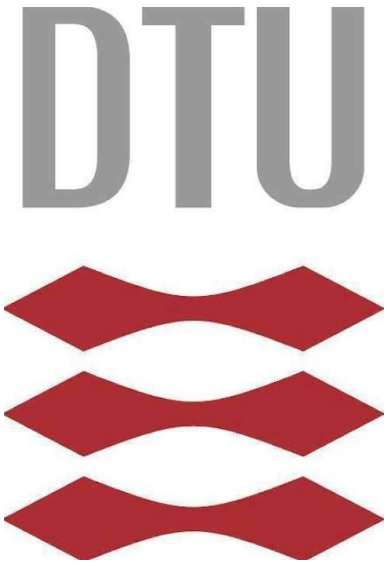
Technical University of Denmark



Markus Sørensen - s183997
Oliver Kinch - s183980

02614 - High Performance Computing

Assignment 1

January, 2023
Technical University of Denmark

Addendum: Individual responsibilities in this assignment

The following table shows who has the main responsibility for the different parts in this project:

Part	studyno.	Name
I.	s183980	Oliver Kinch
II.	s183997	Markus Sørensen
III.	s183997	Markus Sørensen
IV.	s183980	Oliver Kinch

1 I

1.1 Theory

First, we implement the matrix multiplication as it is represented in the assignment, e.g. iterating over m, then n, and then k.

To make the library function of the matrix multiplication we have used the $DGEMM()$ function from the CBLAS library provided by ATLAS. For the arguments, we have used $CblasRowMajor$, as data in C is stored by row-major order. We have chosen $CblasNoTrans$, such that the matrices are not transposed. We have chosen $\alpha = 1$ and $\beta = 0$, such that

$$C = \alpha * A * B + \beta * C = A * B$$

The remaining inputs come directly from the function wrapper.

1.2 implementation/experiments

The two implementations can be seen below.

```
1   void matmult_nat(int M, int N, int K, double **A, double **B, double **C) {
2       for (int m = 0; m < M; m++) {
3           for (int n = 0; n < N; n++) {
4               C[m][n] = 0.0;
5               }
6           }
7       for (int i = 0; i < M; i++) {
8           for (int j = 0; j < N; j++) {
9               for (int k = 0; k < K; k++) {
10                  C[i][j] += A[i][k] * B[k][j];
11              }
12          }
13      }
14  }
```

```
1   void matmult_lib(int m, int n, int k, double **A, double **B, double **C) {
2       cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, 1.0, *A
    , k, *B, n, 0.0, *C, n);
3   }
```

Testing the two functions with the driver tools provided we can see that they both calculate the result correctly. Comparing the two functions we see that the library function is way faster than our own implementation.
aloha: vis at de faktisk beregner rigtigt?

1.3 results/discussion

We got the correct result with our implementations which was expected. It was also expected that the library function would be much faster than our implementation as it optimizes cache usage.

2 II

2.1 Theory

There are $3 \cdot 2 \cdot 1 = 6$ different permutations. The output matrix C will be the same for every permutation, but the compute time will be different. The fastest compute time will be the

permutation that minimizes the number of cache misses. As C stores arrays row-wise, it is expected that the permutations that iterate over n in the last loop will have the fewest cache misses.

2.2 implementation/experiments

To determine which of the permutations is the best we have used square matrices with different sizes and tested the performance of each permutation on these.

