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

Exercises:

1. (20 pts) Get to know your machine
   The following microarchitectural parameters are of Tyler's desktop machine:

   (a) Processor manufacturer, name, and number.
   Solution: Intel(R) Xeon(R) CPU E3-1275 v5

   (b) Number of CPU logical and physical cores.
   Solution: 4 physical, 8 logical.

   (c) CPU-core frequency.
   Solution: 3.6 GHz is the nominal CPU frequency. At the time of writing, it was throttled down to 2.6 GHz

   (d) CPU maximum frequency. Does your CPU support Turbo Boost Technology (or, if not an Intel CPU, something similar)?
   Solution: It does support Turbo Boost, and the maximum frequency is 4.0GHz.

   For one core and **without** using SIMD vector instructions determine:

   (d) Latency [cycles] and throughput [ops/cycle] for floating point additions.
   Solution: Latency: 4 cycles. Throughput: 2 per cycle.

   (e) Latency [cycles] and throughput [ops/cycle] for floating point multiplications.
   Solution: Latency: 4 cycles. Throughput: 2 per cycle.
   Note: Intel intrinsics guide lists the latency of double-precision multiply (`MULSD`) as 3 cycles but both our own and Agner Fog's measurements indicate that it is 4 cycles.

   (f) Latency [cycles] and throughput [ops/cycle] for floating point divisions.
   Solution:
   For single precision (`DIVSS`) Latency: 11 cycles. Throughput: 0.33 per cycle.
   For double precision (`DIVSD`) Latency: 14 cycles. Throughput: 0.25 per cycle.

   (g) Latency [cycles] and throughput [ops/cycle] for fused multiply-add (FMA) operations (if supported).
   Solution: Latency: 4 cycles. Throughput: 2 per cycle.

   (h) Latency [cycles] and throughput [ops/cycle] for converting a double-precision floating-point element to a signed 32-bit integer.
   Solution: Assuming the following instruction is used:
   https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=cvtsd2si&expand=1769, Latency: 6 cycles. Throughput: 1 per cycle.

   (i) Maximum theoretical floating point peak performance in both flop/cycle and Gflop/s.
   Solution: Without SIMD instructions, it can issue two FMA instructions each cycle, giving a maximum of 4 flops/cycle and 16 Gflop/s with turboboost.

2. (10 pts) Cost analysis Consider the following algorithm that solves the linear system $Lx = b$ for x given L and b, where L is an $n \times n$ lower-triangular matrix:

```
1 void lower_triangular_solve (double L[], double x[], double b[], int N)  {
2     for(int i = 0; i < N; i++) {
3         double el10_dot_x0 = 0.0;
4         for(int j = 0; j < i; j++) {
5             el10_dot_x0 += L[i + j*N] * x[j];
6         }
7         x[i] = (b[i] - el10_dot_x0) / L[i + i*N];
8     }
9 }
```

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

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

Solution:

(a) The function `lower_triangular_solve` performs floating point multiplications, divisions, and additions. Therefore,

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

(b) There are N iterations of the outer loop. For each outer loop iteration there are i iterations of the inner loop. From this we can determine that there are a total of $(N-1)N/2$ iterations of the inner loop. Each iteration of the inner loop has 1 floating point addition and 1 floating point multiplication. After the inner loop, the function performs 1 floating point addition and 1 floating point addition. Thus,

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

3. (25 pts) Matrix multiplication

In this exercise, we provide a C source file for multplying an $n \times n$ matrix with a vector and a C header file to time the matrix-vector multiplication using different methods under Windows and Linux (for x86 compatible systems).

(a) Inspect and understand the code.

(b) Determine the exact number of (floating point) additions and multiplications performed by the compute() function in `mvm.c`.

(c) Using your computer, compile and run the code. Compile with the highest level of optimization provided by your compiler (with GCC, compile with the flag "-O3"). A modern compiler will automatically vectorize this very simple routine. Ensure you get consistent timings between timers and for at least two consecutive executions.

