ASL HW1
Student Name: Berke Egeli
Legi. No.: 20 948 220

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:
   Processor Manufacturer: Intel
   Processor Code Name: Ice Lake
   Processor Number: i7-1068NG7

   (b) CPU base frequency.
   Solution: CPU base frequency: 2.30 GHz

   (c) CPU maximum frequency. Does your CPU support Turbo Boost or a similar technology?
   Solution: CPU maximum frequency: 4.10 GHZ / This model support Turbo Boost 2.0 technology

   (d) Phase in the Intel's development model: Tick, Tock or Optimization. (if applicable)
   Solution: Ice Lake is a processor developed in the architecture (Tock) phase.

   (e) Maximum theoretical floating point peak performance in flop/cycle.
   Solution: Without SIMD instructions 3 FMA operations (port 0, port 1 and port 5)(according to page page 39 in the Intel 64 and IA-32 Architectures Optimization Reference Manual which seems to contradict the architecture diagram provided in page 38, so, I blame Intel if my answer is wrong) can be issued per cycle, so, peak performance is 6 flop/cycle.

   (f) Latency [cycles] and throughput [ops/cycle] for fused multiply-add (FMA) operations (if supported).
   Solution: Latecy is 4 cycles and throughput is 1 [ops/cycle].

   (g) Latency [cycles] and throughput [ops/cycle] for floating point comparison operations (e.g. greater- than, less-than, equal-to, etc).
   Solution:
   Different comparison operators (equals/greater than etc. are computed using the same instruction)
   Comparison:

   Single Precision (`CMPSS`): Latency 4 cycles **Throughput** 2 [ops/cycle]
   Note: CMPSS is used to implement pseudo-instructions CMPEQSS (equality), CMPLTSS (less-than), CMPLESS(less-than or equal), CMPUNORDSS, CMPNEQSS, CMPNLTSS, CMPNLESS, CMPORDSS

   Double Precision (`CMPSD`): Latency 4 cycles **Throughput** 2 [ops/cycle]
   Note: CMPSS is used to implement pseudo-instructions CMPEQSD, CMPLTSD, CMPLESD, CMPUNORDSD, CMPNEQSD, CMPNLTSD, CMPNLESD, CMPORDSD

   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 is used in my computer.

(i) Why is the other one still supported?
Solution: The instruction is probably still supported for backwards compatibility.

2. (20 pts) Matrix-vector multiplication

In this exercise, we provide a C source file for multiplying an n × 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.

```
        sum = sum + A[n*i+j]*x[j];
```

Solution: There are 2 floating point operations the line above. Each inner loop iterates for n times. The outer loop iterates for n times as well. So, in total there are n^2 iterations with 2 floating point operations each. Thus, the total number of floating point operations in the compute function is $2n^2$.

(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.

