263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Fr, March 11th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring16/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=2125. 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 or Gagandeep'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 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 their are several flags: use -mno-abm (check the flag), -fno-tree-vectorize should also do the job.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. (15 pts) Get to know your machine
   Determine and create a table for the following microarchitectural parameters of your computer.

   (a) Processor manufacturer, name, and number

   (b) Number of CPU cores

   (c) CPU-core frequency

   (d) Tick or tock model?

   For one core and without considering SSE/AVX:

   (d) Cycles/issue for floating point additions

   (e) Cycles/issue for floating point multiplications

   (f) Maximum theoretical floating point peak performance in both flop/cycle and Gflop/s.

   Tips: On Unix/Linux systems, typing 'cat /proc/cpuinfo' in a shell will give you enough information about your CPU for you to be able to find an appropriate manual for it on the manufacturer's website (typically AMD or Intel). The manufacturer's website will contain information about the on-chip details. (e.g. Intel). For Windows 7 "Control Panel/System and Security/System" will show you your CPU, for more info "CPU-Z" will give a very detailed report on the machine configuration. For Mac OS X there is "MacCPUID."

2. (15 pts) CPU Identification using `cpuid`

   In this exercise we would like to implement a simplified version of CPU-Z and `/proc/cpuinfo`. The `cpuid` instruction can be used to query the CPU and provide information for CPU identification. This instruction is invoked by previously specifying the value of the `eax` register (and sometimes the `ecx` register). The output of the function is then stored in one or several of the `eax`, `ebx`, `ecx` and `edx` registers. Use the skeleton available here and provide the implementation for the following functions:

   (a) CPU vendor ID, model & family

   (b) CPU brand name

   (c) CPU feature flags (SSE, SSE2, SSE3, SSSE3, SSE41, AVX, AVX2)

   For this task is sufficient to support Intel processors only. If you do not own an Intel based CPU, use the public student labs of D-INFK, or the login servers (`optimus6.inf.ethz.ch` or `optimus7.inf.ethz.ch`). Note that Intel provides an extensive manual for CPU identification. Submit only `detect.c`

3. (10 pts) Cost analysis

   For a given number n, such that $n > 1$ consider the following algorithm:

```
1 double calc (int n)  {
2     int a[n+1]; double d = 0;
3     for (int i = 1; i <= n; i += 1) a[i] = 0;
4     for (int i = 1; i <= n; i += 1)
5         for (int j = i; j <= n; j += i) a[j] = !a[j];
6     for (int i = 1; i <= n; i += 1)
7         if (a[i]) d += 1.0 / (double) i;
8     return d;
9 }
```

   (a) Define a (floating point) cost measure $C(n)$ assuming that different floating point operations have different costs.

   (b) Compute the cost $C(n)$ of the function `calc`.

4. (15 pts) Matrix multiplication

   In this exercise, we provide a C source file for multplying two matrices of size n and a C header file that times matrix multiplication using different methods under Windows and Linux (for x86 compatibles).

   (a) Inspect and understand the code.

   (b) Determine the exact number of (floating point) additions and multiplications performed by the compute() function in mmm.c.

   (c) Using your computer, compile and run the code (Remember to turn off vectorization as explained on page 1!). Ensure you get consistent timings between timers and for at least two consecutive executions.

   (d) Then, for all square matrices of sizes n between 100 and 1500, in increments of 100, create a plot for the following quantities (one plot per quantity, so 3 plots total). n is on the x-axis and on the y-axis is, respectively,

      i. Runtime (in cycles).

      ii. Performance (in flops/cycle).

      iii. Using the data from exercise 1, percentage of the peak performance (without vector instructions) reached.

   (e) Briefly discuss your plots.

5. (20 pts) Performance Analysis

   Assume that the elements of vectors u, v, x, y and z of length n are combined as follows:

$$z_i = z_i + u_i \cdot v_i + x_i \cdot y_i \ .$$

(a) Write a C/C++ compute() function that performs the computation described above on arrays of doubles. Save the file as combine.c(pp).

(b) Within the same file create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 4.

(c) Then, for all two-power sizes $n = 2^4, \ldots, 2^{22}$ create a performance plot with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Randomly initialize all arrays. For all n repeat your measurements 30 times reporting the median in your plot.

(d) Briefly explain eventual performance variations in your plot.

(e) Assume that the vectors are now combined as follows:

$$z_i = z_i + u_i \cdot v_i + x_i \cdot y_i + u_i \cdot y_i + x_i \cdot v_i + u_i \cdot x_i \ .$$

Create another performance plot following the instructions in 5c. Briefly explain the performance behaviours and any differences in performance compared to the previous function.

6. (20 pts) Bounds

Consider the below modified version of the Horner's algorithm for evaluating polynomials. The function operates on two matrices and stores the results in an output array:

```
1  // assume cA, cB and res are properly initialized
2  void horner (float x, float y, int n, float * cA [], float * cB [], float * res)  {
3      for (int i = 0; i < n; i += 1) {
4          float d1 = 0.0, d2 = 0.0; d3 = 0.0;
5          for (int j = n - 1; j >= 0; j -= 1) {
6              d1 = d1 * x + cA[i][j];
7              d2 = d2 * y - cB[j][i];
8              d3 += d1 / d2;
9          }
10         res[i] = d3;
11     }
12 }
```

We consider a Core i7 CPU based on a Haswell processor. It offers (as part of AVX2) a so-called fused multiply-add instruction, called FMA, that computes `y = a * x + b` on floating point numbers. Further it offers an instruction (called RCP) that approximates $1/x$ using Newton-Raphson step. The FMA on Haswell is as fast as multiplication, meaning the throughput is two per cycle; the RCP one per cycle.

More information on these instruction and the load bandwidth od the caches in the memory hierarchy can be found in Intel 64 and IA-32 Architectures Optimization Reference Manual. Assume that the load bandwidth from L3 cache is half the one of L2 cache and that the load bandwidth from RAM is half that of the L3 cache.

(a) Determine the exact cost measured in the abovementioned two operations.

(b) Determine an asymptotic upper bound on the operational intensity (assuming empty caches and considering both reads and writes).

(c) Consider only one core and determine hard lower bounds (not asymptotic) on the runtime (measured in cycles):

i. The op count. Assume that the code is compiled using `gcc` with the following flags: `-ffast-math -fno-tree-vectorize -mfma -march=core-avx2 -mrecip=all -O3` and that FMAs are used as much as possible.

ii. Loads, for each of the following cases: All floating point data is L1-resident, L2-resident, L3-resident, and RAM-resident. Consider best case scenario (peak bandwidth).

(d) (enthusiast; no points) Harden the ops-based lower bound by also taking into account the latency of the operations used and their dependencies in the code.