263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Th, March 9th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring17/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=3122. 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, Georg'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. (20 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 logical and physical cores.
   (c) CPU-core frequency.
   (d) CPU maximum frequency. Does your CPU support Intel Turbo Boost Technology?
   (e) Tick or tock model?

   For one core and **without** considering SSE/AVX:

   (d) Latency/Throughput/Gap for floating point additions.
   (e) Latency/Throughput/Gap for floating point multiplications.
   (f) Latency/Throughput/Gap for the `rcp` instruction (if supported).
   (g) Latency/Throughput/Gap for fused multiply - add (FMA) operations (if supported).
   (h) Maximum theoretical floating point peak performance in both flop/cycle and Gflop/s.

Notes:

- Intel calls throughput what is in reality the gap = 1/throughput.
- The manufacturer's website will contain information about the on-chip details. (e.g. Intel 64 and IA-32 Architectures Optimization Reference Manual).
- 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).
- For Windows 7/10 "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".
- Throughout this course, we will consider the FMA instruction as two floating point operations.

2. (10 pts) Cost analysis

   Consider the following algorithm for calculating the sum of first $2n$ terms in the $cot^{-1}(\frac{1}{x})$ series (we assume $n > 1$):

```
1  double calc (int n, double x)  {
2      double num1 = x, num2 = num1*x*x;
3      int den1 = 1, den2 = 3;
4      double sum = num1/(double)den1 - num2/(double)den2;
5      while (den1 < 4*n - 4){
6           num1 = num1 * x * x * x * x;
7           num2 = num1 * x * x;
8           den1 = den1 + 4;
9           den2 = den2 + 4;
10          sum = sum + num1/(double)den1 - num2/(double)den2;
11     }
12     return sum;
13 }
```

   (a) Define a suitable detailed floating point cost measure $C(n)$.

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

   Solution:

   (a) The function `calc` performs floating point multiplications, divisions, additions (subtraction is same as addition) and casts. Therefore,

$$C(n) = C_{add} \cdot N_{add} + C_{mult} \cdot N_{mult} + C_{div} \cdot N_{div} + C_{cast} \cdot N_{cast}.$$

   (b) Before entering the loop, the function performs 1 subtraction, 2 multiplications, 2 divisions and 2 casts. The loop iterates $n - 1$ times, in each iteration it performs, 1 addition, 1 subtraction, 6 multiplications, 2 divisions and 2 casts. Thus,

$$\begin{aligned} N_{add} &= 2n - 1, \\ N_{mul} &= 6n - 4, \\ N_{div} &= 2n, \\ N_{cast} &= 2n, \\ C(n) &= C_{add} \cdot (2n - 1) + C_{mul} \cdot (6n - 4) + C_{div} \cdot 2n + C_{cast} \cdot 2n. \end{aligned}$$

3. (25 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.

Solution:

