DTU Informatics
Department of Informatics and Mathematical Modelling

High-Performance Computing, 02614

Assignment 1: Matrix Multiplication

GROUP:

Alessandro Dal Corso (s120929)
Alexander Birke (s124044)
Daniele Brazzolotto (s111071)

Technical University of Denmark
Lyngby, 15th January 2013

Contents

1 Assignment 1 **3**

1.1 Summary . . . 3
1.2 Problem Statement . . . 3
1.3 Description of Hardware and Software Used . . . 4
1.3.1 Hardware . . . 4
1.3.2 Compiler . . . 4
1.4 Theory . . . 5
1.4.1 Loop Optimization Techniques . . . 5
1.4.2 Loop Blocking . . . 6
1.5 Implementation . . . 6
1.5.1 Basic matrix multiplication . . . 6
1.5.2 Blocked matrix algorithm . . . 7
1.5.3 Scripts and tests . . . 8
1.6 Results and Conclusions . . . 8
1.6.1 Comparison against dgemm . . . 8
1.6.2 Permutations of m, k, n and performance . . . 9
1.6.3 Performance of n,k,m implementations with different sizes . . . 11
1.6.4 Blocked Matrix performance . . . 12
1.7 Compiler Options . . . 14

Chapter 1

Assignment 1

1.1 Summary

The ultimate goal of the assignment is implementing a matrix multiplication algorithm, optimized for single CPU, single core machine. The focus is on how an existing code can be speeded up, without changing (too much) the total number of operations.

An iterative process has been used to improve the code and, after every optimization, an automatic test session has been performed in order to verify the actual performances on different conditions. In some interesting case a profiling tool has been used to investigate in depth some specific behaviour.

The results confirmed that the main bottleneck of the naive algorithm was the poor use of the cache, but they also saw that the best performances can be reach with code that leads to the right compiler optimizations, rather than with code that pretends to optimize everything itself.

1.2 Problem Statement

From the assignment we were given the following tasks

- Write a function that performs matrix matrix multiplication on arbitrary sized two-dimensional arrays.
- Make a separate implementation for each way the loops in the above function can be ordered and test which one is the fastest.
- Compare the result of our implementation to DGEMM(), a similar function in the BLAS library
- All the functions and the call to DGEMM() should be wrapped into a single library.
- Implement a blocked version of the matrix matrix multiplication function and identify the block size that gives the best performance in regards to L1 cache. This result should be compared to the timings for the library function.
- Compare different compiler settings to see how they affect the result.

1.3 Description of Hardware and Software Used

1.3.1 Hardware

To perform the test, it has been used the DTU HPC facility, in particular a Intel Xeon X5550 @ 2.67 GHz.

More technical specification are listed below, in particular please note the cache size, 32 KB L1 and 256 KB L2, and the current CPU speed during the tests, reduced at 1.6 GHz.

DTU provides also some AMD Opteron 6168 machines (technical specifications are listed below as well), but if not different specified, the Intel Xeon X5550 is used.

	Intel Xeon X5550	AMD Opteron 6168
Architecture:	x86_64	x86_64
CPU op-mode(s):	32-bit, 64-bit	32-bit, 64-bit
Byte Order:	Little Endian	Little Endian
CPU(s):	8	24
On-line CPU(s) list:	0-7	0-23
Thread(s) per core:	1	1
Core(s) per socket:	4	12
CPU socket(s):	2	2
NUMA node(s):	2	4
Vendor ID:	GenuineIntel	AuthenticAMD
CPU family:	6	16
Model:	26	9
Stepping:	5	1
CPU MHz:	1600.000	1900.000
BogoMIPS:	5332.58	3800.37
Virtualization:	VT-x	AMD-V
L1d cache:	32K	64K
L1i cache:	32K	64K
L2 cache:	256K	512K
L3 cache:	8192K	5118K

From now on well refer to the Intel Xeon X5550 also as the Intel or the Xeon, while well refer to the AMD Opteron 6168 as the AMD.

1.3.2 Compiler

The compiler that has been used is Sun Compiler 5.12

```
1 suncc -g -fast -fPIC -c -o lib.o lib.c
```

Please note that, although DTU provides some AMD machines as well, it is necessary to compile in the Intel machine, as the compiler is affected by some bugs. If it runs on AMD cpu, it will cause Segmentation Fault on the final application.

Additionally it has been used the Oracle Solaris Studio Performances Analyzer 7.9 to profile the code. This software, that will be referred as the Analyzer from now on, it has been run over the AMD platform as it has a better support for hardware counters.

