263-0007-00: Advanced Systems Lab
Assignment 2: 80 points
Due Date: Th, March 13th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2025/
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(d_mat* A, d_mat* B, d_mat*C, i_mat*D, d_mat* Z) {
2      d_mat_kronecker(B, A, C);
3      d_mat_transpose(C);
4      for (int i = 0; i < C->n1; i++) {
5          for (int j = 0; j < C->n2; j++) {
6              if(i % 2){
7                double t0 = mat_get(C, i, j) * cos(PI / mat_get(D, i, j));
8                mat_set(Z, i, j, t0);
9              } else {
10               double t0 = mat_get(C, i, j);
11               double t1 = sqrt((double)
12                     ((cos(PI / mat_get(D, i, j)) - sin(PI / mat_get(D, i+1, j))) *
13                     (cos(PI / mat_get(D, i, j)) - sin(PI / mat_get(D, i+1, j)))));
14               double t2 = t0 + t1;
15               mat_set(Z, i, j, t2);
16             }
17         }
18     }
19     d_mat_relu(Z);
20 }
```

   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.
- **Important:** the environment on Code Expert has limitations on execution time and memory usage. You may need to run some implementations individually depending on how optimized they are. You can do this by simply commenting the slow implementations out in `register_functions`.
- 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.
- The implementations of the `d_mat` and `i_mat` matrices are given in `mat.cpp`. `d_mat_kronecker` computes the kronecker product of two matrices, `d_mat_transpose` computes the transpose of a matrix (in-place), and `d_mat_relu` computes a (in-place) ReLU activation of a matrix.
- You can assume that the matrices are of the following dimensions:
  - A is 32×32 with floating point values in $[-1, 1]$,
  - B is 8×8 with floating point values in $[10, 20]$,
  - C is 256×256 with floating point values initialized with zeros,
  - D is 256×256 with integer values in $[3, 8]$,
  - Z is 256×256 with floating point values initialized with zeros.

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	Impl. 6
Runtime (cycles)	8353K	5158K	1500K	750K	462K	186K

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, we use lookup tables for the cosine and square root computations. Implementation 4 inlines the transpose function by changing the access to the C matrix. Implementation 5 inlines the kronecker product by adding an outer loop. This allows using scalar replacement on the values of the B matrix. Finally, implementation 6 inlines the ReLU call into the main loop. We achieved a final speedup of $45\times$.

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

Solution: Implementation 6 performs 17 flops in the body of the inner-most loop. The innermost loop is executed $32 * 8 * 8^2 = 2^{14}$ times. We can ignore the cos and $\sqrt{}$ calls as they are done once. Thus, $W = 16 * 2^{14} = 262K$ flops. The performance is therefore $\pi = \frac{278K}{186K} = 1.40$ 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 70% 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 multiplication and division operation on doubles. In addition, the latency and reciprocal throughput of the function $f(a) = \frac{1}{\sqrt{\sqrt{a*a+1}}}$. 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_mul_latency();
double microbenchmark_get_mul_rec_tp();
double microbenchmark_get_div_latency();
double microbenchmark_get_div_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 and division can vary depending on their inputs. Thus, you are also required to find the minimum latency and reciprocal throughput for division and function $f(a)$. Hint: You can try using values where performing those operations becomes trivial.
- 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, i.e the measurements should not oscillate between correct and incorrect. After the deadline, we will run each submission ten times and evaluate the score based on the run with the fewest correct measurements. For instance, if your submission initially achieves a 10/10 score, but upon rerunning it we observe a 6/10, the final score will be recorded as 6/10.

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. You can assume that the $f(a)$ function *does not* use a `rsqrt` instruction.

Discussion:

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

Solution: Yes, the manual reports a latency and reciprocal throughput for `mulsd` 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 reciprocal throughput respectively and Agner reports 13-14 and 4 cycles respectively. The measured latency and reciprocal throughput 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 reciprocal throughput information of the 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, function $f(a)$ consists of a multiplications, an addition, two square root operations, and a division. This gives a theoretical latency of 4 (mul) + 4 (add) + 2 * 18 (sqrt) + 14 (div) = 58 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 52-54 cycles of latency. For the theoretical reciprocal throughput, the square root and the division become the bottleneck because they share the same execution unit in port 0. Thus, the reciprocal throughput is 2*6 (sqrt) + 4 (div) = 16 cycles which is also consistent with the measurements.

(c) Assume that we compile with FMA enabled, i.e using the `-mfma` flag instead of `-mno-fma`. Will the latency and reciprocal throughput of $f(a)$ change? Justify your answer and state the expected latency and reciprocal throughput in case you think it will change.

Solution: Only the latency may change since we can fuse the multiplication and addition together. We would probably get 54 cycles. The reciprocal throughput will remain the same since the bottleneck is unchanged.

(d) Assume that we implement $f(a)$ as follows:

$$f_2(a) = \sqrt{\frac{1}{\sqrt{a * a + 1}}}.$$

Will the latency and reciprocal throughput of $f_2(a)$ change compared to the latency and reciprocal throughput of $f(a)$? Justify your answer and state the expected latency and reciprocal throughput in case you think it will change.

Solution: No, moving the square-root does not change neither the throughput nor the latency. The bottleneck and the number of operation do not change.