263-2300-00: How To Write Fast Numerical Code
Assignment 1: 100 points
Due Date: Fr, March 11th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring16/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=2125. 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 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. (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 tock 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. For Mac OS X there is "MacCPUID."

2. (15 pts) CPU Identification using `cpuid`

   In this exercise we would like to implement a simplified version of CPU-Z and `/proc/cpuinfo`. The `cpuid` instruction can be used to query the CPU and provide information for CPU identification. This instruction is invoked by previously specifying the value of the `eax` register (and sometimes the `ecx` register). The output of the function is then stored in one or several of the `eax`, `ebx`, `ecx` and `edx` registers. Use the skeleton available here and provide the implementation for the following functions:

   (a) CPU vendor ID, model & family

   (b) CPU brand name

   (c) CPU feature flags (SSE, SSE2, SSE3, SSSE3, SSE41, AVX, AVX2)

   For this task is sufficient to support Intel processors only. If you do not own an Intel based CPU, use the public student labs of D-INFK, or the login servers (`optimus6.inf.ethz.ch` or `optimus7.inf.ethz.ch`). Note that Intel provides an extensive manual for CPU identification. Submit only `detect.c`

   Solution:
   One possible solution is available here.

3. (10 pts) Cost analysis

   For a given number n, such that $n > 1$ consider the following algorithm:

```
1 double calc (int n)  {
2     int a[n+1]; double d = 0;
3     for (int i = 1; i <= n; i += 1) a[i] = 0;
4     for (int i = 1; i <= n; i += 1)
5         for (int j = i; j <= n; j += i) a[j] = !a[j];
6     for (int i = 1; i <= n; i += 1)
7         if (a[i]) d += 1.0 / (double) i;
8     return d;
9 }
```

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

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

   Solution:

   (a) The algorithm given by the function `calc` has floating division and addition. Therefore:

$$C(n) = C_{fadd} \cdot N_{fadd} + C_{fdiv} \cdot N_{fdiv}$$

   (b) Observing the code, we see that lines 1-5 set 1 to all elements of `a[i]` such that $i = p^2$ where p is an integer. To determine the iteration of the loop at line 6, we need to determine the count of all numbers i such that $i = p^2$ and $i <= n$, where p is an integer. Therefore:

$$N_{fadd} = N_{fdiv} = \lfloor \sqrt[2]{n} \rfloor$$
$$C(n) = (C_{fadd} + C_{fdiv}) \cdot \lfloor \sqrt[2]{n} \rfloor$$

4. (15 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:

