Question 1:

"Amdahl's Law" Speed up = SU

a) Program:

99.99 % is parallelized -> $p = 0.9999$

- what is the max. speed up if $n \to \infty$

$$\| T_{new} = (1-p) T_{old} + \underbrace{\frac{1}{n} p T_{old}}_{0 \text{ if } n \to \infty} \|$$ .... Amdahl's Law

$$SU = \frac{T_{old}}{T_{new}} = \frac{1}{1-p} = \frac{1}{1-0.9999} = \underline{\underline{10,000}}$$

- SU for different numbers of cores

$$SU = \frac{T_{old}}{T_{new}} = \frac{1}{1-p+\frac{1}{n}p} = \frac{1}{1-p(1-\frac{1}{n})}$$

$$SU(n=20) = \frac{1}{1-0.9999(1-\frac{1}{20})} = 19.96 \approx \underline{20}$$

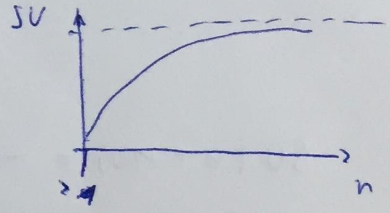
$SU(n=200) = 196.1 \approx \underline{196}$

$SU(n=2,000) = 1,666.8 \approx \underline{1,667}$

$SU(n=20,000) = 6,666.9 \approx \underline{6,667}$

$SU(n=2 \cdot 10^5) = 9,523.86 \approx \underline{9,524}$

$SU(n=2 \cdot 10^6) = 9,950.23 \approx \underline{9,950}$



b) Communication task is involved in the program
The communication scales with 0.01 n (with respect to cores)

This operation adds time to the computation so an additional term has to be introduced to Amdahl's Law.
The communication affects the parallilized tasks.
Modified Amdahl's Law:

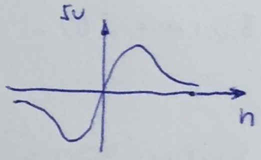
$$T_{new} = (1-p)\,T_{old} + \left(\frac{1}{n} + \underbrace{0.01}_{x}\,n\right) p\,T_{old}$$

$$\frac{T_{old}}{T_{new}} = \frac{1}{1-p+p\left(\frac{1}{n}+xn\right)} = \frac{1}{1+p\left(\frac{1}{n}+xn-1\right)} \qquad (1)$$

In order to maximize (1) the number of n where $f(n)' = \emptyset$ has to be found, i.e. compute the first derivative and set it equal to zero.

$$SU(n)' = \frac{\emptyset - p\cdot\left(-\frac{1}{n^2}+x\right)}{\left(1+p\left(\frac{1}{n}+xn-1\right)\right)^2} \overset{!}{=} \emptyset \iff \cancel{p}\left(\frac{1}{n^2}-x\right) = \emptyset \iff$$

$$n^2 = \frac{1}{x} \iff n = \sqrt{\frac{1}{x}} = \frac{1}{\sqrt{x}}$$



$$n(x = 0.01) = 10$$

$$n(x = 0.001) = 31.62$$

$$SU(n=10) = \frac{1}{1+0.99\left(\frac{1}{10}+0.01\cdot 10-1\right)} = \underline{\underline{4.81}}$$

$$SU(n=32) = \frac{1}{1+0.99\left(\frac{1}{32}+0.01\cdot 32-1\right)} = \underline{\underline{13.77}}$$

c) 1,000 parallelizable ops, 10 serial ops

$$\left.\begin{array}{l} 1{,}000 \text{ parallelizable ops} \\ 10 \text{ serial ops} \end{array}\right\} \quad p = 1 - \frac{10}{1{,}010}$$

$$p \approx 0.99$$

$n = 10$

Execution time for every op is $t = t_1$.

- Execution time with parallel ops

$$T(n) = \frac{O_p}{n} t_1 + O_s \cdot t_1 \qquad (O \dots \text{ops})$$

$$\left\| T(n) = \left(\frac{O_p}{n} + O_s\right) t_1 \right\|$$

- Compute speed up

$$T_{old} = (O_p + O_s)\, t_1 \quad \dots \text{ all ops serial}$$

$$SU = \frac{T_{old}}{T_{new}} = \left(\frac{\left(\frac{O_p}{n} + O_s\right) \cancel{t_1}}{(O_p + O_s)\cancel{t_1}} \right)^{-1} = \left(\frac{\frac{1{,}000}{10} + 10}{1{,}000 + 10} \right)^{-1}$$

$$\underline{\underline{SU = 9.18}}$$

$$\text{Amdahl}: \; SU = \frac{1}{1 - 0.99 + \frac{1}{10} \cdot 0.99} = \underline{\underline{9.18}} \quad ✓$$

- Speed up for imbalanced load. One core takes 1.5 / 3.0 times the load of all the other cores.

