P. Koumoutsakos
S. Martin
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2021

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

Issued: October 1, 2021
Hand in (optional): October 15, 2021 12:00

Grading: For the exercise submission, you only need to submit 3 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 99.99% of the runtime is parallelizable. You want to run this program on a super computer that has up to $2'000'000$ cores.
What is the maximum speed up you can achieve if you had no limitations of processors n? What speed up can you achieve with 20, 200, 2000, $20'000$, $200'000$ and $2'000'000$ cores?

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

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

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 $S_{max} = \frac{1}{1-p} = \frac{1}{0.0001} = 10'000$.

- Replacing in the different number of cores the following speed ups are achieved:

n	$S(n)$
20	19.96
200	196.1
$2'000$	1666.81
$20'000$	6666.89
$200'000$	9523.85
$2'000'000$	9950.25

Points: 6

- 3 for finding the maximum achievable speedup,
- 3 for getting the speedup for the corresponding number of cores n (at least 4 correct)

b) You are executing a code with serial fraction 0.01. However, there are parts of a program that do not benefit from increasing the number of cores n used. Assume that the parallel execution of your code requires a communication operation, which costs a constant amount of time for every core, therefore the corresponding overall time for this operation scales proportionally with the number of cores as $0.01n$. What is the maximum speed up you can achieve with this constraint (and for how many cores)? If you reduce the time spent on this communication operation to $0.001n$, what is the new maximum speedup?

- Let T_0 be the the total time taken by the serial code. The time taken by the parallel version, is given by the sum of the serial, parallel and communication parts:

$$T(n) = T_0 \left(0.01 + 0.01n + \frac{0.99}{n}\right)$$

The speed up is expressed as:

$$\begin{aligned} S(n) &= \frac{T_0}{T(n)}, \\ &= \frac{1}{0.01 + 0.01n + \frac{0.99}{n}} \end{aligned}$$

By taking the minimum of the denominator, you can find the number of cores that will lead to the maximum speedup:

$$\frac{\partial}{\partial n}\left(0.01 + 0.01n + \frac{0.99}{n}\right) = 0.01 - \frac{0.99}{n^2}$$

$$0.01 - \frac{0.99}{n^2} = 0 \Rightarrow n^2 = 99 \Rightarrow n \approx 9.95$$

The optimal number of cores is $n = 9.9$. To find the maximum speedup we solve for $S(n)$ for $n = 9$ and $n = 10$:

$$S(9) = 4.76, S(10) = 4.78$$

The maximum achievable speedup is thus 4.78 using 10 cores.

- Reducing the communication part to $0.001n$ the new speed up is expressed as:

$$\begin{aligned} S(n) &= \frac{T_0}{T(n)}, \\ &= \frac{1}{0.01 + 0.001n + \frac{0.99}{n}} \end{aligned}$$

Finding the optimal number of cores:

$$\frac{\partial}{\partial n}\left(0.01 + 0.001n + \frac{0.99}{n}\right) = 0.001 - \frac{0.99}{n^2}$$

$$\begin{aligned} 0.001 - \frac{0.99}{n^2} = 0 \Rightarrow n^2 = 990 \Rightarrow n \approx 31.46 \\ \Rightarrow S(31) = 13.7107, S(32) = 13.7104 \end{aligned}$$

The maximum achievable speedup is thus 13.7107 using 31 cores.
Note that communication is not exactly an operation and only takes place when you are running a parallel code, for this reason the sum of the serial and parallel part is equal to one even if there is additional communication.

Points: 12

- 3 for the correct derivation of the speedup
- 6 for correctly taking the derivative of the denominator
- 3 for the correct numerical results (0.5 per result)

c) Upon inspection of a code, you see that 1000 of the code's operations are parallelizable and 10 are not. Suppose the time to compute one operation (both serial and parallel) is t_1.

- Compute the time $T(n)$ to execute the code with n cores.
- Using this time $T(n)$, compute the speed-up $S(n)$ for $n = 10$. Verify your answer using Amdahl's law.
- The speed up you get is true only in the case of perfect load balancing. For $n = 10$, re-compute the speed-up assuming one core is assigned 1.5 and 3 times more of the parallel operations.

