J. H. Walther
S. Martin
ETH Zentrum, CLT
CH-8092 Zürich

Fall semester 2022

Set 1 -Amdahl's Law, Roofline Model, Cache

Issued: September 28, 2022
Hand in (optional): October 12, 2022 08:00

Grading: For the exercise submission, you only need to submit 4 out of 5 questions, i.e. the ones not marked with "OPTIONAL".

Question 1: Amdahl's law (30 points)

a) Suppose you have a program where 99% of the runtime is parallelizable. Your boss gives you a computer which has a CPU with 64 cores.

- He wants you to achieve a speedup of 50. Is this possible? Justify.
- What is the maximum speed up you can achieve if you had no limitations of CPU cores n?

- Given the parallelizable fraction p, the speed-up is expressed as:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

with $p = 0.99$ and $n = 64$, we have a speed up of $S(64) = \frac{1}{0.01 + \frac{0.99}{64}} \approx 39.26$. With the given CPU, we are not able to achieve the desired speed up.

- To find the maximum achievable speedup, the limit of $S(n)$ for $n \to \infty$:

$$\lim_{n \to \infty} S(n) = \frac{1}{1-p}$$

The maximum achievable speedup for this code is thus $S_{max} = \frac{1}{1-p} = \frac{1}{0.01} = 100$.

Points: 6

- **3 for showing the achievable speed up of 50 is not doable.**
- **3 for finding the maximum achievable speedup.**

b) Your boss doesn't want to spend too much money on upgrading your Computer.

- What is the minimum amount of cores needed to have at least a speedup of 50?

We are searching for a n such that $S(n) \geq 50$. We do that by solving for n.

$$\frac{1}{(1-p)+\frac{p}{n}} \geq 50$$

$$1-p+\frac{p}{n} \leq \frac{1}{50}$$

$$n \geq \frac{p}{\frac{1}{50}-(1-p)}$$

We need at least $n \geq \frac{0.99}{\frac{1}{50}-0.01} = 99$ cores.

Points: 4

- 2 **For the right calculations and solving.**
- 2 **For the correct number of cores.**

c) You somehow improved your program and now achieve a parallelization fraction of 99.5%.

- How many cores are now needed to reach a speed up of 50?

Same as before but with $p = 0.995$ with gives $n \approx 66.3$. Because we can't have a fractional number of CPU cores we need at least 64 cores to achieve a speed-up of at least 50.

Points: 4

- 2 **For the right calculations and solving.**
- 1 **For the correct number of cores.**
- 1 **For rounding up to the next interger.**

d) Given the same parallelizable fraction from the previous question $p = 99.5\%$, you just found out that your code does not follow Amdahl's law. Part of your code, the CPU cores do have to communicate with each other, which costs a constant amount of time for every core. Therefore the corresponding time for this operation scales proportionally with the number of cores as $0.001(n-1)$[1].

- What is the maximum speed up you can achieve and for how many cores?

The time it takes for your code is given by the sum of the serial, parallel and communication parts:

$$T(n) = (1-p) + 0.001(n-1) + \frac{p}{n}$$

The speed up is expressed as:

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{0.005 + 0.001(n-1) + \frac{0.995}{n}}$$

Due to the communication overhead, we now have to find the maximum of our newly defined speed up function. We can do that by taking the minimum of the denominator and setting it to zero:

$$\frac{\partial}{\partial n}\left(0.005 + 0.001(n-1) + \frac{0.995}{n}\right) = 0.001 - \frac{0.995}{n^2}$$

[1] Note that $(n-1)$ because if you only have 1 core, there is no communication.

$$0.001 - \frac{0.995}{n^2} = 0 \Rightarrow n^2 = 995 \Rightarrow n \approx 31.54$$

Due to n not being a whole number, we have to look for which n we achieve maximum speed up.

$$S(31) \approx 14.9038, S(32) \approx 14.9045,$$

The maximum achievable speed up is 14.9045 with $n = 32$

Points: 16

- 4 **For the right time function.**
- 2 **For the right Speedup function.**
- 3 **For solving for the maximum of the Speedup.**
- 1 **For finding the non interger** $n \approx 31.54$
- 4 **Searching the right** n **by inputing the rounded up and down into the Speedup function**
- 2 **For** $n = 32$

Question 2: Parallel Scaling (20 points)