(d) Then, for all square matrices of sizes n between 200 and 4000, in increments of 200, 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 reached.

(e) Briefly discuss your plots.

Solution:

Intel(R) Xeon(R) CPU E3−1275 v5 @ 3.60GHz
L1: 32KB, L2: 256KB, L3: 8MB
Compiler: icc version 18.0.3, OS: Ubuntu 18.04.2

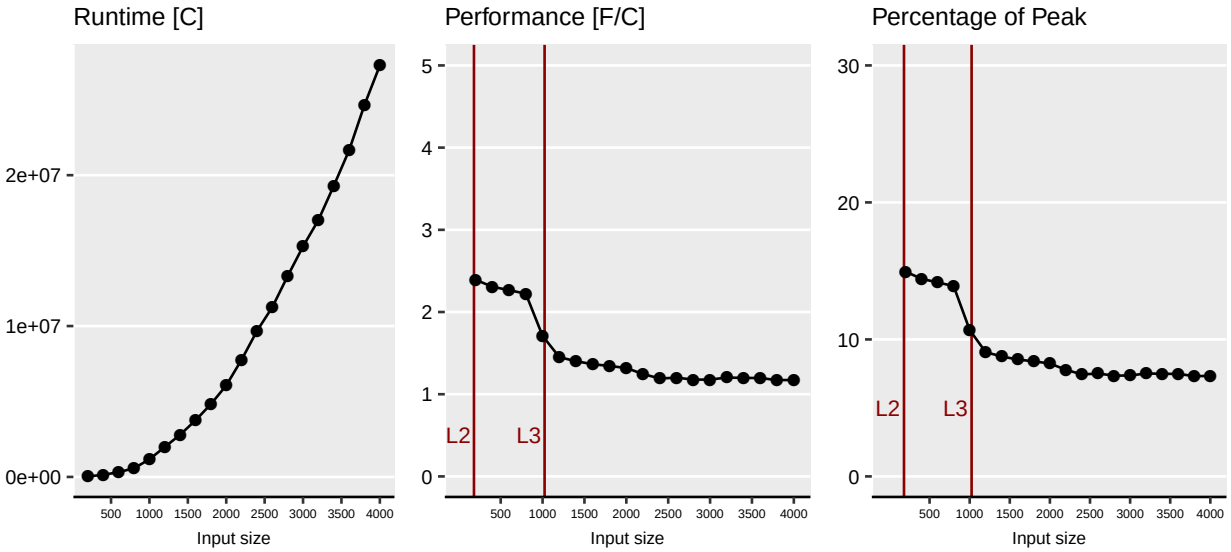

Figure 1: Plots resulting from execution of `mvm.c` on a Skylake CPU (vector peak performance: 16 f/c). The code was compiled with `icc` 18.0.3 with `O3` enabled.

(b) The code performs $2n^2$ 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.

(e) The computation performs well for small problem sizes but performance suffers greatly as soon as the matrices no longer fit in the L3 cache. For large problem sizes, matrix-vector multiplication is an inherently memory-bound operation.

4. (20 pts) Performance Analysis

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

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

(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 $n = 2^4, \ldots, 2^{23}$ 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 series has the major optimizations flags enabled (including 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, but make sure that Intel Turbo Boost is disabled (or enabled 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.

