Student Name: Marios G. Angelis
Student ID: 22-953-186
Spring Semester 2023

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab

Homework 1

---

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(R) Core(TM) i7-8750H (Coffee Lake)

   (b) CPU base frequency.
   Solution: 2.20 GHz is the nominal CPU frequency.

   (c) CPU maximum frequency. Does your CPU support Turbo Boost or a similar technology?
   Solution: The processor supports Turbo Boost, and the maximum frequency is 4.10 GHz.

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

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

   (e) Name three differences between x87 and SSE2.
   Solution:
   • x87 supports 80-bits calculations providing higher precision that SSE2.
   • x87 FPU is a scalar unit whereas SSE2 can process a small vector of operands in parallel.
   • SSE2 uses 16 wide XMM floating point registers, whereas x87 uses 8 80-bit registers.

   For one core and without using SIMD vector instructions, determine the following about your machine. In (g)-(h), 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.

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

   (g) 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/MULSD.

   (h) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision square root.
   Solution:
   For SQRTSS instruction:

- According to uops.info, latency is 12-13 cycles and throughput is 0.33 per cycle.
- According to Agner tables, latency is 12 cycles and throughput is 0.33 per cycle.

For SQRTSD instruction:
- According to uops.info, latency is 13-19 cycles and throughput is 0.22 per cycle.
- According to Agner tables, latency is 15-16 cycles and throughput is 0.16-0.25 per cycle.

(i) Latency [cycles], throughput [ops/cycle] and instruction name for double-precision ceiling operation, i.e., the operation that rounds a floating-point number up to an integer-valued floating-point.
Solution: Latency: 8 cycles. Throughput: 1 per cycle. Instruction: ROUNDSD

2. (20 pts) Matrix multiplication
In this exercise, we provide a C source file for multiplying an n × n matrix with its transpose 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 matrix multiplication. Note that if you have an Apple M1/M2 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 mmm.c and answer the following:
      i. Determine the exact number of floating-point additions and multiplications that it performs.
      Solution: The code performs $2n^3$ floating-point operations. We have 3 nested loops and 2 flops per iteration.