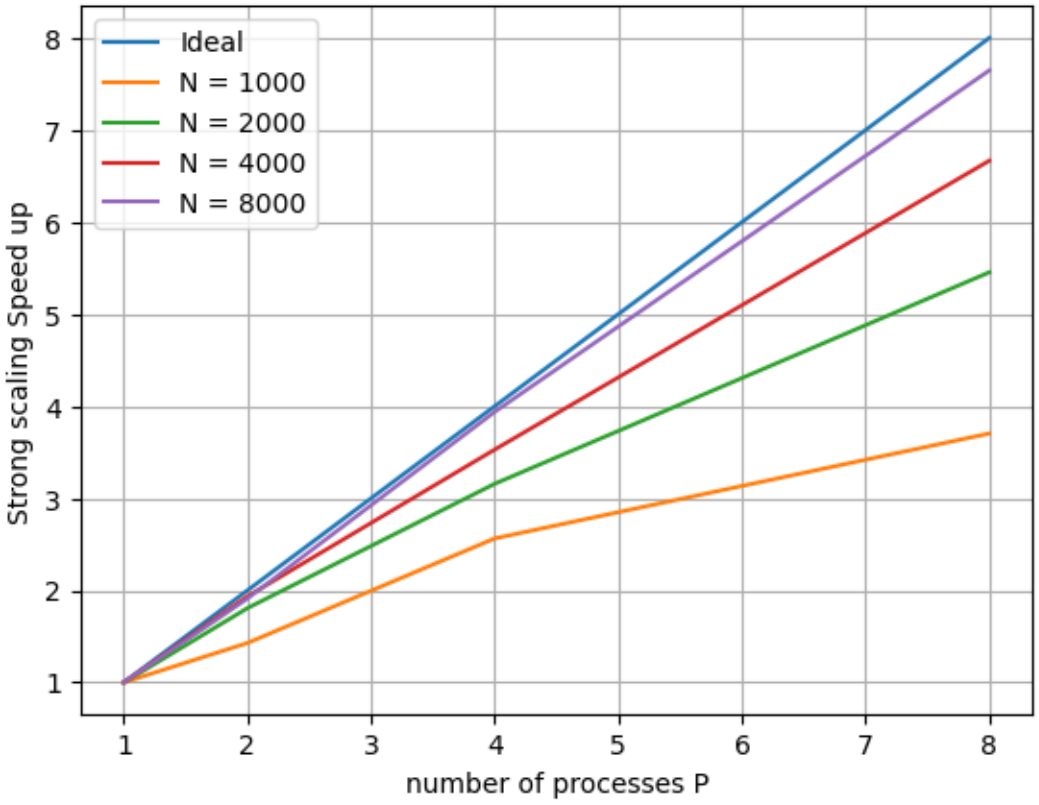
In this exercise, we want to draw the plots of strong and weak scaling. You are given runtime measurements of a simulation with different array sizes N and for different numbers of CPU cores P. For the weak scaling Efficiency, the runtime of the system is increasing linearly by N. You can draw the graphs directly on paper or use any plotting software (MATLAB, Matplotlib, ...).

P\N	1000	2000	4000	8000
1	100	300	600	1300
2	70	165	310	680
4	39	95	170	330
8	27	55	90	170

- Plot the points of the strong scaling speed up, as well as the ideal line.
- Show all steps of the calculations.
- Don't forget to Label the axes.

We find the speed for the different problem sizes:

N\P	1	2	4	8
1000	$\frac{100}{100} = 1.00$	$\frac{100}{70} = 1.43$	$\frac{100}{39} = 2.56$	$\frac{100}{17} = 3.70$
2000	$\frac{300}{300} = 1.00$	$\frac{300}{165} = 1.81$	$\frac{300}{95} = 3.16$	$\frac{300}{55} = 5.45$
4000	$\frac{600}{600} = 1.00$	$\frac{600}{310} = 1.94$	$\frac{600}{170} = 3.53$	$\frac{600}{90} = 6.67$
8000	$\frac{1300}{1300} = 1.00$	$\frac{1300}{680} = 1.91$	$\frac{1300}{330} = 3.93$	$\frac{1300}{170} = 8.65$

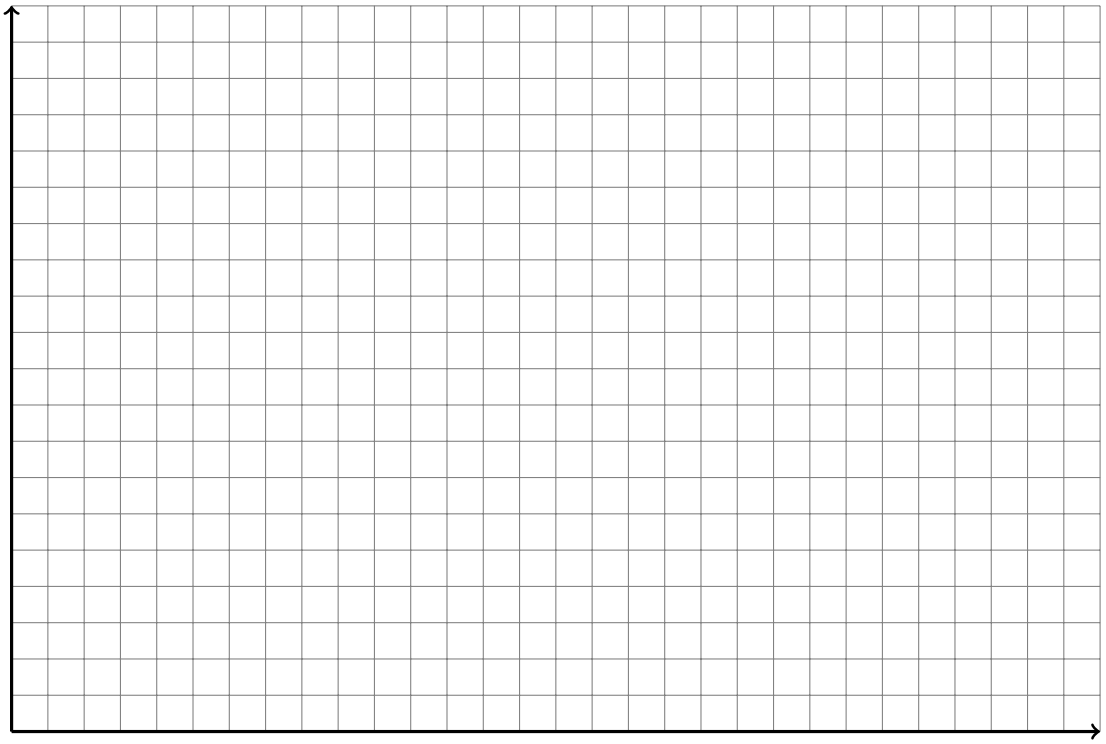

Ideal
N = 1000
N = 2000
N = 4000
N = 8000
Strong scaling Speed up
number of processes P
1
2
3
4
5
6
7
8

resulting in the following strong scaling plot:

Points: 10

- **4 points for correct axis labels (2 per axis)**
- **5 points for correct points in the plot (1.5 per point)**
- **1 for drawing the ideal line**

- Plot four points of the weak scaling Efficiency using $N = 1000$ and $P = 1$ as a reference.
- Show all steps of the calculations. Also, draw the Ideal line and label the axes.



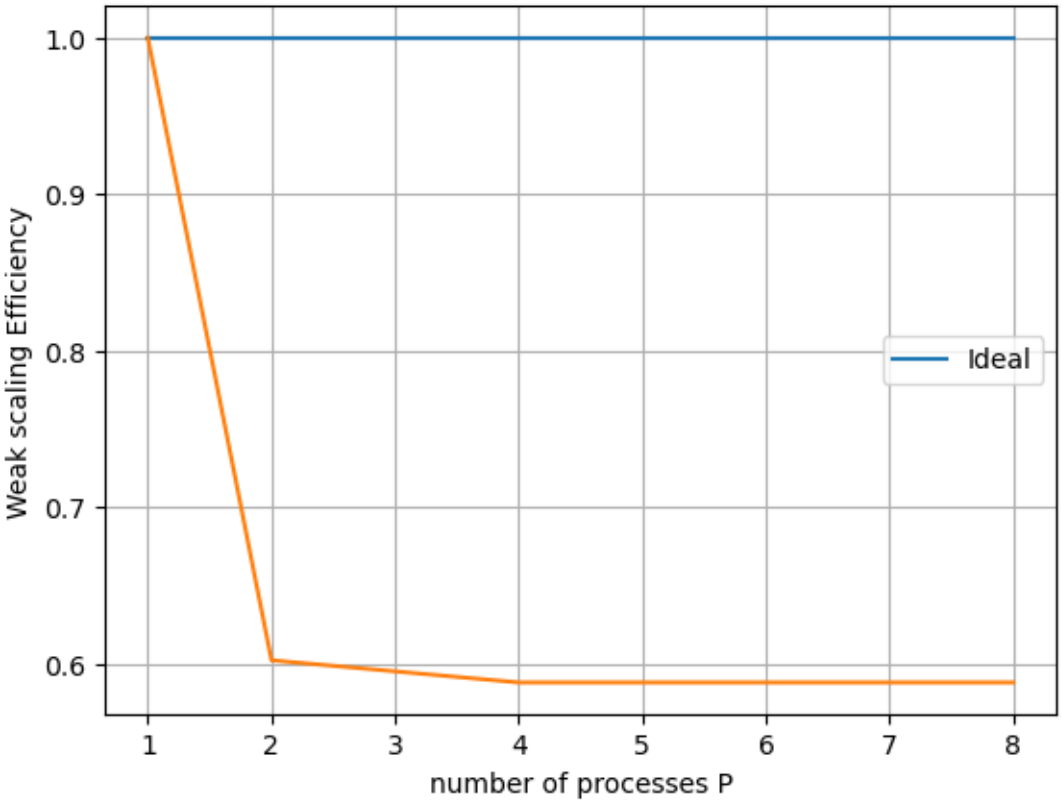
For the weak scaling analysis we scale the problem size N proportional to the number of cores P, i.e. for $N = 2000$ we need a factor x2 in the number of processors and for $N = 4000$ we need a factor x4 and for $N = 8000$ we need a factor of x8. From these we can compute the efficiency $T(1)/T(p)$:

P	1	2	4	8
$T(1)/T(p)$	$\frac{100}{100} = 1.00$	$\frac{100}{165} = 0.61$	$\frac{100}{170} = 0.59$	$\frac{100}{170} = 0.59$

Weak scaling Efficiency plot:

Points: 10

- **4 points for correct axis labels (2 per axis)**
- **5 points for correct points in the plot (1.5 per point)**
- **1 for drawing the ideal line**


1.0
0.9
0.8
0.7
0.6
Weak scaling Efficiency
1
2
3
4
5
6
7
8
number of processes P
Ideal

Question 3: Roofline Model (30 points)

a) Given the following code:

```
1    double a[N], b[N], c[N];
2    ...
3    for (int i = 0; i < N; ++i) {
4
5        c[i] = a[i] * a[i] * a[i] + b[i] - c[i] + 1.0;
6    }
```

- What is the operational intensity of the code, assuming that we have infinite cache size and cache is cold (empty)?
- State all the assumptions you made and show your calculations.