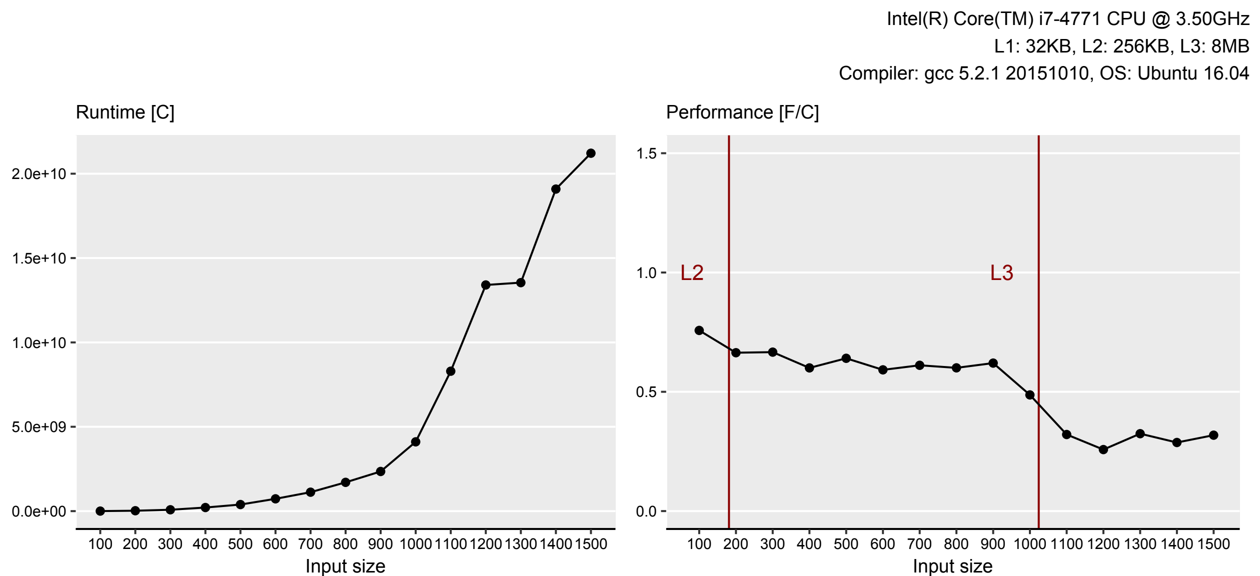

Figure 1: Plots resulting from execution of `mmm.c` on a Haswell CPU (scalar peak performance: 4 f/c). The code was compiled with `gcc` 5.2.1 with `O3` enabled, no FMAs and no vectorization.

(b) The code performs $2n^3$ floating point operations.

(d) See Fig. 1 for part (i) and (ii). The Plot for (iii) is same as for (ii) but with data on y-axis scaled by factor of 25.

(e) The computation is compute bound however, due to dependency in the computation the peak performance cannot be achieved. Every iteration of i-loop loads matrix C, thus the performance drops whenever C does not fit in L2 ($n > 181$) and L3 ($n > 1024$) cache.

4. (20 pts) Performance Analysis

Consider the function *babs*:

```
1 inline double babs (double x) {
2     union {  uint64_t i; double d; } u = { .d = x };
3     u.i = u.i & 0x7FFFFFFFFFFFFFFF;
4     return u.d;
5 }
```

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

$$z_i = z_i + x_i \cdot y_i + u_i \cdot y_i + x_i \cdot z_i + babs(u_i \cdot x_i) \; .$$

(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 3.

(c) Then, for all two-power sizes $n = 2^4, \ldots, 2^{22}$ create performance plot with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Create two series such that the first has all optimization flags disabled, and the second serie has all optimizations flags enabled (except for vectorization). Randomly initialize all arrays. For all n repeat your measurements 30 times reporting the median in your plot.

(d) If you have an Intel processor, run the same tests again, such that you make sure that Intel Turbo Boost is disabled (or enable it if the previous plot was generated with Turbo Boost disabled).

(e) Briefly explain eventual performance variations in your plot and the effects of Turbo Boost.

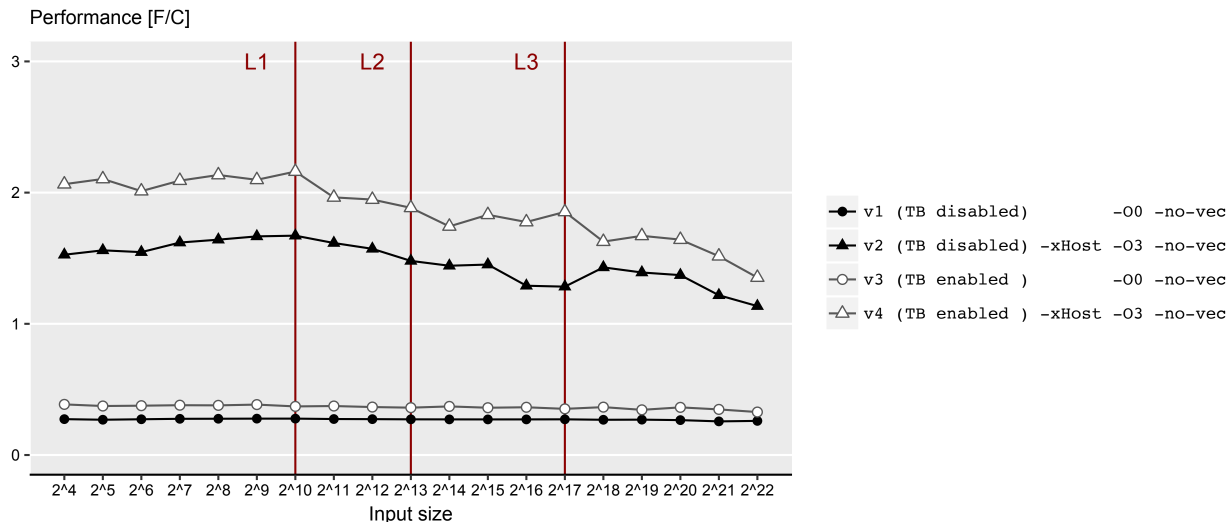

Figure 2: Plots resulting from execution of `combine.c` on an FMA-enabled Haswell CPU (scalar peak performance: 4 f/c). The table reflects the performance values obtained running `v2` series.