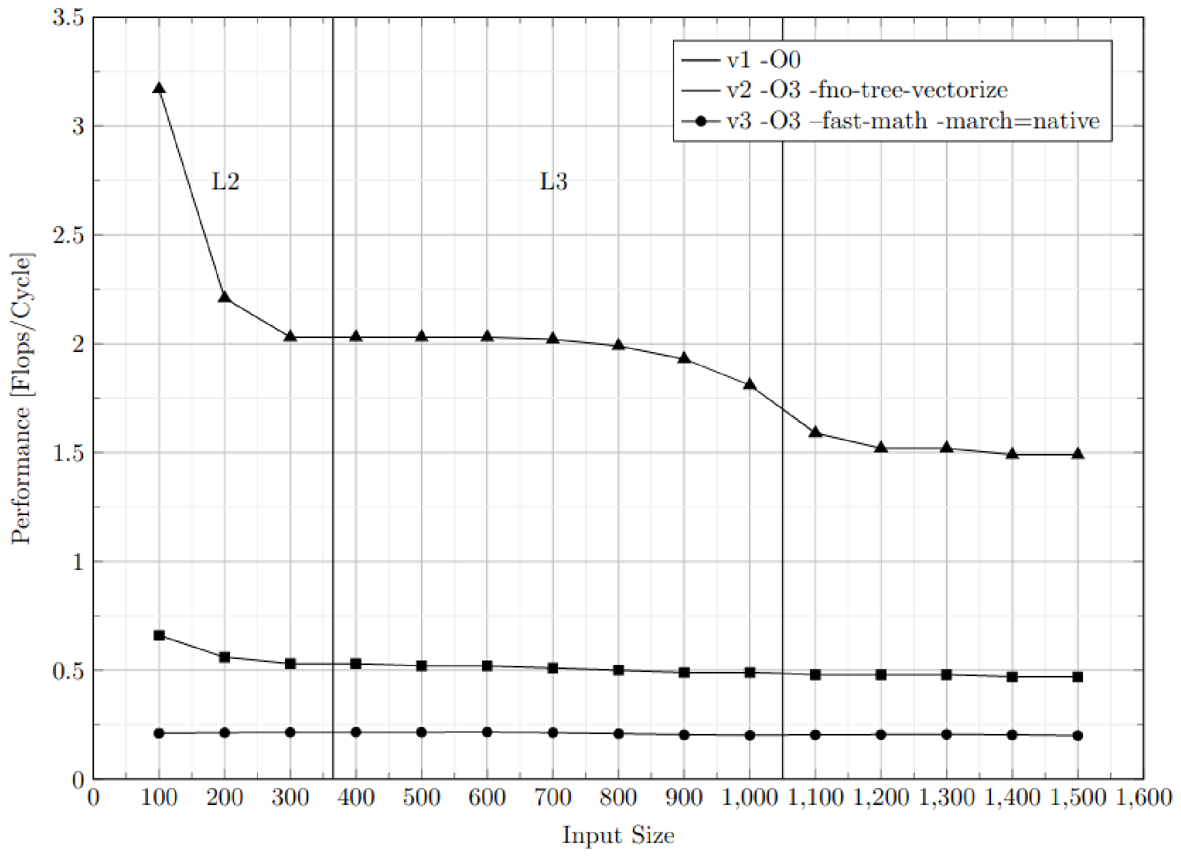
      ii. Determine an upper bound on its operational intensity (consider only reads and assume empty caches).
      Solution:
      $W(n) = 2n^3$ and $Q(n) \geq 2 \cdot 8 \cdot n^2$ because only A and B tables should be loaded from memory. Thus, $I(n) \leq \frac{n}{8}$ flops/bytes.

   (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.
      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 M1 processor and your compiler doesn't support -march=native you can use -mcpu=apple-m1 instead.

Solution:

CPU specifications:

- Intel(R) Core(TM) i7-8750H, 2.20 GHz
- Cache Sizes: L1: 32 KB, L2: 1.5 MB, L3: 9 MB
- Compiler: GCC 11.3.0



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

i. (v1 -O0): Since the first series has all optimizations disabled, we will get the worst performance which is around 0.2 flops/cycle. By disabling compiler optimization flags, no optimization and/or vectorization techniques will be used. Also, the performance is flat.

ii. (v1 -O3 -fno-tree-vectorize): Using these compilation flags, we enable compiler optimizations but not vectorization techniques. Thus, the performance will be slightly better than the previous one but not much faster. Again, we the performance is relatively flat.

iii. (v1 -O3 -fast-math -march=native): Now, we enable vectorization techniques as well as Instruction Level Parallelism (ILP) allowing the compiler to modify the order of instructions to get higher performance. This is done using the flag -ffast-math. Thus, the computation performs well for a small number of matrix sizes but it degrades significantly when the matrix can not fit into the L3 cache. This is done when $n > 1000$ since the L3 cache is 9MB long. The highest performance achieved is 3.17 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:

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

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 max function is translated to a maxsd instruction by the compiler. Answer the following and justify your answers.

(a) Define a suitable detailed floating-point cost measure C(n).
Solution:
$$C(n) = C_{add} \cdot N_{add} + C_{mul} \cdot N_{mul} + C_{max} \cdot N_{max}$$

(b) Compute the cost C(n) of the computation.
Solution:
$$C(n) = C_{add} \cdot (3n) + C_{mul} \cdot (3n) + C_{max} \cdot (n)$$