Note that the assumption to have infinite cache size is not necessary. The arrays only have to be read ones. Cold cache means the arrays are not already present in the cache, hence they have to be read and written.

read: a,b,c & write: c $\Rightarrow$ Q = 4n doubles = 32n bytes

W = 5n flops

I = W/Q = $\frac{5n}{32n}\frac{flops}{bytes} = \frac{5}{32}\frac{flops}{bytes}$

or if we only consider read:

I = W/Q = $\frac{5n}{3n*8}\frac{flops}{bytes} = \frac{5}{24}\frac{flops}{bytes}$

Points: 7

- **3 for the right number of read and write (or just read) operations**
- **3 For the right number of flops**
- **1 For the correct intensity**

b) Given the following code:

```
1    float a[N], C[N*N];
2    ...
3    for (int i = 0; i < N; ++i) {
4        for(int j = 0; j < N; ++j){
5            a[i] = a[i] + C[i*N+j] * 2.0;
6        }
7        C[i*N+i] = C[i*N+i]/(a[i] + 1.0);
8    }
```

- What is the operational intensity of the code, assuming that we have infinite cache size and cold cache?
- State all the assumptions you made and show your calculations.

Same as the previous question but more complicated. Due to having infinite cache size, we only have to read array a and matrix C ones. In total we do $N^2 + N$ read operations and $2N$ write operations hence:

$$Q = 2(N^2 + 3N)floats = 8(N^2 + 3N)bytes$$

$$W = 2(N^2 + N) flops$$

$$I = W/Q = \frac{2(N^2+N)}{8(N^2+3N)} \frac{flops}{bytes} \approx \frac{1}{4} \frac{flops}{bytes}$$

or if we just consider reads:

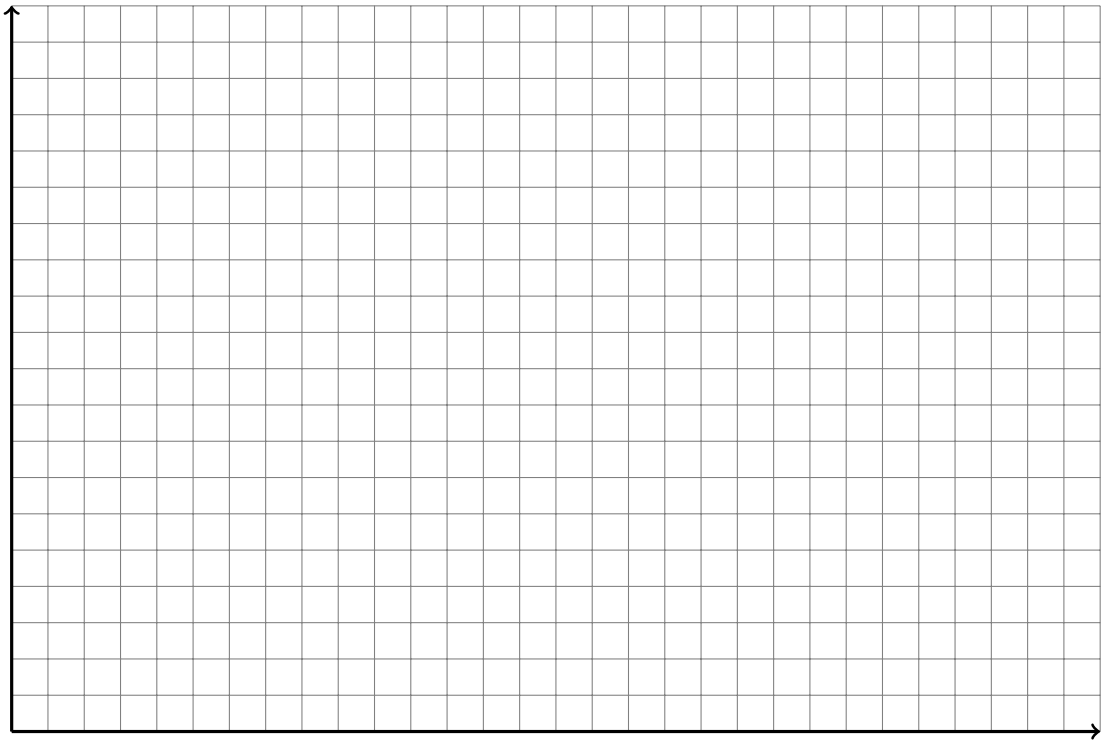
$$I = W/Q = \frac{2(N^2+N)}{8(N^2+N)} \frac{flops}{bytes} \approx \frac{1}{4} \frac{flops}{bytes}$$

Points: 7

- **3 for the right number of read and write (or just read) operations**
- **3 For the right number of flops**
- **1 For the correct intensity**

c) You are given an AMD R9 5950x which has 16 cores. 3.4GHz clock speed. As each core has 2 FMA[2] units and 2 floating point add/sub units. The CPU support AVX2 instructions. For memory, it has a 32KB L1 data cache, 512KB L2 cache and 32MB L3 cache. The measured memory peak bandwidth of this CPU is $\beta = 6.5 bytes/cycle$.

- State the peak performance and peak bandwidth of this CPU for multi-core and with AVX2 vectorization.
- Draw the roofline model of this CPU down below for non-vectorization and only for single-core performance.
- Don't forget to label the axis.
- State the ridge point.



[2]FMA = Fused Multiply Add floating point operation, which does a*b+c in one cycle.