1.4 Theory

The matrix multiplication is a critical operation in computer science, both because it is the base of several practical problems and because it is still relatively expensive.
This operation, in fact, is really the fundamental brick of several interesting problems, from the resolution of big linear systems, often applied to physics, to resolution of discrete Fourier transforms, used for example to DNA splicing problems. On the main challenges of this operation is not related to the pure computational power, but with the amount of data that it requires: nowadays, as CPU/GPU evolved and still evolves faster than the memory chips, the real challenge is not to compute the result in a short time, but making the computational unit able to produce that result. In other words, it is more expensive bringing the right data at the right time, to the right chip, than to actually produce the result.
Algorithms in general nowadays, are so much affected by this behaviour, that the fastest ones are the algorithms that make a better usage of the caches. From this point of view, matrix multiplication is considered a good example of what a typical algorithm should look like these days. It is therefore used as a base to test the ability of high performances computers to deliver outcomes.

1.4.1 Loop Optimization Techniques

Since scientific computing often requires the looping over large data sets there has been developed multiple optimization strategies that targets different areas. These areas are:

Spatial locality
Reuse all the data fetched in a single cache line. Optimization techniques that leads to this is blocking.

Temporal locality Reuse data in previously fetched cache lines. Optimization techniques that leads to this is blocking.

loop bookkeeping
This is the overhead of running the loop in the first place. This happens when the loop only does a small amount of computation. Techniques that addresses this are loop unrolling and loop fussion.

Vectorization
Modern CPU's contain units that can do floating point operations on more than one value at a time. If a loop is organised in a proper way and if the results calculated in the loop is not dependent upon each other it can be recognized by the compiler and assembler code

will be generated that takes advantage of these units. Loop fission can help the compiler to do this.

Since loop blocking will be used later in this report we will cover it in more detail here.

1.4.2 Loop Blocking

A common technique to improve temporal and spatial locality at the cost of loop overhead is blocking. This technique takes a multi dimensional array and splits the computation on it up into smaller blocks. The size of the block is chosen to fit inside the cache size. This ensures that the cache is only filled with data that will be used in the current computations and later implementations. Blocking is implemented by having a set of loops that iterates over the blocks and then have a second set of loops inside the first set that iterates over each element. An implementation of this that calculates the transpose of matrix in C is shown below:

```
1 for(int i = 0 ; i < n ; n += nbi)
2   for( int j = 0 ; j < n ; j++)
3     for( int k = 0 ; k < min(n-i, nbi) ; k++)
4       A[j][i+k] = B[i+k][j]
```

The above loop accesses array A in a column wise order which in C usually results in many cache misses but because this is only done for as much memory as there can be in the cache it means that the data that has been fetched once in one cache page can be used again in a later iteration of the loop.

Blocking is a powerful technique to improve memory use and data reuse but also has its disadvantages. The optimal block size is dependent on the CPU the code is being run on and on what data the algorithm is used on. It is therefore a good idea to have parameters to specify the block size so the implementation can be ported to different architectures more easily.

1.5 Implementation

1.5.1 Basic matrix multiplication

The basic matrix multiplication algorithm follows the most standard n^3 implementation:

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

Its important to note that the C matrix initialization has been moved outside the main loop, in a separate cycle. This doesnt change the code complexity, but it helps to recognize that the core of the multiplication algorithm is completely independent from the order of the 3 nested loops that wrap it, so they can be reordered in any possible permutation (6 different ways).

In order to define with of the 6 permutations is the best, its important to understand that the 3 different matrices are actually linear array accessed by an index in the form:

X[i*c+j] with c constant, i and j loop variables.

The best performances will be reached when the matrix X will be read/written sequentially, so when itll be wrapped by an external loop on i and an internal loop on j. We can express this with the dependency i =¿ j. If we define the dependency graph for all the 3 matrices, we get:

$$C[j*n+i] : j => i : m => n$$

$$A[j*k+l] : j => l : m => k$$

$$B[l*n+i] : l => i : k => n$$

There is only 1 combination that satisfied all these constraints, and that is mkn (external to internal). We expect the best performances with this combination.