- The time to execute the code is given by:

$$T(n) = \frac{1000t}{n} + 10t$$

- The speed up $S(n)$ is given by:

$$S(n) = \frac{T(1)}{T(n)} = \frac{1010t}{\frac{1000t}{n} + 10t} \Rightarrow S(10) = \frac{110t}{1010t} = 9.1818$$

The parallel fraction of the code is given as

$$p = \frac{\#\text{ parallel op}}{\#\text{ total op}} = \frac{1000}{1010} = 0.990099$$

Using Amdahl's law for $n = 10$ we get:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} \Rightarrow S(10) = \frac{1}{0.0099 + \frac{0.99099}{10}} = 9.1818$$

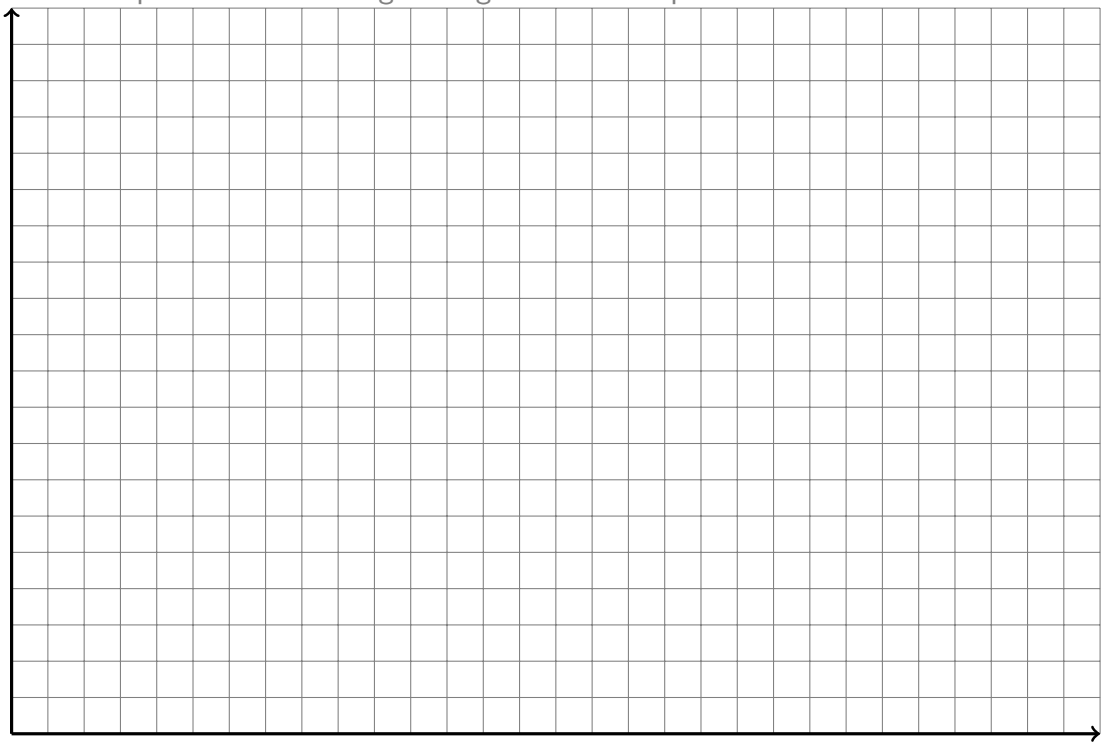
- In the perfect case all cores share the parallel load evenly. For $n = 10$, each core gets 10% of the parallel operations i.e. 100 operations.
  If one core 1 gets 1.5 times more parallel operations, its total execution time is $T_1 = 1.5 * 100t + 10t$. All the other cores share the rest of the parallel operations i.e. $(1000 - 150)/9 \approx 94.4$ operations. The execution time for all the other cores is $T_{2:10} = 94.4t + 10t$.
  As the execution time for core 1 is larger than all other cores, they will idle until core 1 finishes and the total time of the code is $T = T_1 = 160t$. The total speed up in this case is $S = \frac{1010t}{160t} = 6.3125$ instead of 9.1818.
  For an imbalance factor of 3, repeating the same procedure, we end up with a speed up of $S = 3.258$.
  Points: 12
  - 3 for finding the time $T(n)$ and using it to find the speed up $S(n)$
  - 3 for verifying the speedup using Amdahl's law
  - 3 for finding the time of the slowest core for both imbalance factors
  - 3 for finding the speedup for both imbalance factors

Question 2: OPTIONAL - Parallel Scaling (Not graded)

A program simulates N particles. All particles interact with each other and, thus, the number of interactions is proportional to N^2. The runtimes of the program in seconds on P processor cores are in the table:

P\N	500	1000	1500	2000
1	6.00	30.00	72.00	120.00
4	1.50	7.50	18.00	30.00
9	0.75	3.50	9.00	20.00
16	0.50	2.15	6.00	12.00
24	0.40	1.50	4.50	10.00

