263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 7th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2024/
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, number and microarchitecture (e.g. Haswell, Skylake, etc).
   Solution: Intel Xeon Silver 4410Y, Sapphire Rapids

   (b) CPU base frequency.
   Solution: 2.0 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 3.9GHz.

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

   Intel's processors offer two different floating-point instruction sets, namely x87 and SSE/SSE2, that can perform scalar floating-point operations. For example, a floating-point division can be performed using either FDIV (from x87) or DIVSD (from SSE2) assembly instructions.

   (e) Which floating-point division instruction is used when you compile code in your computer and why is the other one still supported?
   Solution: `vdivsd` is used. `fdiv` is still supported for backwards compatibility.

   (f) Explain how the x87 floating-point unit represents floating-point numbers differently from SSE2.
   Solution: x87 also supports 80-bits floating-point numbers, which use 15 bits for the exponent and 63 bits for the mantissa. Sign and integer part both use 1 bit.

   For one core and **without** using SIMD vector instructions, determine the following about your machine. In (g)-(i), make sure to use the correct floating-point instruction (not the one from x87 in case you have an Intel processor) and provide the reference where you found the latency and throughput information.

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

   (h) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision floating-point addition.
   Solution: Latency: 4 cycles. Throughput: 2 per cycle. Instruction: ADDSS(D).

   (i) Latency [cycles], throughput [ops/cycle] and instruction name for single-precision reciprocal square root.
   Solution:
   According to Agner Fog's measurements:
   Latency: 4 cycles. Throughput: 1 per cycle. Instructin: `RSQRTSS`

   (j) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision division operation.
   Solution:
   According to Agner Fog's measurements:
   Skylake: Latency: 11 cycles. Throughput: 0.33 per cycle. Instruction: DIVSS.
   Skylake: Latency: 13-14 cycles. Throughput: 0.25 per cycle. Instruction: DIVSD.

2. (20 pts) LU Factorization

In this exercise, we provide a C source file for computing the LU factorization $A = LU$, with A being a $n \times n$ matrix, and a C header file that allows to read the time stamp counter (TSC) of the processor for x86 compatible systems. The code uses different timers available to time the LU factorization. Note that if you have an Apple M processor, you can still access some of the timers available so you can still complete the homework. Inspect and understand the code and do the following:

(a) 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. Don't forget to disable Turbo Boost. (No need to answer anything here)

(b) Inspect the `compute()` function in `lu.c` and answer the following:

i. Determine the exact number of floating-point additions, multiplications and divisions that it performs.
Solution: The code performs $n(n-1)/2 + n^2(n-1)$ floating-point operations.

ii. Determine an upper bound on its operational intensity (consider only reads and assume empty caches).
Solution:
$W(n) = n(n-1)/2 + n^2(n-1) = \frac{n(2n+1)(n-1)}{2}$ and $Q(n) \geq 8 \cdot n^2$. Thus, $I(n) \leq \frac{(2n+1)(n-1)}{16n}$ flops/bytes.

(c) For all square matrices of sizes n between 100 and 2000, in increments of 100, create a performance plot with n on the x-axis and performance (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`. If you are using the clang compiler, also add `-fno-slp-vectorize` flag to disable vectorization.

iii. The third series has all major optimizations enabled including vectorization: use flags `-O3`, `-ffast-math` and `-march=native`. If you are using an Apple M processor and your compiler doesn't support `-march=native` you can use `-mcpu=apple-mx` (switch `x` for your processor version) instead.

Solution:

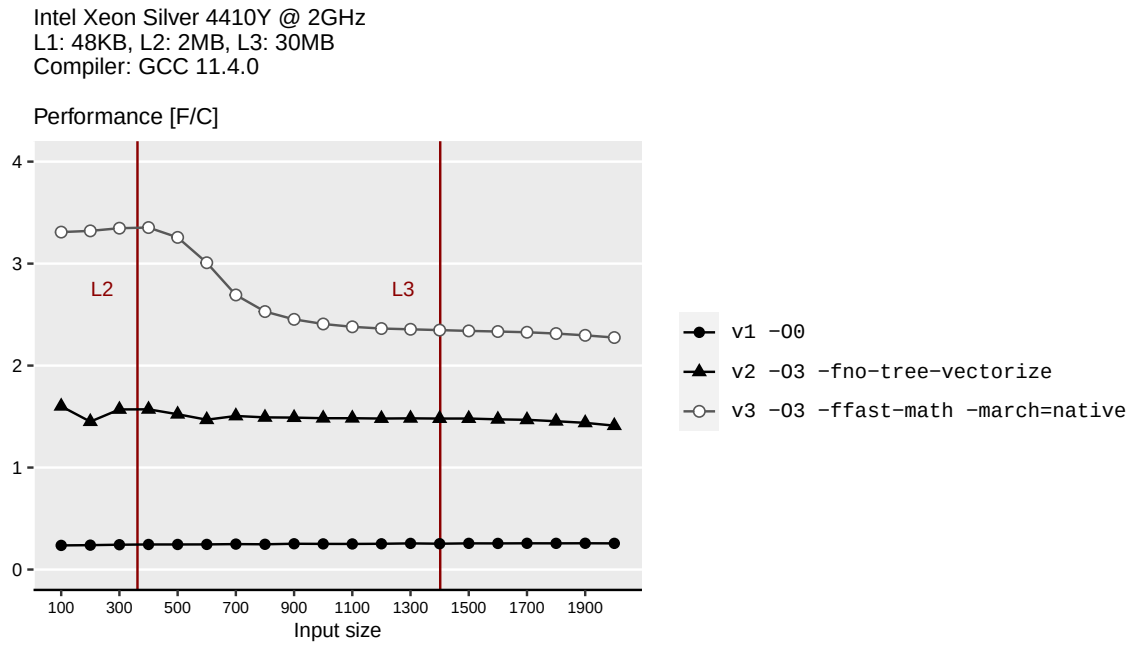

Figure 1: Plots resulting from execution of `lu.c` (vector peak performance: 16 f/c for the given flags).

(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, the compiler is not able to generate scalar FMAs instructions (example).

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 matrix A no longer fits in the cache. The highest performance that we achieve is 3.3 flops/cycle.

3. (25 pts) Performance analysis and bounds

Assume that vectors u, w, x, y and z of length n are implemented using double precision floating-point and combined as follows:

$$z_i = u_i \cdot (w_i + x_i) \cdot (u_i - x_i) + y_i$$

We consider a Core i7 CPU with a Skylake microarchitecture. 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:

$$N_{add} = 3n,$$
$$N_{mul} = 2n,$$
$$C(n) = C_{add} \cdot (3n) + C_{mul} \cdot (2n).$$

(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. The throughput of data reads, for the following two cases: All floating-point data is L3-resident, and all floating-point data is RAM-resident. Consider the best case scenario (peak bandwidth and ignore latency). Note that arrays that are only written are also read and this read should be included.

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 $3n$ floating-point additions and $2n$ floating-point multiplications. All operations can be scheduled in either Port 0 or Port 1. Thus, a lower bound on the runtime is $2.5n$ cycles.

ii. We can only fuse one addition with a multiplication into an FMA. Thus, we have n FMA instructions, $2n$ additions, n multiplications. FMAs can also 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 L3, and an estimate for the RAM throughput. In the computation, at least $5n$ doubles have to be read in total. Thus, $r_{L3} \geq \frac{5n}{4}$ and $r_{RAM} \geq \frac{5n}{2}$.

(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 $I(N) \leq \frac{5nflops}{8(5n)bytes} = \frac{1}{8}$ flops/byte.