1.5.2 Blocked matrix algorithm

In literature, its suggested to use a block version of the naive algorithm to improve data locality: this approach intuitively do a better usage of the cache and improve the overall performances. The idea behind the algorithm is to split both A and B in squared blocks, small enough so the cache L1 can store 3 of them. In this way 2 blocks can be multiplied using the naive algorithm with high performances. Externally, other 3 nested loops will apply the same naive algorithm to combine the blocks:

```
1  void matmult_blk_internal(int m,int n,int k, int sm,int sn,int sk,double *A,double *B,
       double *C, int bs)
2  {
3    int mmin = min(m,(sm+1)*bs);
4    int nmin = min(n,(sn+1)*bs);
5    int kmin = min(k,(sk+1)*bs);
6    int smbs = sm*bs;
7    int snbs = sn*bs;
8    int skbs = sk*bs;
9
10       for(int i = snbs; i < nmin; i++)
11     for(int l = skbs; l < kmin; l++)
12       for(int j = smbs; j < mmin; j++)
13
14         C[j*n+i] += A[j*k+l] * B[l*n+i];
15 }
16
17 void matmult_blk(int m,int n,int k,double *A,double *B,double *C, int bs)
18 {
19   //bs=50;
20   int nm = n*m;
```

```
21   for(int i = 0; i < nm; i++)
22   {
23     C[i] = 0;
24   }
25   int m_bs = m/bs;
26   int n_bs = n/bs;
27   int k_bs = k/bs;
28
29   for(int j = 0; j <= m_bs; j++)
30     for(int l = 0; l <= k_bs; l++)
31       for(int i = 0; i <= n_bs; i++)
32         matmult_blk_internal(m,n,k,j,i,l,A,B,C,bs);
33 }
```

This approach maintain the, not only the same complexity of the naive algorithm, but also the exact same number of floating point operation (the only overhead is on integers), and the same numerical precision of the previous code. The proof is left, but the reader can refer to the literature to further details.

At this point, one important detail, is making sure that the compiler will inline the internal function in order to gain the best from the method. From our test, the most common optimization settings (-fast) included that.

1.5.3 Scripts and tests

In order to test our results, a set of python scripts have been developed. They automatically takes care of:

- rebuilding the library
- executing the tests
- store in a file the compiler option that has been used
- store in a file the description of the machine used
- store the data in a dat file
- generating a eps plot, using a predefined template.

Each test set has been run several time to assure the numerical stability of the results, before selecting an execution as a final

1.6 Results and Conclusions

In this section we present the results we came up during the tests altogether with the conclusions, in order to provide a better overview of the tests that have been performed.

1.6.1 Comparison against dgemm

Against the *dgemm*() library function, even with the same compiling options, the implementation provided of the *matmult_nat*() method has been proven poor, according to

the graph shown in Figure 1.1. The figure shown that, for every possible permutation of the cycle order (including so the naive implementation), dgemm outnumbers it.