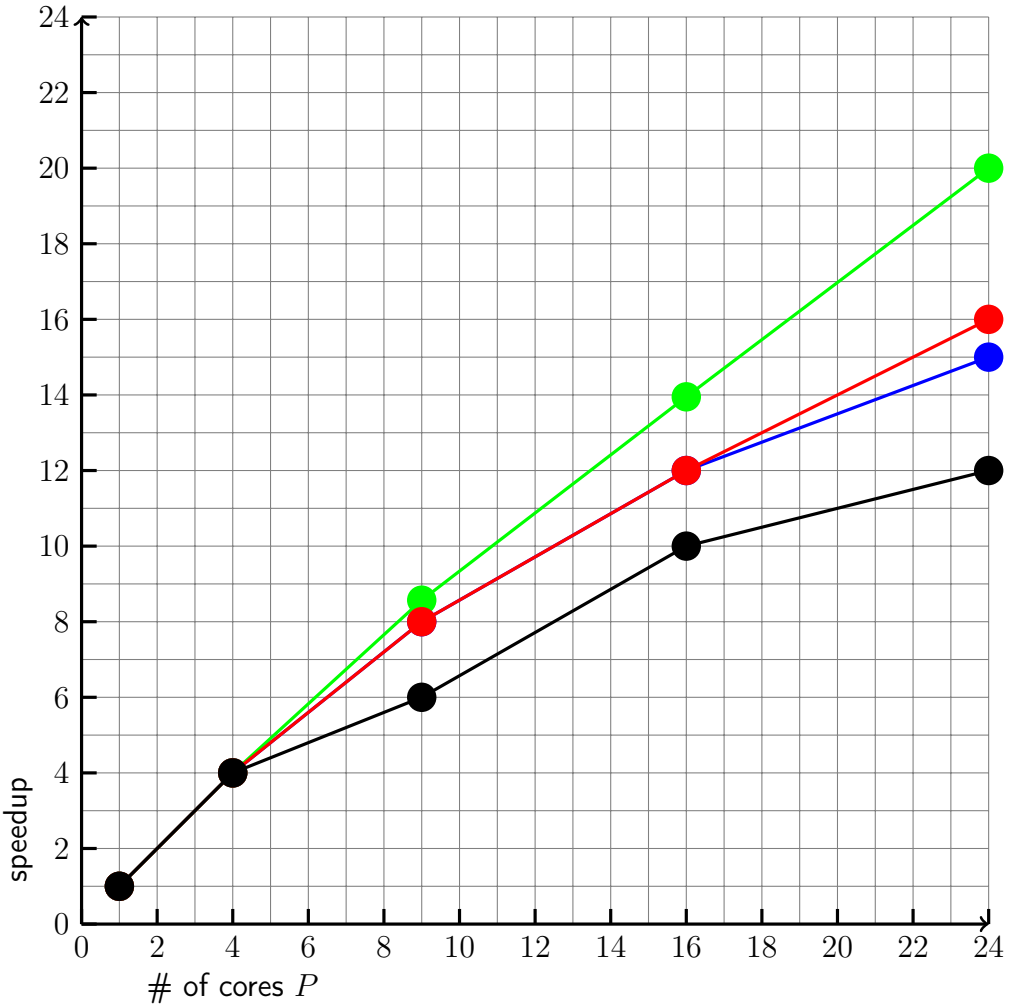
Plot four points of the strong scaling. Show all steps of the calculations. Label the axes.



We find the speed for the different problem sizes:

P	1	4	9	16	24
$S(p) = T(1)/T(p)$	$\frac{6.0}{6.0} = 1.0$	$\frac{6.0}{1.5} = 4.0$	$\frac{6.0}{0.75} = 8.0$	$\frac{6.0}{0.5} = 12.0$	$\frac{6.0}{0.4} = 15.0$
$S(p) = T(1)/T(p)$	$\frac{30.0}{30.0} = 1.0$	$\frac{30.0}{7.5} = 4.0$	$\frac{30.0}{3.5} \approx 8.57$	$\frac{30.0}{2.15} \approx 13.95$	$\frac{30.0}{1.5} = 20$
$S(p) = T(1)/T(p)$	$\frac{72.0}{72.0} = 1.0$	$\frac{72.0}{18.0} = 4.0$	$\frac{72.0}{9.0} = 8.0$	$\frac{72.0}{6.0} = 12.0$	$\frac{72.0}{4.5} = 16.0$
$S(p) = T(1)/T(p)$	$\frac{120.0}{120.0} = 1.0$	$\frac{120.0}{30.0} = 4.0$	$\frac{120.0}{20.0} = 6.0$	$\frac{120.0}{12.0} = 10.0$	$\frac{120.0}{10.0} = 12.0$

resulting in the following strong scaling plot:



Points: 10

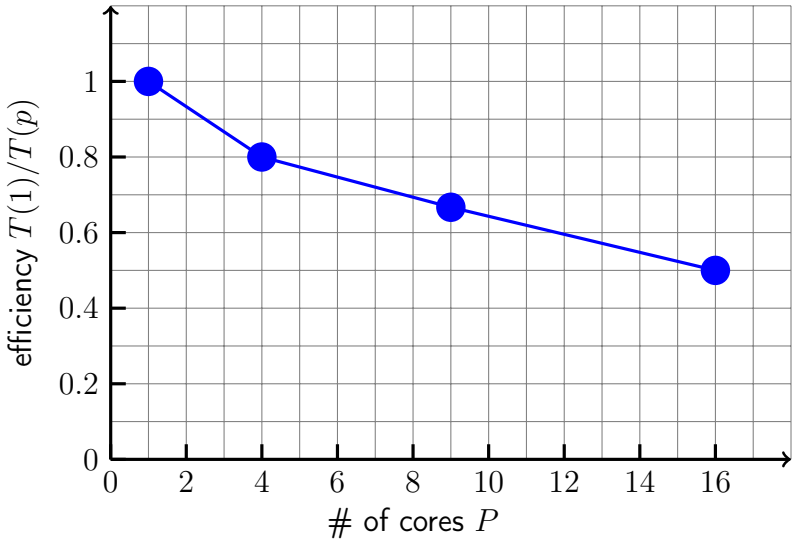
- **4 points for correct axis labels (2 per axis)**
- **6 points for correct points in the plot (1.5 per point)**

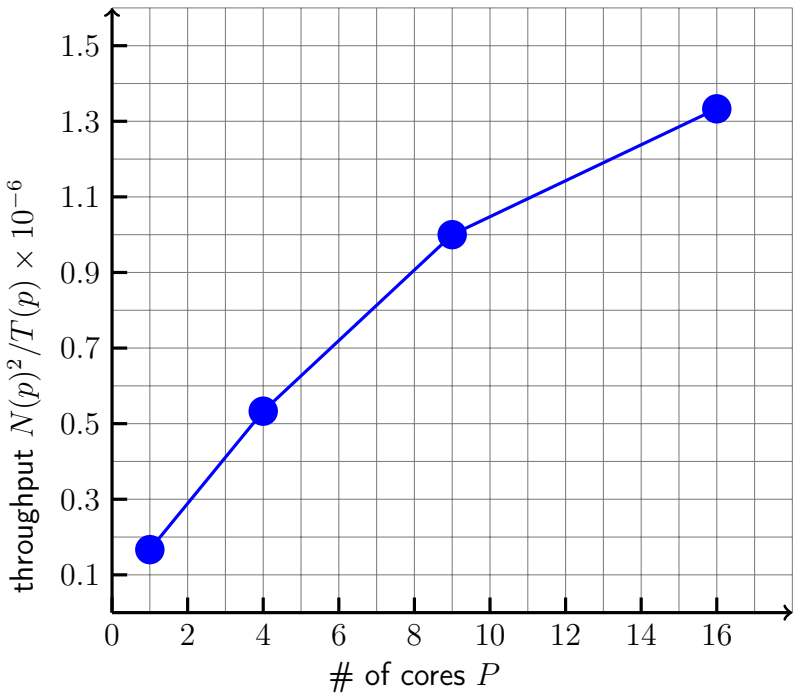
Plot four points of the weak scaling using $N = 500$ and $P = 1$ as a reference. Show all steps of the calculations. Label the axes.

For the weak scaling analysis we scale the problem size $N \times N$ proportional to the number of cores P, i.e. for $N = 1000$ we need a factor x4 in the number of processors and for $N = 1500$ we need a factor x9 and for $N = 2000$ we need a factor of x16. From these we can compute the efficiency $T(1)/T(p)$ or the throughput $N(p)^2/T(p)$:

P	1	4	9	16
$T(1)/T(p)$	$\frac{6.0}{6.0} = 1.0$	$\frac{6.0}{7.5} = 0.8$	$\frac{6.0}{9.0} = 0.667$	$\frac{6.0}{12.0} = 0.5$
$N(p)^2/T(p)/10^6$	$\frac{1.0}{6.0} = 0.1667$	$\frac{4.0}{7.5} = 0.533$	$\frac{9.0}{9.0} = 1.000$	$\frac{16.0}{12.0} = 1.333$

Either of the following weak scaling plots is therefore acceptable:





Points: 10

- **4 points for correct axis labels (2 per axis)**
- **6 points for correct points in the plot (1.5 per point)**

Question 3: Roofline Model (30 points)

Given the following code:

```
float A[N], B[N], C[N];
...
const int P = 2;
for (int i = 0; i < N; ++i) {
    int j = 0;
    while (j < P) {
        A[i] = B[i] * A[i] + 0.5;
        ++j;
    }
    C[i] = 0.9 * A[i] + C[i];
}
```

a) What is the floating point operational intensity of the code? State all the assumptions you made and show your calculations.