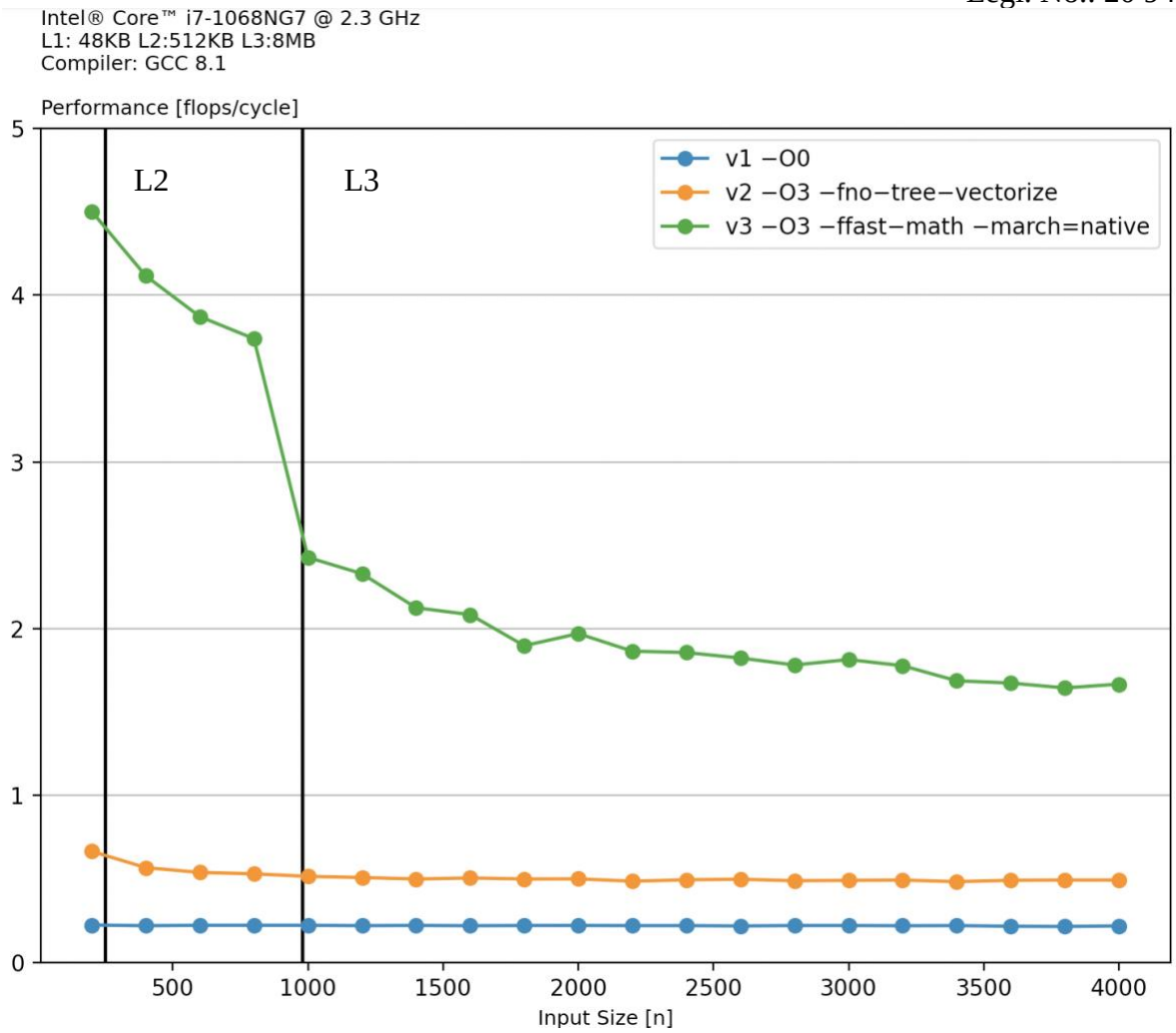

Figure 1: The plot resulting from executing mvm.c on Ice Lake CPU. GCC 8.1 was used to compile the code.

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

Solution:

i. Since no optimizations and vectorizations are used the performance is low.
ii. Because there are optimizations in this step the performance is better, however, accumulator's new value depends on its old value, so, there is a dependency within the loop and it limits the ILP, thus, the performance is still low.
iii. There is vectorization and optimizations in this step, so, performance is much higher than the other two steps. However, we can clearly see that performance drops significantly whenever the data doesn't fit into the caches, L2 and L3 in particular.

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: Since in this computation there are only multiplication and addition operations a suitable cost function would be:

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

Where C_{add} & C_{mul} are costs of an addtion and multiplication operations respectively and N_{add} & N_{mul} are the exact number of addition and multiplication operations respectively.

(b) Compute the cost C(n) of the computation.

Solution:

For vectors of size N,

$N_{add} = 2N$
$N_{mul} = 4N$

So, the total cost C(N) is $C_{add} \cdot 2N + C_{mul} \cdot 4N$

(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).