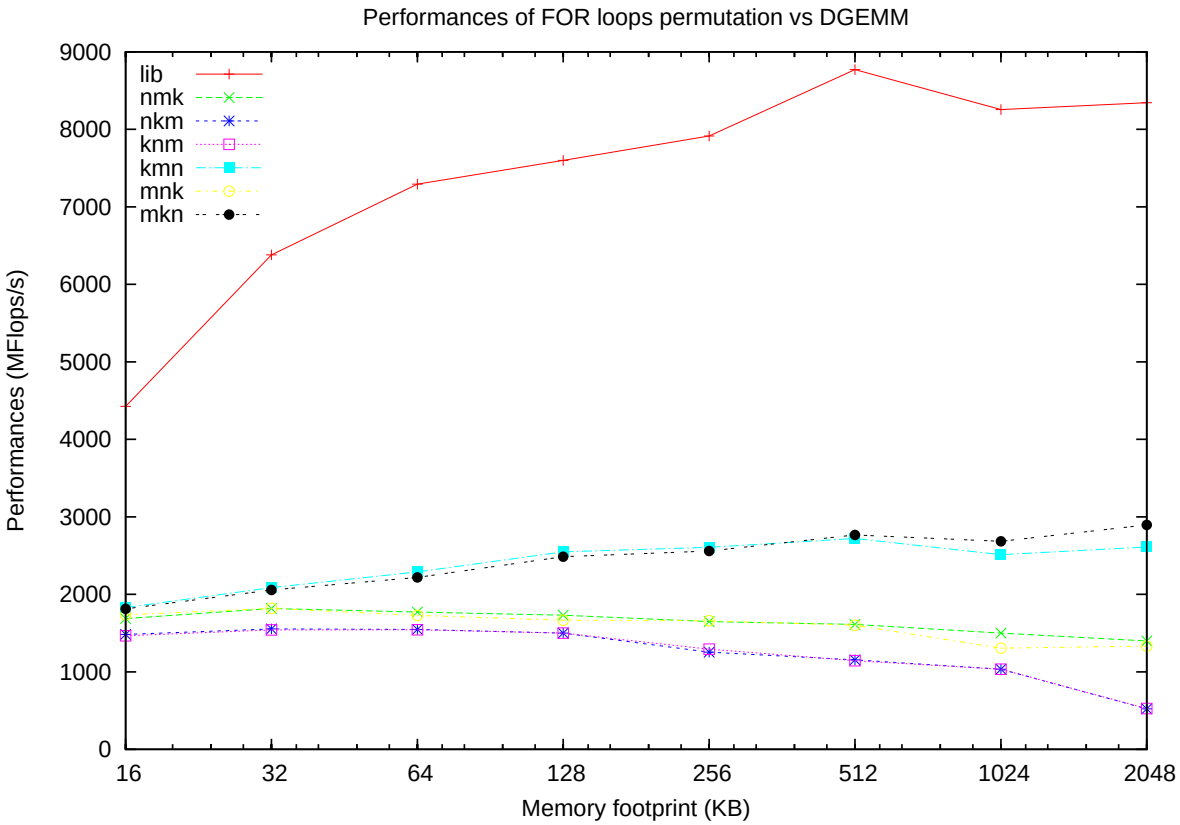

Figure 1.1: Comparison between *dgemm()* and the other implementations.

The reason for that is that the *dgemm*() library has been a lot of years in the industry, and has received a lot of performance improvements during the years. In addition, the library makes use of some inline assembly codes that boost its performance even more.

1.6.2 Permutations of m, k, n and performance

The three cycles performed in the naive algorithm can be rearrenged in six different ways: there are in fact six possible permutations of the three letters. The performance results provided by the execution of the six algorithms on some standards memory footprints are provided in Figure 1.2. The test was made on the product of square matrices $nxnxn$, where n was calculated using the following formula, to relate it to the memory footprint:

$$n = \sqrt{\frac{F_{byte}}{8 \cdot 3}}$$

The best results of the graph are the ones where the most internal cycle is cycled over $[1, n]$, and in particular the best implementation has been proved $m-k-n$, followed by $k-m-n$. The worst ones, instead, have been proved $n-k-m$ and $k-n-m$, the ones with the most internal cycle in m.

The result was what we expected in the Theory section. As a recall, the explanation of this behaviour is that $m-k-n$ is the best implementation because is the one that

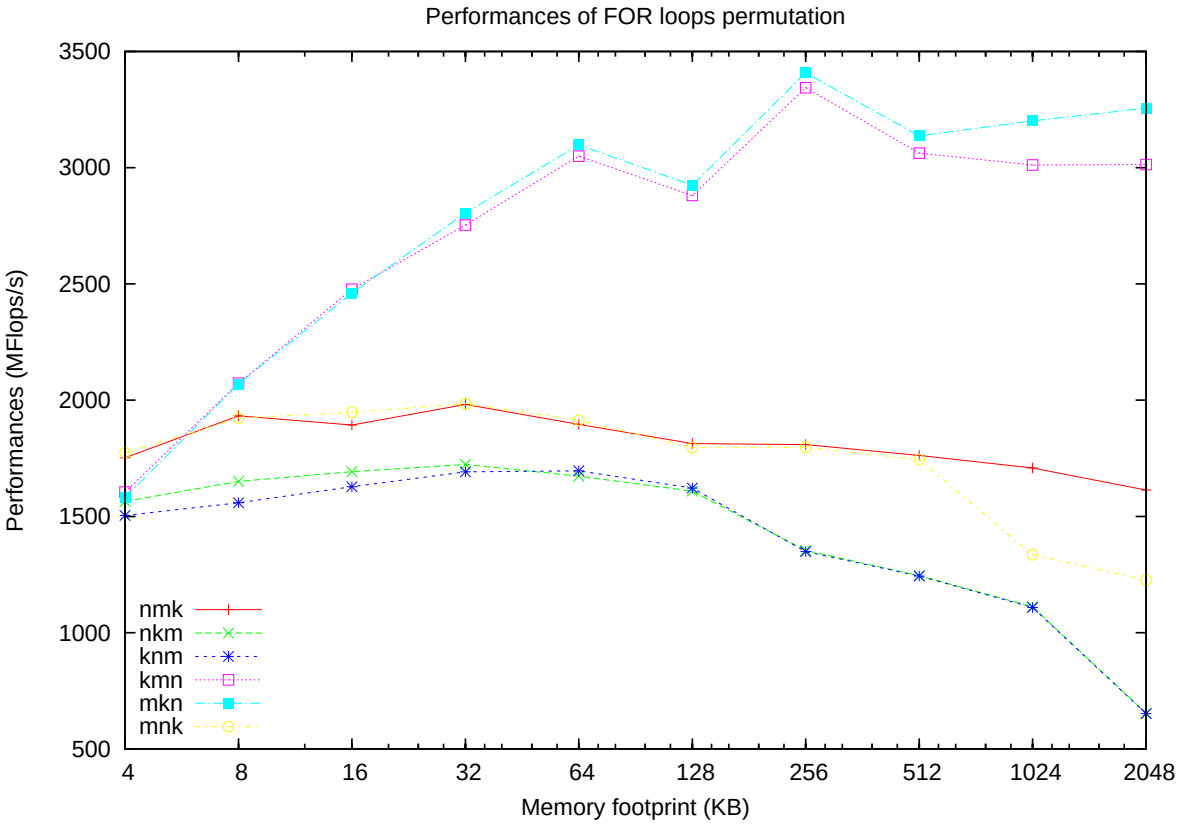

Figure 1.2: The possible combinations of *matmult_xxx()* for the Intel processor on square matrices of growing size.

