263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 6th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2025/
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. Skylake, Ice-Lake, 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 Frequency scaling (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 (Golden Cove).

   X86 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. The x87 instruction set, however, is becoming deprecated but is still supported for backward compatibility.

   Consider the following instruction:

```
1 #define MAX(a, b) ((a) > (b) ? (a) : (b))
```

   Create a C/C++ file that uses this macro to compute the maximum of two *double-precision floating-point* numbers. What instruction is generated by the compiler in the following cases? Does the instruction belong to the x87 instruction set? Explain your results. *Hint*: you can output the assembly using the `-S` flag using `GCC`.

   (e) Compile using `-O0`, i.e no optimizations.
   Solution: Without optimizations the compiler translates the ternary operator as a if-then-else operation. Therefore, it executes the compares as they are written in code.

   (f) Compile using `-O3`.
   Solution: With optimization, GCC compiles the macro by using the `maxsd` assembly instruction from SSE.

   Note:

   - the compiler will remove the computation if you do not use its result,
   - if you use values that are known at compile time, i.e if you hardcode the computation by writing something along the lines of `MAX(5.8, 7.8)`, the compiler will simplify the instruction.

   For one core and **without** using SIMD vector instructions, determine the following about your machine. In (g)-(j), make sure to use the correct floating-point instruction. Be careful not to choose one from the x87 instruction set in case you have an x86 processor. 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. Possibly, one addition can be executed on port 5, meaning the theoretical peak performance is 5 flops/cycle.

(h) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision floating-point multiplication.

Solution: Latency: 4 cycles. Throughput: 2 per cycle. Instruction: MULSS(D).

(i) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision square root operation.

Solution:
According to uops.info measurements:
For single precision (`SQRTSS`) Latency (alder-lake): 12-19 cycles. Throughput: 0.33 per cycle.
For double precision (`SQRTSD`) Latency (alder-lake): 13-19 cycles. Throughput: 0.22 per cycle.

(j) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision division operation.

Solution:
According to uops.info measurements:
For single precision (`DIVSS`) Latency (alder-lake): 11-12 cycles. Throughput: 0.33 per cycle.
For double precision (`DIVSD`) Latency (alder-lake): 13-15 cycles. Throughput: 0.25 per cycle.

2. (20 pts) Symmetric Matrix Multiplication

In this exercise, we provide a folder for computing $C = \alpha AB^T$, with A being a $n \times n$ symmetric matrix and B being a $n \times n$ matrix. Note, since A is symmetric, we only need to store half of it. The content of the folder include

- A C file `symm.c` that includes the actual computation,
- A header file `tsc_x86.h` that allows reading the time stamp counter (TSC) on x86 machines,
- A header file `arm_vct.h` that allows reading the VCT registers[1] on ARM machines,
- A header file `kperf.h` that allows reading the Processor Monitoring Unit (PMU) on ARM machines, which can contain various metrics (cycle count, instructions issued, ...). Requires `sudo` access. More information on VCT and PMU can be found here.

The code uses different timers available to time the matrix multiplication. 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 routine. Ensure you get consistent timings between timers and for at least two consecutive executions. Don't forget to disable Turbo Boost (if you can). (No need to answer anything here)

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

i. Determine the exact number of floating-point additions and multiplications that it performs.
Solution: The code performs $2n^3+n^2$ flops. In particular, it executes n^3+n^2 multiplications, and n^3 additions.

ii. Determine an upper bound on its operational intensity.
Solution: As stated above, $W(n) = 2n^3 + n^2$. The data loaded is: two $n \times n$ matrices of doubles (C and B), and a $n \times n$ symmetric matrix A. To store half of the matrix we need $n(n+1)/2$ doubles. Therefore $Q(n) = 8(2n^2 + n(n+1)/2)$. Thus, $I = \frac{W(n)}{Q(n)} = \frac{2n^3+n^2}{8(2n^2+n(n+1)/2)} \leq \frac{2n^3}{20n^2} = \frac{n}{10}$. In case you don't consider the matrix C that is only read, $I \leq \frac{n}{6}$

(c) For all square matrices of sizes n between 100 and 1500, 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`.

[1] The ARM documentation does not state what VCT stands for, we think it should be virtual cycle timer. For more information read here: https://developer.arm.com/documentation/102379/0101/The-processor-timers.

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` instead, where `x` is the number of your cpu.

Note: it is good practice to set long benchmarks such that they run at night; keep this in mind for your project. You can automate with a simple bash script or Makefile.