The total number of parallelized ops is:

$$O_{p,tot} = (n-1)\frac{O_p}{1 \text{ core}} + 1 \frac{1.5\, O_p}{1 \text{ core}}$$

where O_p is the number of ops of 1 core.

$$O_p = \frac{O_{p,tot}}{(n-1) + 1.5} = \frac{1{,}000}{(10-1) + 1.5} = 95.238$$

$O_p \approx 95$ (round down to guarantee the factor 1.5)

$O_p (\times 3) \approx 83$

x 1.5:

Ops on the 1.5 core = 1,000 - 9 · 95 = 145

x 3.0:

Ops on the 3.0 core = 1,000 - 9 . 83 = 253

$SU(x\,1.5) = \frac{1{,}010\,\cancel{h}}{145\,\cancel{h}} = \underline{\underline{6.97}}$ $(< 9.18\ ✓)$

$SU(x\,3.0) = \frac{1{,}010\,\cancel{h}}{253\,\cancel{h}} = \underline{4.0}$ $(< 6.97\ ✓)$

Question 2: Linear Algebra Operations

Question 2a) Matrix Vector Multiplication

The following two figures Figure 1 and Figure 2 display the results of matrix vector multiplications of square matrices (NxN) of different sizes. The blue bars indicate matrices stored in row major order whereas orange bars represent matrices stored in column major order. Figure 1 contains a subset of the computed matrix results of Figure 2 in order to illustrate the difference in computation time for smaller matrix sizes more clearly.

The obtained results in the following figures clearly reveal that matrices stored in row major order perform significantly better than matrices stored in column major order when multiplied with a vector for increasing matrix sizes. The reason for the obtained results can be found looking at spatial locality and the number of cash hits resulting from it. If a matrix row is traversed at a stride size of k=1, it can be fully exploited as the next elements called for the multiplication have already been loaded into the L1 cache. Therefore, these matrix elements do not need to be fetched from lower level of the memory hierarchy. In case of a 64 byte cache line, the loading of one integer array element from a lower level in the memory hierarchy into the L1 cache causes the next 15 (60 bytes) elements to be loaded into L1 as well. These 15 elements can then be directly fetched from the L1 cache for the multiplication speeding up the performance significantly.

Considering the difference in calculation time between the matrix vector multiplications of row major order and column major order stored matrices, it becomes clear that this difference grows exponentially with increasing matrix dimensions. On the one hand, a matrix of size 512x512, for instance, causes 32 cache misses for one row (512 row elements / 16 elements per cache line) resulting in 16,384 total cache misses for this matrix size (32 cache misses for every row). On the other hand, working with a matrix of the same size stored in column major order causes 262,144 (512 rows * 512 cols) cache misses. Traversing the matrix column-wise makes no use of spatial locality at all, because all elements loaded into L1 with the target element are never made use of, i.e. fetched from the L1 cache. With growing matrix dimensions the number of cache misses grows exponentially resulting in very low performance for column major order stored matrices. Furthermore, huge arrays cannot be stored in any of the L1, L2, L3 aches or RAM anymore and need to be fetched from the SSD. As fetching data from SSD is several orders of magnitude slower than fetching it from RAM or a cache, every single load or store process is a tremendous performance loss.