Solution:

An implementation of the compute function is available here.

The algorithm performs $8 \cdot n$ double precision floating point operations. The compilation of the `babs` function will result in machine code, that (potentially) contains either `andq` x86 instruction or `btrq` instruction which is executed on the integer ALU of the CPU. Haswell CPU has 2 load units (Ports 2 and 3), a separate store unit (Port 4) and 4 integer ALUs, one of which dedicated (Port 6). Therefore the algorithm is bound by the computing runtime whenever data fits in any level of cache.

We use CPU's time step counter (`RDTSC`) to measure performance. While the time step counter ticks at a constant rate within one CPU, each core of the CPU might operate on a different frequency. Turbo Boost does exactly that, boosting the frequency of a particular core. The number of CPU cycles that are needed to complete the algorithm do not change when Turbo Boost is enabled. However, since

RDTSC ticks at a constant rate, while the core frequency is boosted, it gives the perception that the algorithm is completed in less cycles, thus increasing the resulting performance. Therefore v3 and v4 do not reflect the accurate performance result.

Compiling the algorithm with all optimisations disabled, will result in a machine code that corresponds to the C code: the babs function will not be inlined, and the resulting program will not benefits from use of FMAs. This will create a large compute runtime that dominates the runtime required to move the data from and to the cache, resulting in a constant performance across all levels (v1 series).

Once all optimization flags are enabled, the code will benefit from the use of FMAs and babs will be inlined. Reducing the number of flops through the use of FMAs, as well as avoiding the function call, creates improved compute runtime that we observe in L1, L2 and L3 cache. The shorter compute runtime, increases the impact on time spent in moving data, and we observe this with the reduced performance, as soon as the data no longer fits the last level cache (series v2).

5. (20 pts) Bounds

   Consider the three artificial computations below. The functions operate on a input vector and store the results in an output array:

```
1  void artcomp1(float x[], float y[], int N) {
2   for (int i = 0; i < N; i++)
3    y[i] = x[i] * 1.3 + y[i];
4  }
5  void artcomp2(float x[], float y[], int N) {
6   int len = N/2; //assume 2 divides N
7   for (int i = 0; i < 2 * len; i += 2) {
8    y[i] = x[i] * 1.4;
9    y[i + 1] = x[i + 1] + 1.4;    }
10 }
11 void artcomp3(float x[], float y[], float z[], int N) {
12  int len = N/3; //assume 3 divides N
13  for (int i = 0; i < 3 * len; i += 3) {
14   y[i] = x[i] * z[i];
15   y[i + 1] = x[i + 1] * z[i+1];
16   y[i + 2] = x[i + 2] + 3.3;    }
17 }
```

   We consider a Core i7 CPU based on a Haswell processor. As seen in the lecture, it offers FMA instructions (as part of AVX2) that compute `y = a * x + b` on floating point numbers. Consider the information from the lecture slides on the throughput of the according operations. Assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture

   (a) Determine the exact cost (in flops) of each function.

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

   (c) Consider only one core and determine, for each function, a hard lower bound (not asymptotic) on the runtime (measured in cycles), based on:

      i. The op count. Assume that the code is compiled using gcc with the following flags: `-fno-tree-vectorize -mfma -march=core-avx2 -O3` and that FMAs are used as much as possible. Be aware that the lower bound is also affected by the available ports offered for the according computation. (see lecture slides)

      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).

   Solution 1: Note that in the exercise we, by accident, made the variable "len" a float. The solution is for the corrected version where its an Integer, a possible solution with len being a float is shown afterwards. We of course accept versions where the computations involving len are counted.

   (a) The flop cost for each function are

i. $C(N) = 2N$
ii. $C(N) = N$
iii. $C(N) = N$

(b) The operational intensity is

i. $I(N) = \frac{2N flops}{2N doubles} = \frac{2N flops}{16N bytes}$
ii. $I(N) = \frac{N flops}{N doubles} = \frac{N flops}{8N bytes}$
iii. $I(N) = \frac{3N flops}{5N doubles} = \frac{3N flops}{40N bytes}$

Therefore for all cases $I(N) \in O(1)$

(c) i. A. per iteration 2 FMAs can be performed resuling in a lower bound of $\frac{N}{2} cycles$

B. one addition and one multiplication can be performed independent. They are not combineable into an FMA. Therefore the lower bound is $\frac{N}{2} cycles$

