High Performance Computing

Assignment 1: Matrix multiplication

large Miguel Suau de Castro (161333)
Alberto Chiusole (162501)
Anders Jakobsen (122467)

Technical University of Denmark
02614 High-Performance Computing
6th January 2017

Contents

1 | Introduction

This is the first mandatory assignment for the course 02614 High-Performance Computing. The report contains several matrix-matrix multiplications techniques in C and some comparisons between them.

All the computations has been carried out on the DTU HPC cluster on a node with the specifications stated in Table 1.1.

CPU specifications	
L1 cache	32K
L2 cache	256K
L3 cache	30720K

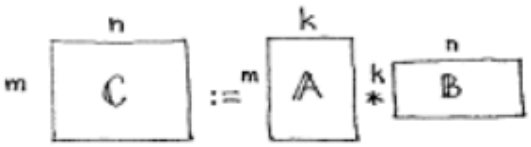
Table 1.1: HPC Cluster memory specification.
The cluster works on top of Intel® Xeon® E5-2670 v3 CPUs.

Throughout this report, all the plot represent the memory footprint of the data the program works on, on the **x** axis, and the Mega Floating Point Operations per Second (MFLOPS), on the **y** axis.

Moreover, three vertical bars are displayed, corresponding to the three levels of cache available, as reported in *table 1.1*.

2 | Native C implementation

To perform a matrix to matrix multiplication ($A \times B = C$) we decided to cycle on the `m` rows of the first matrix, and on the `n` rows of the second.



The routine we wrote is based on three nested *for* loops, which cycle on m, n and eventually on k.

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

We have to point out that before starting saving data into the C matrix we need to initialize it to zero: this operation could be right before creating the last for loop. Doing so, however, would not be fair compared to other combinations of m, n and k, because in this particular one we visit a particular cell [m][n] only once; a different combination could visit cells in multiple occasions, and we cannot recognize if contains uninitialized random data o proper data. For this reason, we had to initialize the output cells to zero with other two more for loops *before* executing the multiplication operation.

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        C[i][j] = 0;
    }
}
```

3 | CBLAS library

Basic Linear Algebra Subprograms (BLAS) are low-level routines that perform linear algebra operations, the library has been thoroughly optimize since it was created. At first it was programmed in FORTRAN, but other versions are now available.

We will be using the CBLAS library which was adapted to C-style to perform the matrix multiplication. There are two important differences between the routines built in C and the original ones. In CBLAS the arguments are passed by value whereas in FORTRAN they are passed by reference. The arrays in CBLAS are stored in rowmajor order avoiding the need to transpose the matrix when using BLAS in C.

We have created a routine that calls DGEMM to perform the matrix multiplication. Notice that the values are passed to the BLAS routine as a single pointer.