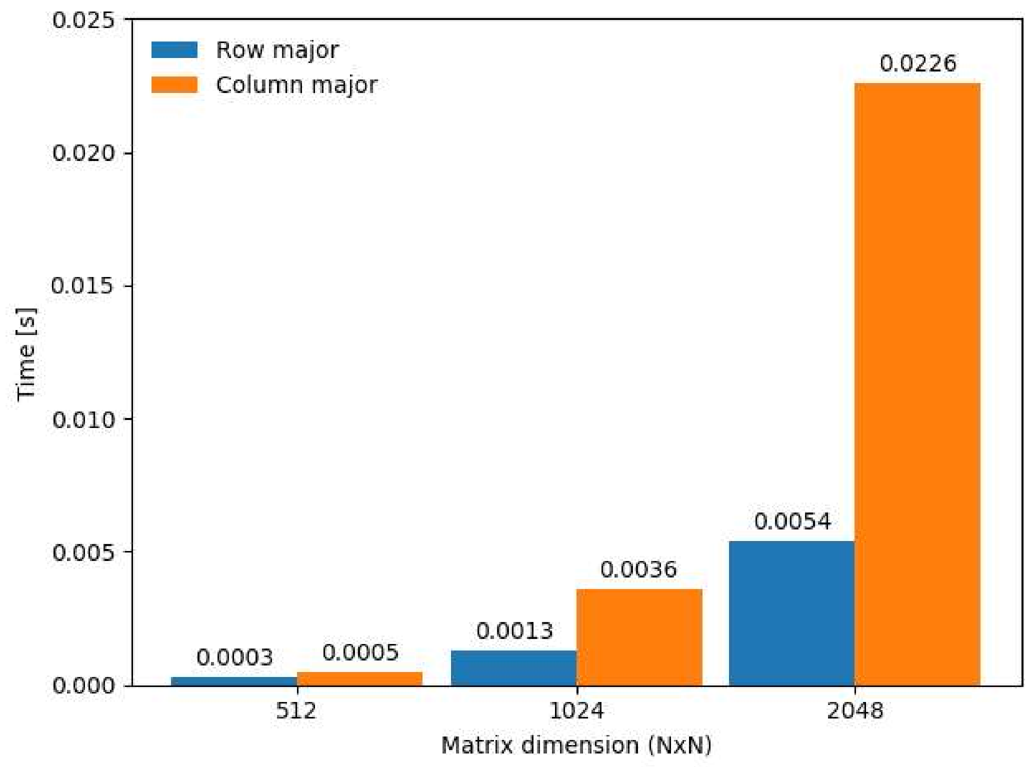


Figure 1: Computation time for matrix vector multiplication of square matrices (NxN) of different sizes. Row major order matrix multiplication is contrasted to column major order multiplication. Zoomed in view of Figure 2. All computations were run 10 times and the average time was computed as final result and is provided in this figure.

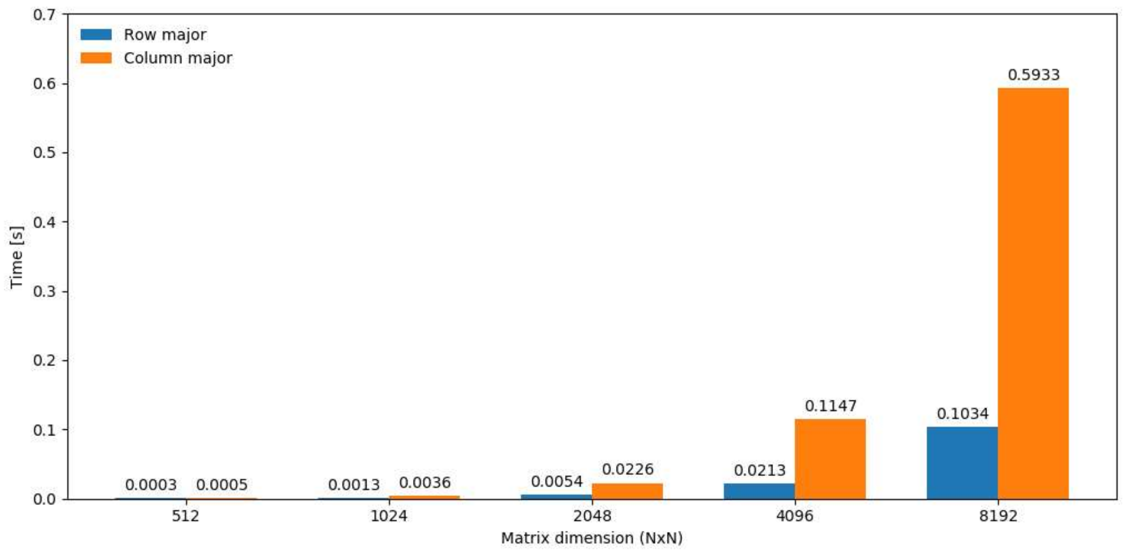
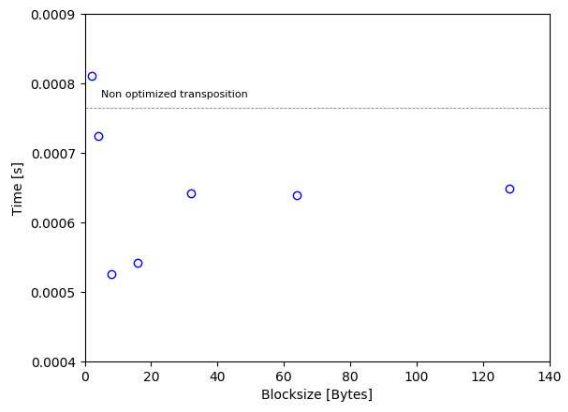


