263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 11th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2021/
Questions: fastcode@lists.inf.ethz.ch

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.
   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:

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

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

   (g) Latency [cycles] and throughput [ops/cycle] for floating point comparison operations (e.g. greater-than, less-than, equal-to, etc).
   Solution:
   For `(U)COMISS/D` instruction:
   - Intel's Optimization Manual reports a latency of 2 cycles and throughput of 1 per cycle.
   - Intel's Intrinsic guide reports a latency of 3 cycles and throughput of 1 per cycle.
   - uops.info reports an upper bound for the latency of 3 cycles and throughput of 1 per cycle.
   - Agner does not specify latency and only reports throughput of 1 per cycle.

   For `CMPSS/D`, all sources report a latency of 4 cycles and throughput of 2 per cycle.

   Intel's processors offer two assembly instructions to compute scalar floating point addition in double precision, namely FADD (from x87) and ADDSD (from SSE2).

   (h) Which instruction is used when you compile code in your computer?
   Solution: ADDSD from SSE2.

   (i) Why is the other one still supported?
   Solution: For backward compatibility.

   Make sure to use the correct floating point operations in parts (f-g).

2. (20 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:

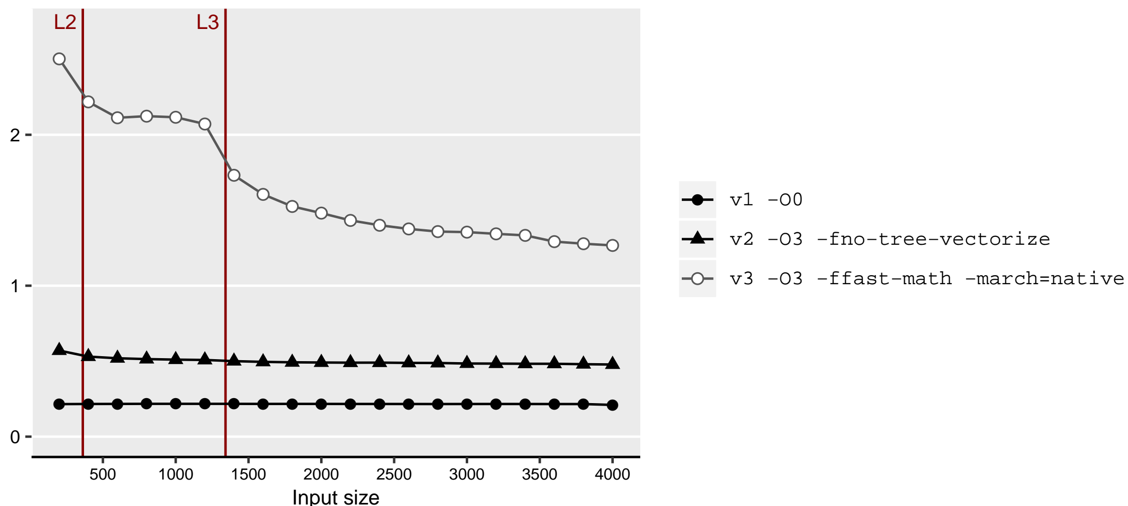

Figure 1: Plots resulting from execution of `mvm.c` (vector peak performance: 16 f/c for the given flags).

(d) Discuss performance variations of 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. (25 pts) Performance analysis and bounds
Assume that vectors x, y, u and z of length n are implemented using double precision floating point and combined as follows:

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

We consider a Core i7 CPU based on a Haswell processor. As seen in the lecture, it offers FMA instructions (as part of AVX2). Recall that we consider cost of the FMA instruction as two floating point operations (an addition and a multiplication). Assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture. Assume that no optimization is performed that simplifies floating point arithmetic (i.e. `-ffast-math` flag is not used). Answer the following and justify your answers.

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

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

(b) Compute the cost $C(n)$ of the computation.
Solution:

$$N_{add} = 2n,$$
$$N_{mul} = 4n,$$
$$C(n) = C_{add} \cdot (2n) + C_{mul} \cdot (4n).$$

(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 no FMA instructions are used. 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 to fuse an addition and a multiplication (i.e. `-mfma` flag is enabled).
iii. Data reads, 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 $2n$ floating point additions and $4n$ floating point multiplications. Multiplications can be scheduled in Ports 0 and 1, whereas additions in Port 1 only. In this case there are more multiplications than additions. Thus, the instructions can be balanced between the two ports, resulting in a lower bound of $3n$ cycles.
ii. By fusing all additions with a multiplication, we have $2n$ FMA instructions and $2n$ multiplications. FMAs and multiplications can be scheduled in either Port 0 or Port 1. Thus, resulting in a lower bound of $2n$ cycles.
iii. Abstracted Microarchitecture shows peak bandwidth of L1, and an estimate for the RAM throughput. In the computation, at least $4n$ doubles have to be read in total. Thus, $r_{L1} \geq \frac{4n}{8} = \frac{n}{2}$ and $r_{RAM} \geq \frac{4n}{2} = 2n$.