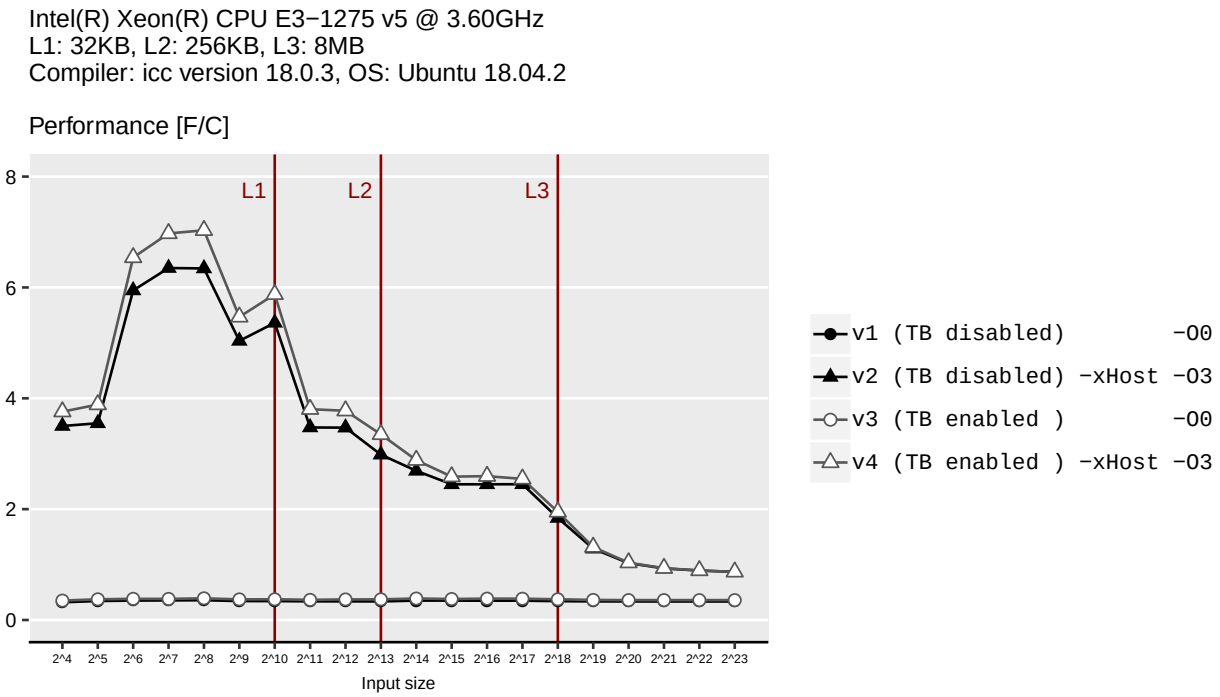

Figure 2: Plots resulting from execution of `combine.c` on an Intel Skylake CPU (peak performance: 16 f/c). The table reflects the performance values obtained running `v2` series.

Solution:

Code can be found here: https://acl.inf.ethz.ch/teaching/fastcode/2019/homeworks/hw01files/combine.zip.

Compiling the algorithm with all optimisations disabled, will result in machine code that is neither optimized or vectorized, and the performance is flat across problem sizes. However with optimizations turned on, we see that performance varies across problem sizes. Performance is great when the data fits in cache, and becomes worse as the size of the data grows. We can even see "steps": performance is greatest when the data fits in L1, and becomes incrementally worse as it no longer fits in subsequent levels of cache. Finally when it no longer fits in the L3 cache, the computation is almost as slow as when optimizations were turned off.

We use the CPU's time step counter (`RDTSC`) to measure performance. The time step counter increments at a constant rate equal to the nominal clock frequency of the CPU, so it is a measure of time rather than cpu cycles. Turbo Boost increases the CPU's frequency above this nomial clock frequency. The number of CPU cycles that are needed to complete the algorithm do not decrease 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.

