Technical University of Denmark

High-Performance Computing

Course 02614

Assignment 1

Authors:
Oskar Hint, s161559
Mikkel Jensen, s123184
Philip Rasmussen, s103124

January 6, 2017

Contents

1 Introduction

The goal of this assignment is to develop different matrix-matrix multiplication functions and to analyze and compare their performance.

1.1 Computing Specifications

Unless otherwise stated a lot of the specifications are fixed throughout the report. The code is written in c++ and compiled with the Sun Studio compiler. We therefore use the matmult_c.studio driver. All the experiments are run on the hpcintro cluster on a single node with one processors per node (ppn), which is specified in the submit.sh script. All the nodes on the cluster are identical. The makefile used is the supplied sunCC makefile with CXX = sunCC for c++ modified for the different cases.

2 Assignment

2.1 The Native Function

First, the goal is to write a native function, which performs a matrix-matrix multiplication. The shape has to be suitable for the operation, but it is arbitrary within the limits. It is defined based upon the dimensions m, n and k, as seen in figure 1, and the matrices A and B computed by the driver.

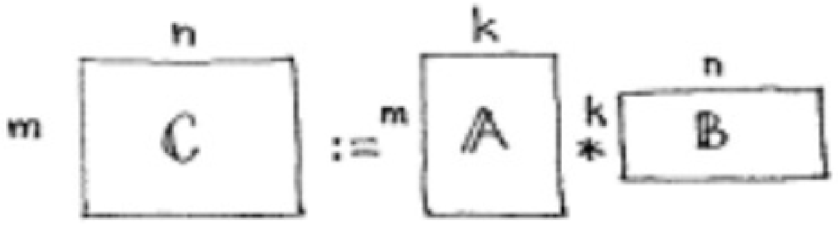


Figure 1: Arbitrary matrix dimensions given in the assignment.

To create the code for such an operation, the native way is chosen, where the matrix is described using double pointers, i.e. A[i][j]. The choice of function prototypes and driver therefore follows, which is described in the given README file.
The general function prototype used is displayed below:

Listing 1: Function Prototype

```
void matmult_NNN(int m, int n, int k, double ** A, double ** B, double ** C)
```

The function prototypes are all declared in the header file ass1_lib.h. The functions themselves are described in the ass1_lib.cpp file, where the ass1_lib.h library is included.

The native function takes the integer arguments m, n and k, as well as the double arguments **A, **B and **C. The double arguments are the actual matrices computed, which in the case of matrix A and B are generated within the supplied driver based on the submitted shape parameters. A dynamic memory allocation function is likewise implemented for the three matrices. In the case of C, only the memory is allocated. To perform the matrix-matrix multiplication 3 nested for loops, one for each of the 3 shape parameters, are required. Three loop variables i, j and l are introduced for the matrix leading dimensions m, n and k. The C matrix has the leading dimensions of m and n, C[i][j]. The function is displayed below:

Listing 2: Function Prototype

```
void matmult_nat(int m, int n, int k, double ** A, double ** B, double ** C){
        for(int i = 0; i < m;i++){
                for(int j = 0; j < n;j++){
                        C[i][j] = 0;
                        for(int l = 0; l < k;l++){
                                C[i][j] += A[i][l]*B[l][j];
                        }
                }
        }
}
```

Notice the += sign, which is caused by the fact, that when the function loops over l for a set (i,j), the multiplications for the different values of l are added together.

2.2 DGEMM and Comparison

For comparison reasons, the important subroutine for matrix matrix multiplication DGEMM (Double Precision General Matrix Matrix) from the BLAS library is implemented in our system. The blas.h library is therefore included by using external C. The library function is implemented taking the same arguments as the nat function.

Listing 3: lib

```
void matmult_lib(int m, int n, int k, double ** A, double ** B, double ** C){
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
  m, n, k, 1.0 , A[0], k, B[0], n, 0.0, C[0], n);
}
```

Comparing the two functions clearly display the limits of the native function. Running the two function on the cluster for a few different matrix sizes.