Figure 2. Computation time for matrix vector multiplication of square matrices (NxN) of different sizes. Row major order matrix multiplication is contrasted to column major order multiplication. All computations were run 10 times and the average time was computed as final result and is provided in this figure.

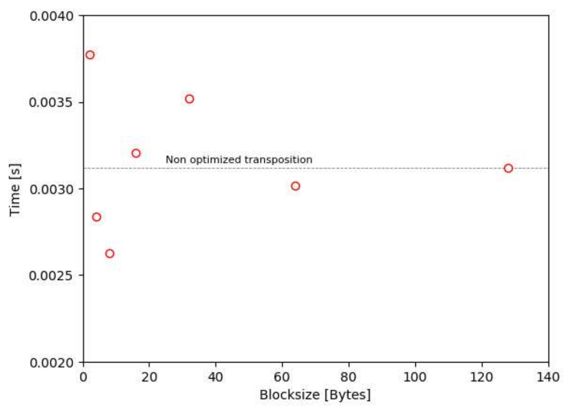
Question 2b) Matrix Transposition

Using too small blocks for the matrix transposition does not generate the ideal efficiency as the spatial locality is not fully exploited. However, too big blocks do not yield optimum performance either as not all block elements fit in the L1 cache at the same time anymore. Therefore, more capacity misses result and diminish performance.

