263-0007-00: Advanced Systems Lab
Assignment 1: 100 points
Due Date: Th, March 5th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2026/
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 Intel's development model: Tick, Tock or Optimization. (if applicable)
   Solution: Opt phase (Golden Cove).

   Intel's processors offer two assembly instructions to compute scalar floating-point multiplication in single precision, namely FMUL (from x87) and MULSS (from SSE).

   (e) Which instruction is used when you compile code in your computer and why is the other one still supported?
   Solution: On our machine, MULSS is generated. FMUL is kept for backwards compatibility.

   (f) What is x87 and why is it called that way?
   Solution: It is a floating-point extension of x86. The name originates from the 8087 co-processor used for math operations. Later, Intel integrated it into its instruciton set with the name x87.

   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 subtraction.
   Solution: According to uops.info measurements:
   For single precision (`SUBSS`) Latency (alder-lake): 2 cycles. Throughput: 2 per cycle.
   For double precision (`DIVSD`) Latency (alder-lake): 2 cycles. Throughput: 2 per cycle.

   (i) 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.

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

Solution:
According to uops.info measurements:
For single precision (`MINSS`) Latency (alder-lake): 4 cycles. Throughput: 2 per cycle.
For double precision (`DIVSD`) Latency (alder-lake): 4 cycles. Throughput: 2 per cycle.

2. (20 pts) Matrix-Vector Multiplication

In this exercise, we provide a folder for computing $y = Ax$, with A being a $n \times n$ matrix and x being a vector of size n. The content of the folder include

- A C file `mvm.c` that includes the 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-vector multiplication. There are comments explaining every aspect of the code. 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 frequency scaling (if you can). (No need to answer anything here)

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

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

ii. Determine a hard lower bound on the data movement (in bytes) between main memory and the CPU.
Note: due to the write-allocate cache policy (this will be covered in detail in a future lecture), arrays that are only written are also read, and this read needs to be considered in the data movement.
Solution: We read a matrix and two vectors. Therefore $Q(n) = 8 \cdot (n^2 + 2n)$

(c) For all $n = 200, \ldots, 4000$, with steps 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` instead, where `x` is the number of your cpu.

Solution:

[1] For more information read here: https://developer.arm.com/documentation/102379/0101/The-processor-timers.

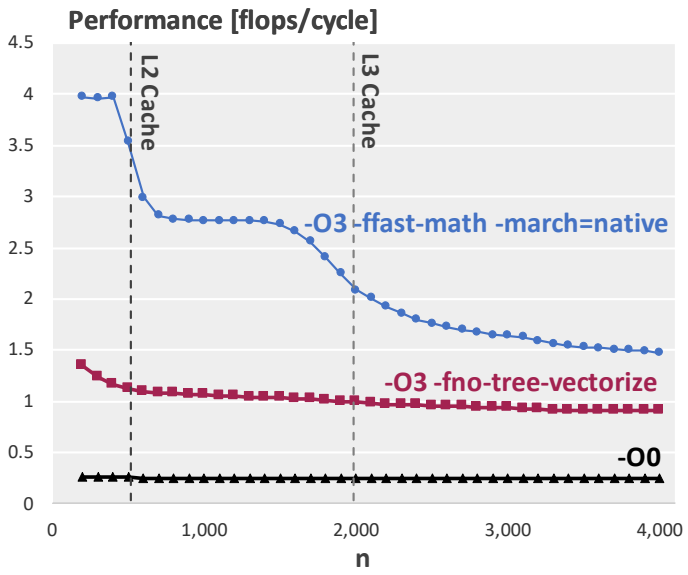

Figure 1: Plots resulting from execution of `mvm.cpp`. For the given flags, scalar peak performance is 5 f/c, vector peak performance is 20 f/c with AVX2.

(d) Discuss performance variations of your plots and report the highest performance that you achieved.

Solution:

i. Non-optimized (`-O0`): 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 (`-O3 -fno-tree-vectorize`): 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 (`-O3 -ffast-math -march=native`): 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. In particular, we have two significant drops in performance at the L2 and L3 cache limits. The highest performance that we achieve is 3.97 flops/cycle.