See solution d) for the roofline plot. The absolute peak performance of this CPU is: $3.4G\frac{cycles}{s} \times 8[AVX2] \times 6\frac{flops}{cycle} \times 16[\#cores] \approx 2611.3\frac{Gflops}{s}$ and the $PB = 3.4G\frac{cycles}{s} \times 6.5\frac{bytes}{cycle} = 22.1\frac{GB}{s}$.
The ridge point in the non-vectorized, single-core peak performance is at an intensity of: $\frac{P_{peak}}{\beta} = \frac{6flops/cycle}{6.5bytes/cycle} \approx 0.92flops/byte$ and at the peak of $6flops/cycle$. In units of seconds it would be: $\frac{P_{peak}}{\beta} = \frac{3.4GHz \times 6flops/cycle}{22.1GB/s} \approx 0.92flops/byte$ and at a peak of $20.4Gflops/s$

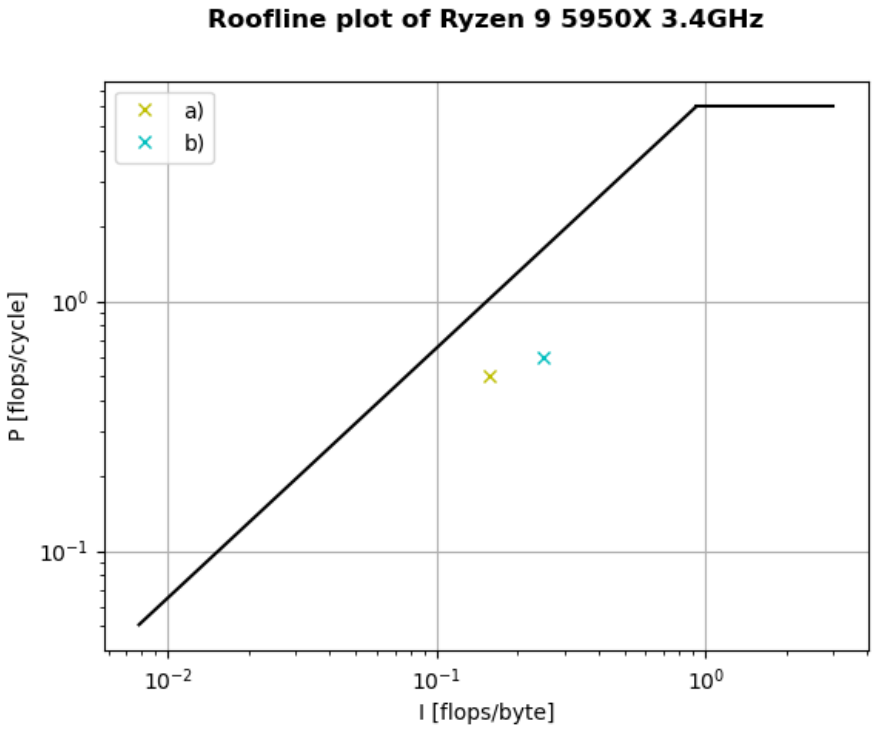
Points: 10

- **2 for the correct peak performance for vectorization and multi-core (either in units per cycle or per second).**
- **2 for the correct peak bandwidth for vectorization and multi-core (either in units per cycle or per second).**
- **2 For the labelling axis correctly (1 point each) (PP in cycles or in seconds is ok)**
- **3 for the right roofline plot**
- **1 For the correct ridge point**

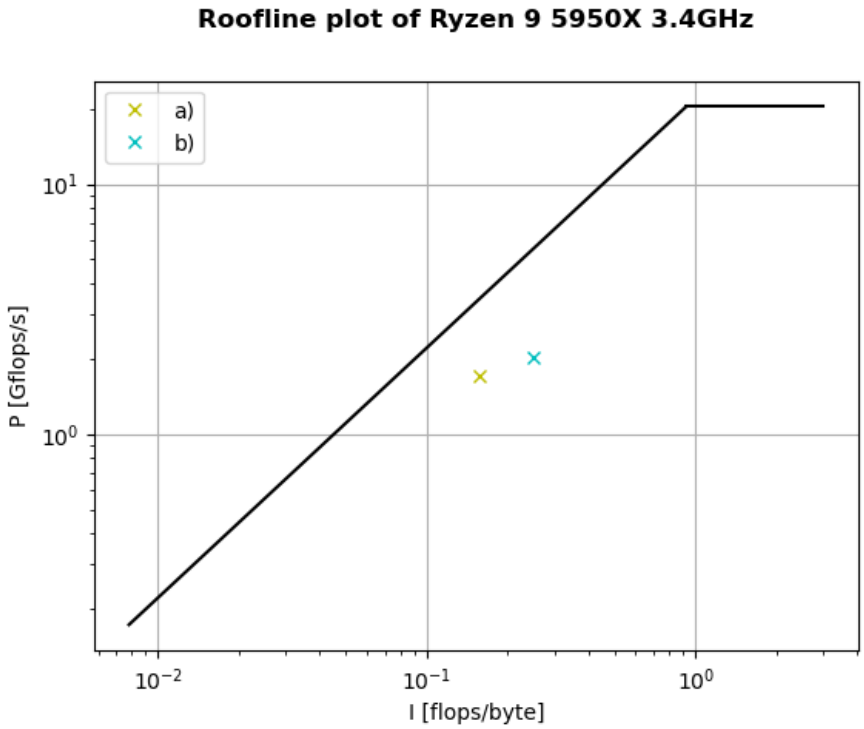
d) You measured the performance of a) being 0.5 flops/cycle and from b) being 0.6 flops/cycle.

- Draw those points inside your roofline model.
- Are they memory bound or compute bound?
- How do they perform?

Both are memory bound. Algorithm b) performes better than a) but a) is closer to its limit than b). Therefore a) performs relatively better than b).