Solution:

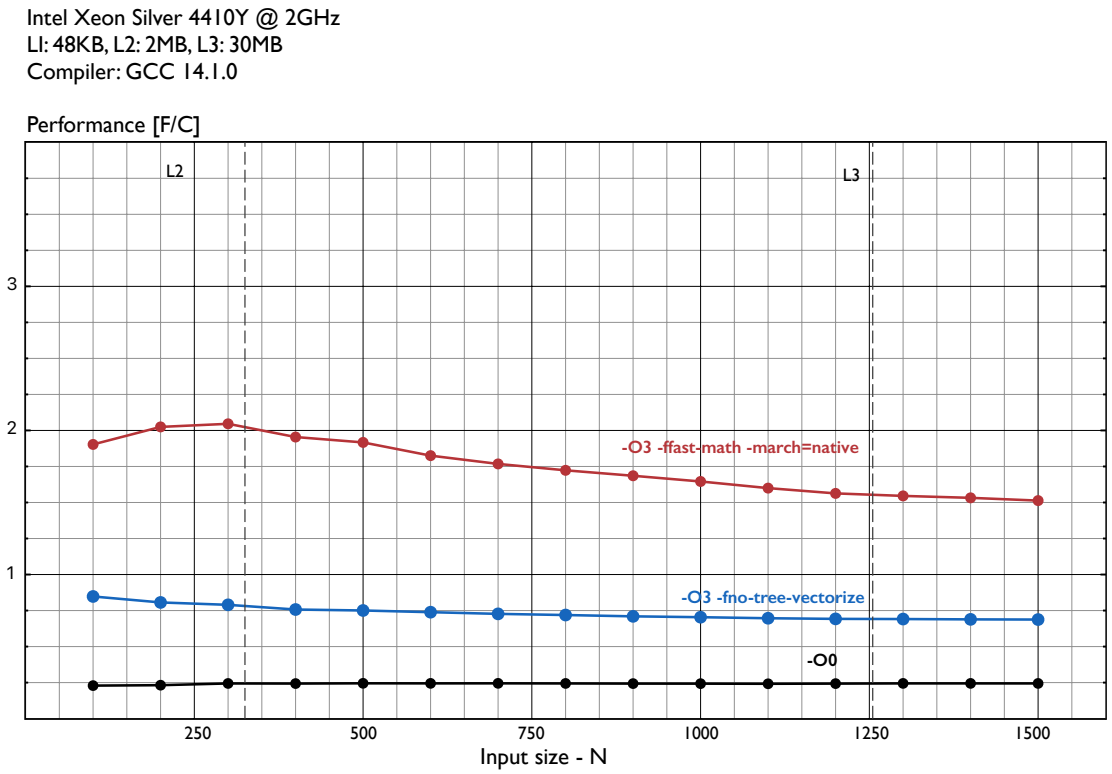

Figure 1: Plots resulting from execution of `symm.c`. For the given flags, scalar peak performance is 5 f/c, vector peak performance is 24 f/c with AVX2.

(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 starts to degrade steadily as soon as the matrices no longer fits in the cache. The highest performance that we achieve is 2.04 flops/cycle.

3. (20 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:

$$z_i = z_i + ceil(u_i/w_i) \cdot x_i + x_i \cdot y_i.$$

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) and that the ceil function is translated to a `roundsd` instruction by the compiler. Answer the following and justify your answers.

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

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

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

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

(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 each of the following cases:

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 L1-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 $2n$ floating-point additions and $2n$ floating-point multiplications, n floating-point ceil operations and n floating point division. The division has a throughput of 1/4, meaning the execution unit (not the port) remains occupied for 4 cycles. All other instructions can be executed on the remaining execution units. Thus, the lower bound is $4n$.

ii. The situation remains unchanged. While some instructions can be fused into FMAs, the division remains the bottleneck, leading to the same lower bound of $4n$.

iii. Abstracted Microarchitecture shows peak bandwidth of L1, and an estimate for the RAM throughput. In the computation, at least $5n$ doubles have to be read in total. Thus, $r_{L1} \geq \frac{5n}{8}$ 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 $I(N) \leq \frac{6n flops}{8(5n) bytes} = \frac{6}{40}$ flops/byte.

4. (25 pts) Basic optimization

   Consider the following function that computes the square euclidian norm $\|x - y\|_2^2$, where x, y are vectors of doubles of length n.

```
1  void comp(double *x, double *y, int n) {
2      double s = 0.0;
3      for (int i = 0; i < n; i++) {
4          double m = x[i] - y[i];
5          s += m * m;
6      }
7      x[0] = s;
8  }
```

   (a) Create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 2 for you and for all two-power sizes $n = 2^6, \ldots, 2^{23}$ 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`[2] `-fno-tree-vectorize`. If you are using clang, add also the `-fno-slp-vectorize` and `-ffp-contract=fast` flags.

   (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 an inter loop dependency when accumulating the values in `s`. On Skylake, the latency of the addition is 4 cycles. Thus, $T(n) \geq 4n$. Since $W(n) = 3n$, the performance is upper bounded by $\pi(n) \leq 0.75$ flops/cycle. On some modern CPUs (see plot), additions and subtractions are executed on a specialized FastADD execution unit. The latency of the addition is instead 2-3 cycles. Thus, $T(n) \geq 2n$. Since $W(n) = 3n$, the performance is upper bounded by $\pi(n) \leq 1.5$ 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:

[2] For Apple M1/M2 processors, the flag `-mfma` may not be supported. If this is the case, use instead `-mcpu=apple-m1` or `-march=native`.