causes the less cache misses. If the most internal variable looped is n, in fact, we read a whole row in the matrix each time we start reading an element from matrices B and C (that have both n columns). This causes less misses in the cache, leading to a better perfomance for both the $m-k-n$ and $k-m-n$ behaviours. In addition, since the A matrix has k columns, also if we loop through k before looping though m we got a better performance, for the same reason. This explains why $m-k-n$ is slighly better that the $k-m-n$ implementation.

The worst case scenarios $n-k-m$ and $k-n-m$, instead, do the opposite: they privilegiate columns over rows, resulting in a better performance if the matrix was stored column-wise, like in Fortran, but leading to disastruous results in C, where the most performant is the rowmajor order.

To verificate our assumption, we checked the number of cache misses for both the best and the worst implementation, resulting in:

	knm	mkn
MFlops:	440.342	1548.554
Misses in L1-Data:	3 969 996 464	2 239 998 044
Misses in L2-Data:	99 000 394	102 000 309

That confirmed our assumptions, seeing that the best implementation has fewer cache misses that the worst one.

1.6.3 Performance of n,k,m implementations with different sizes

The results and conclusions presented in the previous section held keeping in mind that the matrices we tested were square ones, where $n = m = k$. We repeated the same tests for some odd-sized matrices, i.e. where one of the three dimensions n,m,k was kept at a low value (in these tests 4 elements). The other two values were raised accordingly, in order to maintain the same memory footprint and have meaningful results. The results we came up for n and am were pretty much the same, but we k we found some different results, which are presented on Figure 1.4.

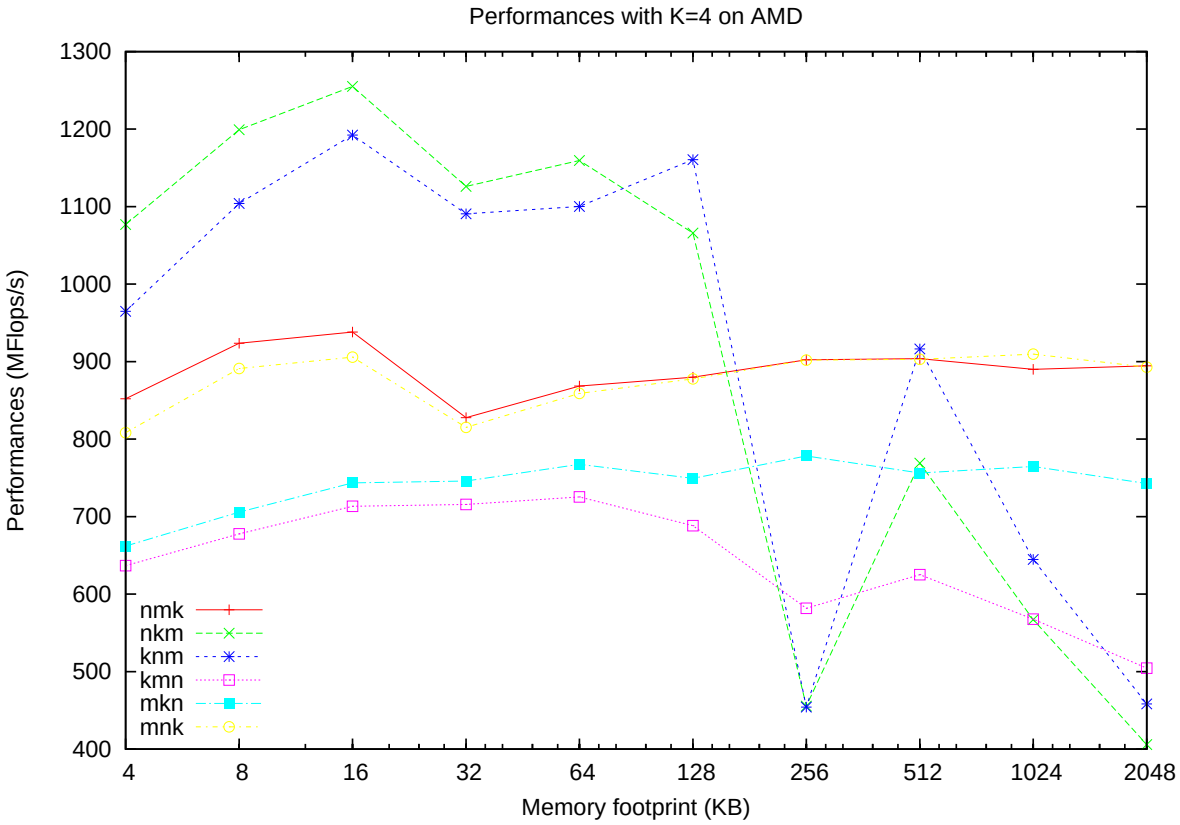

Figure 1.3: The possible combinations of *matmult_xxx()* with AMD processor.

