, create a new function in `comp.cpp` that has the same signature as *slowperformance1* and register it to the timing framework through the *register_function* function in `comp.cpp`. Let it run and, if it verifies, it will print the runtime in cycles.

- Implement in function *maxperformance* the implementation that achieves the best runtime. This is the one that will be autograded by Code Expert.
- For this task, the Code Expert system compiles the code using GCC 11.2.1 with the following flags: `-O3 -march=skylake -mno-fma -fno-tree-vectorize`. Note that with these flags vectorization and FMAs are disabled. It is also not allowed to use pragmas that modify the compilation environment.
- It is not allowed to use vector intrinsics or FMAs to speedup your implementation.
- You can assume that the matrices are of size $n \times (n+1)$ with $n = 100$;

Discussion:

(a) Create a table with the runtime numbers of each new function that you created (include at least 3). Briefly discuss the table explaining the optimizations applied in each step. Mention also the maximum speedup that you achieved.

Solution:

Implementation	Impl. 1	Impl. 2	Impl. 3	Impl. 4	Impl. 5
Runtime (cycles)	1260K	490K	289K	78.7K	24.5K

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

Implementation 1 is the original code. Implementation 2 removes calls to `mat_set` and `mat_get` by accessing the arrays directly. Implementation 3 unrolls the outer loop once. This allows to remove the if-condition. In addition, some computations are precomputed. Implementation 4 unrolls the inner loop three times to remove cosine operations. In addition, values that stay constant across iterations are precomputed. Finally, implementation 5 removes extra memory accesses that are repeated between iterations. We achieved a final speedup of 51.

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

Solution: Implementation 5 performs 29 flops in the body of the inner-most loop and 30 flops in the outer-most loop. The outer and inner loops perform 50 and 32 interations respectively. Thus, $W = 29 \cdot 32 \cdot 50 + 30 \cdot 50 = 47.9K$ flops. The performance is therefore $\pi = \frac{47.9K}{24.5K} = 1.95$ flops/cycle.

(c) Consider the theoretical peak performance for one core, without SIMD vector instructions and without FMAs of the machine running the programs submitted to Code Expert. What percentage of this theoretical peak performance did you achieve?

Solution: The theoretical peak performance is 2 flops/cycle with the given flags. Thus, we achieved 97% of the peak performance.

3. *Microbenchmarks(40 pts)*

Your task is to write a program (without vector instructions, i.e., standard C) in Code Expert that benchmarks the latency and inverse throughput (also called "gap" in the class) of the division operation on doubles and the max operation (which returns the larger of two double precision floating-point numbers). In addition, the latency and gap of the function $f(a) = \frac{1}{\sqrt{a^2-1}}$. We provide the implementation of $f(a)$ in `foo.h`. More specifically:

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

```
void    initialize_microbenchmark_data (microbenchmark_mode_t mode);
double  microbenchmark_get_max_latency();
double  microbenchmark_get_max_gap();
double  microbenchmark_get_div_latency();
double  microbenchmark_get_div_gap();
double  microbenchmark_get_foo_latency();
double  microbenchmark_get_foo_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).
- Note that the latency and gap of floating-point square root and division can vary depending on their inputs. Thus, you are also required to find the minimum latency and gap for division and function $f(a)$. Hint: You can try using values where performing those operations becomes trivial.
- Note that the compiler usually transforms short implementations of the max operator that use the ternary operator (also called conditional operator) into its respective max assembly instruction.
- It is not allowed to manually inline the function in `foo.h` into the implementation of your microbenchmarks.
- Make sure that your benchmarks yield stable measurements between runs.
- It is not allowed to use assembly nor intrinsics.

Additional information:

- Our Code Expert system already has Turbo Boost disabled. However, note that CPUs may throttle their frequency below the nominal frequency. To ensure that the CPU is not throttled down when running the experiments, one can **warm up** the CPU before timing them.
- For this task, our Code Expert system uses GCC 11.2.1 to compile the code with the following flags: `-O3 -fno-tree-vectorize -march=skylake -mno-fma -fno-math-errno`. Note that with these flags vectorization and FMAs are disabled. The `-fno-math-errno` flag is used to guarantee that the `sqrt` function call is converted to its respective instruction.

Discussion:

(a) Do the latency and gap of floating point max and division match what is in the Intel Optimization Manual? If no, explain why. (You can also check Agner's Table).

Solution: Yes, the manual reports a latency and gap for `maxsd` instructions (Skylake) of 4 and 0.5 cycles respectively which is consistent with the microbenchmarks. For the division (*divsd*), the manual reports 14 and 4 cycles for latency and gap respectively and Agner reports 13-14 and 4 cycles respectively. The measured latency and gap in the microbenchmarks are 14 and 4 cycles for a random value and 13 and 4 cycles for a trivial input value. These values are consistent with both, the Intel's manual and Agner's measurements.

(b) Based on the dependency, latency and gap information of the floating point operations, is the measured latency and gap of function $f(a)$ close to what you would expect? Justify your answer.

Solution: Yes, function $f(a)$ consists of a multiplications, an addition, a square root operation, and a division. This gives a theoretical latency of 4 (mul) + 4 (add) + 18 (sqrt) + 14 (div) = 40 cycles which is consistent with the measurements. Note that Agner reports a latency of 15-16 cycles for square root. If we consider his numbers, then we would expect 37-38 cycles of latency. For the theoretical gap, the square root and the division become the bottleneck because they share the same execution unit in port 0. Thus, the gap is 6 (sqrt) + 4 (div) = 10 cycles which is also consistent with the measurements.

(c) Assume that we implement two different versions of $f(a)$ as follows:

$$f_2(a) = \frac{1}{\sqrt{(a+1)(a-1)}}, \qquad f_3(a) = \sqrt{\frac{1}{a^2-1}}.$$

Further, assume that we compile them with FMA enabled (i.e., we remove the `-mno-fma` flag). Will the latency and gap be different for these new implementations of $f(a)$ compared to the original one which was compiled without FMAs? Justify your answer and state, for each function, the expected latency and gap in case you think it will change.

Solution: No FMA instruction will be generated in $f_2(a)$ and the addition and subtraction can be scheduled in parallel. Thus, the latency will stay the same as the original: 4 (add | sub) +

4 (mul) + 18 (sqrt) + 14 (div) = 40 cycles. In $f_3(a)$ the multiplication and subtraction will be fused into an FMA. Thus, the latency will decrease to 4 (fma) + 14 (div) + 18 (sqrt) = 36 cycles. The gap will not change for both functions since the square root and division are the bottleneck in all functions.