3. (20 pts) Performance analysis and bounds

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

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

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. There is no algebraic manipulation, the computation is executed as it is written). 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:

$$N_{add} = 2n,$$
$$N_{mul} = 5n,$$
$$C(n) = C_{add} \cdot (2n) + C_{mul} \cdot (5n).$$

(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: due to the write-allocate cache policy (this will be covered in more detail in a future lecture), arrays that are only written are also read, and this read needs to be counted.

Solution:

i. The instruction mix in this case consists of $2n$ floating-point additions and $5n$ floating-point multiplications. All these instructions share the same ports. The bound on the runtime is thus $7/2n$ cycles.

ii. Since `-ffast-math` is disabled, we can only fuse one addition and one mulitplication into an FMA, giving a new instruction mix of n FMAs and $4n$ multiplications and n additions. These instructions can be balanced on the two ports giving a bound of $3n$ cycles. We considered also a bound of $5/2n$ obtained by using two FMAs as correct.

iii. Abstracted Microarchitecture shows peak bandwidth of L1, and an estimate for the RAM throughput. In the computation, at least $3n$ doubles have to be read in total. Thus, $r_{L1} \geq \frac{3n}{8}$ and $r_{RAM} \geq \frac{3n}{2}$.

(d) Determine a lower bound on the data movement. Assume empty caches and consider only reads. Again, arrays that are only written are also read, see point (iii.) above.

Solution: We need to load three arrays of doubles. Therefore data movement is $Q(n) \geq 8(3n) bytes$.

4. (25 pts) Basic optimization

Consider the following function that computes a combination of three vectors x, y, z of doubles of length n.

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

(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 -march=native` and `-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:
Since the `y[i] * y[i]` multiplication is independent, it is possible for the CPU to execute the next iteration (i.e., `y[i+1] * y[i+1]`) speculatively. Then we again have the FMA and addition in the main loop. If we could reorder operations, the `y[i] * y[i]` multiplication and the `s + x[i]` addition can be done in parallel. Then a final FMA can be executed to accumulate in `s`. Therefore the bottleneck is made up of one addition/multiplication and an FMA. On Skylake, the latency of addition, multiplication and FMA is 4 cycles. Thus, $T(n) \geq 8n$. Since $W(n) = 4n$, the performance is upper bounded by $\pi(n) \leq 0.5$ 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, $T(n) \geq 6n$. Since $W(n) = 4n$, the performance is upper bounded by $\pi(n) \leq 0.66$ 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: In the original code, the performance suffers from inter loop dependency, which limits

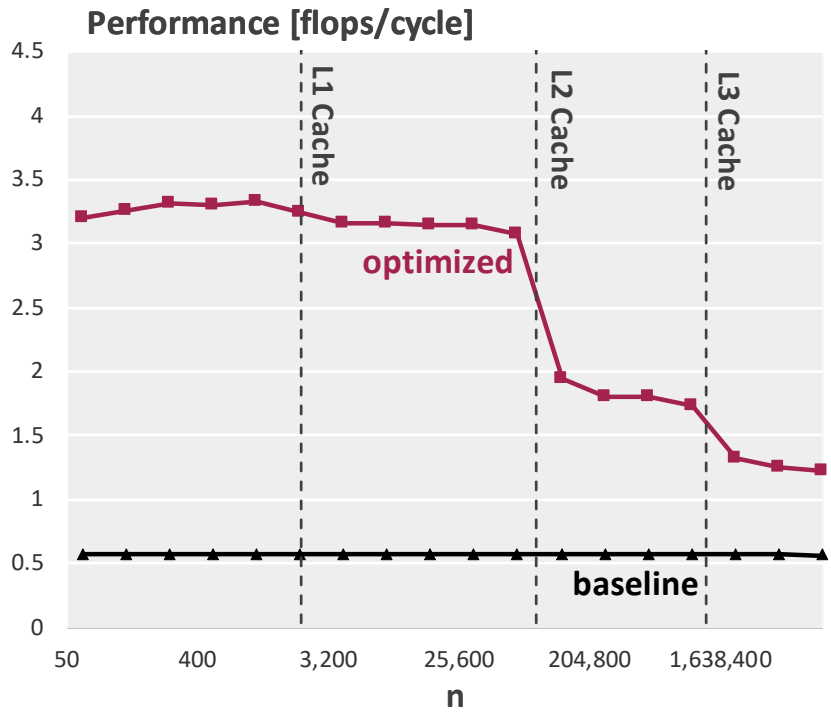

Figure 2: Performance plot (peak performance: 5 f/c for the given flags).

the amount of ILP. Thus, the performance is 0.57 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 3.26 flops/cycle.

