

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

Assignment 1: 100 points

Due Date: Th, March 5th, 17:00

<http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/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=1317>. Before submission, you must enroll in the Moodle course. Enrollment key is “263-2300”.
- (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. 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 Daniele’s office. Late homeworks have to be submitted electronically.
- (Plots)
For plots/benchmarks, be concise, but provide necessary information (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 (soon in lecture).
- (Code)
When compiling the final code, ensure that you use optimization flags. Disable SSE for this exercise when compiling. Under Visual Studio you will find it under Config / Code Generator / Enable Enhanced Instructions (should be off). With gcc there 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 tok 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.

2. (25 pts) Cost analysis

Consider the following code from the domain of numerical program analysis (it computes the strong closure of a set of octagonal inequalities over N variables).

```
void strong_closure(double A[N][N]){
    int i,j,k;
    for(k = 0; k < N/2; k++){
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                A[i][j] = min(A[i][j], A[i][2k] + A[2k][j]);
                A[i][j] = min(A[i][j], A[i][2k+1] + A[2k+1][j]);
                A[i][j] = min(A[i][j], A[i][2k] + A[2k][2k+1] + A[2k+1][j]);
                A[i][j] = min(A[i][j], A[i][2k+1] + A[2k+1][2k] + A[2k][j]);
            }
        }
        for(i = 0; i < N; i++){
            for(j = 0; j < N; j++){
                A[i][j] = min(A[i][j], (A[i][i^1] + A[j^1][j])/2);
            }
        }
    }
}
```

- Define a suitable cost measure $C(N)$ assuming that different floating point operations have different costs.
- Compute the cost $C(N)$ of the function `strong_closure`.
- How would you change the definition and value of $C(N)$ assuming that all operations have the same cost?

Note: `^` stands for bitwise xor. Integer operations are ignored. Lower-order terms (and only those) may be expressed using big-O notation (this means: a result like $3n + O(\log(n))$ is ok but $O(n)$ is not).

3. (15 pts) Floyd Warshall

The standard Floyd Warshall kernel computes the shortest paths between all pairs n vertices. We provide a C source [file](#) and a C header [file](#) that times this kernel using different methods under Windows and Linux (for x86 compatibles).

- Inspect and understand the code.
- Determine the exact number of (floating point) additions and min operations performed by the `compute()` function in `fw.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.
- Then, for all square matrices of sizes n between 100 and 2000, 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,
 - Runtime (in cycles).
 - Performance (in flops/cycle).
 - Using the data from exercise 1, percentage of the peak performance (without vector instructions) reached.
- Briefly describe your plots, and submit your modified code to the Moodle and call it also `fw.c`.

4. (20 pts) Convex Combination

Consider the computation of the convex combination of the elements of two arrays x and y of length n :

$$z_i = w_i \cdot x_i + (1 - w_i) \cdot y_i . \quad (1)$$

The arrays z and w are also of length n and contain respectively the result and the weights used.

- (a) Write a C/C++ `compute()` function that performs the computation described above on arrays of doubles. Save the file as `convex.c(pp)`.
- (b) Within the same file create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 3.
- (c) Then, for all sizes n between 2 and 5M (with increment of n) create a performance plot with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Use random initialization for arrays x , y , and w . For all n repeat your measurements at least 15 times reporting the median in your plot.
- (d) Briefly motivate eventual variations in performance in your plot.

5. (20 pts) Bounds

We consider a slightly modified implementation of a so-called stencil computation using a nine-point stencil h over an $N \times N$ grid G as shown below.

```

for (i = 1; i < N-1; i++) {
  for (j = 1; j < N-1; j++) {
    G[i][j] =
      h[0][0]*G[i-1][j-1]
      + h[1][0]*G[i][j-1]   + h[1][1]*G[i][j]
      + h[2][0]*G[i+1][j-1] + h[2][1]*G[i+1][j] + h[2][2]*G[i+1][j+1];
  }
}

```

- (a) Determine the exact cost measured in flops (floating point operations).
- (b) Determine an asymptotic upper bound on the operational intensity (assuming empty caches and considering both reads and writes).
- (c) On a Core i7 Sandy Bridge, consider only one core and determine hard lower bounds (not asymptotic) on the runtime (measured in cycles) based on
 - i. The op count (floating point ops only, no vectorization).
 - ii. Loads, for each of the following cases: All floating point data is L1-resident, L2-resident, RAM-resident.

6. (0 pts: for the enthusiast) Intel PCM v.s. Linux `perf`

Another way to perform cost analysis of a given algorithm is by using CPU performance counters. Accessing these counters requires OS support and is different on each CPU type. `perf` is a convenient Linux tool that abstracts away CPU hardware differences and is part of the Linux Kernel 2.6+. If your machine is not running Linux, but is powered by an Intel-based CPU, Intel provides a cross-platform performance monitoring tool called [Intel Performance Counter Monitor](#) (Intel PCM). Choose **one of the two**, and do the following:

- (a) Determine the performance monitoring events available on your CPU that monitor double precision floating point operations. Provide a list of mnemonic names of the events, usually available in the CPU vendor architecture manual (For example Sandy Bridge: `FP_COMP_OPS_EXE:X87`, `FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE` and `FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE`).
- (b) For Linux `perf`:
 - i. Convert the obtained events into raw PMU events, in the form of `rNNN` where `NNN` is a hexadecimal event descriptor. Use `libpfm`'s tool `examples/check_events` to perform the conversion. Extended `perf` tutorial is available [here](#).
 - ii. Determine the number of flops and cycles of the MMM program by feeding the raw PMU events to `perf`.
 - iii. Determine the performance of the MMM loop by invoking `perf` tool inside your MMM program. To achieve this level of granularity, `perf` must be invoked as a system call, with proper instantiation of the PMU events. Modify the skeleton available [here](#) to obtain precise results.

(c) For Intel PCM:

- i. Follow the OS specific instructions to setup the Intel PCM tool on your machine. Verify your installation by running `./pcm.x 1`.
- ii. Convert the obtained events into raw PMU events, by determining their event number and unmask value, available in the Intel [manual](#).
- iii. Determine the performance of the MMM loop inside your MMM program. Modify the skeleton available [here](#) to obtain precise results.

Note that Intel PCM tool is not stable on Mac OS X 10.10 (Yosemite). Although the MSR driver could be compiled and loaded, a simple invocation of the `pcm.x` tool will result in a kernel panic, which will force your Mac OS X to restart.