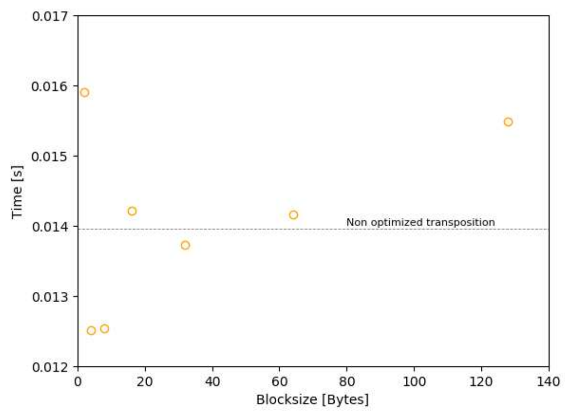
(a) 256x256



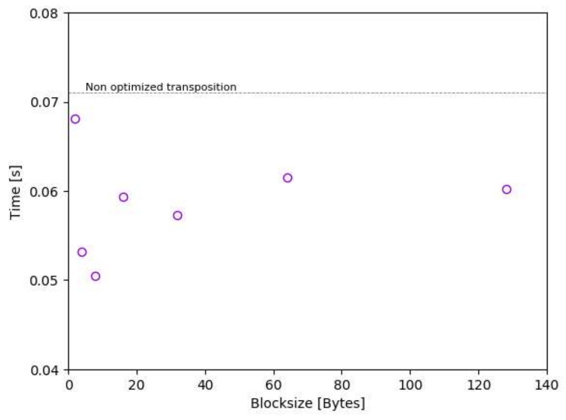
(b) 512x512



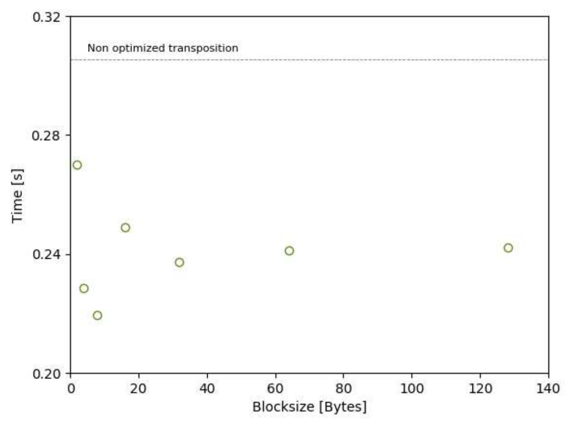
(c) 1024x1024



(d) 2048x2048



(e) 4096x4096



(f) 8192x8192

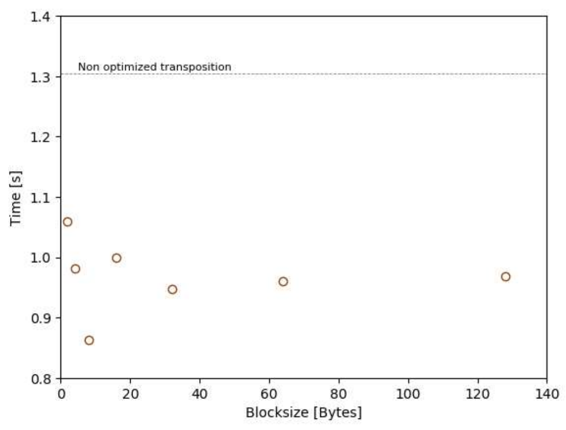


Figure 3: Square matrix transposition for different matrix dimensions. Each figure maps the computation time against the used block size chosen for the algorithm. The following block sizes were used for the tests: 2, 4, 8, 16, 32, 64, and 128. A horizontal reference line is provided in each plot that indicates the performance of the traditional algorithm.

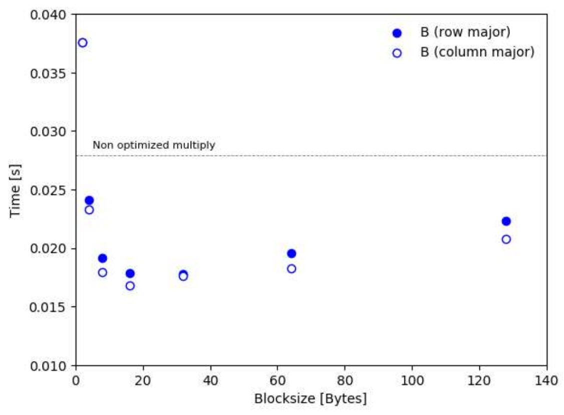
Question 3b) Matrix Matrix Multiplication

Figure 4 indicates that too small block used for the block algorithm do not provide the maximum efficiency. A block of size 2 results in an even worse performance than the traditional algorithm. The problem with small blocks is that the algorithm does not fully exploit the spatial locality. A cache line of size 64 bytes stores 8 double values, therefore, a block size below 8 refrains from using all the blocks that were loaded into the L1 cache. This results in a performance decrease.

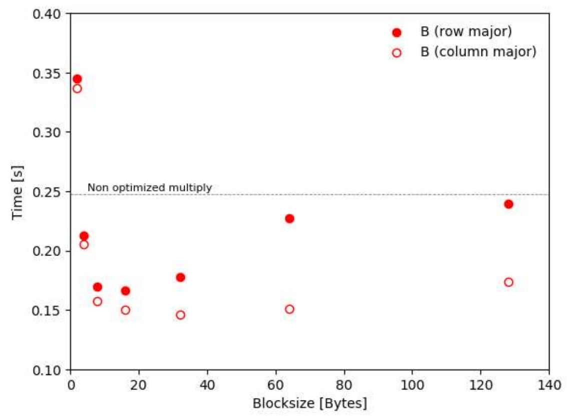
According to the results below, the ideal block size is 16. Block sizes of 64 bytes and 128 bytes result in more cache misses than necessary as a block cannot fit entirely in the L1 cache anymore. Thus, more capacity misses will result and cause a lower performance. For block sizes of 32 bytes and below the whole block (for each matrix) fits in the L1 cache and allows for temporal locality during its entire computation.

Comparing the block algorithms of B stored in row major order to B stored in column major order the plots below reveal that storing B in column major order provides a performance benefit. When B is stored in column major order the elements that are multiplied with a row of matrix A are directly below each other. This is effectively multiplying a row of matrix A with a row of matrix B. Traversing these two rows with a stride size of k=1 fully exploits spatial locality really well and makes the block algorithm with B stored in column major order more efficient.