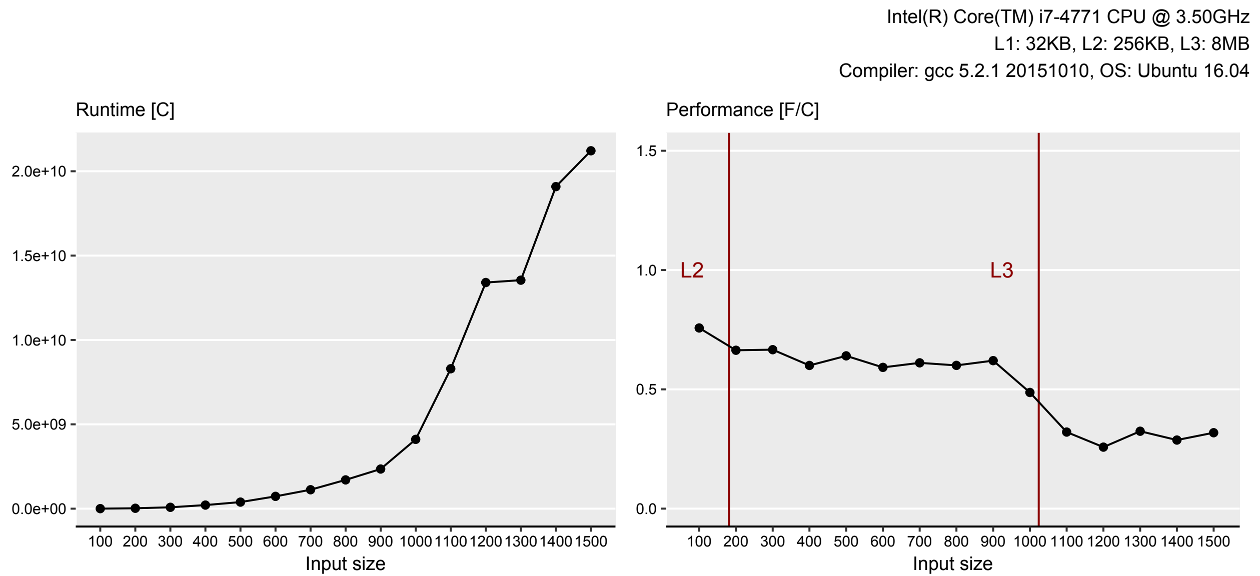


Figure 1: Plots resulting from execution of `mmm.c` on a Haswell CPU (scalar peak performance: 4 f/c). The code was compiled with `gcc` 5.2.1 with `O3` enabled, no FMAs and no vectorization.

(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.

5. (20 pts) Performance Analysis
Assume that the elements of vectors u, v, x, y and z of length n are combined as follows:

$$z_i = z_i + u_i \cdot v_i + x_i \cdot y_i \ .$$

(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 4.

(c) Then, for all two-power sizes $n = 2^4, \ldots, 2^{22}$ create a performance plot with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Randomly initialize all arrays. For all n repeat your measurements 30 times reporting the median in your plot.

(d) Briefly explain eventual performance variations in your plot.

(e) Assume that the vectors are now combined as follows:

$$z_i = z_i + u_i \cdot v_i + x_i \cdot y_i + u_i \cdot y_i + x_i \cdot v_i + u_i \cdot x_i \ .$$

Create another performance plot following the instructions in 5c. Briefly explain the performance behaviours and any differences in performance compared to the previous function.

Solution

As a possible solution, we consider the scalar execution of compute() on a Sandy Bridge CPU[1].

(a-b) See the file combine.cpp.

(c) See Fig. 2(a).

(d) The code is always bound by data movement:

- When data fits in L1, performance is bound by 1.5 f/c. On Sandy Bridge we can either issue two loads or one load and one store at any given cycle (on port 2 and 3 of the execution core) with an average bandwidth of 1.5 doubles per cycle. The runtime for the computation is therefore approximately $5n$ doubles/1.5 doubles/c $= 3.3n$ cycles.
- When data fits in L2 we can fetch at most 1 double per cycle from L2 bounding the runtime to $5n$ cycles and therefore performance by 0.8 f/c.
- Also from L3 we can fetch at most 1 double per cycle, not introducing any further bottleneck.
- Note that full bandwidth for the three levels of cache (i.e., 48 B/c for L1 and 32 B/c for L2/L3) can only be approached by using 256-bit memory accesses (with AVX instructions).
- When data exceeds the last-level cache size the runtime becomes bound by the off-chip bandwidth ($10n$ cycles).

e See Fig. 2(b). In this case the new function computes $f = 10n$ flops. The major difference comparing with the initial function is that the new one is bound by the computation runtime whenever data fit in any level of cache:

