263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 6th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2020/
Questions: fastcode@lists.inf.ethz.ch

Exercises:

1. (15 pts) Get to know your machine
   The following microarchitectural parameters are of an Intel(R) Xeon(R) CPU E3-1275 machine:

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

   (b) CPU base frequency.
   Solution: 3.6 GHz is the nominal CPU frequency.

   (c) CPU maximum frequency. Does your CPU support Turbo Boost or a similar technology?
   Solution: It does support Turbo Boost, and the maximum frequency is 4.0GHz.

   (d) Phase in the Intel's development model: Tick, Tock or Optimization. (if applicable)
   Solution: Tock phase (Skylake).

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

   (d) Maximum theoretical floating point peak performance in flop/cycle.
   Solution: Without SIMD instructions, two FMAs can be issued per cycle. Thus, 4 flops/cycle.

   (e) Latency [cycles] and throughput [ops/cycle] for floating point division.
   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.

   (f) Latency [cycles] and throughput [ops/cycle] for floating point square root.
   Solution:
   According to Intel:
   For single precision (`SQRTSS`) Latency: 13 cycles. Throughput: 0.33 per cycle.
   For double precision (`SQRTSD`) Latency: 18 cycles. Throughput: 0.166 per cycle.
   According to Agner Fog's measurements:
   For single precision (`SQRTSS`) Latency: 12 cycles. Throughput: 0.33 per cycle.
   For double precision (`SQRTSD`) Latency: 15-16 cycles. Throughput: 0.166-0.25 per cycle.

   (g) Latency [cycles] and throughput [ops/cycle] for floating point approximate reciprocal (`rcp`) instruction (if supported).
   Solution:
   For single precision (`RCPSS`) Latency: 4 cycles. Throughput: 1 per cycle.

2. (25 pts) Matrix-vector multiplication

   In this exercise, we provide a C source file for multiplying 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). Inspect and understand the code and do the following:

   (a) Using your computer, compile and run the code for $n = 400$. Compile using GCC with the highest level of optimization (use 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. Don't forget to disable Turbo Boost. (No need to answer anything here)

   (b) Determine the exact number of (floating point) additions and multiplications performed by the `compute()` function in `mvm.c`.
   Solution:
   The code performs $2n^2$ floating point operations.

(c) For all square matrices of sizes n between 200 and 4000, in increments of 200, create a performance plot with n on the x-axis and performance (in flops/cycle) on the y-axis. Create three series such that:

i. The first series has all optimizations disabled: use flag `-O0`.

ii. The second series has the major optimizations except for vectorization: use flags `-O3` and `-fno-tree-vectorize`.

iii. The third series has all major optimizations enabled: use flags `-O3`, `-ffast-math` and `-march=native`.

Solution:

