263-0007-00: Advanced Systems Lab

Assignment 2: 80 points
Due Date: Th, March 14th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2024/
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:

```c
void slow_performance1(mat* x, mat* y, mat*z) {
    double t1;
    for (int i = 0; i < z->n1; i++) {
        for (int j = 0; j < z->n2; j++) {
            if (i % 2) {
                t1 = mat_get(z,i,j)/cos(C*M_PI*j) + mat_get(x,i,j);
            } else {
                t1 = mat_get(z,i,j)/(sqrt(mat_get(y,i,0)) * sqrt(mat_get(x,0,j%6)))
                    + mat_get(y,i,j);
            }
            mat_set(z,i,j,t1*cos(mat_get(x,0,j%3)));
        }
    }
    for (int i = 0; i < z->n1; i++) {
        for (int j = 0; j < z->n2; j++) {
          mat_activate(z, i, j, ALPHA);
        }
    }
}
```

   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.
- 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. Example.
- 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 $n \times n$ with $n = 100$;
- You can assume $0 < x_{ij} < \pi/2$, $y_{ij} > 0$, and $z_{ij} > 0$ for all $0 \leq i, j \leq n$;

(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	Baseline	Impl. 1	Impl. 2	Impl. 3	Impl. 4
Runtime (cycles)	873K	611K	291K	47.8K	27.8K

The table above reports runtime in cycles for five 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.

Baseline 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 to avoid executing the branch and precomputes constant values of the square root and the cosine. Implementation 4 unrolls the inner loop six times to remove cosine operations. Finally, implementation 5 hoists computations from the innermost loop, unrolls the cleanup code, and removes reduntant calls to abs and max. We achieved a final speedup of 31.

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

Solution: Implementation 4 performs 57 flops in the body of the inner-most loop and 57 flops in the outer-most loop. The outer and inner loops perform 50 and 16 interations respectively. Thus, $W = 57 \cdot 16 \cdot 50 + 57 \cdot 50 = 48.4K$ flops. The performance is therefore $\pi = \frac{48.4K}{27.8K} = 1.75$ 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 87% 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 reciprocal throughput of the addition and square root of doubles. In addition, the latency and reciprocal throughput of the function $f(a) = \sqrt{a^3}$. 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_add_latency();
double  microbenchmark_get_add_rec_tp ();
double  microbenchmark_get_sqt_latency();
double  microbenchmark_get_sqt_rec_tp ();
double  microbenchmark_get_foo_latency();
double  microbenchmark_get_foo_rec_tp ();
```

- 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 reciprocal throughput of floating-point square root can vary depending on their inputs. Thus, you are also required to find the minimum latency and reciprocal throughput for square root and function $f(a)$.
- 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 reciprocal throughput of double-precision floating point addition and square root 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 reciprocal throughput for `addsd` of 4 and 0.5 respectively. For the square root (`sqrtsd`), the manual reports 18 and 6 cycles for latency and gap respectively and Agner reports 15-16 and 4-6 cycles respectively. The measured latency and gap in the microbenchmarks are 18 and 6 cycles for a random value and 13 and 4.5 cycles for a trivial input value. These values are consistent with the Intel's manual. Further, the numbers by Agner are within the measured range.

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

Solution: Yes, the latency of $f(a)$ is given by the sum of the latencies of its components since all results are dependent. Therefore, we have latency of $f(a) = 4(\text{mul}) + 4(\text{mul}) + 18(\text{sqrt}) = 26$. The bottleneck is the square root computation, meaning that the reciprocal throughput of $f(a)$ is equal to the reciprocal throughput of `sqrtsd`, i.e 6 cycles

(c) Assume that we now implement $f(a)$ as

$$f_2(a) = a \cdot \sqrt{a},$$

will the latency and reciprocal throughput be different for this new implementation? Justify your answer and state the expected latency and reciprocal throughput in case you think it will change.

Solution: Yes. Now the latency of $f_2(a)$ is the sum of the latencies of a mul and a sqrt. Therefore, we have latency of $f_2(a) = 4(\text{mul}) + 18(\text{sqrt}) = 22$. The reciprocal throughput will not change since the bottleneck is still the `sqrtsd`.