Points: 6



- 4 for assuming both being memory bound (2 points each)
- 2 stating a) performing relatively better than b)

Question 4: Linear Algebra Operations (30 points)

In this exercise we will implement some basic linear algebra operations. We study the effects of storing and using data in different ways on the efficiency of the code.

a) We want to implement the element-wise matrix-matrix multiplication (also known as the Hadamard product). Implement it ones in row-wise and ones in column-wise. You are provided with a skeleton code and only have to fill in the TODO parts.

- State the time difference between T(col-wise)/T(row-wise) for different sizes of N and for different compiler flags.
- What do you observe?
- Play around with the compiler optimization flags to see if something changes with the runtime of the code.

On my laptop I got those measurements with different -O compiler flags of T(col-wise)/T(row-wise):

N	100	200	400	800
-O0	1.01	1.03	1.26	3.60
-O1	0.95	1.31	1.93	14.52
-O2	1.54	1.61	2.30	15.96
-O3	1.91	2.62	2.90	18.61
-Ofast	1.80	2.35	3.03	18.30

Hence column wise implementation is always slower than row-wise. Even the compiler flag -Ofast doesn't improve this human mistake of reading column-wise matrices. At very big matrices (like $N^2 = 800^2$) it is getting worse due to having to read from my RAM. $800^2 * 8 * 2bytes = 10.24MB$ which is more than all the caches of my laptop together[3]. Ignore the 0.95 in -O1. This could be the problem of not having warm up runs in the code.

Points 10:

- 1 **for initializing the matrices with random numbers**
- 1 **for implementing Hadamard product row-wise**
- 1 **for implementing Hadamard product col-wise**
- 2 **for measuring the time using chrono and averaging with the given reps.**
- 4 **for measuring for at least two different compiler flags and stating the time differences of T(col-wise)/T(row-wise)**
- 1 **state what you observe**

b) In this exercise we will compare our implementation of matrix-matrix multiplication with LAPACK subroutine function dgemm(). You are provided with a skeleton code and you only have to fill in the TODO parts.

c) We want to compute the product of two square matrices $A, B \in \mathbb{R}^{N \times N}$ The matrices A, B are stored in row major order in our implementation. One way to do the multiplication is the straightforward way,

[3] i7 7660U: L1 128KB, L2 512KB, L3 4.0MB

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \, .$$

As discussed in the previous subquestion, you have already seen the reasons this is not the most efficient way of implementing the product.

First, implement the straightforward multiplication algorithm in the provided skeleton code. Then implement the block-multiplication algorithm that was explained in the classroom. The block sizes are given. Finally, implement a second version of the block-multiplication algorithm where the matrix B_col is stored in column-major order. Initialize all the matrices with random numbers (not too big numbers). Note that B_row & B_col have to have the same values but in their respective major order stored, because you want to get the same matrix-matrix multiplication results for all implementations.

- Run your code over increasing matrix dimensions and vary the block size.
- What do you observe? How do you explain it?
- How much faster is the Lapack function?
- In the end, the results are saved in a file. Plot the results with the given python script or with your favourite tool.
- Analyze the results and draw your conclusion. Which one is the most efficient algorithm? Explain in terms of cache usage.

All implementations (naive, with blocking, and with blocking and column-major B) are given in the solution code. Figure 1 shows the performance versus the block size for the two implementations. The code is compiled with G++ with the -O3 optimization flag and executed on a compute node of Euler with `Intel Xeon Gold 6150` CPUs. The reported timings are the mean over 10, 5 or 1 samples depending on the matrix size. The version with blocking with the optimal block size is about five times faster than the naive. With the column-major matrix B, the optimal block size is larger since the inner loops read the matrices contiguously. Lapack is unbeatable and at least 14.8 times faster than any blocking implementations. This shows you should never implement something on your own. "Do not reinvent the wheel"!

Points 20:

- **2 for initializing the matrices with random numbers and B_col is the transpose of B_row.**
- **4 for implementing the matrix-matrix multiplication algorithm, naive + B row-major.**
- **4 for implementing the blocking matrix-matrix multiplication algorithm, B row-major.**
- **4 for implementing the blocking matrix-matrix multiplication algorithm, B column-major.**
- **2 for implementing and successfully running Lapack dgemm() function (also visible in the plot by being dominantly fast).**
- **2 for plotting the results.**
- **1 for showing that B column-major is more efficient.**
- **1 for explaining that B column-major is more efficient.**

