263-0007-00: Advanced Systems Lab

Assignment 2: 80 points
Due Date: Th, March 18th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2021/
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 with the following about your project (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 available in Code Expert:

```
#define C1 0.2
#define C2 0.3
void slowperformance1(float *x, float *y, float *z, int n) {
    for (int i = 0; i < n - 2; i++) {
        x[i]      = x[i] / M_SQRT2 + y[i] * C1;
        x[i+1]   += z[(i % 4) * 10] * C2;
        x[i+2]   += sin((2 * M_PI * i) / 3) * y[i+2];
    }
}
```

   This is part of the supplied code in Code Expert:

   - 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, loop unrolling and scalar replacement as discussed in the lecture, strength reduction, inlining, 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 8.3.1 with the following flags: `-O3 -fno-tree-vectorize`. Note that with these flags vectorization and FMAs are disabled.
   - It is not allowed to use vector instrinsics or FMA to speedup your implementation.

- Hint: The code above may present additional overhead due to casts between double precision and single precision floating point. It is advisable to avoid these casts by using only one precision accross all operations.

Discussion:

(a) Create a table with the runtime numbers of each new function that you created (include at least 2). Briefly discuss the table explaining the optimizations applied in each step.
Solution:

Implementation	Impl. 1	Impl. 2	Impl. 3	Impl. 4	Impl. 5
Runtime (cycles)	53.33K	53.21K	7.60K	3.97K	2.27K

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 restructures the loop body to only process one element of array `x` per iteration. Since no optimizaitn is applied yet, the runtime stays the same. Implementation 3 unrolls the loop three times. This allows to also replace the function calls to `sin` function with constant values. This achieves a 7× speedup. Implementation 4 removes divisions by multiplying by the inverse of `M_SQRT2`. In addition, values that stay constant across iterations are precomputed, achieving a 13.4× speedup compared to the original code. Finally, implementation 5 unrolls the loop four times. This allows to precompute the values of `z[(i%4)*10]*C2`. After this, we achieve a 23.5× speedup compared to the orginal code.

(b) What is the speedup of function *maxperformance* compared to *slowperformance1*?
Solution: The speedup of implementation 5 is 23.5.

(c) What is the performance in flops/cycle of your function *maxperformance*.
Solution: Implementation 5 performs $4n+4$ flops. The performance is 1.8 flops/cyc for $n = 1022$.

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") of floating point addition and division on doubles. In addition, the latency and gap of function $f(x) = \frac{x}{1+x^2}$. We provide the implementation of $f(x)$ 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_add_latency();
double  microbenchmark_get_add_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 division can vary depending on its inputs. Thus, you are also required to find the minimum latency and gap for division and function $f(x)$. Hint: You can try using values where performing a division becomes trivial.
- It is not allowed to manually inline the function in `foo.h` into the implementation of your microbenchmarks.

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 8.3.1 to compile the code with the following flags: `-O3 -fno-tree-vectorize -march=skylake`. Note that with these flags vectorization is disabled but FMAs are enabled.

Discussion:

(a) Do the latency and gap of floating point addition 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 *addsd* (Skylake) of 4 and 0.5 cycles respectively. For division (*divsd*), the manual reports 14 and 4 cycles for latency and gap respectively. These numbers are consitent with the microbenchmarks.

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

Solution: Yes, function $f(x)$ consists of a multiplication, an addition and a division. Note that the addition and multiplication will be fused into an FMA instruction. According to the Intel's manual, an FMA instruction has a latency and gap of 4 and 0.5 cycles. This gives a theoretical latency of 4+14 = 18 cycles which is consistent with the measurements. For the theoretical gap, the division become the bottleneck. Thus, the gap is 4 cycles which is also close to the measurements.

(c) Will the latency and gap of $f(x)$ change if we compile the code with flags `-O3 -fno-tree-vectorize` (i.e., with FMAs disabled)? Justify your answer and state the expected latency and gap in case you think it will change.

Solution: Since FMAs are disabled, the addition and multiplication in function $f(x)$ will not fuse. Thus, the new latency will be $4 + 4 + 14 = 22$. The gap will remain the same as division is still the bottleneck.