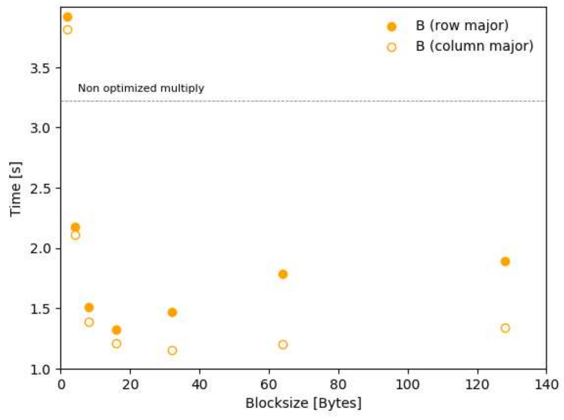
(a) 256x256



(b) 512x512



(c) 1024x1024



(d) 2048x2048

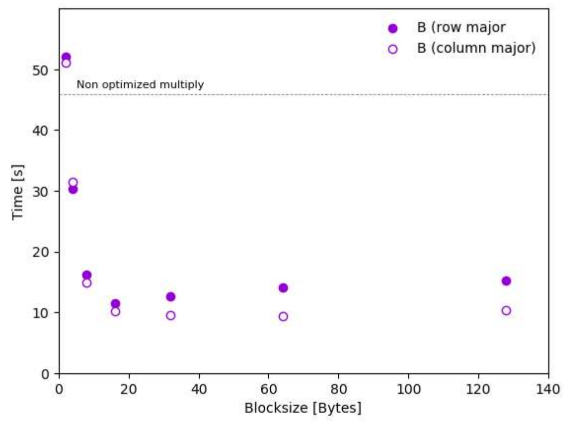


Figure 4. Matrix matrix multiplication of square matrices of different dimensions. Each figure maps the computation time against the used block size chosen for the algorithm. The following block sizes were used for the tests: 2, 4, 8, 16, 32, 64, and 128. A horizontal reference line is provided in each plot that indicates the performance of the traditional algorithm.

Question 3: Cache Size

Note: As my code did not produce the correct outputs on the Euler cluster, but on my personal machine, and as TAs have not been allowed to comment on this specific problem, I decided to use the results produced on my own machine. These results come quite close to the hint sketch provided in the handout.

Question 3a)

The following table presents the total size (size) and cache line size (coherency line size) of the different cache levels of the node on the Euler cluster. The command used to obtain the following values is:

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

Table 3.1: Cache information of Euler cluster node.

Cache level	Size [KB]	Coherency line size [B]
L1	32	64
L2	256	64
L3	6144	64

Question 3b) Variant 1

The task was to create an array of different sizes, which contain 4 byte integer elements. This array was then stepped through according to Sattolo's algorithm, which makes sure that each element in the array is hit at least once. The results are displayed in the graph below and labelled Variant 1 (blue data points).

The plot clearly indicates the performance differences between the three cache levels. Arrays with a size below 32 KB, which is the cache level 1 size on this machine, have the highest performance as this cache level is closest to the CPU and considerably the fastest cache hardware of the three different cache level. Hence, all data points below the size of 32 KB have similar performance due to temporal locality, that is, elements that have been cached and are used at some time in the near future. As long as all elements can fit in the L1 cache there is no difference in performance as the amount of data stored has no impact on how fast elements can be read from the cache.

Considering the transition from cache level L1 to L2, which is exactly at 256 KB, it is to say that the results match the expectations. As the array does no longer fit into the cache level L1, it is cached in level L2. Level 2 is further away from the CPU and slower than L2 (reading from L2 takes about 10 clock cycles, whereas reading from L1 takes only 2-3 clock cycles). A similar behaviour can be observed taking the transition from cache level L2 to L3 into consideration. As it takes about 40-50 clock cycles to read data from the cache L3, another performance drop is expected.

The transitions from one cache level to another are expected to be relatively steep as all the elements are all of a sudden loaded into a higher cache level increasing the time it takes to fetch elements from the cache. As far as the transition from L2 to L3 is concerned, this transition can be considered as sharp as the first 5 data points are within a range of 130.2 KB. The other 16 data points are within a range of 1520.3 KB (note the logarithmic scale of the x-axis). With regards to the transition from L1 to L2, it is to say that this transition is still quite sharp, but smoother compared to the transition between the L2 and L3 cache. One factor that can play a key role in this transition is spatial locality, that is, elements are loaded into cache that are placed near the target element in the memory and are used in the near future. Smaller array sizes increase the likelihood that spatial locality can be exploited when stepping through the array with randomly chosen step sizes. Other factors that can affect the drop are cache mapping and other processes currently running on the machine.