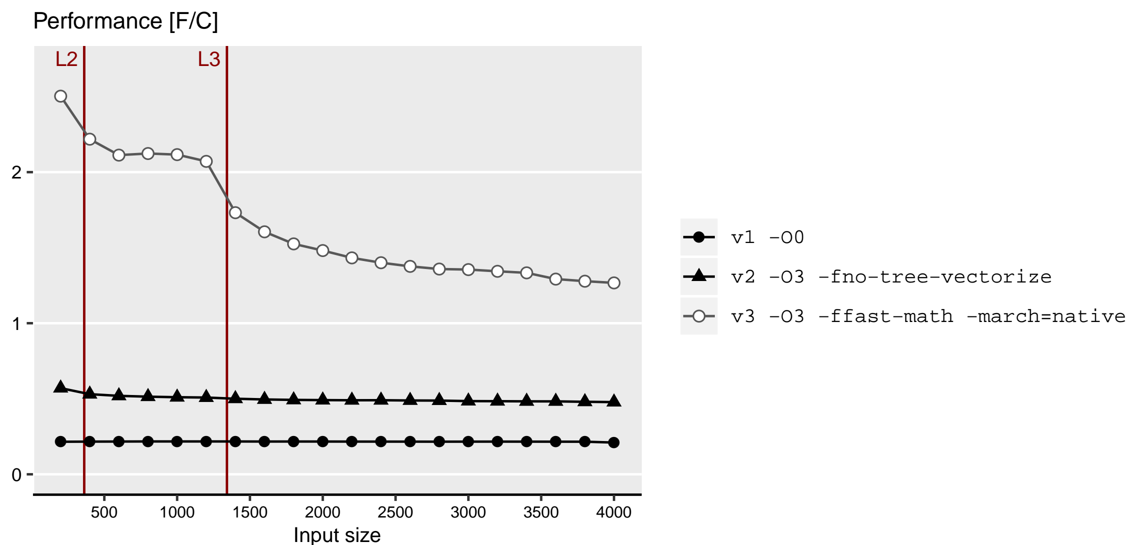

Figure 1: Plots resulting from execution of `mvm.c` on a Cascade Lake CPU (vector peak performance: 16 f/c for the given flags). The code was compiled with GCC 8.1.

(d) Briefly discuss your plots and report the highest performance that you achieved.
Solution:

i. Non-optimized (`v1`): This results in machine code that is neither optimized or vectorized. This is nice for debugging. However, the performance is low and flat across problem sizes.

ii. Optimized but non-vectorized (`v2`): The performance is better than in the previous case. However, the performance suffers due to the limited amount of ILP caused by inter loop dependencies.

iii. Fully optimized (`v3`): The `-ffast-math` flag enables ILP which is combined with vectorization and significantly improves performance. 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.

3. (10 pts) Performance Analysis
Assume that the elements of vectors x, y, u and z of length n are combined as follows:

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

(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 2.

(c) Then, for all two-power sizes $n = 2^4, \ldots, 2^{23}$ 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. Compile your code using GCC with flags `-O3` and `-march=native`.

(d) Briefly explain performance variations in your plot.

(e) Enroll and submit the code in Code Expert.

Solution:

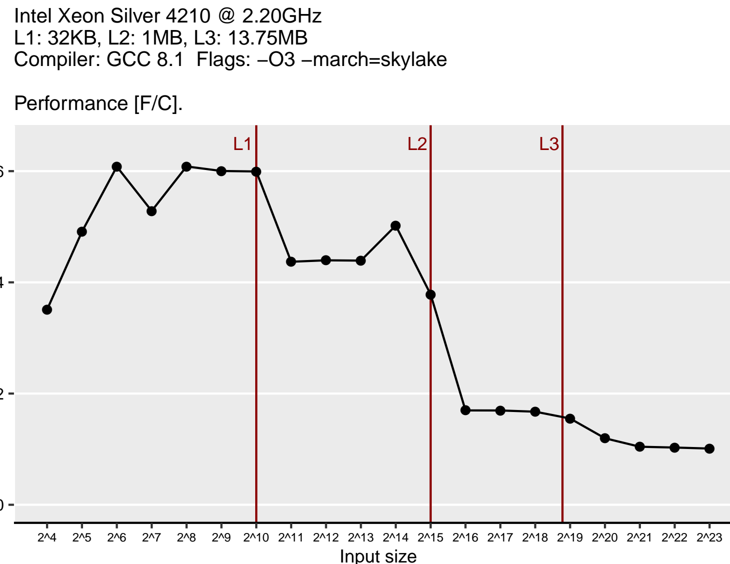

Figure 2: Plots resulting from execution of `combine.c` on a Cascade Lake CPU (vector peak performance: 16 f/c for the given flags). The code was compiled with GCC 8.1.

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.