```
1 #include <stdio.h>
2 #include "cblas.h"
3 #include <stdlib.h>
4
5 void matmult_lib(int m, int n, int k, double **A, double **B, double **C) {
6   double alpha = 1.0, beta = 0.0;
7   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, A[0], k, B[0],
      n, beta, C[0], n);
8 }
```

It is not a surprise that the graph in figure 3.1 shows a better performance for the CBLAS routine that for the native version.

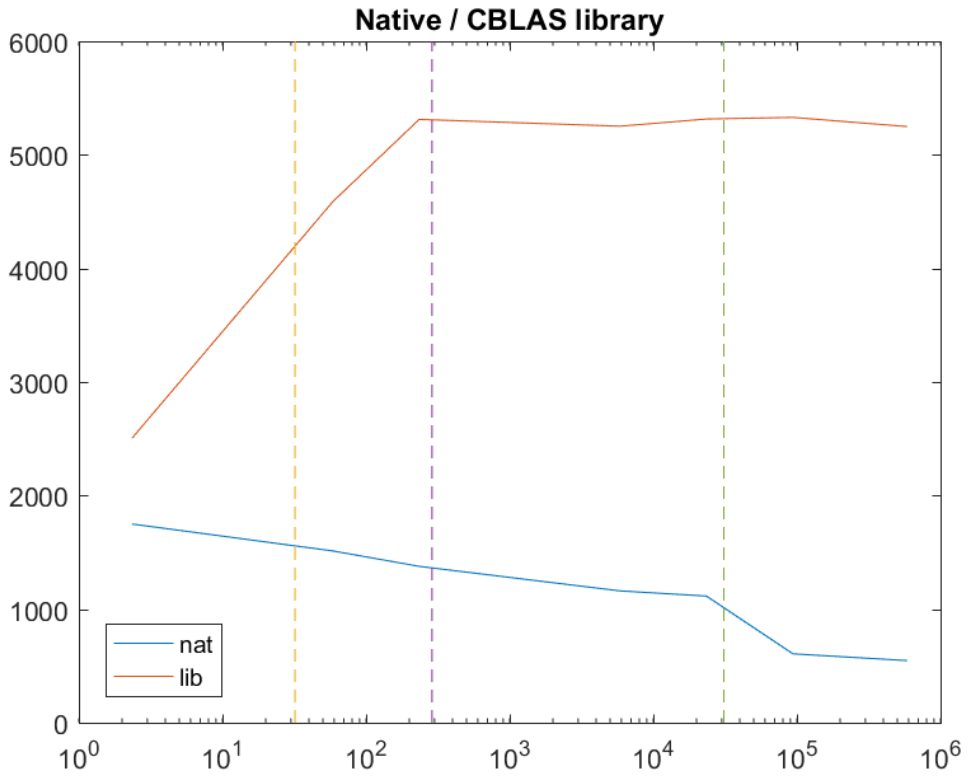


Figure 3.1: Comparison between the native implementation and the OpenBLAS function.

4 | Loop interchange

Loop interchange is a technique that can be applied to two or more nested loops in order to improve the code performance. The improvement has to do with the way the programming language access and stores the memory. The matrix multiplication routine consists of 3 nested loop that can be interchanged with no effect in the result. We applied loop interchange and created 6 different routines with 6 different loop combinations. We wrote a bash script that performs multiple tests for the 6 different routines varying the problem size. The results of these test are shown in figure 4.1

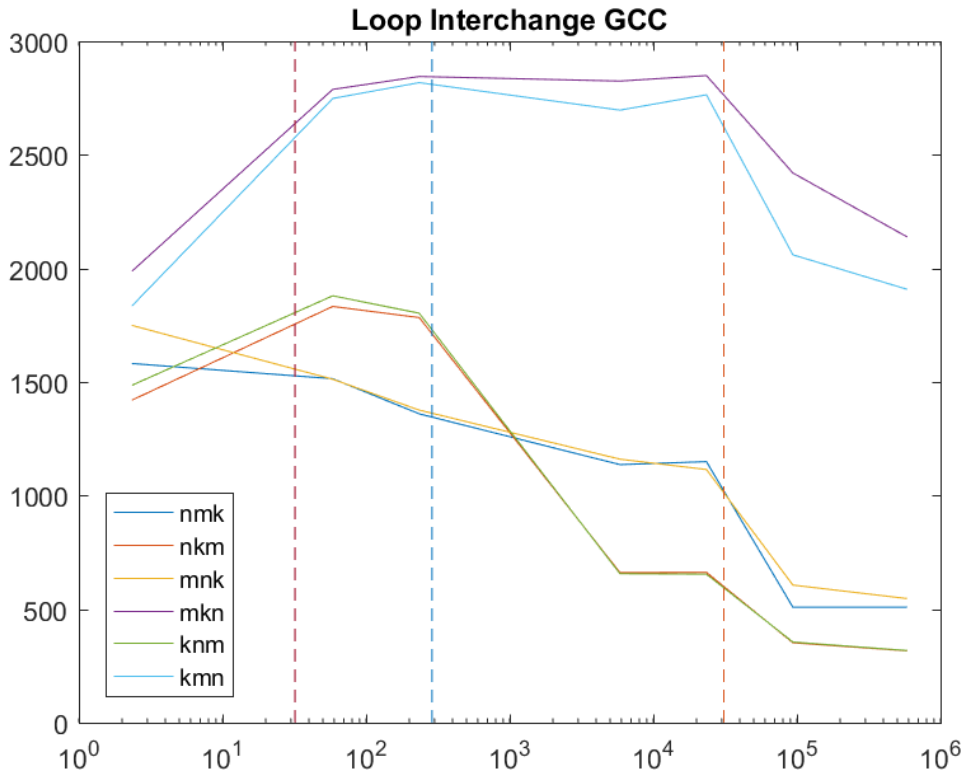


Figure 4.1: Loop interchange comparison (Memory footprint / MFlops)