Table 1: This table displays the performance (MFlop/s) of the two functions matmult_nat() and matmult_lib() with a square matrix size of 512x512 and 2048x2048.

m, n, k dimension	nat() MFlop/s	lib() MFlop/s
512	529.2	20767.1
2048	407.2	21521.5

2.3 The Permutations

Since the algorithm consists of three nested loops iterating over the three dimensions m, n and k, it is clear through basic combinatorics, that there is 3! = 6 possible ways to order the loops:
mnk - mkn - kmn - knm - nmk - nkm
The 6 permutations are implemented as separate functions in the library with names matmult_NNN. The functions are obviously very similar to the already described nat function, but the function are however displayed below:

Listing 4: lib

```
void matmult_mkn(int m, int n, int k, double ** A, double ** B, double ** C){
        for(int i = 0; i < m;i++){
                for(int j = 0; j < n;j++){
                        C[i][j] = 0;
                }
        }


        for(int i = 0; i < m;i++){
                for(int l = 0; l < k;l++){
                        for(int j = 0; j < n;j++){
                                C[i][j] += A[i][l]*B[l][j];
                        }
                }
        }
}
```

Here, the C matrix is defined in an earlier loop. The mkn permutation is here shown, since it will also turn out to be an important permutation throughout this report.

2.4 Performance analysis

The performance of the six different functions resulting from the possibilities of permuting mnk are tested with square matrices with memory footprints ranging from a few kB to hundreds of MB. The results are seen in figure 2 with no compiler options. All six permutations perform equally well when the memory footprint is less than the L2 cache. For memory footprints larger than L2, the permutations with the loop over n maintain the same performance, while the other four permutations have a significant reduction of performance. This is due to the fact that m is the row index for matrices B and C, and that works well with the C language which is row major, meaning that the caches lines go along the rows of the matrices. The performance does not degrade after exceeding L3 which shows that program does not utilize the caches optimally. If that was the case, the smaller bandwidth between the memory and the processor would cause a significant drop in the performance when the memory footprint exceeds L3. To investigate this, the same graph is produced, but this time the −fast option is activated in the Sun Studio compiler, and the results are shown in figure 3. All six functions have improved performance compared to the case without compiler options, but the functions with k as the inner loop, are a factor of ~ 2 slower for memory footprints which fit in either L1 and L2 caches. This is due to the fact that when k is the

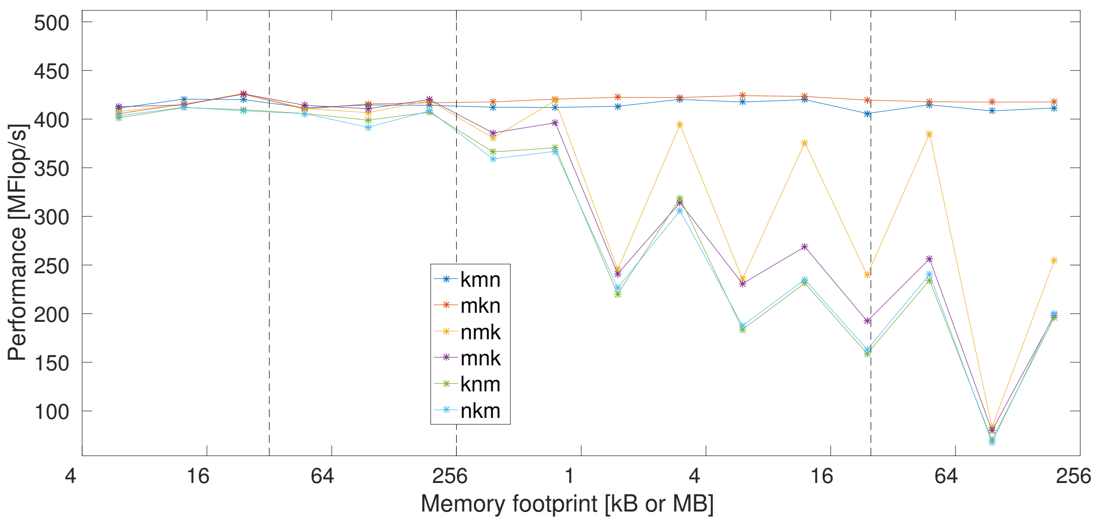


Figure 2: The performance of the different permutations versus the memory footprint, with L1, L2, and L3 caches shown as the vertical black dotted lines.