Assumptions: we have enough registers.

```
    A[i] = B[i] * A[i] + 0.5;
    A[i] = B[i] * A[i] + 0.5;
    C[i] = 0.9*A[i] + C[i];
```

memory operations: read A,B, C write A, C, (3+2)*4=20 bytes, fp operations: 3 mul + 3 add = 6 flops

OI = 6/20

Make sure they don't count ++j.

DA:

- With cache: 5*4 byte transferred, P*2 + 2 flops, OI = (P*2+2)/20
- Without cache: P*3*4 + 3*4 bytes transferred, P*2 + 2 flops, OI = (P*2+2)/([P+1]*12)

Points: 10

- **5 points 1 per correct read/write operation found**
- **3 points for correct flops**
- **2 points for correct operational intensity**

b) For a peak performance of 409.7 GFLOP/s (single precision) and a memory bandwidth of 34 GB/s, find all positive P for which the code is memory bound. Assume an infinite cache and state further assumptions you made. Show your calculations.

Ridge point = 409.7 / 34 = 12.05 Assuming perfect caching A[i], B[i], C[i] are read only once from the memory and A[i], C[i] are written only once to the memory.

```
    A[i] = B[i] * A[i] + 0.5;        \\ 2 FLOP
    ...
    A[i] = B[i] * A[i] + 0.5;        \\ 2 FLOP
    C[i] = 0.9*A[i] + C[i];          \\ 2 FLOP
```

OI > Ridge point = 12.05

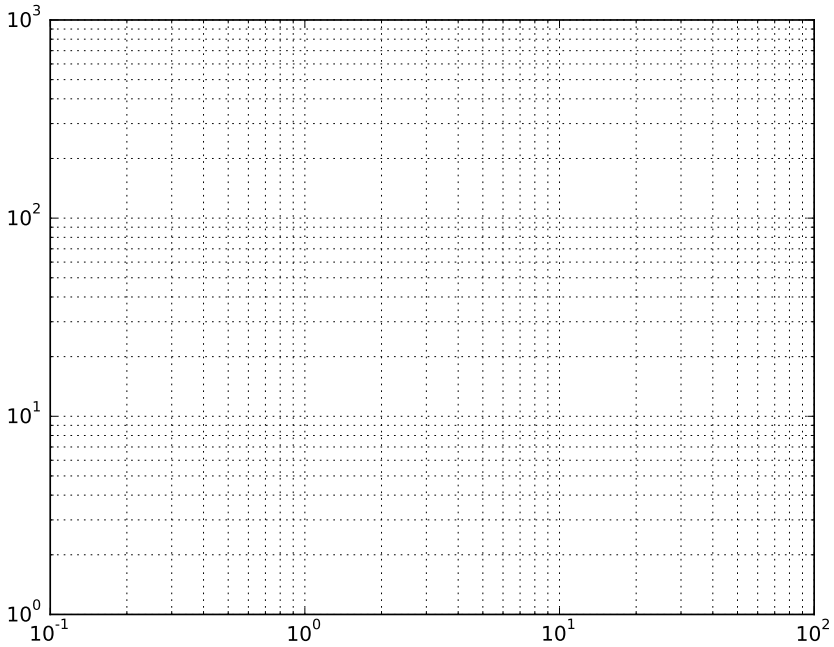
Flops = (2 + 2P), Bytes = 5*4 = 20

(2 + 2P)/20 < 12.05
2 + 2P < 20 * 12.05 = 241
2P < 239
P < 239/2 = 119.5
P < 119
Less means less or equal here

Points: 12

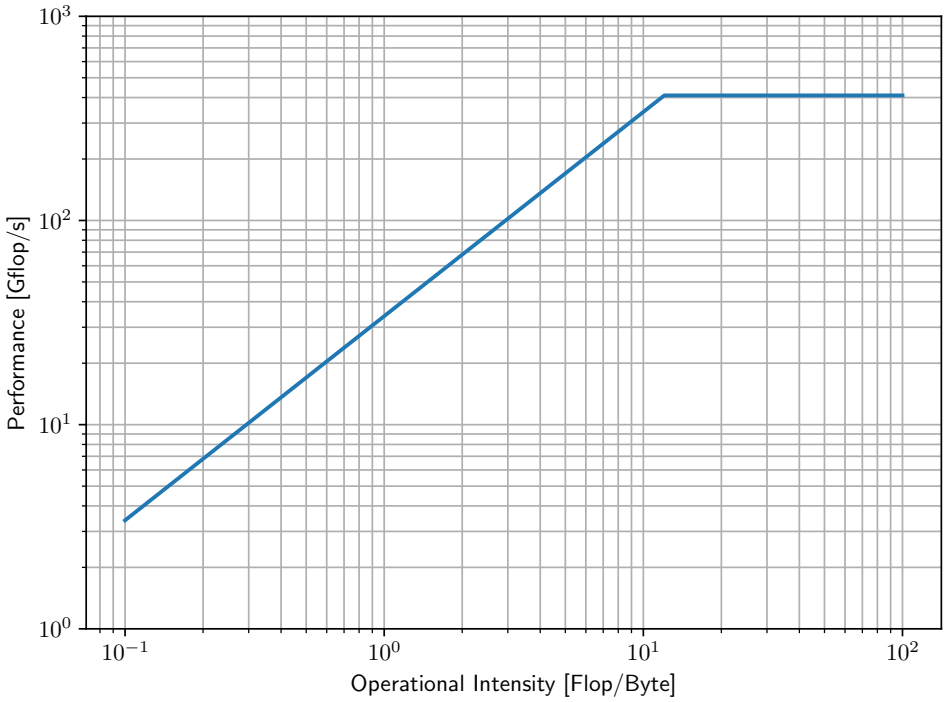
- **4 points for correct ridge point**
- **4 points for assuming perfect caching and that A,B,C are read only once**
- **4 points for correct inequality for P**