4. (30 pts) Cost analysis and bounds

Consider the following algorithm that accurately multiplies a vector x with an $n \times n$ matrix A.

```
1  void accurate_mvm (double A[], double x[], double y[], int N)  {
2      double sum_h, sum_l, Aij, Xj, th, tl, sh, sl, a, b;
3      for(int i = 0; i < N; i++) {
4          sum_h = 0.0;
5          sum_l = 0.0;
6          for(int j = 0; j < N; j++) {
7              Aij = A[n*i + j];
8              Xj  = x[j];
9              th  = Aij * Xj;
10             tl  = fma(Aij, Xj, -th);
11             sh  = th + sum_h;
12             a   = sh - sum_h;
13             b   = sh - a;
14             sl  = (th - a) + (sum_h - b);
15             sum_h = sh;
16             sum_l += tl + sl;
17         }
18         y[i] = sum_h + sum_l;
19     }
20 }
```

We consider a Core i7 CPU based on a Haswell processor. As seen in the lecture, it offers FMA instructions (as part of AVX2). Assume that function `fma(a,b,c)` always is translated into an FMA instruction that computes `y = a * b + c`. Further, assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture. Finally, assume that no optimization is performed that simplifies floating point arithmetic (i.e. `-ffast-math` flag is not used). Recall that we consider the FMA instruction as two floating point operations (an addition and a multiplication).

(a) Define a suitable detailed floating point cost measure $C(n)$.
Solution: The function performs floating point multiplications and additions. Therefore,

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

(b) Compute the cost $C(n)$ of the function `accurate_mvm`.
Solution: There are N iterations of the outer loop and N^2 iterations of the inner loop. Each iteration of the inner loop has 9 floating point additions and 2 floating point multiplications. After the inner loop, the function performs 1 floating point addition. Thus,

$$\begin{aligned} N_{add} &= 9N^2 + N, \\ N_{mul} &= 2N^2, \\ C(n) &= C_{add} \cdot (9N^2 + N) + C_{mul} \cdot (2N^2). \end{aligned}$$

(c) Consider only one core without using vector instructions (i.e. using flag `-fno-tree-vectorize`) and determine a hard lower bound (not asymptotic) on the runtime (measured in cycles), based on:

i. The throughput of the floating point operations. Assume that the only FMA instruction used is the one given by function `fma`. Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).

ii. The throughput of the floating point operations where FMAs are used as much as possible to fuse an addition and a multiplication. For this exercise, it is also allowed to transform additions ($c := a + b$) into FMA instructions (of the form $c := 1.0 \cdot a + b$).

iii. Loads, for the following two cases: All floating point data is L1-resident, and all floating point data is RAM-resident. Consider the best case scenario (peak bandwidth and ignore latency).

Solution: We can obtain bounds by examining which execution ports the instructions are scheduled to and the throughputs of those instructions.

i. The instruction mix in this case consists of $8N^2 + N$ floating point additions, N^2 floating point multiplications and N^2 FMA instructions. Additions can only be scheduled in Port 1 of a Haswell computer. Thus, the bottleneck is in this port, resulting in a lower bound of $8N^2 + N$ cycles.

ii. Considering that all additions are performed using FMA instructions of the form $c := 1.0 \cdot a + b$, we have now $9N^2 + N$ FMA instructions and N^2 multiplications. FMAs and multiplications can be scheduled in either Port 0 or Port 1. Thus, resulting in a lower bound of $\frac{1}{2}(10N^2 + N) = 5N^2 + \frac{N}{2}$ cycles.

iii. https://acl.inf.ethz.ch/teaching/fastcode/2020-/slides/arch.pdf shows peak bandwidth of L1, and an estimate for the RAM throughput. Note that when data is L1-resident (warm cache) only $N^2 + N$ doubles have to be read thus: $r_{L1} \geq \frac{1}{8}(N^2 + N)$. When data is in RAM (cold cache), arrays that are only written are also read, thus the lower bound for this case is: $r_{RAM} \geq \frac{1}{2}(N^2 + 2N)$.
Note: Since there may be a confusion about when to consider reads for arrays that are only written, for this homework we will accept also the answer $r_{L1} \geq \frac{1}{8}(N^2 + 2N)$ for the case of L1-resident data.