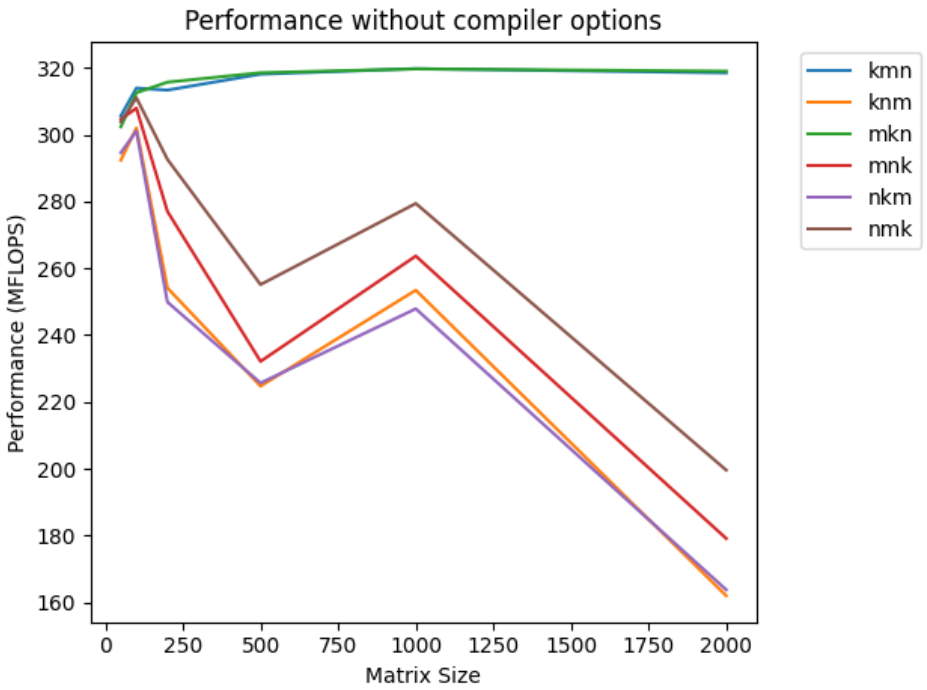


Figure 1: Matrix multiplication permutation tested on different matrix sizes.

As expected, it is seen that the permutations that iterate over n in the last loop perform the best. It is also seen that these permutations are not very sensitive to the matrix sizes - compared to the other implementations that have L1, L2, and L3 cache misses. Therefor we make the plot looking at the memory usage instead of the sizes and get the following result.

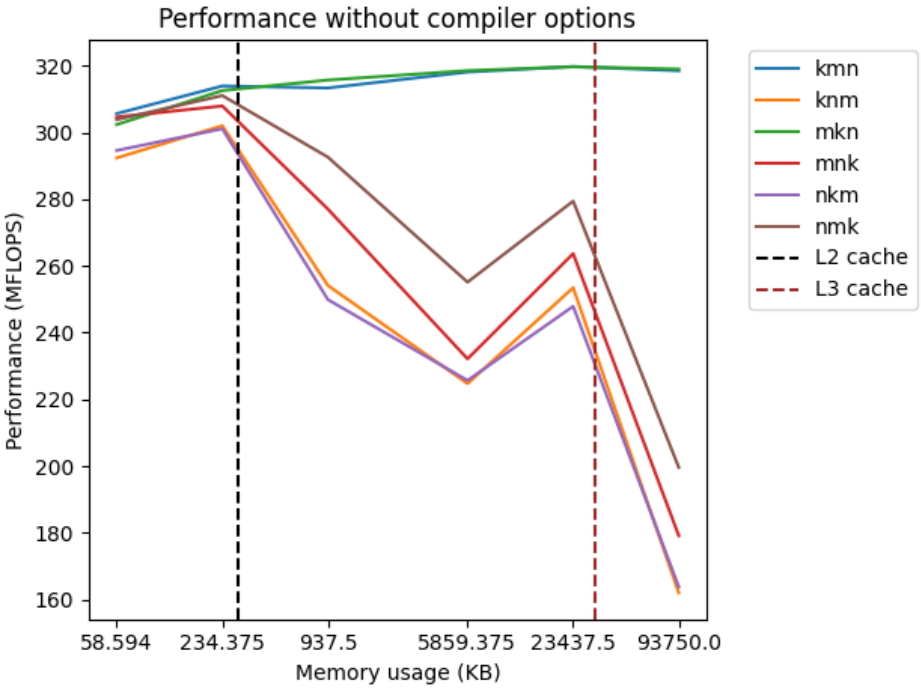


Figure 2: Matrix multiplication permutation tested on different matrix sizes, plotted with memory usage and L2 and L3 cache lines.

We see a clear trend that the performance of the permutations not ending on n, decreases significantly when a cache line is crossed.