Solution: Addition operations can only be issued in port 1 of Haswell architecture whereas multiplication operations can be issued in both port 0 and port 1. So, 2N addition operations can be issued to port 1 back to back. While additions operations are issued, in port 0 2N multiplication operations can be issued to port 0. After all addition operations are issued, remaining multiplication operations can be issued to both port 0 and port 1, with N multiplication operations each. So, the **hard lower bound** for total operation is **3N cycles**.

(I am ignoring the dependencies between operands in these computations because it is asked to find the hard lower bound based on throughput only and unlike questions 5 it isn't explicitly stated that we should consider the dependencies. That said, it is very unlikely that all operations can be completed by the lower bound stated in the answer. **Addition operations require the outputs of the multiplications, so, multiplications must be completed first. This prevents parallel execution alongside addtions which is what the lower bound I calculated depends on.**)

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).

Solution: If we can use FMA instructions, then we can issue them to both port 0 and port 1, like multiplication instructions. In this case, in total there will be 2N FMA instructions and 2N multiplication instructions. So, the **hard lower bound will be 2N cycles**.

(I am ignoring the dependencies between operands in these computations because it is asked to find the hard lower bound based on throughput only and unlike questions 5 it isn't explicitly stated that we should consider the dependencies. That said, it is very unlikely that all operations can be completed by the lower bound stated in the answer. **One of the inputs of the FMA instruction is the output of the other fma instruction, so, technically they cannot be issued at the same time as implied by my answer.)**

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: Throughput of L1 cache is 8 doubles/cycle and main memory is 2 doubles/cycle. In both cases we read 4N doubles because all vectors are used to write the new value. Thus,

$$r_{L1} \geq \frac{1}{8}4N = \frac{1}{2}N \; cycles,$$

$$r_{MM} \geq \frac{1}{2}4N = 2N \; cycles$$

(c) 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:

$$I(N) \leq \frac{6N}{8(4N)} = \frac{3}{16} flops/byte$$

4. (25 pts) Scalar product
Consider the following function that computes the scalar product of vectors x and y of size n:

(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 run- time. 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.