It can be seen that the mkn combination is faster for problems of any size. We have used this routine for further studies. The vertical lines indicate the three cache sizes. It is easy to see that the speed either stays at the same level within an specific cache or drops when the memory access a larger cache.

5 | Optimization

Different compiler optimization options have been applied in order to study the changes in performance.

Using Sun Studio compiler we first tried to apply loop interchange by switching on the -xrestrict option. By doing this we tell the compiler that the pointers do not overlap and thus, there is no risk for loop interchange. We expected the compiler to rewrite the code so that the 6 combinations took the form of the MKN routine. However, since we are passing double pointers the compiler does not switch the loops and that's why there is no performance improvement.

We also tried to set the -xprefetch option to level 3, so that the machine requires the next chunk of memory while computing the previous operation. Again, as shown in figure 5.1 we did not obtain more speed either.

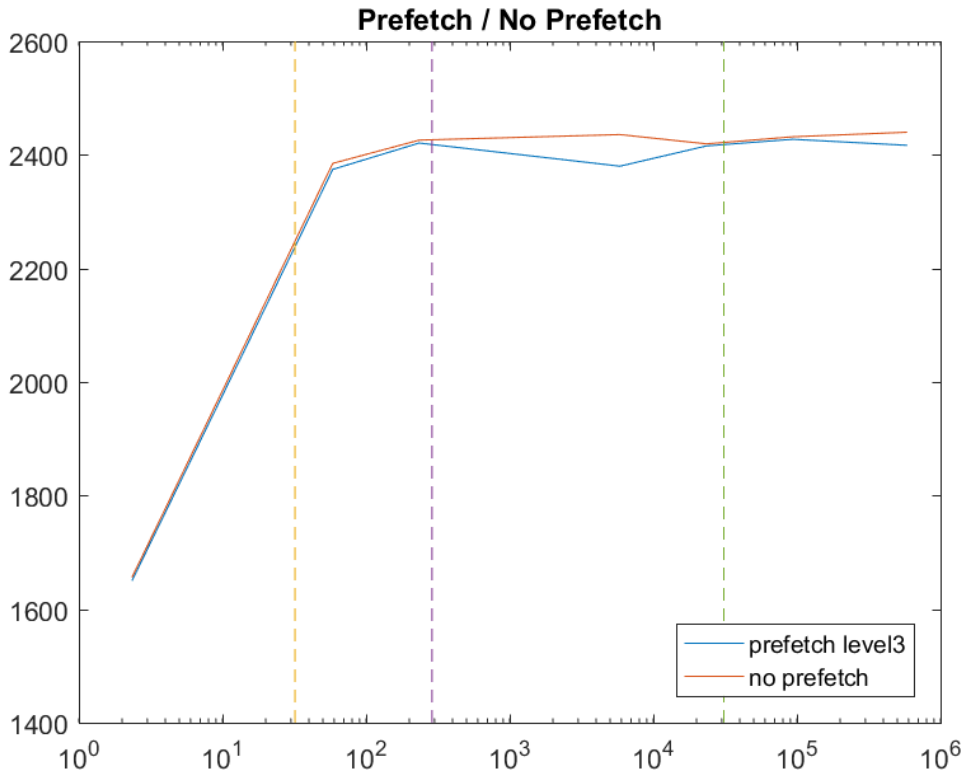


Figure 5.1: Comparison between Prefetch and no prefetch
(Memory footprint / MFlops)

Nevertheless, we did find out that the Sun Studio compiler performs better than the GCC compiler for large problem sizes.