The performance of all permutations could potentially be improved by choosing some good compiler flags. First, we add $-O3$ which gives the following results

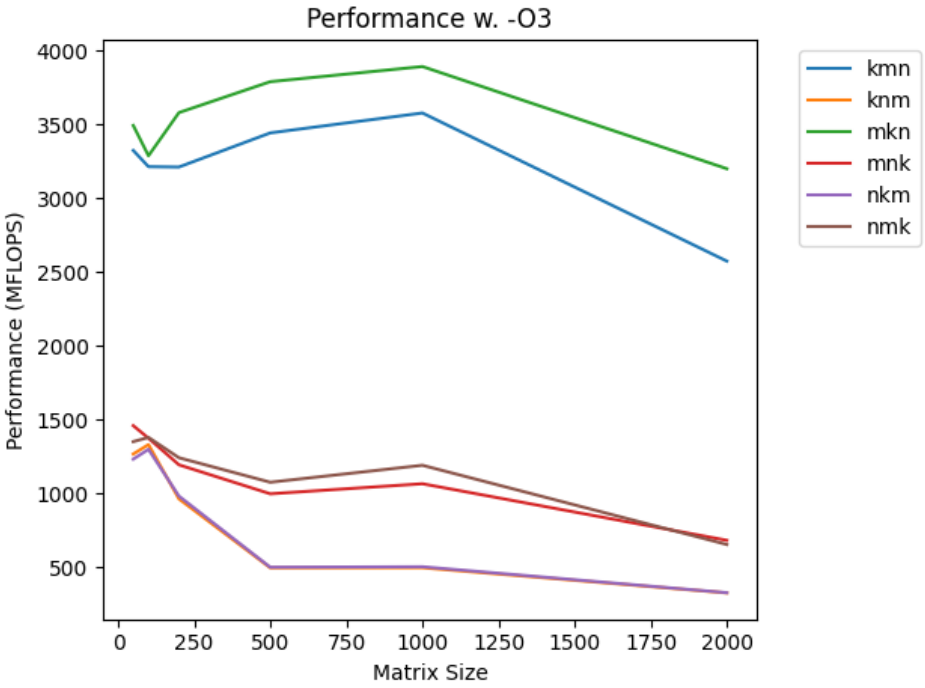


Figure 3: Matrix multiplication permutation tested on different matrix sizes, with the -O3 compiler flag.

This increased the performance by a factor of around 5. We also try to add $-funroll-loops$, which gives the following results

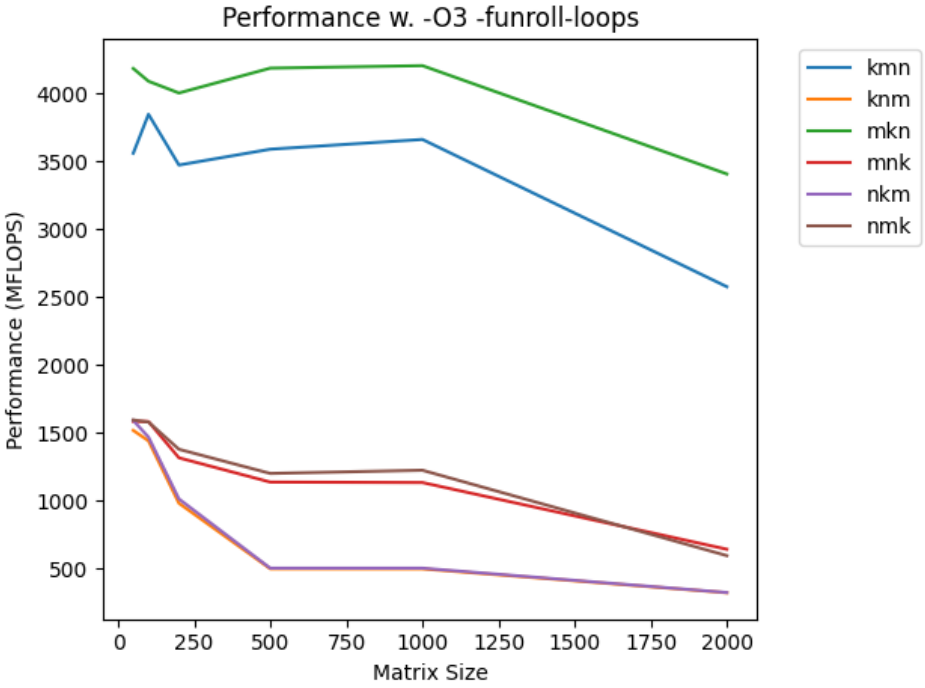


Figure 4: Matrix multiplication permutation tested on different matrix sizes, with the -O3 -funroll-loops compiler flags.