Question 3c) Variant 2

Variant 2 (red data points) displays the performance results mapped against the array size for a constant step size of one, i.e. stepping from array index k to array index k+1. This step size can be considered as the ideal one according to spatial locality. Whenever a cash miss is detected and new elements need to be loaded into the L1 cache, the next 15 elements are loaded as well. Due to a step size of one, these 15 elements are all cash hits and allow a perfect exploitation of spatial locality. The plot below shows that there is no noteworthy performance drop when elements do not fit in one cache level anymore. The reason is that spatial locality comes to the rescue and amortizes the performance loss because of a longer data fetching time-span. The memory mountain for Intel core i7 processors also indicates that there is almost no performance loss at a stride size of k=1. These processors are cleverly designed to understand that a stride size of k=1 is used and already load the required data into cache before it is actually accessed. This strategy boosts the performance tremendously.

Question 3d) Variant 3

In this Variant the step size is set to k=16 elements, that is, 64 bytes. As the cache line size is 64 bytes, too, the concept of spatial locality is not applicable as neighbouring elements in memory that are fetched into L1 are not used by the algorithm.

The lower performance of this variant for array sizes that fit in the L1 cache, compared to Variant 1 and Variant 2, cannot be a result of temporal locality. As all array elements are stored in L1 it is not a matter of when these elements are accessed. Maybe spatial locality plays a role here. The memory mountain in reference [1], however, displays a steady performance loss for the L1 cache for Intel core i7 processors with an increasing stride size. The book does not further explain why this is the case.

The performance drop between L1 and L2 is very sharp as one would expect due to the longer time it takes to fetch elements from L2 compared to fetching data from L1. It is noteworthy to mention that this performance drop is sharper than the one of Variant 2. As Variant 3 cannot exploit spatial locality at all due to its step size of 64 bytes, no performance gain can be obtained for the smaller array sizes in this cache level. The performance for arrays of sizes that fit entirely in the L2 cache can be claimed as constant with regards to Figure 5.

It is important to mention that the performance behaviour of arrays that are stored in the L3 cache is quite unintuitive at first glance, as no performance drop can be observed although the elements are now loaded into the L3 cache, which is significantly slower than L2. However, performance degrades continuously with an increasing arrays size. Spatial locality cannot cause the unexpected behaviour, as stepping through the array at a size that matches the cache line size allows no exploitation of it. Temporal locality should perform less well in L3 than in L2 as it is simply hardware dependent in this case, however, as described in reference [1], stride sizes of k=1 are dealt with extremely cleverly by the Intel i7 processors; maybe there is a similar behaviour if a constant stride of 64 bytes is detected.