c) Draw below the roofline corresponding to (b) and label the axes.



Points: 8

- **4 points for correct axis labels**
- **2 points for correct ridge point**
- **2 points for correct slope**


10^3
10^2
10^1
10^0
Performance [Gflop/s]
10^{-1}
10^0
10^1
10^2
Operational Intensity [Flop/Byte]

Question 4: Linear Algebra Operations (20 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 multiplication of a square matrix $A \in \mathbb{R}^{N \times N}$ with a vector $\boldsymbol{x} \in \mathbb{R}^N$. Notice that we have two options, to store the matrix A row-wise (row major order) or column-wise (column major order). You are provided with a skeleton code and you are asked to implement the product $A\boldsymbol{x}$ storing A with row major order first and then with column major order. You can initialize $A_{i,j} = i + j$ and $x_i = i$.

Which implementation is faster and why? What do you observe as the dimension of the matrix increases? How do you explain it?

In the following table the ratio of the execution time of the row-major over the column-major order implementation is presented for increasing values of the N. Both row-major and column-major implementations make a good use of the cache (see solution code).

N	100	200	400	1000	2000	5000	10000
`CXXFLAGS=-O0`	1.02	1.0	1.01	1.02	1.01	1.03	0.99
`CXXFLAGS=-O3`	1.81	1.87	1.97	2.02	1.61	1.63	1.64

With no optimization enabled (`-O0`), both implementations perform equally and use the cache in the same manner. Increasing the compilation optimization (`-O3`), on the other hand, favors the column-wise approach. The compiler is able to vectorize more easily the column major implementation than the row-major one. This difference between `-O0` and `-O3` happens because the problem is limited by the computation rather by the bandwidth (we used no multithreading, no instruction level parallelism and no vectorization in the `-O0` version).

Points 4:

- 2 **for implementing the matrix-vector multiplication.**
- 1 **for observing that the speedup is increasing as the N increases.**
- 1 **for explaining using cache line.**

b) We want to compute the transpose of a square matrix $A \in \mathbb{R}^{N \times N}$ The matrix A and its inverse $A^\top$ are stored in row major order in our implementation. One way to do the transposition is the straightforward way, where we loop over all the indices and we set the corresponding (i, j) element of $A^\top$ equal to the (j, i) element of A. As you have seen in the lecture, this is not the most efficient way in terms of cache usage due to compulsory misses. A solution to this problem is to work in blocks.

First implement the straightforward transposition algorithm in the provided skeleton code. Then, imitate the multiplication algorithm that was explained in classroom and use the provided skeleton code to implement a block version of the transposition algorithm.

Run your code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results with your favourite tool. Analyze the results and draw your conclusion.

Both implementations (naive and with blocking) are given in the solution code. Figure 1 shows the performance versus the block size for the two implementations. The code is compiled with GCC 4.8.2 and executed on a compute node of Euler with `Intel Xeon`