inner loop, the program needs to fetch two new values to multiply together, and the compiler optimizer apparently has some issues optimizing this. When the memory footprint exceeds L2, the functions with m as the inner loop drop off to the level of the functions with k as the inner loop. This is because m is the column size of matrices A and C, and looping over this as the inner loop goes against the row major format of C. For smaller memory footprints, the compiler is able to mitigate this issue, but after L2, the compiler optimizer can not fix the issue, and the performance drops. When the memory footprint does not exceed the size of the L3 cache, the performance of the functions with n perform at a constant rate. This is again due to the row major format of C and n being the row size of matrices B and C. After the L3 cahce size, the performance drops $\sim$ 10%, which is less than expected, but that is probably due some extensive prefetching performed by the compiler when the $-$fast option is enabled. It is also worth noting that mkn performs better than kmn for all memory footprints, and the reason here is that the loop over m is the slowest, and thus to achieve the best performance, this loop must placed as the outmost loop.

One function prom each pair sharing the same inner loop is selected to investigate the effect of the different compiler options on the performance. Three options are chosen: General optimization -xO5, loop unrolling -unroll=8, and prefetching -xprefetch_level=3. The three plots are seen in figure 4. As seen, neither prefetching nor unrolling does anything measureable to the performance, while -xO5 is responsible for the entire performance increase from the -fast option. We expected a performance increase from the unrolling because the code only specifies one multiplication per loop, and unrolling should decrease the relative amount of overhead calculations, thus increasing the performance even for small memory footprints, but it is not observed. This could be because the hard-coded unrolling value is not optimal for all matrix sizes, and thus it does not give a performance increase. The -fast option does unrolling, but probably with more dynamically. The prefetching was expected to give an increased performance for memory footprints exceeding the L3 cache, but that is not observed either, again most likely due to something else being the bottleneck, as is also evident from the fact that performance with no options is more or less constant for all memory footprints

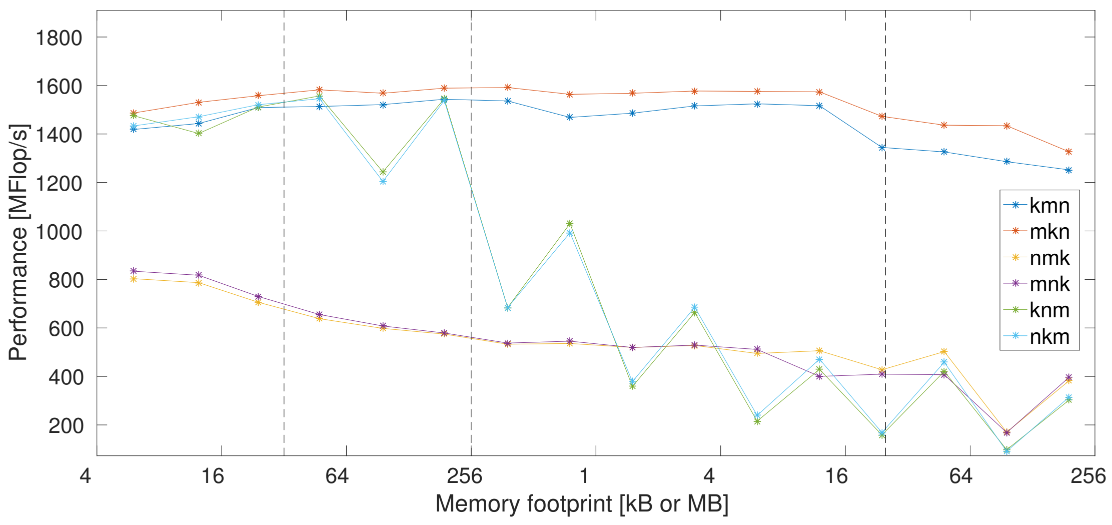


Figure 3: The performance of the different permutations versus the memory footprint with compiler option −fast activated. L1, L2, and L3 caches shown as the vertical black dotted lines.