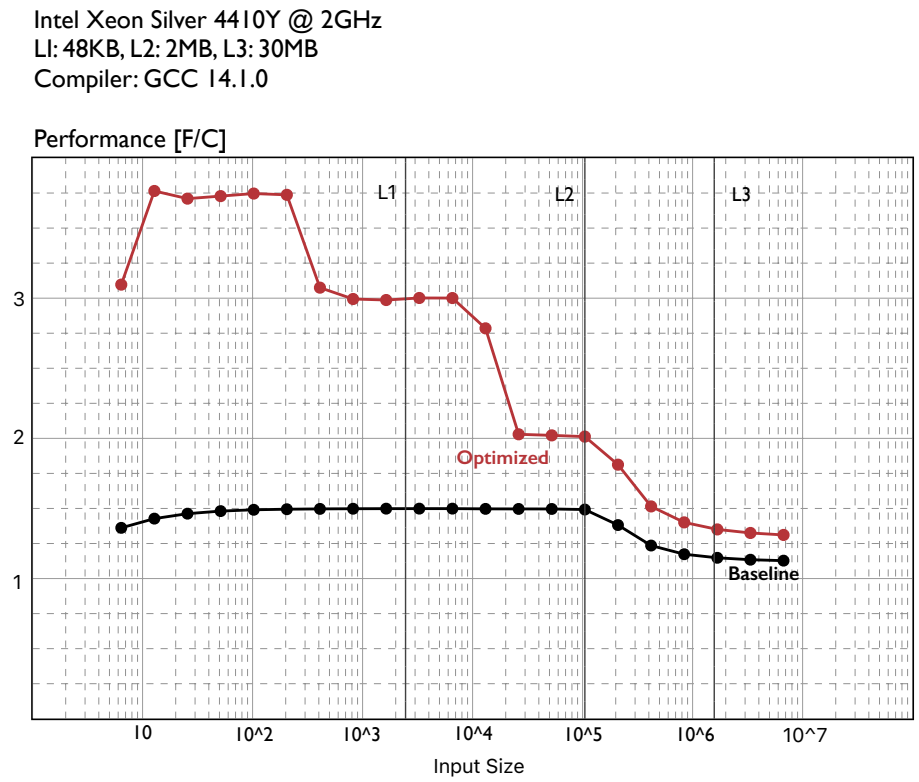

Figure 2: Performance plot (peak performance: 4 f/c for the given flags).

In the original code, the performance suffers from inter loop dependency which limits the amount of ILP. Thus, the performance is 1.5 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 3.74 flops/cycle.

5. (15 pts) ILP analysis

   Consider the following computations:

```
1 double comp(double a, double b, double c, double d) {
2     double t0 = a * b;
3     double t1 = b * c;
4     double t2 = c * d;
5     double t3 = t0 + t1;
6     double t4 = t3 + t2;
7     double t5 = t2 / a;
8     return t4 + t5
9 }
```

   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).

   (a) State the latency, throughput and port usage of the double-precision floating-point addition, multiplication and division instructions.

Solution: From uops.info:
Addition: p01, lat: 4, throughput: 0.5
Multiplication: p01, lat: 4, throughput: 0.5
Division: p0, lat: 13-15, throughput: 4

Determine hard lower bounds (not asymptotic) on the runtime (measured in cycles) for the following cases. Base your analysis on the *latency*, *throughput*, and *dependencies* of the floating-point operations. 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.

(b) Assume that the operations are issued in the order they are written, i.e., `t1` is computed after (or simultaneously with) `t0`, `t2` after (or simultaneously with) `t1` and `t0`, and so on. Determine a hard lower bound on the runtime for `comp`.

Solution: At least 26 cycles, If we follow the order of instruction in the code, the div is scheduled just before the last instruction.

(c) Now assume that operations can be issued out-of-order, meaning that `t1 = b * c` can be issued before `t0 = a * b`. However, data dependencies must still be respected. Determine a hard lower bound on the runtime for `comp`.

Solution: At least 22 cycles, The div can be issued after four cycles, after t2 is computed.

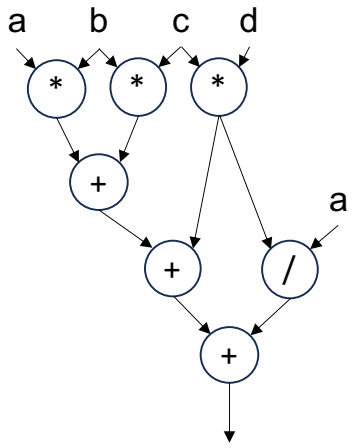

Figure 3: Dependency graph for `comp`. It is the same for both cases.