263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Th, March 6th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring14/course.html
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  We set up a SVN Directory for everybody in the course. The Url of your SVN Directory is https://svn.inf.ethz.ch/svn/pueschel/students/trunk/s14-fastcode/YOUR.NETZH.LOGIN/ You should see subdirectory for each homework.
- (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. Late submissions have to be submitted completely electronically and emailed to fastcode@lists.inf.ethz.ch.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert them to PDF and submit to svn in the top level of the respective homework directory. Call it homework.pdf. Handwritten parts can be scanned and included or brought (in time) to Alen's or Daniele's office.
- (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 (lecture 5).
- (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 their are several flags: use -mno-abm (check the flag), -fno-tree-vectorize should also do the job. Submit all the code you write into the according folders in your SVN directory.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. (25 pts) Cost analysis

   Consider the following code that computes the Cholesky decomposition of a given $N \times N$ symmetric positive definite matrix A.

```
void chol(float A[N][N]) {
  int i,j,k;
  double c;

  for (j = 0; j < N-1; j++) {

    A[j][j] = sqrt(A[j][j]);

    for (i = j+1; i < N; i++)
      A[i][j] = A[i][j]/A[j][j];

    for (k = j+1; k < N; k++)
      for (i = k; i < N; i++)
        A[i][k] = A[i][k] - A[i][j]*A[k][j];
  }
}
```

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

   (b) Compute the cost $C(N)$ of the function `chol`.

(c) How would you change the definition and value of $C(N)$ assuming that all operations have the same cost?

Note: 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).

2. (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

   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.

3. (15 pts) MMM
The standard matrix multiplication kernel performs the following operation : $C = AB + C$, where A, B, C are matrices of compatible size. 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).

   (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 of the code.

   (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 2, percentage of the peak performance (without vector instructions) reached.

   (e) Briefly describe your plots, and submit your modified code to the SVN and call it also mmm.c.

4. (20 pts) MVM
We consider matrix-vector multiplication of the form $y = A * x + y$, where A is an $n \times n$ square matrix and y and x are vectors of length n.

   (a) Create a new file mvm.c. The code should contain a compute() function that implements matrix-vector multiplication in the form given above using a double loop, and an rdtsc() function for timing it (you can use the one in mmm.c from exercise 3).

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

(c) 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 2, percentage of the peak performance (without vector instructions) reached.

(d) Compare the performance obtained with MVM and MMM and briefly discuss.

5. (20 pts) Bounds
We consider a double loop 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[0][1]*G[i-1][j] + h[0][2]*G[i-1][j+1]
      + h[1][0]*G[i][j-1]   + h[1][1]*G[i][j]   + h[1][2]*G[i][j+1]
      + 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) MMM `perf`
Another way to perform cost analysis of a given algorithm is by using CPU performance counters. Hardware counters are a set of special-purpose registers that store counts of hardware-related events within a CPU. Accessing these registers requires OS support and is different on each CPU type, as each CPU model might monitor different set of events. `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, use the student lab machines `stud{1..27}-h{56,57}.inf.ethz.ch` to complete this exercise.

(a) Determine the number of cycles of the MMM program in exercise 3, invoking `perf` command line tool to gather performance statistics (extended `perf` tutorial is available here).

(b) 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`).

(c) 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.

(d) Determine the number of flops of the MMM program by feeding the raw PMU events to `perf`.

(e) 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.