(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:

i. The instruction mix consists of 3n multiplications, n additions, 2n subtractions and n max operations. According to uops as well as Agner tables, all operations (mults,adds,subs,max) can be scheduled in ports 0 and 1 in parallel. Thus, we have 3n additions/subtractions (3n/2 cycles to execute), 3n multiplications (3n/2 cycles to execute) and n max operations (n/2 cycles to execute). A lower bound on the runtime will be 3.5n cycles.
ii. According to Skylake microarchitecture, 2 FMAs can be executed in parallel. However, only additions and multiplications are executed from the FMA. Thus, we have n FMA instructions (n/2 cycles to execute), 2n subtractions (n cycles to execute), 2n multiplications (n cycles to execute) and n max operations (n/2 cycles to execute). A lower bound on the runtime will be 3n cycles.
iii. According to Skylake microarchitecture, the peak bandwidth of L3 cache is 4 loads/cycle and the peak bandwidth of RAM is 2 loads/cycle. In the computation, at least 5n doubles should be loaded in total. Thus, we have $r_{L3} \geq \frac{5n}{4}$ 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:
Operational intensity $I(n) = \frac{W(n)}{Q(n)}$ where W(n) is the number of floating point operations executed and Q(n) is the number of bytes transferred between memory and cache. In this case, $I(n) \leq \frac{7n}{5n \cdot 8} = \frac{7}{40}$ flops/byte since we have 7n flops in total and at least 5n doubles should be loaded in total.

4. (25 pts) Basic optimization
Consider the following function:

```c
void comp(double *x, double *y, int n) {
    double s = 0.0;
    for (int i = 0; i < n; i++) {
        s = (s + x[i]*x[i]) + y[i]*y[i]*y[i];
    }
    x[0] = s;
}
```

(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 = 24, . . . , 223 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 -fno-tree-vectorize.

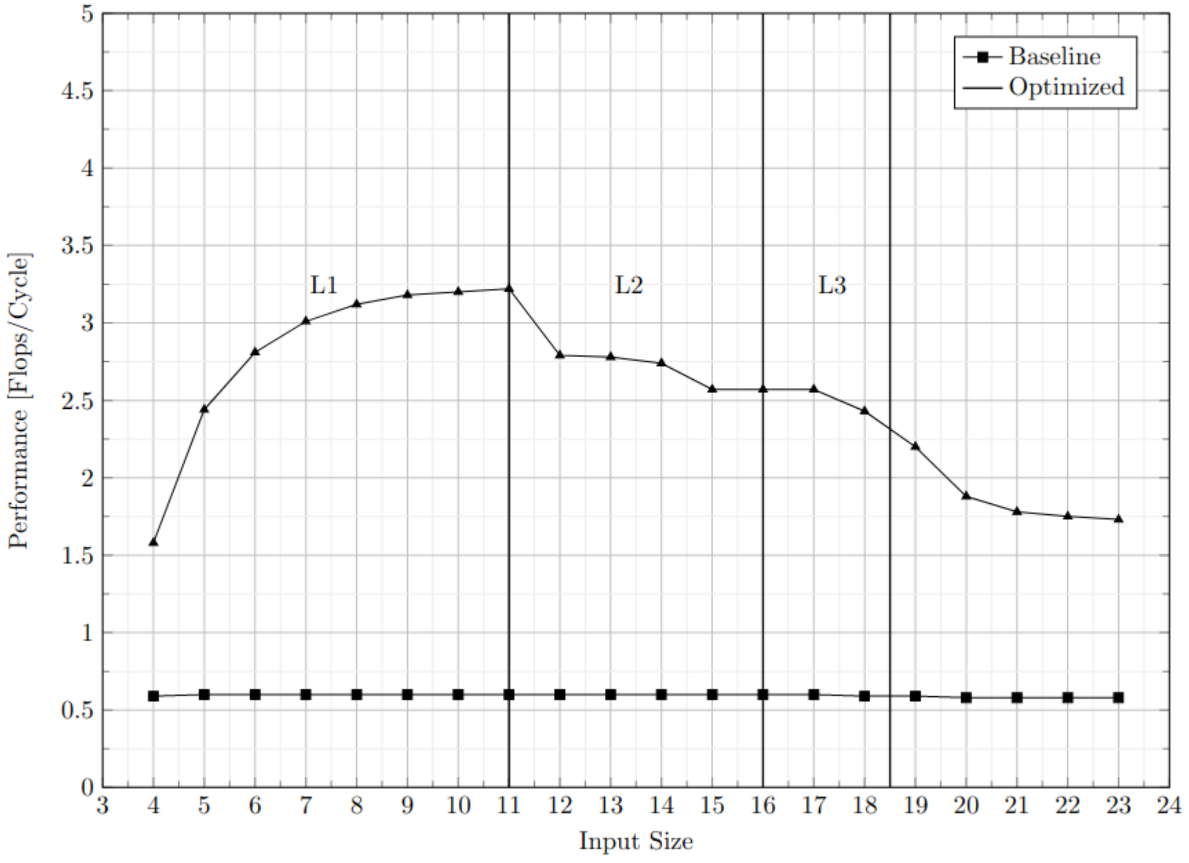
(b) Considering the latency and throughput information of floating-point operations in your machine, and the dependencies in comp, derive an upper bound on the performance (flops/cycles) of comp when using the specified flags in (a), i.e., when FMA instructions are enabled (-mfma) but vectorization is disabled (-fno-tree-vectorize).
Solution:
The latency of an FMA is 4 cycles in CoffeeLake. Thus, we need 8 cycles to perform 5 operations (1 FMA and 1 multiplication during the first 4 cycles and 1 FMA during the remaining 4 cycles). Thus, $T(n) \geq 8n$. Since $W(n) = 5n$, the performance is upper bounded by $\pi(n) \leq 0.625$ 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 flags -O3 -mfma -fno-tree-vectorize. If you are using clang, add also the -fno-slp-vectorize flag to disable vectorization.

