263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Thu March 8 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring12/course.html
Questions: fastcode@lists.inf.ethz.ch

1. (25pts) Cost analysis

   Consider the following Matlab function. Input and output are both vectors of length n, $n = 2^k$ is assumed to be a two-power.

```
1  function y = func(x)
2
3  n = length(x);
4  y = zeros(n,1); % allocate the result y which is a column vector
5
6  % base case
7  if n == 1
8      y(1) = x(1); return
9  end
10
11 m = n/2;
12
13 % recurse
14 t1 = 2 * func(x(1:m));
15 t2 = func(x(m+1:n));
16
17 for i = 1:m
18   y(2*i-1) = t1(i) + t2(i);
19   y(2*i) = t1(i) - t2(i);
20 end
21
22 y(1) = y(1) + 1;
23 return
```

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

   Note: Only consider floating point ops.

   Solution: In order to build the two recurrence relations for $A(n)$ and $M(n)$, we first need to compute:

   - Number of recursive calls = 2 (lines 14 and 15).
   - Number of additions during a conquer step = $n + 1$ (lines 18, 19, and 22).
   - Number of multiplications during a conquer step = $\frac{n}{2}$ (line 14).
   - $A(1) = M(1) = 0$ (line 8).

   Computation of $A(n)$:

$$\begin{cases} A(n) = 2A\left(\frac{n}{2}\right) + n + 1 \\ A(1) = 0 \end{cases} \xrightarrow{f_k = A(2^k)} \begin{cases} f_k = 2f_{k-1} + 2^k + 1 \\ f_0 = 0 \end{cases}. \tag{1}$$

   Using the formula $f_k = af_{k-1} + s_k = f_0 a^k + \sum_{i=0}^{k-1} a^i s_{k-i}$, $k \geq 1$ we obtain

$$f_k = 2f_{k-1} + 2^k + 1 = \sum_{i=0}^{k-1} 2^i \left(2^{k-i} + 1\right) \tag{2}$$

$$= \sum_{i=0}^{k-1} 2^k + \sum_{i=0}^{k-1} 2^i = k2^k + 2^k - 1, \tag{3}$$

which leads to $A(n) = f_{\log n} = n \log n + n - 1$.

Similarly, we compute $M(n)$:

$$\begin{cases} M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2} \\ M(1) = 0 \end{cases} \xrightarrow{f_k = M(2^k)} \begin{cases} f_k = 2f_{k-1} + 2^{k-1} \\ f_0 = 0 \end{cases}. \tag{4}$$

Again, using the formula $f_k = af_{k-1} + s_k = f_0 a^k + \sum_{i=0}^{k-1} a^i s_{k-i}$, $k \geq 1$ we obtain

$$f_k = 2f_{k-1} + 2^{k-1} = \sum_{i=0}^{k-1} 2^i \left(2^{k-1-i}\right) = k2^{k-1}, \tag{5}$$

which leads to $M(n) = f_{\log n} = \frac{n}{2} \log n$.

2. (10pts) 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.

   Solution:
   Basically everyone managed to solve this question without problems, therefore just some general remarks:

   - If you are using a mobile processor you most likely will have frequency scaling enabled. This can be switched off in the operating system and/or via the BIOS.
   - On newer systems (i7) the Turboboost feature starts to seriously affect the measured performance (It can happen that you measure way higher performance that you should). The easiest way of avoiding this is to simply disable it via BIOS.

3. (30pts) 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. You need to setup the timer for your machine by setting the number of runs NUM_RUNS and the machine frequency FREQUENCY. For the number of runs per n use the formula $2 * (ceil(1000/n))^3$.

(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 seconds)
   ii. performance (in Mflop/s or Gflop/s).
   iii. Using the data from exercise 2, percentage of the peak performance reached.

(e) Submit your modified code to the SVN and call it also mmm.c

Solution: Also this worked for everyone. You saw that the runtime increases with size and that the triple loop MMM yields terrible performance. Within the next weeks you will learn how to improve on this.

4. (30pts) Bounds
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 straightforward implementation uses this double loop:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
     y[i] = y[i] + A[i][j]*x[j];
```

On a Core 2, consider only one core and detemine hard lower bounds on the runtime (measured in cycles) based on

(a) The op count (floating point ops only, no vectorization)

(b) Loads, for each of the cases L1-resident, L2-resident, RAM-resident (again only floating point data)

Finally, assume the Core 2 runs at 3 GHz, and

(c) Compute for each of the four lower bounds on the runtime an upper bound for the performance (in Gflop/s).

Solution:

(a) The op count is one add and one mul per inner loop yielding a total of n^2 muls and n^2 adds. On the Core 2 we can do one add and one mul in parallel per cycle. Therefore the computational bound is n^2.

(b) The number of loads required for this double loop is:
- n^2 for $A[i][j]$ cause every access depends on both i and j.
- n for $x[j]$, assuming it fits into registers (n^2 if it does not fit - but since we asked for the lower bound we take only n).
- n for $y[i]$ cause it is only depended on the outer loop variable i and stays the same within the inner loop

This yields a total of $n^2 + 2n$ loads.

Cache	Throughput	Bound
L1	$2\frac{doubles}{cycle}$	$\frac{n^2+2n}{2}$ cycles
L2	$1\frac{double}{cycle}$	$(n^2 + 2n)$ cycles
RAM	$\frac{1}{4}\frac{double}{cycle}$	$(n^2 + 2n) * 4$ cycles

(c) In the following table we present the upper performance bounds derived from the runtime bounds given in the previous section. Please note that the real bounds will be lower, because we are ignoring the stores and assume we can fit $x[j]$ into registers.

Location	Bound	Performance
CPU	n^2 cycles	$\frac{2n^2 flops}{n^2 cycles} * 3$ GHz $= 6$ GFlop/s
L1	$\frac{n^2+2n}{2}$ cycles	$\frac{2n^2 flops}{\frac{n^2+2n}{2} cycles} * 3$ GHz $= \frac{12n}{n+2}$ GFlop/s
L2	$(n^2 + 2n)$ cycles	$\frac{2n^2 flops}{(n^2+2n) cycles} * 3$ GHz $= \frac{6n}{n+2}$ GFlop/s
RAM	$4(n^2 + 2n)$ cycles	$\frac{2n^2 flops}{4(n^2+2n) cycles} * 3$ GHz $= \frac{3n}{2(n+2)}$ GFlop/s