5. (15 pts) ILP analysis

   Consider the following computations:

```
1 double comp(double a, double b, double c) {
2     return (a * a + b + b * b + c * c) * (a * b + c);
3 }
```

   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). 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.

   (a) Determine a hard lower bound on the runtime for `comp` if FMAs are not generated.

   Solution: Without -ffast-math, the operations happen from left to right. We give the compiled version of the `comp` function in Listing 1 in Figure 3 (left). Additionally we provide a dependency DAG in Figure 4 where we highlight the critical path, and a possible scheduling of the ASM code in Figure 9. The whole operation takes at least **20 cycles** to finish.

Listing 1: Compiled without -ffast-math

```
comp(double, double, double):
        vmulsd  xmm3, xmm0, xmm0
        vmulsd  xmm4, xmm1, xmm1
        vmulsd  xmm0, xmm0, xmm1
        vaddsd  xmm3, xmm3, xmm1
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm3, xmm3, xmm4
        vmulsd  xmm4, xmm2, xmm2
        vaddsd  xmm3, xmm3, xmm4
        vmulsd  xmm0, xmm3, xmm0
        ret
```

Listing 2: Compiled with -ffast-math

```
comp(double, double, double):
        vmulsd  xmm4, xmm0, xmm0
        vmulsd  xmm3, xmm1, xmm1
        vmulsd  xmm0, xmm0, xmm1
        vaddsd  xmm3, xmm3, xmm4
        vmulsd  xmm4, xmm2, xmm2
        vaddsd  xmm0, xmm0, xmm2
        vaddsd  xmm4, xmm4, xmm1
        vaddsd  xmm3, xmm3, xmm4
        vmulsd  xmm0, xmm3, xmm0
        ret
```

Figure 3: ASM of the code in comp compiled with gcc 15.2 with flags `-march=skylake -O3 -mno-fma`. The left code is compiled with `-fno-fast-math` while the right code is compiled with `-ffast-math`. The initial values of the `xmm0, xmm1, and xmm2` registers are a, b, and c respectively.

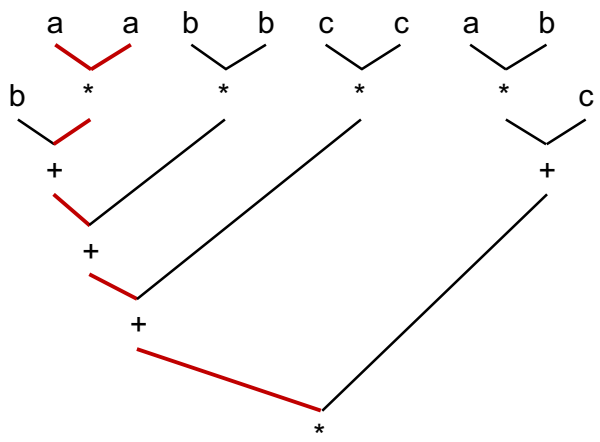

Figure 4: Dependency DAG without FMAs. The critical path is in red. Each opeartion has a latency of 4 cycles, therefore the total latency is 20 cycles.