This additionally increased the performance of all permutations. We also tried to add $-flto$,

which gives the following results

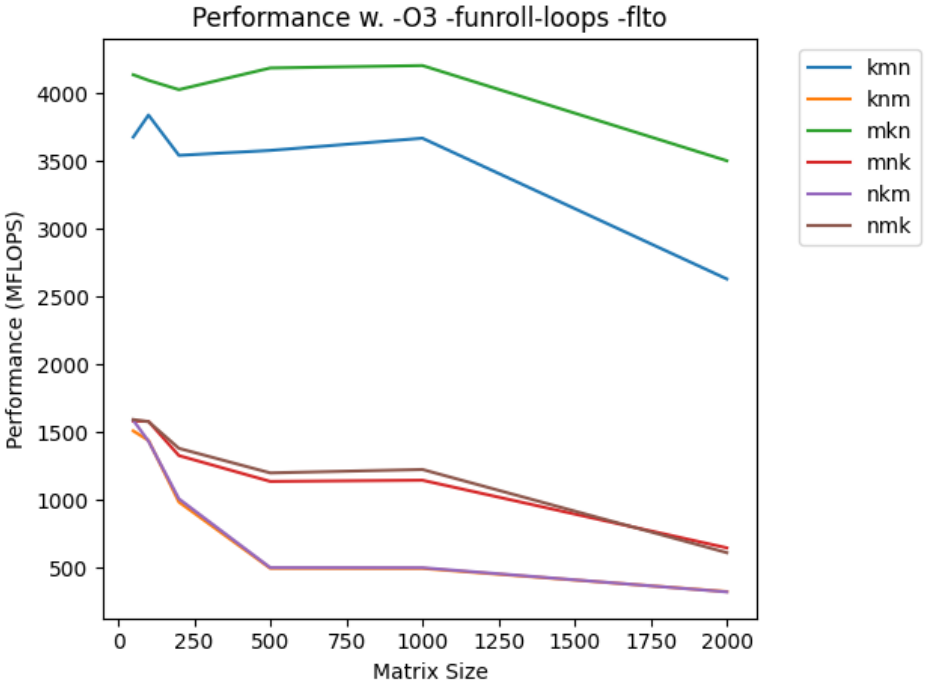


Figure 5: Matrix multiplication permutation tested on different matrix sizes, with the -O3 -funroll-loops -flto compiler flags.

This is seen to not increase performance and we choose not to keep it. Then we try the $-march = native$ compiler flag.

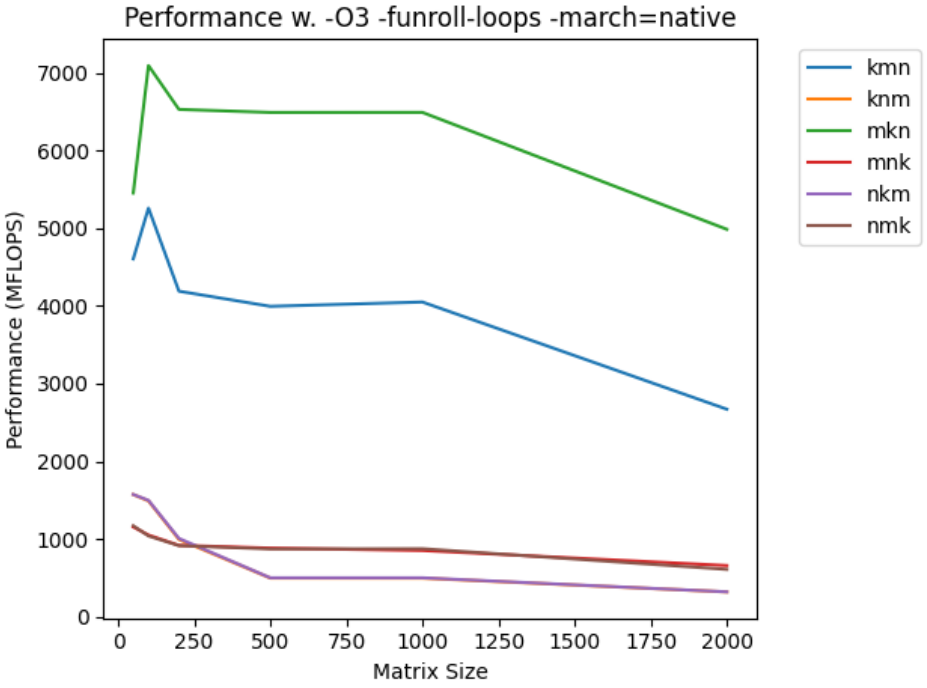


Figure 6: Matrix multiplication permutation tested on different matrix sizes, with the -O3 -funroll-loops -march=native compiler flags.