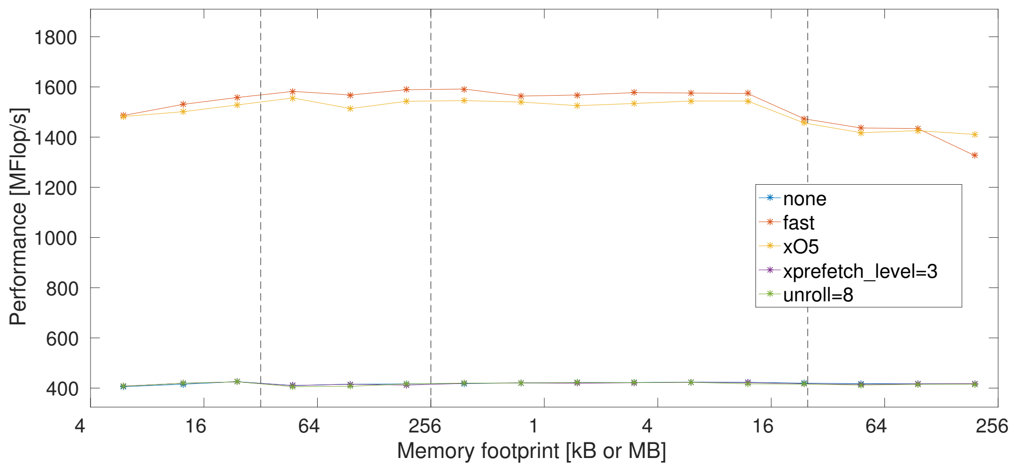
tested here.

In conclusion, the compiler optimizations can, for the right loop order, increase the performance by a factor of 4, but it was not possible to observe the effect from the individual options enabled alone.

