# 263-2300-00: How To Write Fast Numerical Code

Assignment 2: 80 points
Due Date: Th, March 14th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2019/
Questions: fastcode@lists.inf.ethz.ch

**Submission instructions (read carefully)**:

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=10968.
- (Late policy)
  **You have 3 late days, but can use at most 2 on one homework**, meaning submit latest 48 hours after the due time. For example, submitting 1 hour late costs 1 late day. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's, Tyler's or Gagandeep's office.
- (Plots)
  For plots/benchmarks, **provide (concise) necessary information for the experimental setup (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions**. Follow (at least to a reasonable extent) the small guide to making plots from the lecture.
- (Code)
  When compiling the final code, ensure that you use optimization flags (e.g. for GCC use the flag "-O3"). **Disable SSE/AVX for this exercise when compiling**. Under Visual Studio you will find it under Config / Code Generator / Enable Enhanced Instructions (should be off). With `gcc` you can use the flag `-fno-tree-vectorize`.
- (Neatness)
  5% of the points in a homework are given for neatness.

**Exercises**:

1. *Short project info (10 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):

   **Solution:** This will be different for each student.

2. *Optimization Blockers (20 pts)* Code needed

In this exercise, we consider the following short computation:

```c
void slowperformance1(double *w, double *x, double *y, double *z, int n) {
  for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
      z[i] += x[k] * y[k] / sqrt(w[i]);
    }
  }
}
```

This is part of the supplied code.

**Solution:** A zip file containing a sample solution is found here.

| | Impl. 1 | Impl. 2 | Impl. 3 | Impl. 4 | Impl. 5 | Impl. 6 |
|---|---|---|---|---|---|---|
| -O0 | 0.221 F/C | 0.259 F/C | 0.242 F/C | 0.322 F/C | 0.539 F/C | 0.589 F/C |
| -O3 | 0.190 F/C | 0.190 F/C | 0.247 F/C | 0.744 F/C | 1.966 F/C | 1.971 F/C |

The table above reports flops/cycle for six different implementations of the above code, with optimizations turned off and turned on. These numbers were recorded on a Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz Skylake with hyperthreading disabled, and compiled with gcc 7.3.2.

Implementation 1 is the original code. Implementation 2 has scalar replement, which, by itself, does not improve performance significantly. Implementation 3 has scalar replement and loop unrolling with 4 iterations unrolled. When these optimizations are combined, the latency of the floating point instructions can be overcome, and hence we see an improvement with optimizations turned on. Implementation 4 has the `sqrt` moved outside of the inner loop. Implementation 5 has all prior optimizations. Implementation 6 is the same as 5 but with 8 interations unrolled instead of 4. Skylake has a FMA latency of 4 cycles and FMA throughput of 2, therefore it requires 8 accumulators to not be latency bound. Implementation 6 is accordingly a little bit faster than Implementation 5. Ultimately, we achieve a speedup of factor 10.37x.

A curiosity is that Implementations 1 and 2 are faster with optimizations turned off! One possible explaination for this is that with `-O0`, the code calls the `sqrt` function, and with `-O3`, the code branches, calling `vsqrtsd` if the value is positive and calling `sqrt` if it is negative.

It is possible to linearize the code in the following way:

```c
void linear_implementation(double *w, double *x, double *y, double *z, int n) {
  double x_dot_y = 0.0;
  for (int k = 0; k < n; k++) {
    x_dot_y += x[k] * y[k];
  }

  for (int i = 0; i < n; i++) {
      z[i] += x_dot_y / sqrt(w[i]);
  }
}
```

This linear implementation has a smaller runtime but fewer ops, (and we count it as one of the three possible implementation we asked for in (c)).

3. *Microbenchmarks(45 pts)* Code needed

Write a program (without vector instructions, i.e., standard C, and compiled with autovectorization disabled) that benchmarks the latency and throughput of a floating point multiplication and division instructions on doubles. Use the skeleton available provided and:

- Implement the functions provided in the skeleton:

```
void           microbenchmark_mode (microbenchmark_mode_t mode);
double         microbenchmark_get_mul_latency    ();
throughput_t   microbenchmark_get_mul_throughput ();
double         microbenchmark_get_div_latency    ();
throughput_t   microbenchmark_get_div_throughput ();
double         microbenchmark_get_sqrt_latency    ();
throughput_t   microbenchmark_get_sqrt_throughput ();
```

**Solution:** The sample solution can be found here.

Results for latency in cycles and throughput in operations per cycle can be found in the table below. These results were obtained with an Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz on Ubuntu 18.04.2 LTS and compiled using gcc 7.3.0 with optimization flags `-O3` and `-fno-tree-vectorize`. Hyper-threading was disabled for these measurements. Sqrt A measures sqrt given arbitrary data as inputs, Sqrt B measures sqrt given the number 1, and Sqrt C measures sqrt for powers of 2.

| | Mul | Div | Sqrt A | Sqrt B | Sqrt C |
|---|---|---|---|---|---|
| Latency (cycles) | 4.01 | 14.035 | 18.667 | 13.500 | 18.667 |
| Throughput (instructions per cycle) | 1.99 | 0.249 | 0.055 | 0.077 | 0.077 |

Discussion:

(a) Can you reach the theoretical latency / throughput of the instructions? Yes for mul and div. On Skylake, multiplication and division have a latency of 4 and 14 cycles, respectively, and a throughput of 2 and .25 instructions per cycle. For sqrt, we do not reach it. On Skylake, sqrt has a latency of 15 to 16 cycles and a throughput of 4 to 6 instructions per cycle.

(b) Does the sqrt instruction have the same latency and throughput for every input? Above, we report measurements for sqrt for arbitrary data, 1.0, and power of 2 as inputs, and show that the three classes differ from each other on at least one measurement.

(c) Do the throughput and latency of the sqrt instruction match what is in the manual? They do not. With `-O3` turned on, the compiler first checks to see if the input is negative, and if so it follows a different code path, and this affects the performance of the operation.

Note that you might have to run the same instruction many times in order to get precise benchmark results. The code skeleton uses CMake to compile and uses `RDTSC` to measure performance. To compile it, you can use the following:

```
cd microbenchmark
mkdir build
cd build
cmake ..
cd ..
cmake --build build --config Release
```

Then run it, using:

```
./build/ubenchmark
```

The `zip` bundle contains detailed informations on the compilation steps in Linux / Windows / Mac OS X, available in `README.md`, as well as in the `doc` folder.