263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Th, March 7th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring13/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/s13-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. 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 Georg/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 Matlab function. The input vectors x_1 and x_2 are both of length $m = 2^k$, while the input vector y is of length n (not necessarily a two-power). The output z is a scalar value.

```
function z = func(x1, x2, y)

m = length(x1);
n = length(y);

% base case
if m == 1
    z = x1(1)+sum(y); % sum(y) adds together all the vector's elements
    return
end

k = m/2;

t  = func(x1(1:k), x2(1:k), y) * func(x1(k+1:m), x2(k+1:m), y);

x3 = x1+x2; % addition of vectors
y1 = pi*y; % multiplication of a vector by a scalar

z = t*func(x3(1:k), x3(k+1:m), y1);

end
```

   We assume the floating point cost measure $C(m, n) = (A(m, n), M(m, n))$, where $A(m, n)$ is the number of additions and $M(m, n)$ the number of multiplications. Compute $C(m, n)$. Show your work.

   Note: Only consider floating point ops.

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 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!) . Ensure you get consistent timings by at least 2 different timers and for at least 2 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 reached.

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

4. (20 pts) Daxpy
We consider vector addition of the form $y = \alpha x + y$ (often called daxpy) , where $\alpha = 1.1$ is a scalar constant value, and y and x are vectors of doubles of length n.

   (a) Create a new file daxpy.c. The code should contain a compute() function that implements vector addition of the form given above, and a rdtsc() function for timing it (you can use the one in mmm.c from exercise 3).

   (b) Create a plot for the same quantities requested in 3d. The vector length n should vary between 1000 and 2000000 (2 millions) incrementing of 1000.

   (c) Briefly describe your plots, and submit the file daxpy.c to the SVN.

5. (20 pts) Bounds
We consider convolution of two vectors: $y = h * x$, where y, x are vectors of length n, and h is a vector of length $k < n/2$. A straightforward implementation may use this double loop:

```
for (i = k-1; i < n-k; i++)
  for (j = 0; j < k; j++)
     y[i] +=  h[j] * y[i-j];
```

(a) Determine the exact cost measured in flops (floating point operations).

(b) Determine an asymptotic upper bound on the operational intensity (assuming empty caches).

(c) Now assume $k = 4$. On a Core i7 Sandy Bridge, consider only one core and detemine 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 (do not consider writes, just reads). All floating point data is L1-resident, L2-resident, RAM-resident.