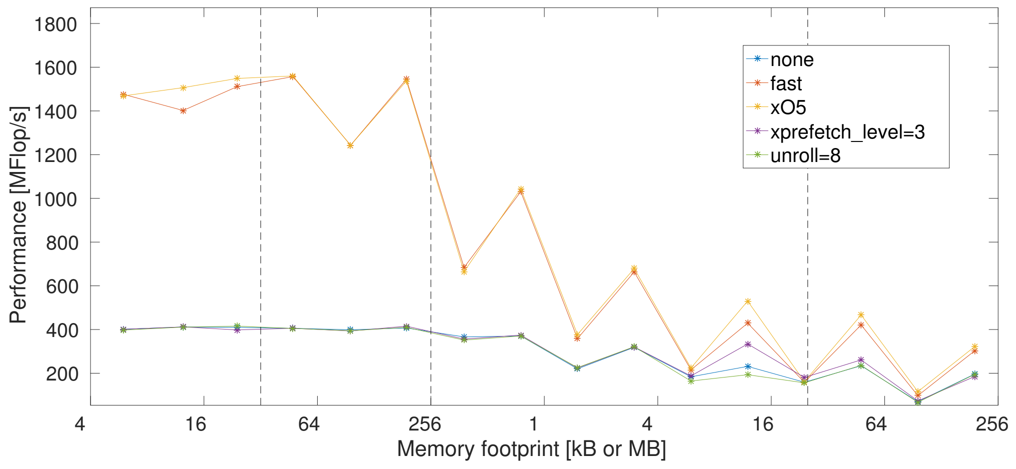
2.4.1 Analyzer Tool

We expect better-performing permutations to have less cache misses. To validate our hypotheses regarding cache hits we conducted a profiling experiment. That is, we fixed the matrix dimension size to 1000 which corresponds to 24MB memory footprint and ran each of the six permutations with and without compiler optimizations (with and without -fast). The results were collected and viewed using Oracle Solaris Studio Performance Analyzer. To be able to make direct comparisons, we set the minimum runtime to zero and max iterations to one. This way each permutation is run for a single iteration and we can compare cache hits in absolute terms. To directly compare, the relative hit ratio is computed for each permutation for both compiler settings and for both the L1d and L2d cache.
Table 2 shows the various hit ratios for the different permutations. It is clear, that both mkn and kmn has the fewest cache misses in the in L1 for both with and without fast. The hit ratios are worse in the L2 cache for the good performing mkn and kmn functions, but this is most likely due to the very few hits in that cache, since more of the L2 cache is used due to the fact of the faster computations. It is likewise found that nmk and mnk also perform averagely regarding the cache misses, which is also apparent in the memory foodprint since they are fluctuating between the bottom perfoming knm and nkm permutations and the top performing permutations of the mkn and kmn.

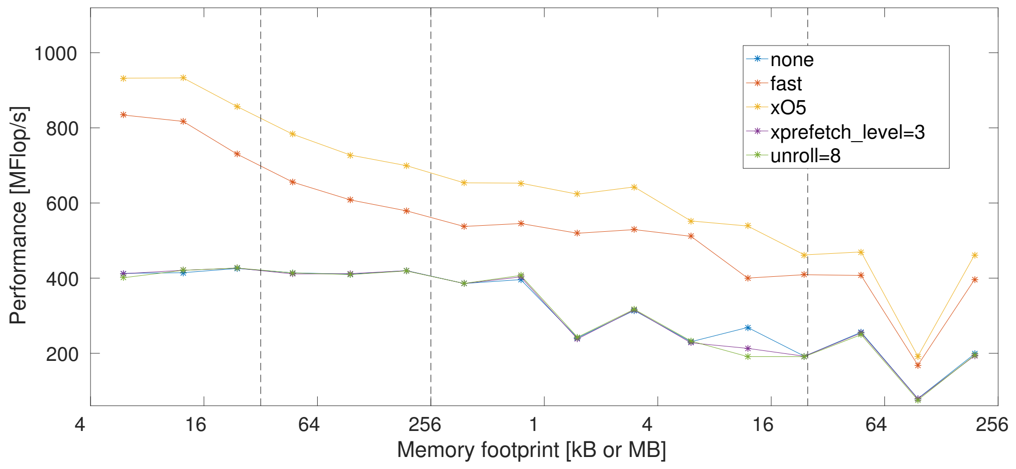
Since there were no hits in L2 for the mkn permutation, we performed a computation for a 2000x2000 matrix, just to provide the whole picture. Here the L1 ratio is found to be 99.91% and 86.67% for the L2 cache.



(a)



(b)



(c)

Figure 4: Performance versus memory footprint with different compiler options for (a) mkn, (b) knm, and (c) mnk.

Table 2: Displaying L1 and L2 cache hit ratio for the various permutations for a 1000x1000 matrix.

	No compiler optimization		With -fast	
	L1 hit ratio	L2 hit ratio	L1 hit ratio	L2 hit ratio
mkn	99.98%	75.96%	99.94%	0 hits
nkm	90.22%	93.18%	62.78%	93.53%
nmk	95.19%	99.69%	84.38%	87.60%
mnk	99.49%	87.43%	84.46%	99.39%
kmn	99.97%	61.24%	99.94%	75.96%
knm	90.20%	93.43%	62.94%	93.08%

2.5 Loop Blocking

Loop blocking is a technique to improve the memory access and data re-use, and thereby reduce the cache misses. To perform this, the matrices is essentially cut into smaller blocks, which are then solved, where more of the information can be reused, since it is already available in the cache. By implementing loop blocking, we intend to compare the performance of the new blocked loop function with the best performing permutation, which is mkn. The function prototype is almost identical to the earlier used with the new addition, that it takes a specified integer bs as an argument.