This is seen to improve performance of the best performing permutations but decrease the performance of some of the others. We choose to keep it as we do not care about the bad performing permutations as we only need one good method. We also try the $-mfpmath = sse$ which gives the following results

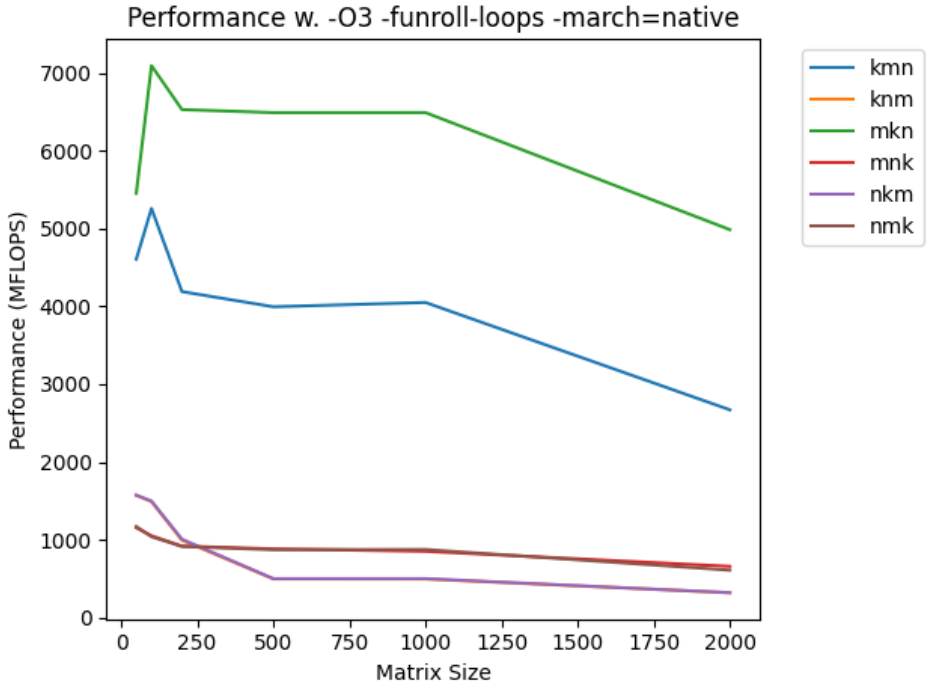


Figure 7: Matrix multiplication permutation tested on different matrix sizes, with the -O3 -funroll-loops -march=native -mfpmath=sse compiler flags.

We do not see an increase in performance and we choose not to keep it. We see that the best-performing method is the *mkn* method which has a major improvement in performance from the compiler flags added. It is however still not performing as well as the library implementation

2.3 results/discussion

It has been seen that the mkn version is the fastest permutation, which is not surprising as C arrays are stored row-wise. E.g. by iterating over the rows of the matrices first, we ensure that we are accessing elements that are close to each other in memory, which reduces the number of cache misses and leads to better performance.

For the compiler option, we found that $-O3$ gave a huge improvement in performance. This was also expected as it adds a whole set of tools that the compiler can use. In our case, it does loop vectorization, which makes the compiler operate on multiple elements in the loop simultaneously, but also adds many other optimization techniques which probably also impact the performance positively.

For the $-funroll-loops$ it adds loop unrolling which reduces the overhead of the loop by manually expanding the loop body and performing multiple iterations in a single pass. This also allows the compiler to schedule the iterations of the loop more efficiently.

The $-flto$ flag did not improve performance, but this was also expected as one of the main features of this is to be able to see the whole code of a complicated program at once, which makes it able to move code around, not run unused variables etc. Our code is however so simple that this does not have any impact.

The $-march = native$ flag did improve performance of the best permutations but decreased the performance of some of the worst permutations. The flag enables optimization targeting the specific processor, and it makes sense that the performance could increase by adding this. A reason that it may decrease performance is that the worse permutations can not use this because of their structure and it, therefore, decreases their performance.

The $-mfpmath = sse$ flag is seen to not give any improvement. This flag makes the compiler use the SSE method to perform floating-point arithmetic. The reason that this does not improve performance might be that e.g. the $-O3$ flag already used some optimization methods for floating point arithmetic.