We can see that the performances of the two best version in the square case are dropped dramatically, especially for little memory footprints. Since the test is on the AMD processor, we are able do see a drop in the performance due to the filling up of the L1 cache around $128kB$ (the cache is actually 64K, but it seems a little larger due to prefetch). For matrix sizes, we can see that the results of the cache analysis still confirm our previous assumptions of $m - k - n$ being faster than knm, as shown in the following table:

	knm	mkn
MFlops:	474.607	741.996
Misses in L1-Data:	2 419 072 580	6 600 200
Misses in L2-Data:	272 000 807	164 000 494

As a side note, even the *dgemm*() library drops its performances roughly from 8GFlops to 2 GFlops, as it is optimized for square memory sizes.

1.6.4 Blocked Matrix performance

We recall that the blocked version, introduced in section Implementation, uses the best algorithm for naive multiplication, $m-k-n$, in its inner loop. We experimented the block size against the MFLOPs generated for a generally big square matrix (with $2048kB$ memory footprint). The results are reported in Figure **??** (green line). The performances of the algorithm increased at the rising of the block size. This means that the optimum is the highest possible value of the block size, i.e. the matrix itself. In conclusion, the blocking algorithm can not really improve performance, because it will be always beaten by the $m-k-n$ algorithm.

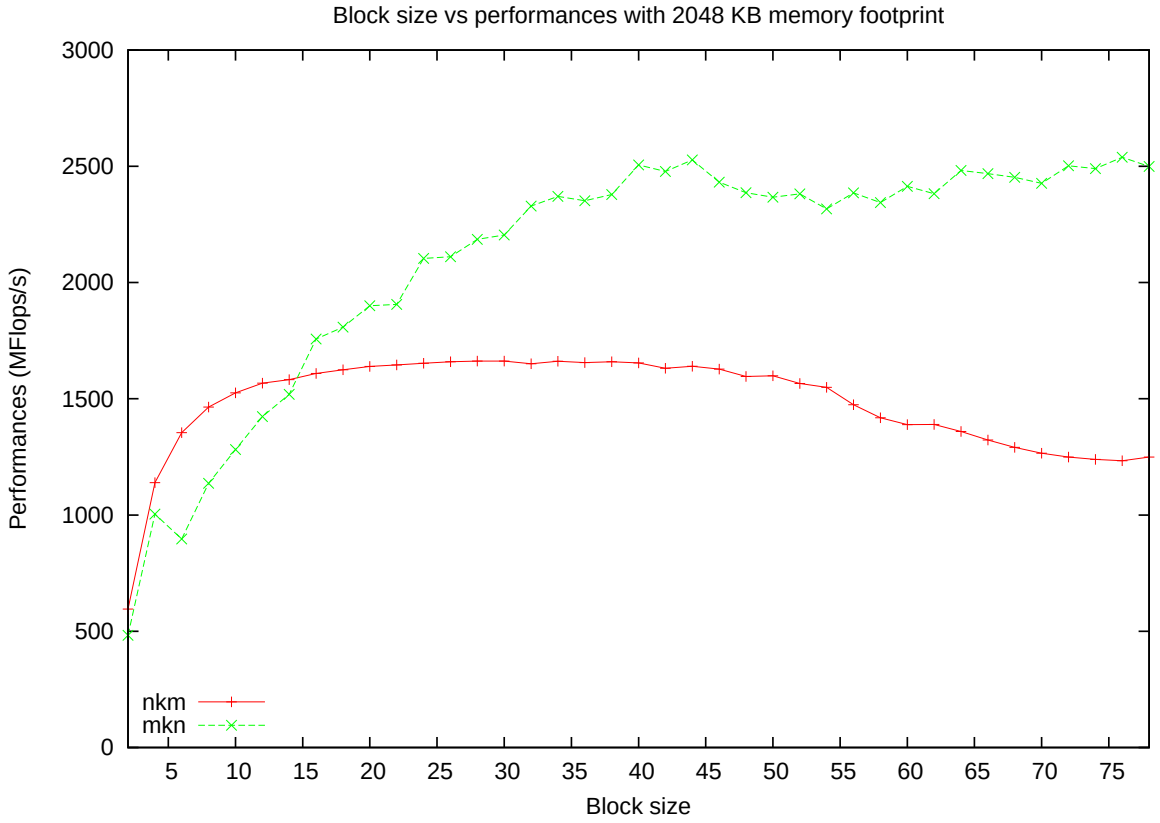

Figure 1.4: The possible combinations of *matmult_xxx()* with AMD processor.