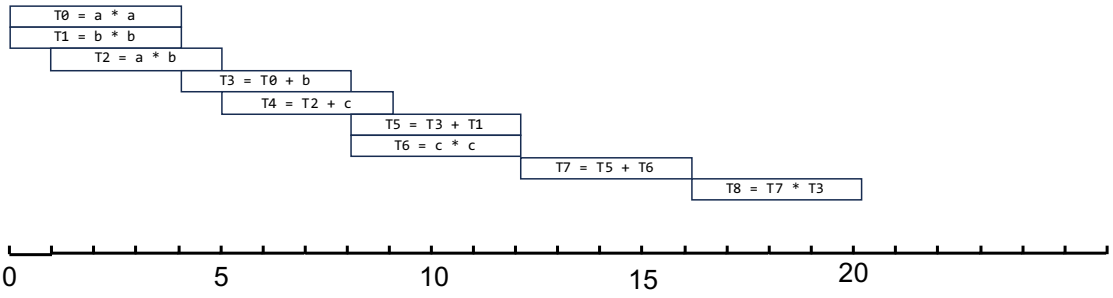

Figure 5: Possible scheduling of the instructions when no FMAs are generated. We obtain again 20 cycles.

(b) Determine a hard lower bound on the runtime for `comp` if FMAs are generated.

Solution: Again, without -ffast-math, the operations happen from left to right, and we give the compiled version of the `comp` function in Listing 3 in Figure 8 (left). Additionally we provide a dependency DAG in Figure 6 where we highlight the critical path, and a possible scheduling of the ASM code in Figure 10. The whole operation takes at least **16 cycles** to finish.

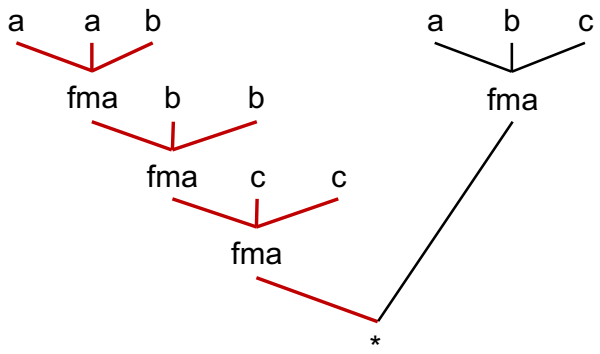

Figure 6: Dependency DAG with FMAs. The critical path is in red. Each opeartion has a latency of 4 cycles, therefore the total latency is 16 cycles.

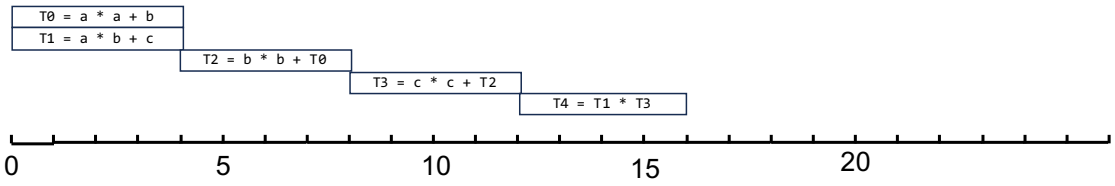

Figure 7: Possible scheduling of the instructions when FMAs are generated. We obtain again 16 cycles.

What happens with `-ffast-math`? In this exercise, we asked explicitly for the case **without `-ffast-math`**, so if you answered the above, you receive full marks. Here, we explore what happens if instead we consider the case with `-ffast-math enabled`. If `-ffast-math` is enabled, the compiler can reorder the operations. We give the actual compiled assembly in Listings 2 and 4 in Figure 3 and 8 (right snippet). In both cases, we see that the compiler moved some operations around. If we do a basic scheduling of the compiled code, we get the results in Figures 9 and 10 for the no FMA and FMA cases, respectively. In both cases, the bound on the cycles remains unchanged.

However, for the no FMA code, we can find a scheduling that achieves a lower bound of **17 cycles**. Such scheduling is given in Figure 11. The difference is that here we use two different accumulators to compute the result of the left parenthesis.

Listing 3: Compiled without -ffast-math

```
comp(double, double, double):
        vmovapd xmm3, xmm0
        ; a is copied in the accumulator (xmm3)

        vfmadd132sd     xmm3, xmm1, xmm0
        vfmadd132sd     xmm0, xmm2, xmm1
        vfmadd231sd     xmm3, xmm1, xmm1
        vfmadd231sd     xmm3, xmm2, xmm2
        vmulsd  xmm0, xmm0, xmm3
        ret
```