Listing 5: Blocked blk Prototype

```
void matmult_blk(int m, int n, int k, double ** A, double ** B,
double ** C, int bs)
```

To implement the blocking itself for arbitrarily sized matrices, we introduce 3 blocking size variables in the loop for each dimension. Since the blocking is performed for each dimension, we need six nested loops. The first three are the blocking loops to increases in increments of the blocking size. These include an if conditional, in the case, when the specific dimension is not dividable by the block variable. In the last run of the loop, there may be a case where the blocking size is actually bigger than the missing number of loop variables (dimension size - loop variable), in which case the bs variable is set equal to the number of missing rowx/columns for that dimension. This secures that no segmentation error occur.

The three inner loops basically runs over the according block sizes for the three dimension variables.

Listing 6: Blocked Loop Function

```
int bsi=bs;
int bsj=bs;
int bsl=bs;

for(int i1 = 0; i1 < m;i1+=bsi){
        if(m-i1 < bs) {bsi=m-i1;
        }
        for(int l1 = 0; l1 < k;l1+=bsl){
                if(k-l1 < bs) {bsl=k-l1;
                }
                for(int j1 = 0; j1 < n;j1+=bsj){
                        if(n-j1 < bs) {bsj=n-j1;
                        }
                        for(int i2 = 0; i2 < bsi; i2++){
                                for(int l2 = 0; l2 < bsl;l2++){
                                        for(int j2 = 0; j2 < bsj; j2++){
                                                C[i1+i2][j1+j2] +=
                                                A[i1+i2][l1+l2]*B[l1+l2][j1+j2
                                        }
                                }
                        }
                }
        }
}
```

It would also be easy to implement different initial block sizes to experiment with different block dimensions and structures, i.e. a rectangle, but this is however not researched in our report. It could also be interesting to look into how the blocking perform with different type of rectangles, but these experiments have been performed on square matrices.

2.5.1 Block Performance and Comparison

Since the optimal blocking size varies with different matrix dimension, we seek to determine a good overall block size to use the same parameter for a memory footprint comparison with the best permutation mkn and the library function.
An experiment with varying block sizes are run for a 2000x2000, which is displayed in figure 5. The different blocksizes found can then also be compared to the L1, L2 and L3 caches. So the variations however small, clearly show the advantage of the correct block size and how it suits the different cache sizes. The peak right before the L2 cache is with a block size of 100. This block size is then chosen to use for the rest of the experiments.

The memory footprint for the blk and mkn is created, using a blocking size of 100. It should be noted, that the blk function is using the same loop structure as mkn. There are multiple factors in play here, and they perform almost identical with matrices of small sizes, but the mkn function do however perform better. This is probably caused by the additional loops and if statements in the blocking function. It can be seen that mkn performs surprisingly well for matrices of bigger sizes, and the blk function falls of a bit around 4 MB. The decent performance of the mkn function is partly caused by the Sun Studio compiler, which is very optimized already. Furthermore, the mkn permutation is already performing the calculation in the best suitable way memory wise. The mkn function do however start to fall off in MFlop/s, once the matrix sizes increases above the L3 cache size, which makes sense, since there is a penalty on accessing the RAM, whereas the blk function largely avoids that, and does not face the same downward trend. If we had run the experiment for bigger matrix sizes, the result may very well have been that the mkn function would continue to fall of, and the blk function would perform better on matrices of sizes above 1 GB.

We are now ready to compare the blk function with the library function DGEMM, but we quickly realize that these are not in the same league since the MFlop/s of the lib function