To find an approximate optimum for the block size, it was necessary to substitute the algorithm in the inner loop with the worst one ($k-n-m$). Repeating the test, the data showed a plateau curve, reaching its maximum in the range 27-37. After 40, the performances start to decrease, because the L1d cache starts to fill up. At 40, in fact, we have an estimated data cache size of $3 \cdot 40^2 \cdot 8 = 37.5kB$, which is sligthly higher than the L1-data cache size ($32kB$) due to the prefetching effect.

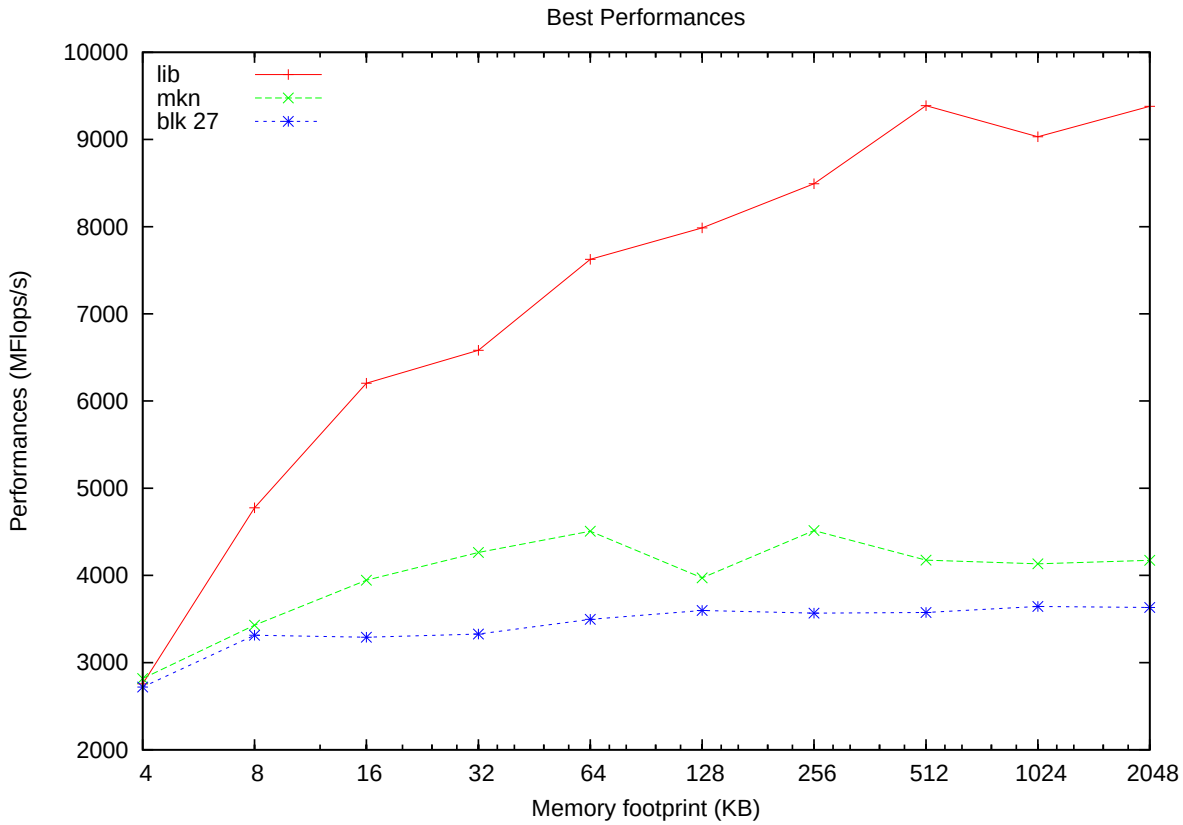

Figure 1.5: Comparison between $dgemm()$, the best blocked version and the $m-k-n$ Implementation.

1.7 Compiler Options

The compiler options tested where:

- **none** No optimization
- **-xO1 to -xO5** Different levels of optimization. The higher the number the more aggressively the compiler tries to optimiz the code for performance.
- **-xrestrict** Treats pointer-valued function parameters as restricted pointers. A restricted pointers are assumed not to be aliased. This flag improves vectorization.
- **-fast** Macro that applies various options including -xO5.