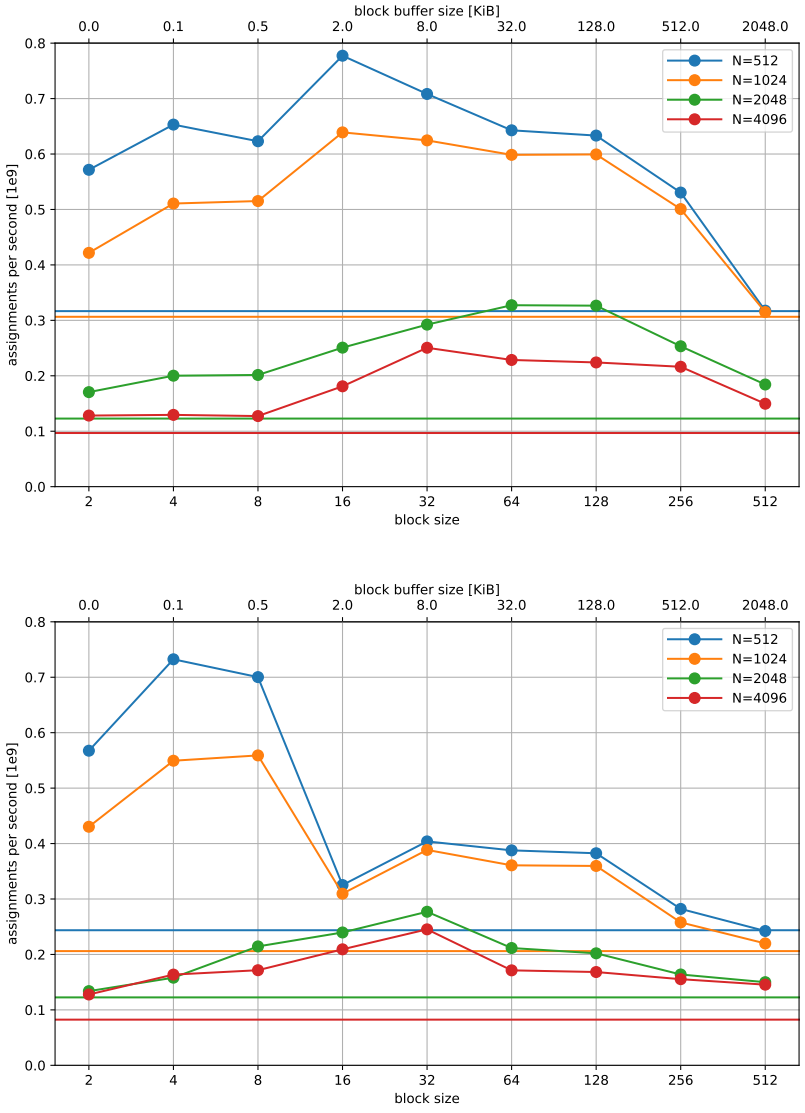


Figure 1: Performance of transposition versus the block size for various matrix sizes. Naive implementation (horizontal lines) and implementation with blocking (circles). Version with consecutive writes (top) and strided writes (bottom).

Gold 6150 CPUs. The reported timings are the minimum over 10 samples, each sample is the mean time over a batch of $4096^2/N^2$ runs to reduce the measurement overhead for small matrices. The cache is flushed before each new sample (but not withing the batch) by writing arbitrary data to a sufficiently large block of memory.

The cache sizes of the processor are 32K for L1, 1024K for L2, and 25M for L3. Performance is higher for matrices that fit in L3 cache ($N = 512$ and 1024). Matrix with $N = 2048$ takes 32M and no longer fits. Implementation with blocking and the optimal block size is about two times faster that the naive. The highest values are achieved when the block fits in L2 (for example, block of size 64 taking 32K).

One detail is about the access pattern for writes. The solution code contains two versions, the second one triggered by macro `WRITESTRIDE` (for example, by passing `-DWRITESTRIDE` to the compiler). One version has the inner loop over index `k` writing to consecutive locations:

```
for (size_t l = j; l < j + blockSize; l++)
    for (size_t k = i; k < i + blockSize; k++)
        AT[k + l * N] = A[l + k * N];
```

The other version version has the inner loop over index `l` writing to strided locations:

```
for (size_t k = i; k < i + blockSize; k++)
    for (size_t l = j; l < j + blockSize; l++)
        AT[k + l * N] = A[l + k * N];
```

The version with strided writes is 30% slower (Figure 1, bottom). A possible explanation to this is that a write-miss is more expensive than a read-miss. According to the write-back policy, a write-miss requires first reading the data for the cache line from memory, and then the actual writing when the cache line is evicted. Both versions are equally fast for blocks of size 4 and 8 for matrices $N = 512$ and 1024. In this case the whole matrix fits in L2 and each block fits in L1 along with `N/blockSize` blocks required for one outer loop.

Points 6:

- 4 **for implementing the transposition algorithms.**
- 1 **for finding an optimal block size.**
- 1 **for plotting the results and making at least one relevant observations.**

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,

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

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. Finally, implement a second version of the block-multiplication algorithm were the matrix B is stored in column major order. You can initialize $B_{i,j} = 2i + j$.