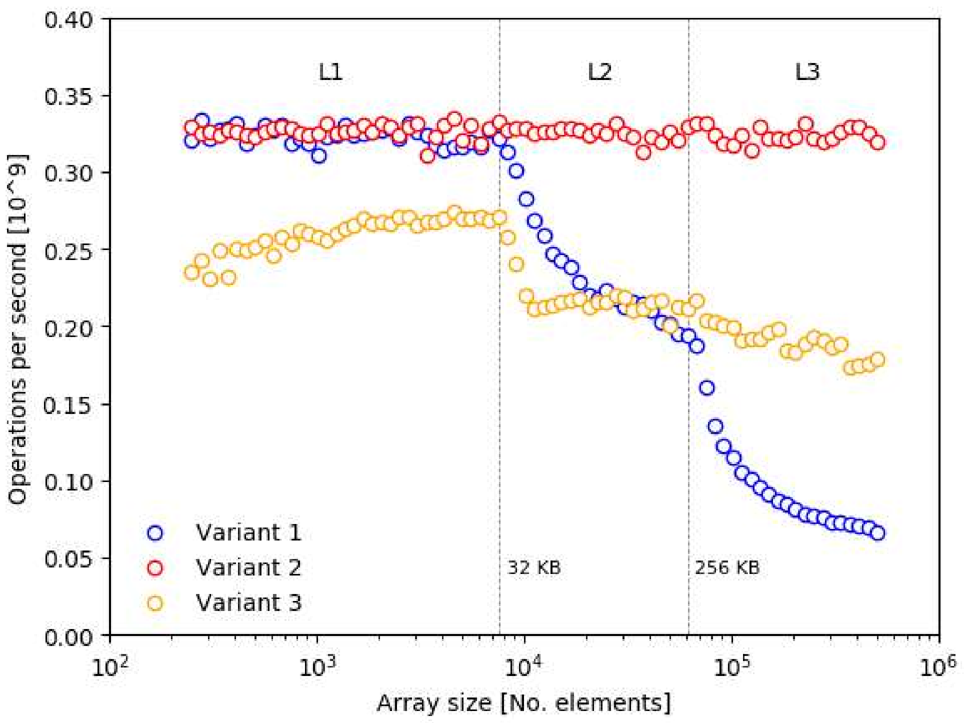


Figure 5: Maps the operations per second against the array size (array has integers of size 4 bytes as its entries) for different step sizes moving inside the array. **Variant 1:** Step size is computed randomly by Sattolo's algorithm. **Variant 2:** Step size is one. **Variant 3**: Step size is 16 (equal to 64 bytes, which is also the size of a cache line).

The following figure illustrates the memory mountain of an Intel core i7 processor that is referred to in the text above. The image is taken from reference [1].

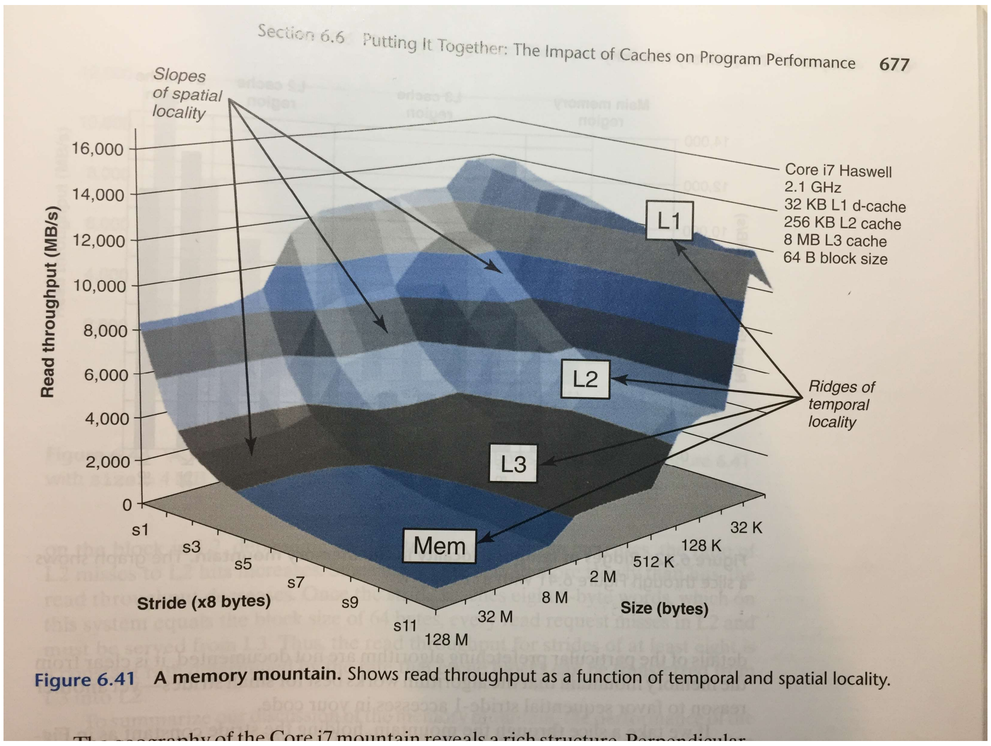

Section 6.6 Putting It Together: The Impact of Caches on Program Performance 677

Figure 6.41 A memory mountain. Shows read throughput as a function of temporal and spatial locality.

Figure 6. Memory mountain of an Intel core i7 processor [1].

Appendix

References

1. Randal E. Bryant, David R. O'Hallaron, "Computer Systems A Programmer's Perspective", Pearson Education Limited, Edinburgh Gate, Harlow, 2016