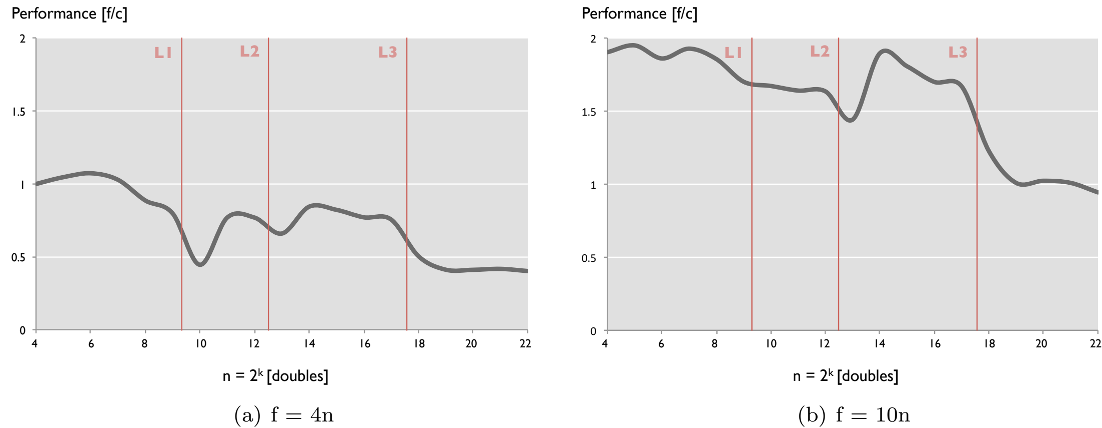
$$\begin{aligned}
\text{L1}: \frac{10n \text{ flops}}{2 \text{ f/c}} &= 5n \text{ cycles} > \frac{5n \text{ doubles}}{1.5 \text{ doubles/c}} = 3.3n \text{ cycles}\\
\text{L2-L3}: \frac{10n \text{ flops}}{2 \text{ f/c}} &= 5n \text{ cycles} = \frac{5n \text{ doubles}}{1 \text{ double/c}} = 5n \text{ cycles}\\
\text{RAM}: \frac{10n \text{ flops}}{2 \text{ f/c}} &= 5n \text{ cycles} < \frac{5n \text{ doubles}}{0.5 \text{ double/c}} = 10n \text{ cycles}
\end{aligned}$$

6. (20 pts) Bounds

Consider the below modified version of the Horner's algorithm for evaluating polynomials. The function operates on two matrices and stores the results in an output array:

```
1  // assume cA, cB and res are properly initialized
2  void horner (float x, float y, int n, float * cA [], float * cB [], float * res)  {
3      for (int i = 0; i < n; i += 1) {
4          float d1 = 0.0, d2 = 0.0; d3 = 0.0;
5          for (int j = n - 1; j >= 0; j -= 1) {
6              d1 = d1 * x + cA[i][j];
7              d2 = d2 * y - cB[j][i];
```

[1] For technical details we refer to the Intel Optimization Manual, Sec. 2.3.



(a) f = 4n

(b) f = 10n

Figure 2: Plots resulting from execution of combine.cpp. The code was compiled with icc v.15 and executed on a Sandy Bridge CPU (Intel Core i7-2600: 32 kB L1-D, 256 kB L2, 8 MB L3; scalar peak performance: 2 f/c; L1 tp: 1.5 doubles/c; L2/L3 tp: 1 double/c; RAM tp: 0.5 double/c.

```
8                 d3 += d1 / d2;
9             }
10            res[i] = d3;
11        }
12    }
```

We consider a Core i7 CPU based on a Haswell processor. It offers (as part of AVX2) a so-called fused multiply-add instruction, called FMA, that computes `y = a * x + b` on floating point numbers. Further it offers an instruction (called RCP) that approximates $1/x$ using Newton-Raphson step. The FMA on Haswell is as fast multiplication, meaning the throughput is two per cycle; the RCP one per cycle.

More information on these instruction and the load bandwidth od the caches in the memory hierarchy can be found in Intel 64 and IA-32 Architectures Optimization Reference Manual. Assume that the load bandwidth from L3 cache is half the one of L2 cache and that the load bandwidth from RAM is half that of the L3 cache.

(a) Determine the exact cost measured in the abovementioned two operations.

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

(c) Consider only one core and determine hard lower bounds (not asymptotic) on the runtime (measured in cycles):

  i. The op count. Assume that the code is compiled using `gcc` with the following flags: `-ffast-math -fno-tree-vectorize -mfma -march=core-avx2 -mrecip=all -O3` and that FMAs are used as much as possible.

  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).

(d) (enthusiast; no points) Harden the ops-based lower bound by also taking into account the latency of the operations used and their dependencies in the code.

Solution:

(a) Note that the use of RCP transforms line 8 as: `d3 = d1 * (1 / d2) + d3` - which is another FMA instruction. Since FMA does addition and multiplication, it is counted as two flops. Therefore $C(N) = N^2 \cdot (2 \cdot 3 + 1) = 7 \cdot N^2$.