(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 this read should be included.
Solution: The operational intensity is $I(N) \leq \frac{6n flops}{8(4n) bytes} = \frac{3}{16}$.

4. (25 pts) Scalar product

Consider the following function that computes the scalar product of vectors x and y of size n:

```
1  double scalar_product(double *x, double *y, int n) {
2      double s = 0.0;
3      for (int i = 0; i < n; i++) {
4          s += x[i] * y[i];
5      }
6      return s;
7  }
```

(a) Create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 2 and for all two-power sizes $n = 2^4, \ldots, 2^{23}$ create a performance plot for the function `scalar_product` 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 -fno-tree-vectorize`.

(b) Perform optimizations that increase the ILP of function `scalar_product` to improve its runtime. It is not allowed to use FMA or vector instructions. Add the performance to the previous plot (so one plot with two series in total for (a) and (b)). Compile your code using GCC with flags `-O3 -fno-tree-vectorize`.

(c) Discuss performance variations of your plot and report the highest performance that you achieved.

(d) Enroll and submit the code of your optimized function in Code Expert. Carefully read and follow the instructions given in Code Expert to submit your code.

Solution:

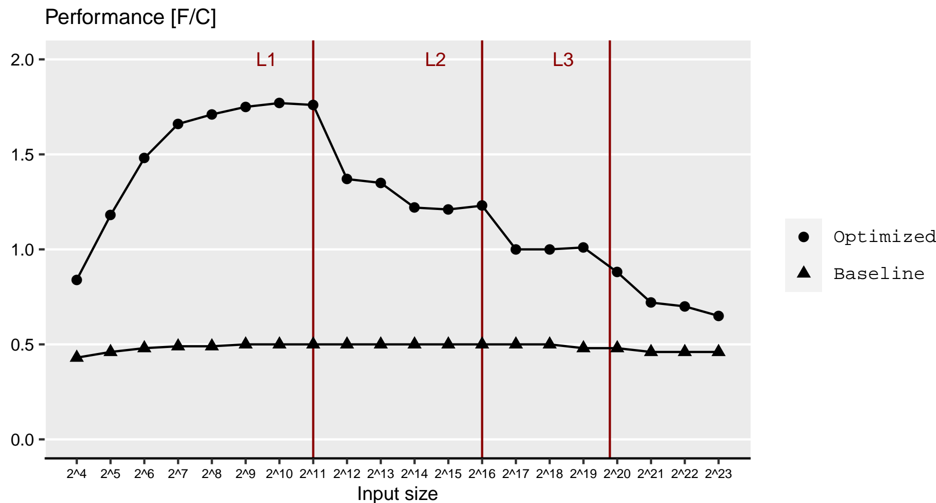

Figure 2: Performance plot (peak performance: 2 f/c for the given flags).

In the original code, the performance suffers from inter loop dependency which limits the amount of ILP. Thus, the performance is 0.5 flops/cycle across all problem sizes. Unrolling the loop and using separate accumulators increases the ILP. For this case, 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. The maximum performance achieved is 1.77 flops/cycle.

5. (10 pts) ILP analysis

Consider the following computation:

```
1  double artcomp(double a, double b, double c) {
2      double r;
3      r = (a + b) * (b + c) + (a * c);
4      return r;
5  }
```

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 the following. Justify your answers.

(a) The latency, throughput and dependencies of the floating point arithmetic 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).

Note: It may be useful to draw the Directed Acyclic Graph (DAG) of the computation.

Solution: The critical path of the DAG (see Figure 3) is 11 cycles. Note that the first two additions cannot start at the same time because additions can only be issued in Port 1. Thus, the computation is delayed by an additional cycle. The runtime is therefore at least 12 cycles.

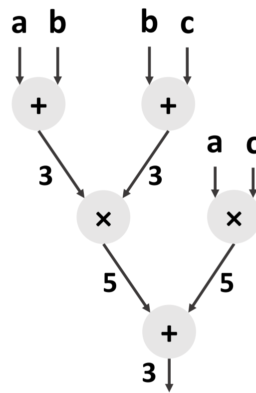

Figure 3: Dependency graph for `artcomp`.