Student Name: Marios G. Angelis
Student ID: 22-953-186
Spring Semester 2023

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab

Homework 2

---

1. 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 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:

Impl. Num	Impl1	Impl2	Impl3	Impl4	Impl5 (max)
Runtime (cycles)	1247710	487958	319102	33583	28425

The table above reports runtime in cycles for 5 different implementations of the above code. These numbers were recorded on a Intel(R) Xeon(R) Silver 4210 @ 2.20GHz Cascade Lake with hyper-threading disabled, and compiled with GCC 11.2.1.

- Implementation 1 is the original code.
- In implementation 2, we removed all the calls to mat_set and mat_get functions by accessing the arrays directly. This achieves a 2.6x speedup since the code does not perform all these function calls and also does not need to execute the bound checking in each iteration.
- In implementation 3, we removed the imod operation by applying loop unrolling to the outer loop 2 times (i+=2). This achieves a 4x speedup.
- In implementation 4, we removed cos function calls by applying loop unrolling in the inner loop 3 times (j+=3) since we noticed that cos values are repeated per 3 iterations of j (-0.5,-0.5,1). Also, we removed fmax operator using ternary expressions and pre-computed values that stay constant across iterations (like sqrt). This achieves a 37.6x speedup.
- In implementation 5, after observing the equations carefully, we refactored the code trying to perform fewer operations and we applied the load/compute/store pattern. Thus, we achieved a 44× speedup compared to the original code.

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

Solution: Implementation 5 performs around $5 \cdot n1 \cdot n2$ flops. Thus, the performance is 1.81 flops/cycle for n1 = 100 and n2 = 100.

(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: Without vectorization techniques and FMA units, the theoretical peak performance for a Cascade Lake processor is 2 flops/cycle. Thus, we achieve 90.5 % of the theoretical peak performance of this processor.

2. 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:
- You can use the initialize microbenchmark data function for any kind of initialization that you may need (e.g. for initializing the input values).

```
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();
```

- 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(x). 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: Agner's table reports a latency and gap for **div** instruction running in a Cascade Lake processor of 13-14 and 4 cycles respectively. The measured latency and gap in our microbenchmark are 13.7 and 4 cycles for a random value and 13 and 4 cycles for a trivial input value (for example 0). For **max** operation, Agner's table reports 4 and 0.5 cycles for latency and gap respectively. The measured latency and gap in our microbenchmark are 4 and 0.5 cycles respectively. Both latency and gap values are consistent with the Intel's manual.

(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 $f(a) = \frac{1}{\sqrt{a^2-1}}$ consists of a multiplication, a subtraction, a square root operation and a division. The measured latency and gap in our microbenchmark are 40 and 10 cycles for $a = \sqrt{\frac{1+\sqrt{5}}{2}}$ and 34 and 8.6 cycles for a random input value a. 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. Also, for the division (DIVSD), both the manual and

Agner reports 13-14 and 4 cycles for latency and gap respectively. This gives a theoretical latency of 4 (mult) + 4 (subtraction) + 18 (sqrt) + 14 (div)= 40 cycles which is consistent with the measurements. It is important to mention that when $a = \sqrt{\frac{1+\sqrt{5}}{2}}$, the result of f(a) is never nan. Thus, using this input, we measure the maximum latency and throughput of the foo function. For the theoretical gap, the square root becomes the bottleneck. Thus, the gap is around 6 cycles which is also close to the measurements since a number of the total cycles is about the function calls for the foo function.

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

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

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: For $f_2(a)$, FMA cannot be used. The processor will execute the addition and the multiplication in parallel, then the sqrt and finally the division. This gives a theoretical latency of 4 (add and sub) + 4 (multiplication) + 18 (sqrt) + 14 (div)= 40 cycles. Also, for the theoretical gap, the square root becomes the bottleneck. Thus, the gap will be again 6 cycles. For $f_3(a)$, FMA can be used. The processor will execute the multiplication in parallel with the subtraction, then the division and finally the sqrt. This gives a theoretical latency of 4 (multiplication and subtraction) + 14 (div) + 18 (sqrt)= 36 cycles. Also, for the theoretical gap, the square root becomes the bottleneck. Thus, the gap will be again around 6 cycles.