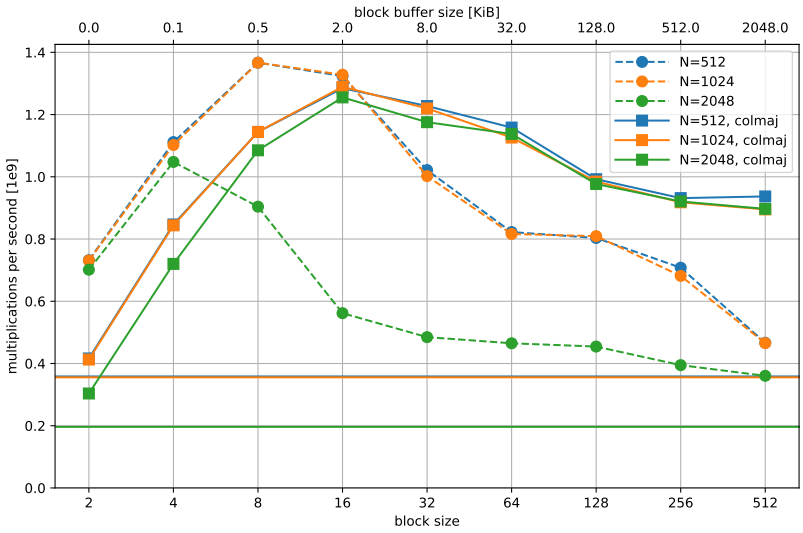


Figure 2: Performance of matrix-matrix multiplication versus the block size for various matrix sizes. Naive implementation (horizontal lines), blocking with row-major B (circles) and blocking with column-major B (squares).

Run you code over increasing matrix dimensions and vary the block size. What do you observe? How do you explain it? In the end, the results are saved in a file. Plot the results 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 2 shows the performance versus the block size for the two implementations. The code is compiled with GCC 4.8.2 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.

Points 10:

- 4 **for implementing the matrix-matrix multiplication algorithm, naive +** B **row-major.**
- 2 **for implementing the matrix-matrix multiplication algorithm,** B **column-major.**
- 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.**

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[1].

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[2]:

- 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.

Points 1:

- **1 for getting the results (on any Euler node or other CPU).**

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.

[1] 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.

[2] 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.

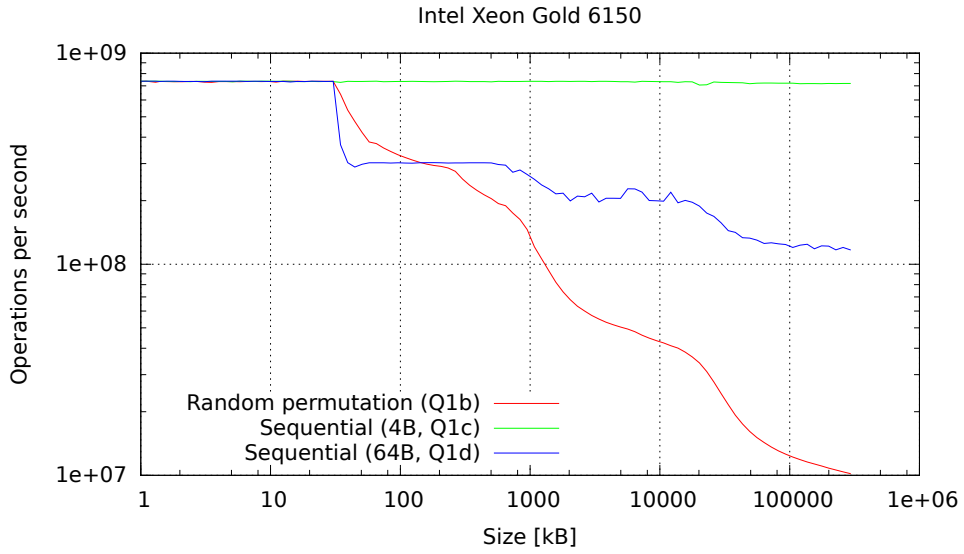


Figure 3: Performance of the permutation traversal for Question 5b, 5c and 5d on an Euler IV (Xeon Gold 6150) node. Note: The plot shows the performance for array size of up to 300 MB, but in the exercise you were asked to analyze only up to 20 MB. See text for more details.

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 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 3 (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.

Points 9:

- 2 **for implementing the jump for-loop,**
- 2 **for implementing time measurement,**
- 2 **for finishing and running the 5b part of the code and generating plots,**
- 1.5 **for matching the plot shape to the cache size,**
- 1.5 **for explaining smoothness.**

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 3 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[3]. 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.

Points 4:

- 3 **for implementing the sequential case 5c,**
- 1 **for arguing about the prefetcher (1 pt if you only mentioned that a single cache line has multiple a_k elements).**

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 3 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

[3] 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.

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).

Points 6:

- 3 **for implementing the sequential case with 64 B stride (5d),**
- 1.5 **for saying that random permutation case is limited by the latency to DRAM (or cache miss cost),**
- 1.5 **for saying that the 64 B stride case is related to memory bandwidth.**