5. (20 pts) Bounds

Consider the three artificial computations below. The functions operate on input arrays of length N and store the results in an output array of length N:

```
1  void artcomp1(float alpha, float x[], float y[], float z[], int N) {
2    for (int i = 0; i < N; i++)
3      y[i] = alpha * (x[i] + y[i] + z[i] + 1.0);
4  }
5  void artcomp2(float alpha, float x[], float y[], int N) {
6    for (int i = 0; i < N; i++) {
7      y[i] = alpha / x[i];
8    }
9  }
10 void artcomp3(float x[], float y[], float z[], int N) {
11   //Assume 3 divides N
12   for (int i = 0; i < N; i += 3) {
13     y[i] = x[i] * z[i];
14     y[i + 1] += x[i + 1] * z[i + 1];
15     y[i + 2] = x[i + 2] + z[i + 2];
16   }
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 that divisions are performed with the regular div on Port 0. 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 upper bound on the operational intensity of each function. Assume empty caches and consider only reads but note: arrays that are only written are also read and the read should be included.

(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 computation (see lecture slides).

ii. Loads, for each of the following cases: All floating point data is L1-resident, L2-resident, and RAM-resident. Consider best case scenario (peak bandwidth).

Solution:

(a) The flop cost for each function are

i. $C(N) = 4N$

ii. $C(N) = N$

iii. $C(N) = 4N/3$

Note that 1.0 is a double, and `artcomp1` therefore has two conversions. If these are included, $C(N) = 6N$ for that operation.

(b) The operational intensity is

i. $I(N) = \frac{4N flops}{3N float} = \frac{1 flops}{3 bytes}$

ii. $I(N) = \frac{N flops}{2N floats} = \frac{1 flops}{8 bytes}$

iii. $I(N) = \frac{4N/3 flops}{3N floats} = \frac{1 flop}{9 bytes}$

(c) For all cases $I(N) \in O(1)$

i. These computations are all throughput rather than latency bound. We can obtain bounds by examining which execution ports the instructions are scheduled to and the throughputs of those instructions.

A. There are a few different answers to this question that we consider valid.

- If one interpreted "FMAs are used as much as possible" to mean that FMAs are used whenever an addition and a multiplication can be fused, then each iteration can be executed using one FMA and two addition instructions. The FMA can be co-issued with one of the additions, but the addition instructions must be scheduled on the same port. This resulting in a lower bound of $2N$ cycles.
- If "FMAs are used as much as possible" is instead interpreted to mean additions ($c := a + b$) are performed using FMA instructions (of the form $c := 1.0 \cdot a + b$), but the additions (implemented as FMA instructions) do not need to be scheduled on the same port. This resulting in a lower bound of $1.5N$ cycles.
- Without fast math turned on, the compiler is not allowed to apply transformations such as

  `alpha * (x[i] + y[i] + z[i] + 1.0)` $\rightarrow$ `alpha * (x[i] + y[i] + z[i]) + alpha`

  Then there are 3 additions and 1 multiplication that cannot be fused. If the additions are implemented as addition instructions, they must all be executed on the same port, giving a lower bound of $3N$ cycles. If they are instead FMA instructions, we get a lower bound of $2N$ cycles as there are 4 instructions that can all be executed on either of two ports.
- The literal `1.0` should have instead been `1.0f`. In the present code, The present code must convert the result of $x[i] + y[i] + z[i]$ must be converted to double-precision, and the result must be converted to single-precision before storing it in $y[i]$. then there must be two conversion instructions: `CVTSS2SD`, which can be executed on ports 0 or 5, and `CVTSD2SS`, which can be executed on ports 1 or 5.

B. Single-precision division has a throughput of 1 per 7 cycles on haswell. Therefore the lower bound is $7N$ cycles.

C. We must consider two iterations of the loop at a time.
Each iteration, we have a multiplication, FMA, and addition. The multiplication of the first iteration and the FMA of the first iteration can be co-issued, and the addition of the first iteration can be co-issued with the multiplication of the second iteration. Then the FMA of the second iteration and the addition of the second iteration can be co-issued, and this process can repeat for subsequent iterations. Two iterations must take 3 cycles, and there are $N/3$ iterations. The lower bound is $N/2$ cycles.

ii. A. $C_{\text{double loads}}(N) = 3/2N$

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

C. $C_{\text{double loads}}(N) = 3/2N$

iii. http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring17/slides/arch.pdf shows peak bandwidth of L1, L2 and an estimate for the RAM throughput. It follows that:

A. $r_{L1} = \frac{3N}{16}$, $r_{L2} = \frac{3N}{16}$, $r_{RAM} = \frac{3N}{4}$

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

C. $r_{L1} = \frac{3N}{16}$, $r_{L2} = \frac{3N}{16}$, $r_{RAM} = \frac{3N}{4}$