(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 is upper bounded by $\pi(n) \leq 0.625$ flops/cycle. This is because the performance suffers from interloop dependency and as a result, the amount of Instruction Level Parallelism (ILP) is limited. In order to improve the performance, we apply Loop Unrolling with Separate Accumulators in order to explore the highest amount of Instruction Level Parallelism that can be achieved. We use 8 separate accumulators, each of which consists of 2 FMAs and 1 multiplication operation. More specifically, if we start processing at cycle 0, after 4 cycles, we will have the results of the first operations that can be an FMA and a multiplication running in parallel. Thus, during cycles 0 to 4, we can start 4 FMAs and 4 multiplications in total. We could have used 4 accumulators instead of 8, but we noticed that the processor remained underutilized after cycle 4. More specifically, at cycle 4, the result from the FMA and the multiplication we started at cycle 0 will be ready, so we can start the second FMA of the first accumulator. However, at this point, if we had only 4 accumulators, the other port will be not utilized. For this reason, we use 8 accumulators to expand the Instruction Level Parallelism and keep the processor busy at every cycle. The maximum performance achieved is 3.22 flops/cycle. Also, we can notice that the performance is significantly greater when the data fits in the L1 cache and becomes worse as data no longer fit in L2 and L3 cache respectively.

5. (10 pts) ILP analysis
Consider the following computations:

```c
#include <math.h>

double artcomp1(double a, double b, double c, double d) {
    double r;
    r = (a*a + b*b) - (c*c + d*d);
    return r;
}

double artcomp2(double a, double b, double c, double d) {
    double r;
    r = ceil(a)*b - ceil(c)*d;
    return r;
}
```

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. 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). 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 artcomp1.
Solution:
The runtime for artcomp1 function is at least 13 cycles. Since we have a Skylake processor, both the addition and the multiplication operations have a throughput of 2 cycles and a latency of 4 cycles. Thus, we can start a multiplication and an addition in parallel using ports 0 and 1, or 2 multiplications or 2 additions in the same way. More specifically, multiplications a*a and b*b can start in parallel at the first cycle and multiplications c*c and d*d can start in parallel at the second cycle. Then, we can start the addition a*a + b*b at cycle 5, when the results of the corresponding multiplications are ready, and the addition c*c + d*d at cycle 6, when the results of the corresponding multiplications are ready. Finally, we can start the subtraction at cycle 10 when the results from the additions are ready. The DAG is shown in Figure 1.