3 III

3.1 Theory

A good performance measure to check for effecient code is to look at cache misses and hits. As explained earlier a cache miss is when the system looks for some data in the cache but is unable to find it, and therefor has to look further out in memory.
We will use the cache hits and misses to calculate the cache hit ratio

$$Cache\_hit\_ratio = \frac{Number\_of\_cache\_hits}{Number\_of\_cache\_hits + Number\_of\_cache\_misses}. \quad (1)$$

If a program finds all the data needed in a specific layer of the cache the cache hit ratio will be 1. Therefor a result coming as close to 1 as possible is best.

3.2 implementation/experiments

To analyze the number of cache hits and misses for the different permutations we have used the Oracle Developer Studio Analyzer and a matrix size of $500x500$. From the analysis we got the cache hits and misses, which we used to calculate the cache hit ratio. These can be seen for the library implementation and all permutations in the table below

Permutation	L1 Hit Ratio	L2 Hit Ratio	L3 Hit Ratio
lib	0.9935	0.9732	1.0
mkn	0.9014	0.9390	1.0
kmn	0.9065	0.6886	1.0
mnk	0.6224	0.9390	1.0
nmk	0.6320	0.9971	1.0
nkm	0.5093	0.9406	0.9989
knm	0.5378	0.9403	0.9988

Here we clearly see why the *lib* method is best followed by the permutation ending with n, then k and at last m, as these have the highest ratio in L1 cache.

3.3 results/discussion

When looking at the results above it makes perfect sense why none of our permutations come close to the *lib* method in performance, as this method has over 99% hit ratio in L1 cache. When looking at our permutations we see that those ending with n come closest to the library implementation in the L1 cache hits with just above 90%. Comparing the two methods ending on n we see that mkn has a much higher L2 hit ratio compared to kmn, but this also makes sense as it iterates over k second which is the second fastest of the three. Therefor it also makes sense that this method performs the best of the permutations and that the difference in performance between the permutations increases as the size of the matrices increases. For the rest of the permutations we see that the methods iterating over k to start with are better than those iterating over m. This is also expected as iterating over m to start with needs a lot of cache. These are also the only method with L3 cache misses. The permutation with the highest L2 cache hit ratio is the nmk method, which is a little surprising, but might be due to the specific matrix size.

4 IV

4.1 Theory

To improve the performance of the matrix multiplication function, we can use blocking, a technique that involves breaking the matrices down into smaller blocks that will fit in the cache. This can reduce the number of cache misses, as the blocks used in the computation will stay in the cache for a longer time.

4.2 implementation/experiments

For the blocked version, the mkn version is used as the starting point, as it has been seen to be the best-performing permutation.

The following plot shows the performance of the blocked version as a function of block size, for three different square-matrix sizes.

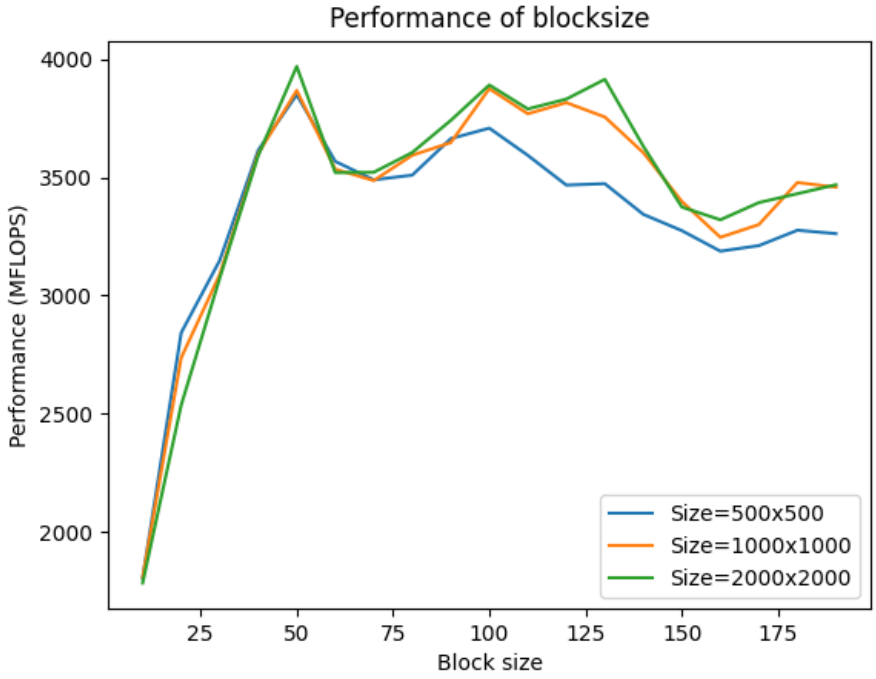


Figure 8: Performance as a function of block size for three different sizes of square matrices