The different options was tested for different memory footprints on both an the AMD and Intel CPU. The results are given on figure 1.7 and 1.6.

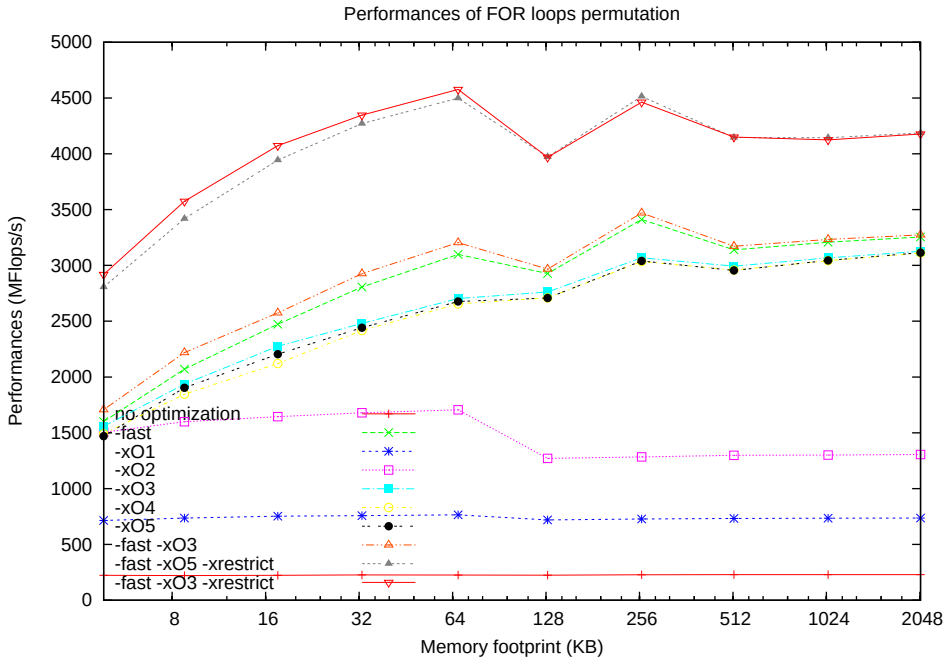

Figure 1.6: Timings of different compiler options on the intel CPU

The fastest set of options on both the AMD and Intel is -fast -xO3 -xrestrict. The missing data points on figure 1.7 is because test failed because of a compiler error whenever -fast.

As it can be seen from figure 1.5 and figure 1.6 that even though we use the fastest options we cannot compile a version of our implementation that is faster than the library function.

It is also interesting to note that options -x04 and -x05 performs worse than -xO3 on both the Intel and AMD.

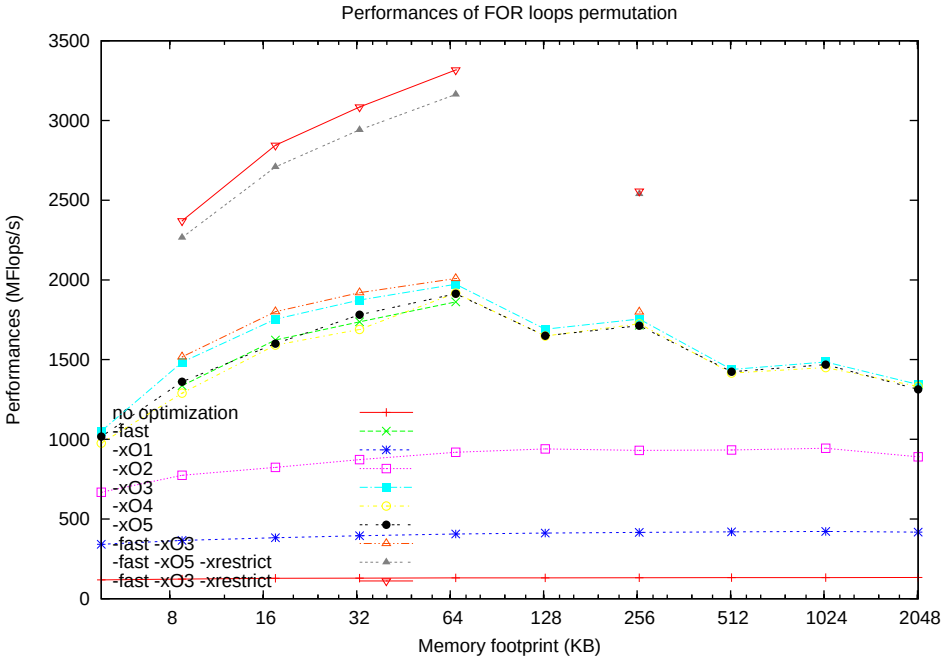

Figure 1.7: Timings of different compiler options on the AMD CPU