C. the computation conflicts on the ports as only either two mults, or a mult and an add can be performed in parallel. Therefore we can perform at the best 2 adds and 4 mults in 3 cycles. This results in a lower bound of $\frac{N flops}{\frac{6 flops}{3 cycles}} = \frac{N}{2} cycles$

ii. A. $C_{\text{double loads}}(N) = 2N$

B. $C_{\text{double loads}}(N) = N$

C. $C_{\text{double loads}}(N) = \frac{5}{3}N$

http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring17/slides/arch.pdf shows peak bandwidth of L1, L1, L3 and an estimate for the RAM troughput. It follows:

A. $r_{L1} = \frac{2N}{8*2}$, $r_{L2} = \frac{2N}{8*2}$, $r_{L3} = \frac{2N}{4*2}$,$r_{RAM} = \frac{2N}{2*2}$

B. $r_{L1} = \frac{N}{8*2}$, $r_{L2} = \frac{N}{8*2}$, $r_{L3} = \frac{N}{4*2}$,$r_{RAM} = \frac{N}{2*2}$

C. $r_{L1} = \frac{\frac{5}{3}N}{8*2}$, $r_{L2} = \frac{\frac{5}{3}N}{8*2}$, $r_{L3} = \frac{\frac{5}{3}N}{4*2}$,$r_{RAM} = \frac{\frac{5}{3}N}{2*2}$

Solution 2: This solution is for the original version of the task that included the mistake where "len" was a float. "artcomp1" stays unchanged and is therefore omitted. We assume that the compiler moves the 2 * len / 3 * len computation out of the loop.

(a) The flop costs are given by:

i. $C(N) = (N+1)_{\text{arithmetic flops}} + \frac{N}{2}_{\text{comparison flops}} + (\frac{N}{2}+1)_{\text{cast flops}}$
ii. $C(N) = (N+1)_{\text{arithmetic flops}} + \frac{N}{3}_{\text{comparison flops}}(\frac{N}{3}+1)_{\text{cast flops}}$

The $+1 flops$ are from the inital multiplication. The cast are performed for len before the loop and then for the loop index i before each comparison. We assume the following cast instructions is used `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=cvtsi2ss&expand=1321`

(b) The operational intensity is

i. $I(N) = \frac{(N+1)_{\text{arithmetic flops}} + \frac{N}{2}_{\text{comparison flops}} + (\frac{N}{2}+1)_{\text{cast flops}}}{N doubles}$
ii. $I(N) = \frac{(N+1)_{\text{arithmetic flops}} + \frac{N}{3}_{\text{comparison flops}}(\frac{N}{3}+1)_{\text{cast flops}}}{5N doubles}$

Therefore for all cases $I(N) \in O(1)$

(c) i. We assume that the comparision is performed with the instruction `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=1321,721,722,723,731,720,1143&cats=Compare&text=_mm_comilt_ss%2520`.

A. one addition and one multiplication can be performed independent. They are not combineable into an FMA. In addition this version performs a comparison every loop iteration. As this falls into the SIMD log category we can issue it on port 5 whenever they appear. The cast on the other hand has to be performed on port 1 and therefore competes with the addition. The resulting lower bound is therefore N cycles for the loop in terms of throughput.

B. the computation conflicts on the ports as only either two mults, or a mult and an add can be performed in parallel. In addition the cast conflicts with the addition on the port. If we assign the mults to port 0 and the addition plus the cast to port 1 we can perform each loop iteration in 2 cycles. Once again the comparison can be issued on a different port and therefore won't impact on the bound. This results in a lower bound of $\frac{N flops}{\frac{4flops}{2cycles}} = \frac{N}{2} cycles$ for the loop.

ii. A. $C_{\text{double loads}}(N) = 2N$

B. $C_{\text{double loads}}(N) = N$

C. $C_{\text{double loads}}(N) = \frac{5}{3}N$

http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring17/slides/arch.pdf shows peak bandwidth of L1, L1, L3 and an estimate for the RAM troughput. Note that we deal with floats - therefore have the double bandwidht compared to doubles. (therefore the *2 in the divisor). It follows:

A. $r_{L1} = \frac{2N}{8*2}$, $r_{L2} = \frac{2N}{8*2}$, $r_{L3} = \frac{2N}{4*2}$,$r_{RAM} = \frac{2N}{2*2}$

B. $r_{L1} = \frac{N}{8*2}$, $r_{L2} = \frac{N}{8*2}$, $r_{L3} = \frac{N}{4*2}$,$r_{RAM} = \frac{N}{2*2}$

C. $r_{L1} = \frac{\frac{5}{3}N}{8*2}$, $r_{L2} = \frac{\frac{5}{3}N}{8*2}$, $r_{L3} = \frac{\frac{5}{3}N}{4*2}$,$r_{RAM} = \frac{\frac{5}{3}N}{2*2}$