(d) Determine an upper bound on the operational intensity. Assume empty caches and consider only reads but note: arrays that are only written are also read and the read should be included.
Solution: The operational intensity is $I(N) \leq \frac{11N^2 + N flops}{8(N^2 + 2N) bytes} \approx \frac{11 flops}{8 bytes}$

5. (20 pts) ILP analysis

Consider the artificial computation below. The function operates on input arrays of length N.

```
1 double artcomp(double a[], double b[], double c[], int N) {
2     double s = 0.0;
3     for (int i = 0; i < N; i++) {
4         s = (s + a[i] + b[i]) * c[i] + (a[i] * c[i] + s) * b[i];
5     }
6     return s;
7 }
```

Make the same assumptions as in the previous exercises, i.e., consider a Haswell processor, only one core without using vector instructions (using flag `-fno-tree-vectorize`), and assume that no optimization is performed that simplifies floating point arithmetic (i.e. `-ffast-math` flag is not used). Thus, it is not allowed to apply associativity and distributivity laws to rearrange the computation. Determine hard lower bounds (not asymptotic) on the runtime (measured in cycles), based on:

(a) The latency, throughput and dependencies of the floating point operations. Assume that no FMA instruction is generated (i.e. `-mfma` flag is not used). Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).
Solution: We have to determine the critical path of the dependency graph. Thus, the runtime is at least $14N$ cyles (see Figure 3).

(b) The latency, throughput and dependencies of the floating point operations where FMAs are used as much as possible to fuse an addition and a multiplication (i.e. `-mfma` flag is enabled). For this exercise, it is **not** allowed to transform additions ($c := a + b$) into FMA instructions (of the form $c := 1.0 \cdot a + b$).
Solution: $15N$ cycles (see Figure 4).

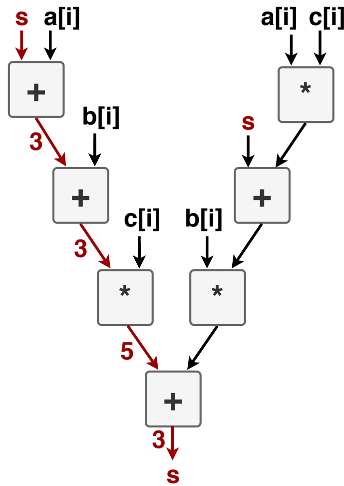

Figure 3: Dependency graph for one iteration of `artcomp`. The critical path is 14 cycles.

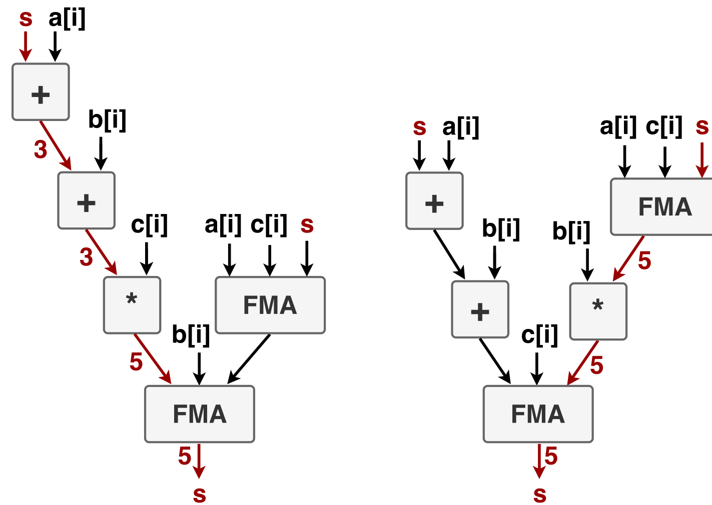

Figure 4: Two possible Dependency Graphs when using FMAs. The minimum critical path of the two is 15 cycles.