4. (25 pts) Basic optimization

Consider the following function:

```
1 void comp(double* x, double *y, double *z, int n) {
2   for (int i = 0; i < n; i++) {
3     for (int k = 0; k < 2; k++){
4       z[k] += x[i+1-k] * y[i+k];
5     }
6   }
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 $n = 2^4, \ldots, 2^{27}$ create a performance plot for the function `comp` 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 with flags `-O3 -mfma`[1] `-fno-tree-vectorize`. If you are using clang, add also the `-fno-slp-vectorize` and `-ffp-contract=fast` flags.

[1]For Apple M processors, the flag `-mfma` may not be supported. If this is the case, use instead `-mcpu=apple-mx`, where `x` is your processor version, or `-march=native`.

(b) Considering the latency and throughput information of floating-point operations in your machine, and the dependencies in `comp`, derive an upper bound on the performance (flops/cycles) of `comp` when using the specified flags in (a), i.e., when FMA instructions are enabled (`-mfma`) but vectorization is disabled (`-fno-tree-vectorize`).

Solution:
The runtime is limited by a dependency when accumulating the values in `z[0]` and `z[1]`. The innermost loop is a single FMA, as we can issue two independent FMAs per cycle we can compute the two FMAs in parallel. The latency of FMA is 4 cycles (Skylake) and there are two in parallel in every outer-loop iterations. Thus, $T(n) \geq 4n$. Since $W(n) = 4n$, the performance is upper bounded by $\pi(n) \leq 1$ flops/cycle.

(c) Perform optimizations that increase the ILP of function `comp` to improve its runtime. It is not allowed to use vector instructions. Add the performance to the previous plot (so one plot with two series in total for (a) and (c)). Compile your code with the same flags as before.

(d) Discuss performance variations of your plot and report the highest performance that you achieved. Also discuss the optimizations that you performed to increase the ILP.

(e) 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:

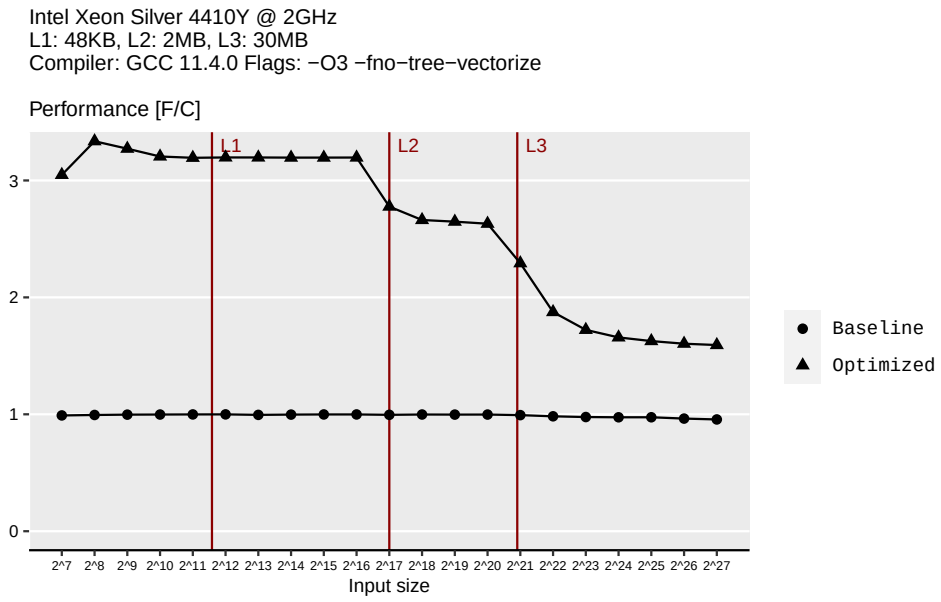

Figure 2: Performance plot (peak performance: 4 f/c for the given flags).

In the original code, the performance suffers from inter loop dependency which limits the amount of ILP. Thus, the performance is 1 flops/cycle across all problem sizes and it's consistent with the upper bound derived in (b). Unrolling the loop and using separate accumulators increases the ILP. For the given machine, we need at least 8 accumulators. 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 3.3 flops/cycle.

5. (10 pts) ILP analysis

Consider the following computations:

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

Make the same assumptions as in Exercise 3, i.e., consider a Skylake 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) for the following cases, based on the latency, throughput and dependencies of the floating-point operations only. Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides). It may be useful to draw the dependency graph of the computation. Justify your answers.

(a) Determine a hard lower bound on the runtime for `artcomp` when no FMA instruction is generated.

Solution: Multiplications and additions are issued on the same port. Since two independent additions and multiplications can be issued in the same cycle we need to wait one cycle. This increases the critical path by one. The runtime is at least **23 cycles**. as shown in the critical path of the dependency graph in Figure 3 (left).

(b) Determine a hard lower bound on the runtime for `artcomp` when FMA instructions are generated.

Solution: Now we can fuse an addition and a multiplication. This removes the initial contention on the floating point port. The runtime is at least **22 cycles** as shown in the critical path of the dependency graph in Figure 3 (right).

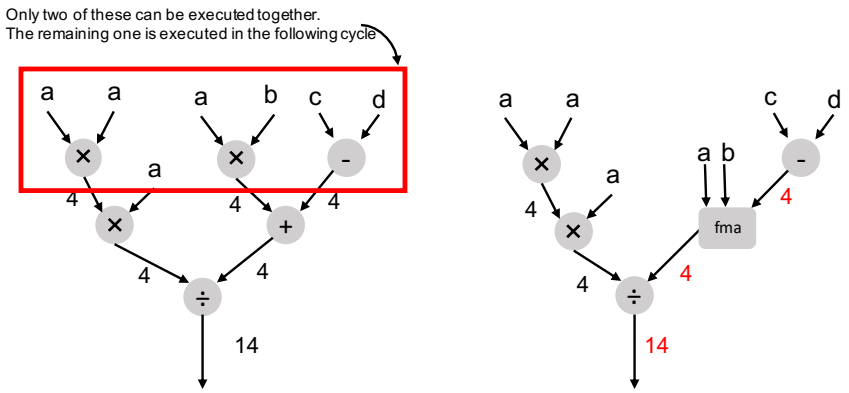

Figure 3: Dependency graph for `artcomp`.