(b) $I(N) = \frac{7 \cdot N^2}{4 \cdot (2 \cdot N^2 + N)}$ flops/B. Therefore $I(N) \in O(1)$

(c) i. There are N^2 iterations, and in each iteration we have 3 FMAs and one RCP. There are two FMA units on Haswell on different ports, and one RCP unit, that shares a port with one FMA. Therefore, ignoring the dependencies - in the best case scenario, $2 \cdot N^2$ FMAs can be executed in parallel, in N^2 cycles, and the rest of RCP and FMA can be executed in parallel also in N^2 cycles. Therefore lower bound is $2 \cdot N^2$

ii. Intel 64 and IA-32 Architectures Optimization Reference Manual (section 2.2.4 Cache and Memory Subsystem) suggests that peak bandwidth of L1 and L2 is 64 bytes / cycle. Therefore $tp_{L1} = tp_{L2} = 16$ floats/cycle.

$$r_{L1} = r_{L2} = \frac{2 \cdot N^2}{16}, \; r_{L3} = \frac{2 \cdot N^2}{8}, \; r_{RAM} = \frac{2 \cdot N^2}{4}$$

(d) To harden the bound, we take into consideration the dependencies imposed on the FMAs and RCP. Obviously, even if we disregard line 8, the FMAs on lines 6 & 7, will be forced to run in latency mode. Therefore we will need at least $5 \cdot N^2$ cycles. However, the combination of FMAs and RCP, does not force the whole algorithm to run in latency mode.

Haswell, has two ports with FMA units, namely Port 0 and Port 1. RCP is located on Port 0 (since it is part of the DIV unit) and since we deal with non-vectorized code, it has latency of 5 cycles (using `rcpss` instruction). Compiling the code with `-O3` might hint `gcc` to unroll the loop fully or partially. Then it takes up to the CPU out of order execution engine to take over, and reorder the instructions for best performance. Note that if we execute FMAs on one port only, then all FMAs will work in throughput mode, whenever possible.

Since we need to calculate lower bounds, lets assume a perfect out of order execution and consider Figure 3. We can start executing the FMA that calculates `d2`, followed by the one that calculates `d1`. Since those are pipelined on the same port, `d2` will be available after 5 cycles, and `d1` after 6 cycles. After the 5th cycles, `d2` from the next iteration gets inside the pipeline, followed by `d1` from the same iteration. Meanwhile in the 5th cycles, RCP gets executed on Port 0. And after 5 cycles (10 from the start of the iteration), we have already calculated `1/d2`. At this point in time, `d2` and `d1` are already in the pipeline execution of Port 1, and we need to wait for 2 more cycles, till `d3` is placed in the pipeline. Meanwhile, `d2` and `d1` from the second iteration of the innermost loop have become available, and we can proceed executing RCP from the second iteration. And so on and so forth.

This means that while all FMAs are going to operate in throughput mode (whenever possible), RCP will only operate in latency mode. Therefore the hardened lower bound will be:

$$5 + 5 \cdot N^2 + 7 = 12 + 5 \cdot N^2 \text{ cycles}$$

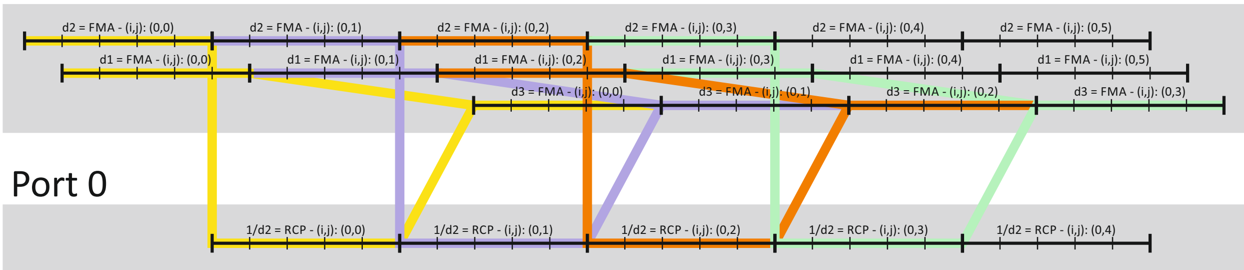


Figure 3: Optimal out of order execution of the `horner` function. Each color represents a complete execution of the computation contained in a single iteration of the innermost loop.