(b) Determine a hard lower bound on the runtime for artcomp2. Assume that the compiler transforms the ceil function to the respective assembly instruction performing this operation (so no function call to the math library occurs).
Solution:
The runtime for artcomp2 function is at least 17 cycles. Since we have a Skylake processor, both the addition and the multiplication operations have a throughput of 2 cycles and a latency of 4 cycles. Also, the ROUNDSD operation used for the ceiling has a latency of 8 and a throughput of 1. Thus, we can start ceil(a) operation at the first cycle and ceil(c) operation at the second cycle. Then, we can start the multiplication ceil(a)*b at cycle 9, when the results of the ceil(a) operation are ready, and the multiplication ceil(c)*d at cycle 10, when the results of the ceil(c) operation are ready. Finally, we can start the subtraction at cycle 14 when the results from the multiplications are ready. The DAG is shown in Figure 2.

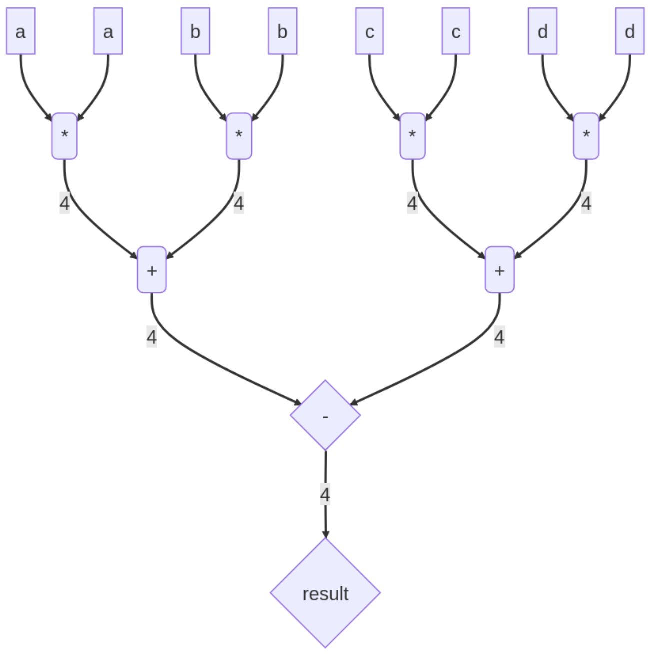


Figure 1: DAG for artcomp1

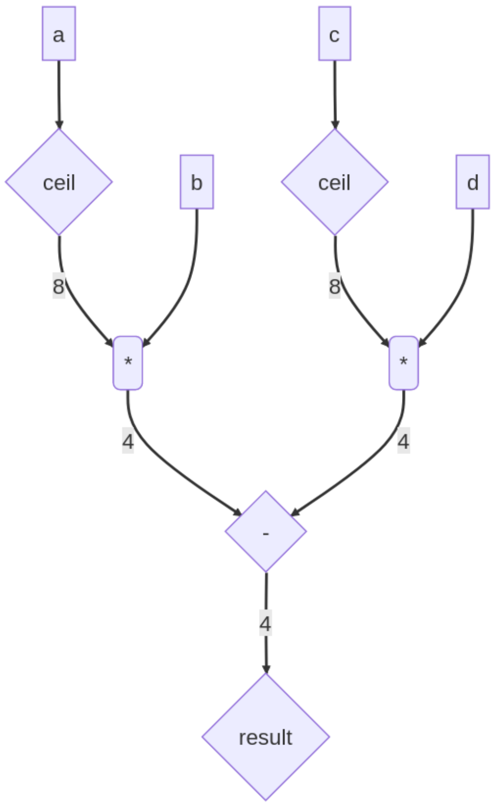


Figure 2: DAG for artcomp2