It is seen that the optimal block size varies depending on the matrix sizes. However, for this experiment, it is seen that a block size of 50 is an approximate optimum for all three matrix sizes. In general, larger block sizes may be more effective for larger matrix sizes, as the increased size allows for more data to be stored in the cache and reused. However, for very large block sizes, the overhead of the loop may become significant, leading to a decrease in performance.

The following plot shows how the blocked version (with a block size of 50) performs compared to the mkn and lib versions.

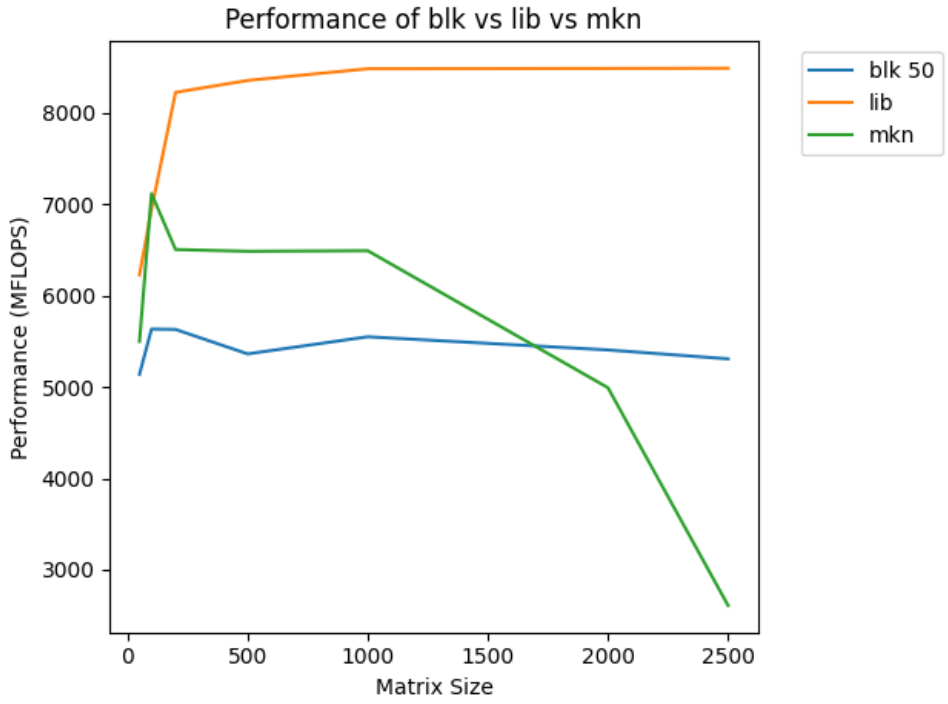


Figure 9: Performance of the blocked version compared to non-blocked and lib

It is seen that the blocked version, compared to the mkn version, is very insensitive to the matrix size. The mkn version's performance decreases with respect to the matrix sizes, and it is seen that the blocked version is faster than the non-blocked version, for matrix sizes larger than $\sim 2000 \times 2000$. The blocked version is still not nearly as fast as the lib version.

4.3 results/discussion

Does blocking improve the performance, and where do you expect and get the largest effect?

In general, blocking can be especially effective on processors with small cache sizes or when multiplying large matrices. However, there is still a trade-off between the benefits of reducing cache misses and the overhead of using blocking. Therefore, it is expected (and also seen in the experiments), that blocking improves performance when operating with very large matrices.

Can it be faster than the fastest non-blocked version, and if yes, for which sizes of the block size and which matrix sizes?

Yes, it is possible for the blocked version of the matrix-matrix multiplication function to be faster than the fastest non-blocked version. Our experiments have shown that blocking with a block size of 50 improves the performance for matrices larger than $\sim 2000 \times 2000$.

Hardware

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
Stepping:              1
CPU MHz:               2200.000
CPU max MHz:           2900.0000
CPU min MHz:           1200.0000
BogoMIPS:              4389.48
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-11
NUMA node1 CPU(s):     12-23
```

Figure 10: Computer architecture