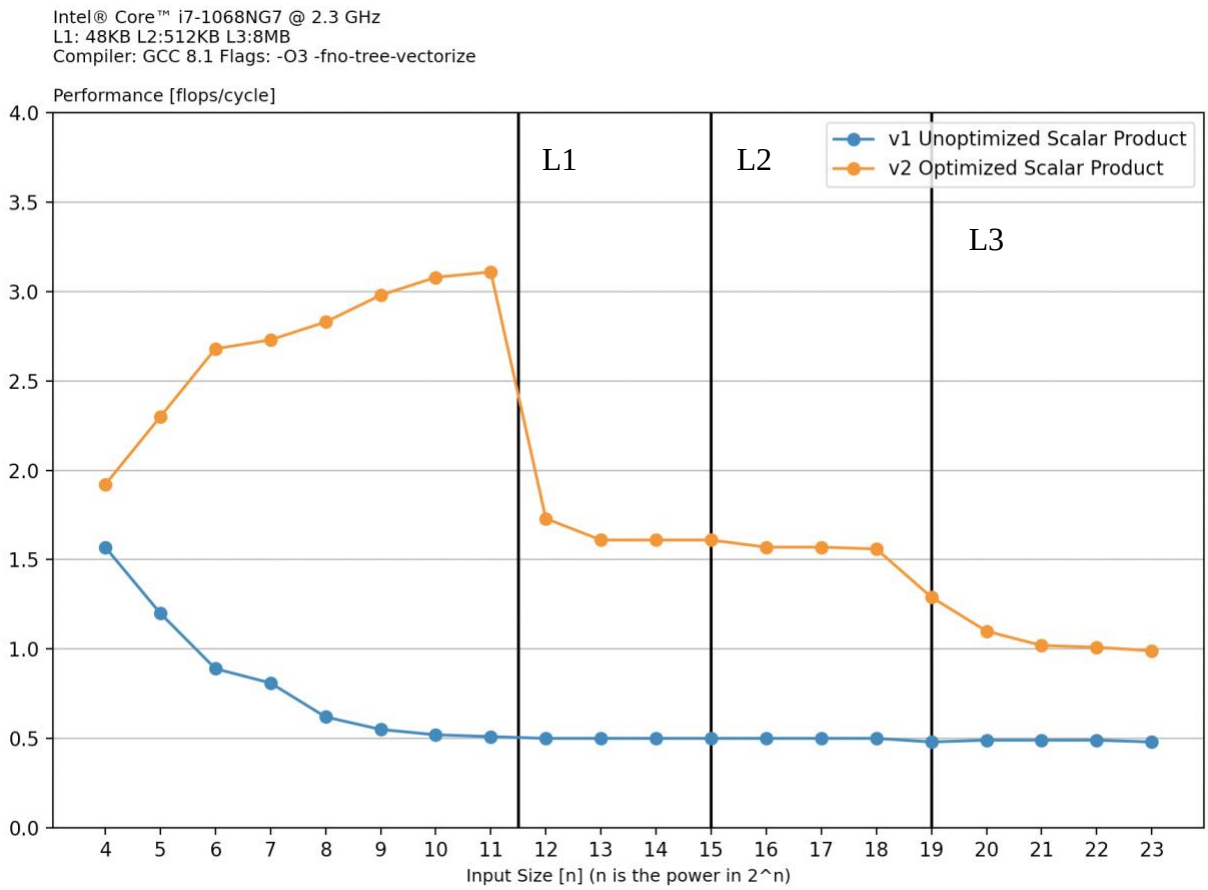

Figure 2: The plot resulting from executing scalar_product and optimized_scalar_product on a Ice Lake CPU. The code was compiled on GCC 8.1

Solution: For the unoptimized scalar product benchmark, we can see that the performance stays relatively the same even though the problem size increases. This is because of the accumulator's dependency to its old value, so, the execution cannot benefit from any ILP and its performance stays flat.

For the optimized method benchmark, the performance varies with the input size. Specifically, whenever data starts not fitting into the caches the performance decreases significantly. (I do not know why performance drop is not significant when data does not fit into L2 cache, but it is consistently lower than the previous input sizes.)

(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.

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

…

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: We have to find the critical path in the DAG of the computation. Because Haswell has only one addition port and we cannot use FMA instructions, (a + b) and (b + c) cannot be issued at the same time (because there is only one port that supports addition), even though there is no dependency between them. One of them has to wait one clock cycle for the other one to begin execution. Therefore, our critical path takes at least 12N cycles.

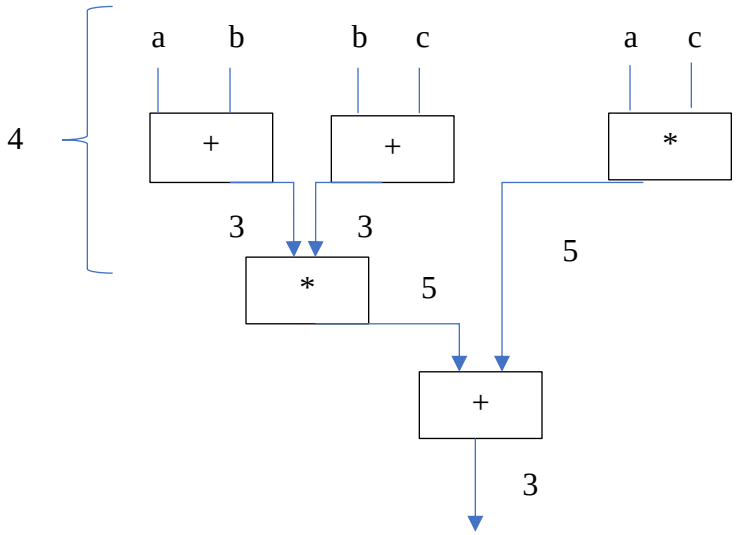

Figure 3: DAG of artcomp function provided in the homework sheet.