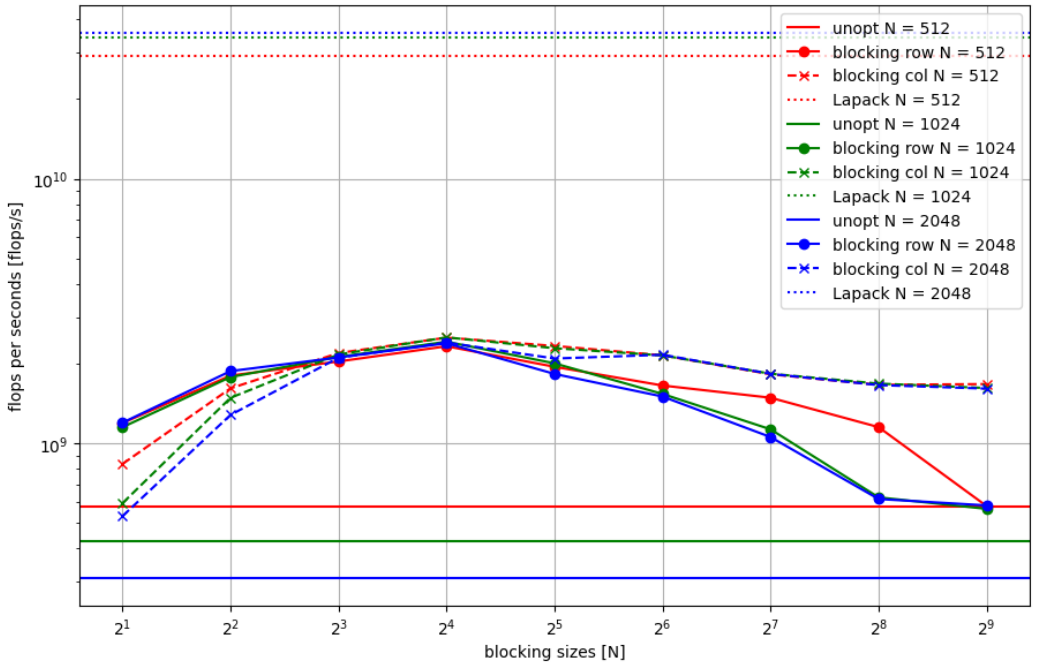


Figure 1: Performance of matrix-matrix multiplication versus the block size for various matrix sizes as well as compared to Lapacks function dgemm(). Naive implementation (horizontal straight lines), blocking with row-major B (circles) and blocking with column-major B (X's), Lapack function (horizontal dotted lines).

Question 5: OPTIONAL - Cache size and cache speed (Not graded)

This exercise shows how the performance of a program can be affected by the size of the data it operates with. Depending on whether the data fits into L1, L2, L3 cache, or not at all, we expect different memory access time. Furthermore, in which order we access the elements will also have an effect.

We will demonstrate this by traversing a linked list in the form of a permutation of size N, for different values of N. In other words, we will have an array of integers $a_0, \ldots, a_{N-1}$, where each a_i is a unique value from 0 to $N-1$. We start with the index $k = 0$, and then repeat M times the operation $k \leftarrow a_k$, for some $M \gg N$. This way we minimize memory-unrelated operations and measure virtually only the memory access time[4].

a) Before writing and running the code, check the sizes of L1, L2 and L3 cache by running the following command on an Euler compute node:

`grep . /sys/devices/system/cpu/cpu0/cache/index*/*`

The output will contain information from four different `index*` folders, each of which represents one cache level. Two are L1 caches (one for data, one for program code), one L2 and one L3 (both unified data and code). Extract the following information[5]:

- total size (property `size`),
- cache line size (property `coherency_line_size`).

The following tables includes cache and cache line sizes for different CPU models available on Euler cluster:

Model	L1D	L1I	L2	L3	Cache line
XeonGold_6150 (Euler IV)	32 kB	32 kB	1024 kB	25344 kB	64 B
XeonE3_1585Lv5 (Euler III)	32 kB	32 kB	256 kB	8192 kB	64 B
XeonE5_2680v3 (Euler II)	32 kB	32 kB	256 kB	30720 kB	64 B

L1D and L1I denote two parts of the L1 cache, one for data, one for instructions (the program itself). L2 and L3 cache are unified, they can store both data and instructions.

For the interested reader: As can be seen from the table, the fastest cache has only 32 kB (typical values are 32–64 kB). Thus, if our code is memory-intensive, if possible, we should split the data into chunks of size about 30 kB and do as much computation as possible before going to the next chunk (alternatively, one can aim to fit into L2 or L3). This is a basic idea behind, for example, cache-efficient matrix multiplication algorithms.

To select a specific CPU model on Euler, add a flag `-R "select[model==MODELNAME]"`. See `get_euler_cache_info.sh` for more info.

b) You are provided with a skeleton code for sampling the execution time for different values of N. The code already selects the values of N and outputs the results.

Fill out the `TODO` sections marked with *Question 5b* with the code for linked list traversal and time measurement. Use the provided `sattolo` function to generate a random one-cycle permutation. This function guarantees that the permutation is such that all of the N

[4]To be precise, we measure latency of reading a_k from memory (or cache), plus latencies of CPU instructions themselves. Thus, this will only give as an approximation of the memory and cache latencies.

[5]Depending on the node type your program assigned to, you will get different values. Optionally, you might also want to run `lscpu` (in the same run), to get the exact CPU model name.