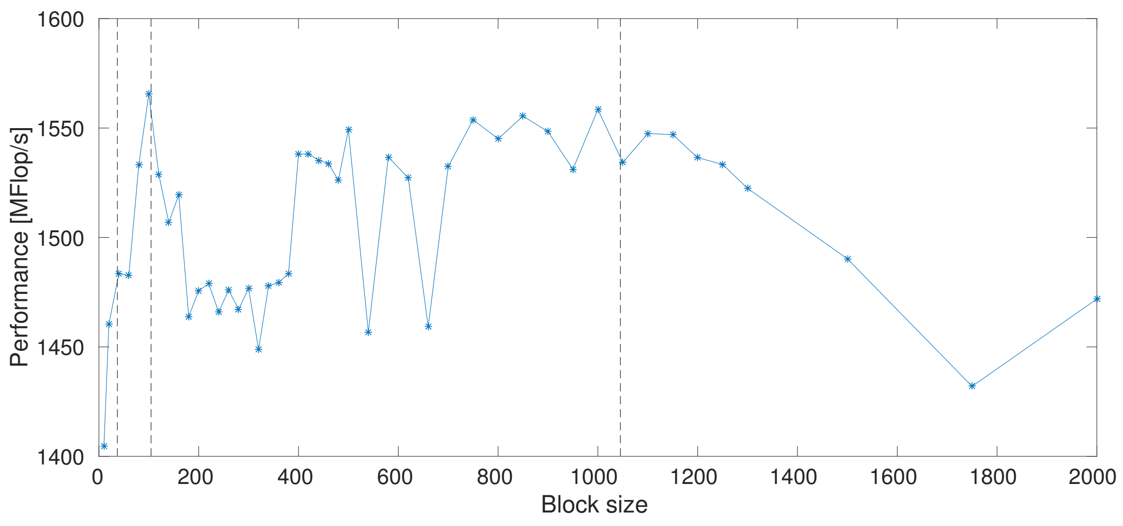


Figure 5: d

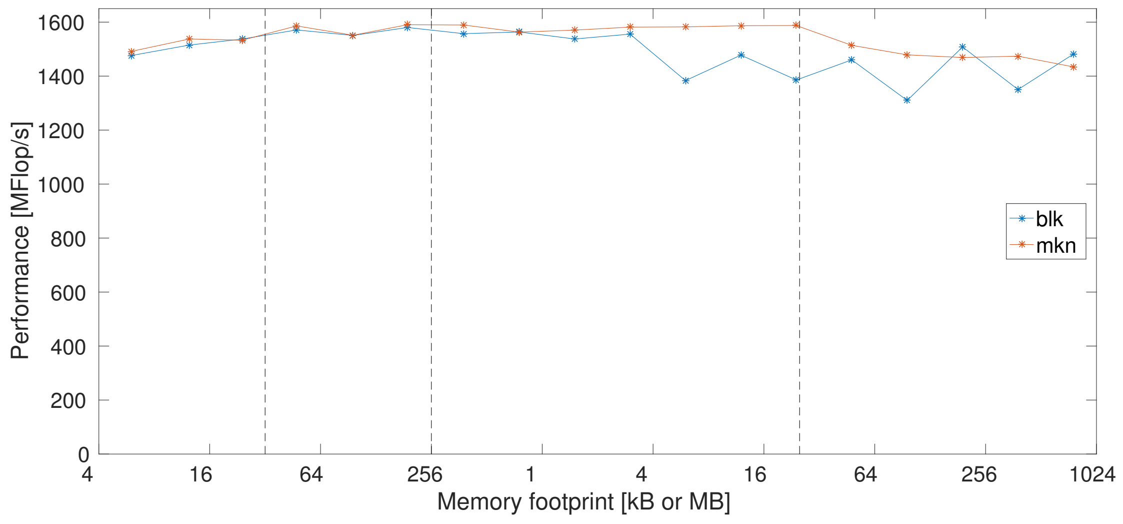


Figure 6: The memory footprint of the blk and mkn functions.

is mostly above 20000. The performance of the library function is far superior to our own novice functions.

3 Conclusion

We have implemented 6 functions all computing a matrix matrix product, and each having a different permutation of loop orders. Using different compiler optimizations, the best one was found to be mkn, topping at $\sim$ 1600 MFlop/s, opposed to the DGEMM library function which tops at $\sim$ 22000 MFlop/s. Cache hits and misses were analyzed to highlight the difference in performance between the permutations, and the analysis showed that the quickest functions has the highest hit-to-miss-ratio as expected. Finally, blocking was implemented as an attempt to better the performance degradation from the memory footprint exceeding the L3 cache size. The implementation was successful, but performance-wise, it only showed a minor improvement as opposed to the best non-blocked version. This is partly due to the Sun Studio compiler doing a great job at prefetching. However, we expect that the blocked version will be better than the non-blocked version for matrices larger than the one tested here.