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

Assignment 2: 80 points
Due Date: Th, March 16th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring17/course.html
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=3122. Before submission, you must enroll in the Moodle course.
- (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, Gagandeep's or Georg's office. Late homeworks have to be submitted electronically by email to the fastcode mailing list.
- (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. **Disable auto-vectorization for this exercise when compiling**. Under Visual Studio you will find it under Config / Code Generator / Enable Enhanced Instructions (should be off). With `gcc` their are several flags: use `-mno-abm` (check the flag), `-fno-tree-vectorize` should also do the job and for `icc`, the flag `-no-vec`.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Short project info (10 pts)* Go to the list of mile stones 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.

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

   In this exercise, we consider the following short computation:

```c
void slowperformance1(double *x, double *y, double *z, int n) {
 double sum = 0;
  for (int i = 0; i < n; i = i + 1)
   for (int k = 0; k < 4; k++)
    sum += x[i] * y[(i + k) % 5];
  z[0] = sum; //we to this to avoid deadcode elimination
}
```

   This is part of the suplied code.

   (a) read and understand the supplied code. It enables you to register functions with the same signature, which will be timed in a microbenchmark fashion.

   (b) Create new functions where you perform some loop unrolling and scalar replacement as discussed in the lecture to increase the performance. Explore at least three possible choices in this space, as different as possible.

   (c) For every optimization you perform, create a new function in `comp.c` 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, determine the performance.

   (d) When done, rerun all code versions also with optimization flags turned off ($-O0$ in the Makefile).

   (e) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of *slowperformance1*. Briefly discuss the table.

   (f) Submit your `comp.cpp` to Moodle.

   What speedup do you achieve?

3. *Microbenchmarks(45 pts)* Code needed

   Write a program (without vector instruction, i.e, standard C, and compiled without autovectorization) 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:

```c
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 ();
```

   - Use `microbenchmark_mode` function bellow for initialization and test modes.
   - Report your CPU, operating system and compiler.
   - Report any optimization flags that you used.
   - Report the results in cycles: latency and throughput, and calculate the gap.
   - Generate `microbenchmark.s` - the assembly version of the benchmark.
   - Submit only `microbenchmark.c` and `microbenchmark.s`.

   Discussion:

   (a) Can you reach the theoretical latency / throughput of the instructions?

   (b) Does the multiplication instruction exhibit consistent latency / throughput on any input? Do you observe any changes if one operand is set to `0x0000000000000001`? Demonstrate this behaviour in the `MUL2_LATENCY` / `MUL2_THROUGHPUT` mode.

(c) Does the division instruction exhibit consistent latency / throughput on any input? Do you observe any changes if the divisor is set to 2? Demonstrate this behaviour in the `DIV2_LATENCY` / `DIV2_THROUGHPUT` mode.

(d) Do you observe any changes when flags like `-ffast-math -funsafe-math-optimizations` are enabled. If so, why?

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 Intel Performance Counter Monitor to measure performance on Intel based CPUs. If an Intel CPU is not available, it is possible to compile the code in "failback" mode that uses `RDTSC` only. In a nutshell, the compilation goes as follows:

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

Then run it, using:

```
./bin/IntelPCM
```

For the failback mode:

```
cd microbenchmark
mkdir build
cd build
cmake -DRDTSC_FAILBACK=1 ..
cd ..
cmake --build build --config Release
```

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.