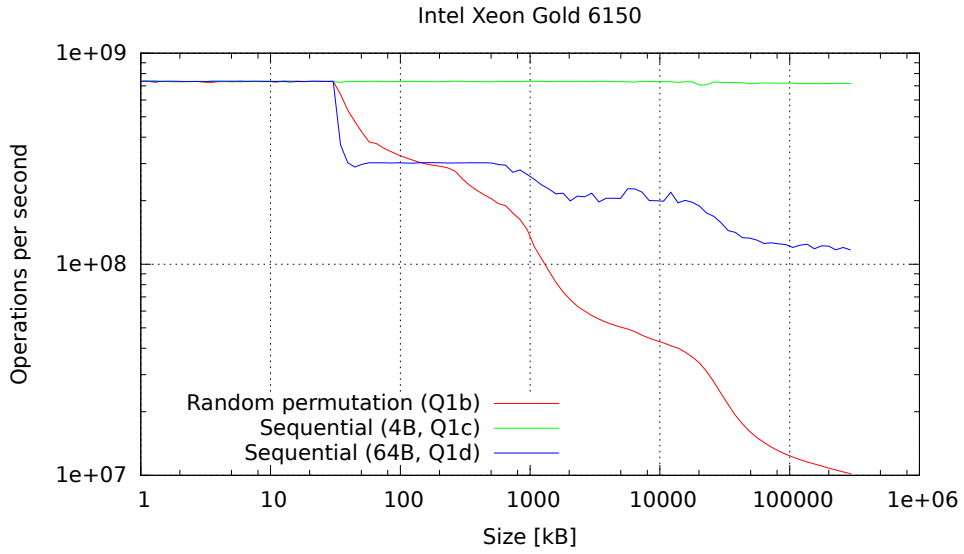


Figure 2: Performance of the permutation traversal for Question 5b, 5c and 5d on an Euler IV (Xeon Gold 6150) node. Note: The plot shows the performance for array size of up to 300 MB, but in the exercise you were asked to analyze only up to 20 MB. See text for more details.

elements are visited. Compile the code with `make`, run with `bsub ... make run` and plot the results with `make plot`.

What do you observe, do the drops in performance match the cache sizes? Are the transitions smooth or sharp, why?

The results for Intel Xeon Gold 6150 can be see in the Figure 2 (the red line). As we increase the array size, there is a clear drop in performance at 32 kB and at about 1024 kB, which amount to L1 and L2 cache sizes, respectively. The transitions are smooth because in a random permutation there is always a non-zero probability of jumping to a cache line that is already present in the cache, even though the total array is much larger (especially due to the fact that one cache line fits 16 32-bit integers). If we knew the exact cache policy determining how old entries are removed from the cache, we would in principle be able to construct a permutation which would force the CPU to load a new line from DRAM after every jump.

For the interested reader: For completeness, the plot was extend from 20 MB to 300 MB to show what happens the when array does not fit the L3 cache, which allows us to estimate the latency between CPU and DRAM. Considering that the execution time per jump for small arrays is negligible to those for very large arrays, we can estimate that the memory read has a latency of about 100 ns. Assuming the CPU frequency of 2.7 GHz, this amounts to about 270 cycles. Note that the memory throughput here is only 0.64 GB/s, much smaller than the maximum (you fetch 64 B [cache line] per element!).

Note: The exact plot shape (for all three lines) may change for different architectures! You should expect only qualitatively similar results. This applies to Question 4 as well.

c) Instead of jumping randomly through memory, initialize the array a such that k goes repeatedly as $0, 1, 2, \ldots, N-1, 0, 1, 2 \ldots$ Compare the performance of this (mostly) continuous access to the random access from the previous subquestions.

How would you explain the result?

The performance is shown in the same Figure 2 as the green line.

There are two reasons why the performance here is much better, and in fact does not depend on the array size. First, our data element a_k is only 4 bytes large (`sizeof(int)`), thus after we read 1 element from memory, we get another 15 for free. Secondly, the CPU is *prefetching* data from the memory. Namely, CPU detects that we are reading memory sequentially and fetches the data in advance. By the time we want to read a specific array element, it is already present in the cache.

For the interested reader: What this benchmark measures is in fact the total latency of instructions required to perform the operation `k = a[k]`. It takes 1.4 ns to do one jump, which amounts to 5 cycles[6]. If we relate this to instruction pipelines shown in the lecture, we see that the pipeline is greatly underutilized. The reason is that each jump operation must wait for the previous one to finish. This means that if, in the same for-loop, we add another "jumper" k_2 with `k2 = a[k2]`, we could expect the number of for-loop iterations per second not to drop at all (for small array sizes). Of course, to be sure, benchmarking would be required.

d) The previous subquestion was somewhat unfair. We would load a single cache line (of 64 bytes), and then do several jumps for free, because the data is already there. Implement the third variant of the permutation, where k jumps by 64 bytes (how many elements is that?). If that would cause k to go above $N-1$, take the modulo N. It does not matter if not all elements are visited this way, we still do force the CPU to load the whole array from memory, as we read from every cache line.

Compare the results with the previous two cases. What limits the performance for very large N in this case, and what in the case of a random permutation?

See the blue line in the Figure 2 for results.

As explained before, for very large N, the random permutation case is limited by the latency to access DRAM. In other words, it measures the cost of a *cache miss*. This case, on the other hand, is closer to benchmarking maximum memory bandwidth, as the CPU prefetcher already "knows" what to read (see below for details).

For the interested reader: For very large N, the number of jumps per second is about $0.11 \cdot 10^9$, amounting to 7 GB/s, which is still several times smaller than the expected maximum bandwidth, which is of the order of 50 GB/s. This should be expected, as we use a single core only. On the Euler IV nodes, if instead of doing `k = a[k]` we simple copy or read large arrays, we can achieve about 14 GB/s (because there are no data interdependencies). The single-core bandwidth is related to many factors, one of which could be the prefetcher and its limitations. For example, see Section 7.5.2 in the Intel Optimization Reference Manual.

The performance of a jump for different array sizes in this case can be partially related to cache latencies (see Agner's Optimization Guide, Section 11.12).

[6] If the base frequency of 2.7 GHz is assumed, the value of 3.66 cycles/jump is retrieved, but if boost frequency of 3.7 GHz is assumed the result is 5.02 cycles/jump. Thus, probably the CPU was in the boost mode, as that number is closer to an integer.