Listing 4: Compiled with -ffast-math

```
comp(double, double, double):
        vmulsd  xmm4, xmm1, xmm1
        vmovapd xmm3, xmm2
        ; c is copied in the accumulator (xmm3)
        vfmadd132sd     xmm3, xmm1, xmm2
        vfmadd231sd     xmm4, xmm0, xmm0
        vfmadd132sd     xmm0, xmm2, xmm1
        vaddsd  xmm3, xmm3, xmm4
        vmulsd  xmm0, xmm0, xmm3
        ret
```

Figure 8: ASM of the code in comp compiled with gcc 15.2 with flags `-march=skylake -O3 -mfma`. The left code is compiled with `-fno-fast-math` while the right code is compiled with `-ffast-math`. The initial values of the `xmm0, xmm1, and xmm2` registers are a, b, and c respectively. Lookup the differences between `vfmadd231sd` and `vfmadd132sd`.

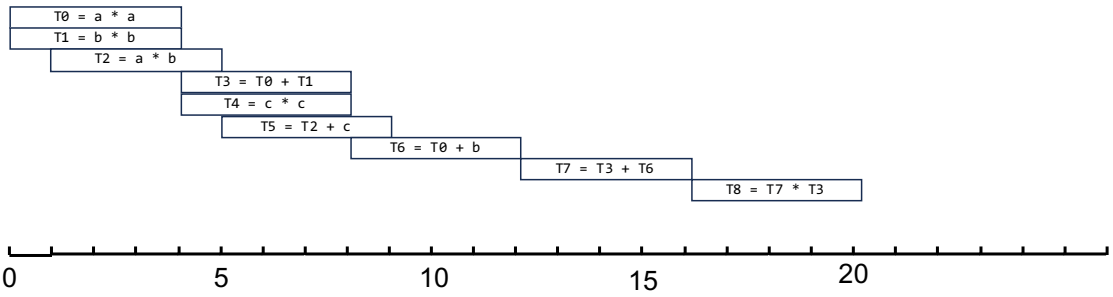

Figure 9: Possible scheduling of the instructions when no FMAs are generated and `-ffast-math` is enabled. Again we get 20 cycles

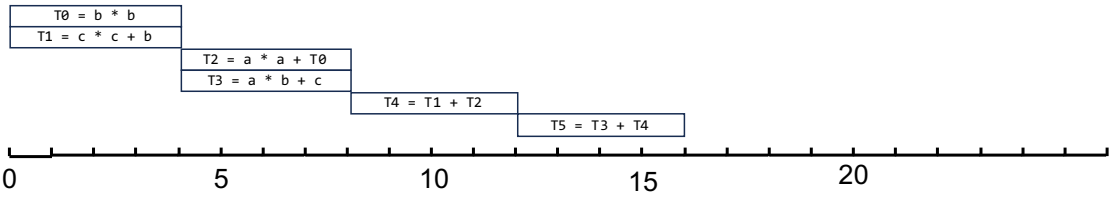

Figure 10: Possible scheduling of the instructions when FMAs are generated and `-ffast-math` is enabled. Again we get 16 cycles

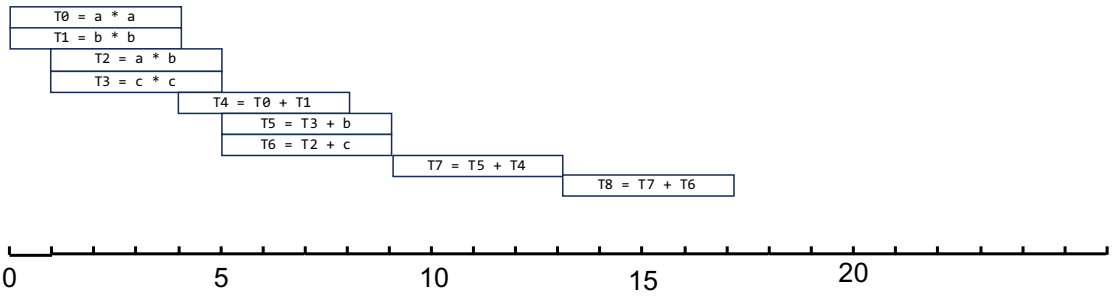

Figure 11: Optimal scheduling of the instructions when no FMAs generated. Now we get 17 cycles.