We then decided to repeat the loop interchange test using Sun Studio Compiler with the flag -fast disabled. The graph in figure 5.2 shows how when disabling this option there is a fall around the L3 cache. Our guess is that the compiler applies loop blocking when asked for code optimization.

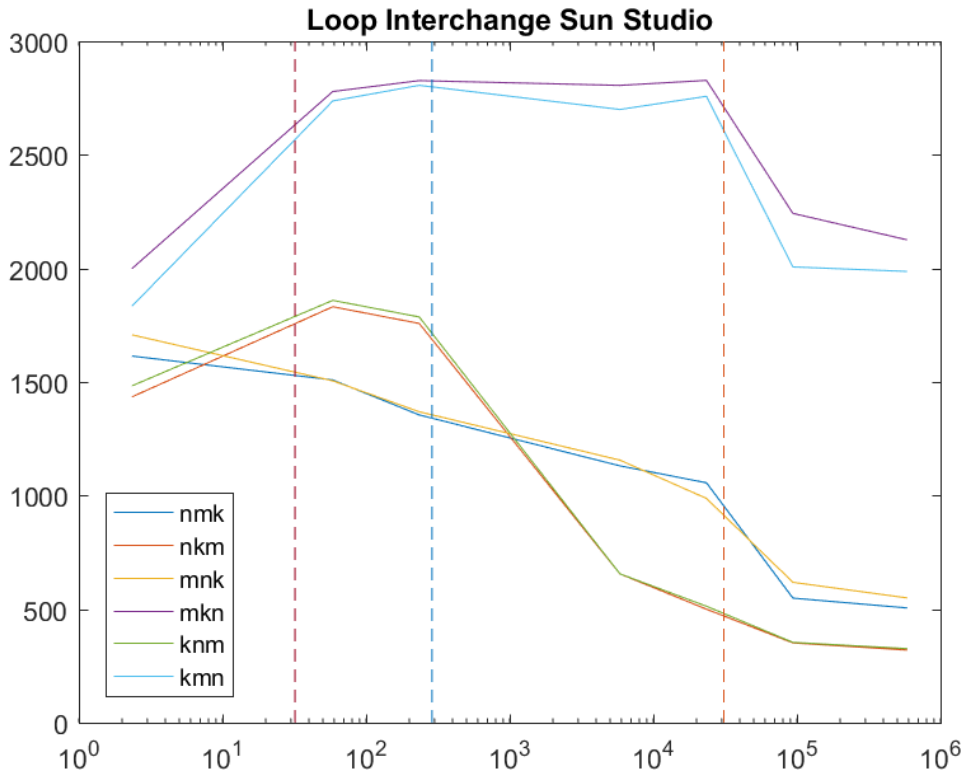


Figure 5.2: Loop interchange Sun Studio compiler with no optimization (Memory footprint / MFlops)

6 | Loop blocking

We have implemented a *loop tiling*, or *loop blocking*, technique, which can improve the performances of a program by increasing the locality of reference. Instead of loading the same cell multiple times, a loop blocking optimization loads similar areas in the input matrices and performs all the possible operations in it.

The algorithm that performs loop blocking selects a tile in both the two original matrix and uses it as a sub-matrix, on which computes operations until exhaustion of the cells to work on.

In a native version there are three for loops, one for each dimension m, n and k: in a loop blocking there are six loops, the usual three and three more that cycle on the chunks of matrix used at every time.

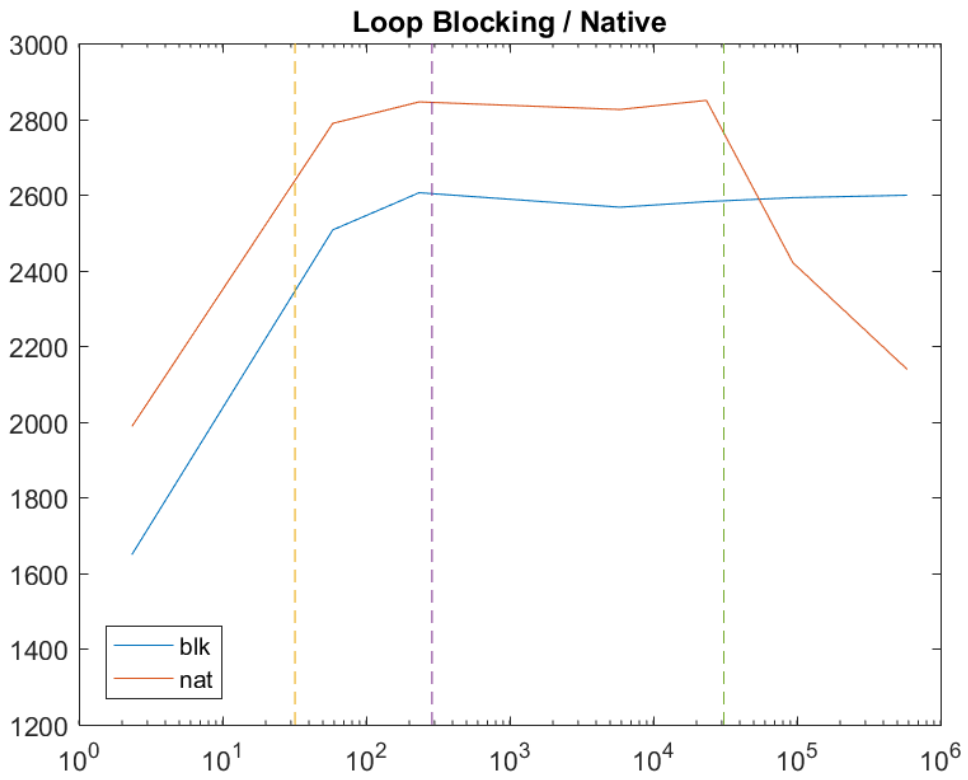


Figure 6.1: Comparison between the native and the blocking implementations, built with GCC.

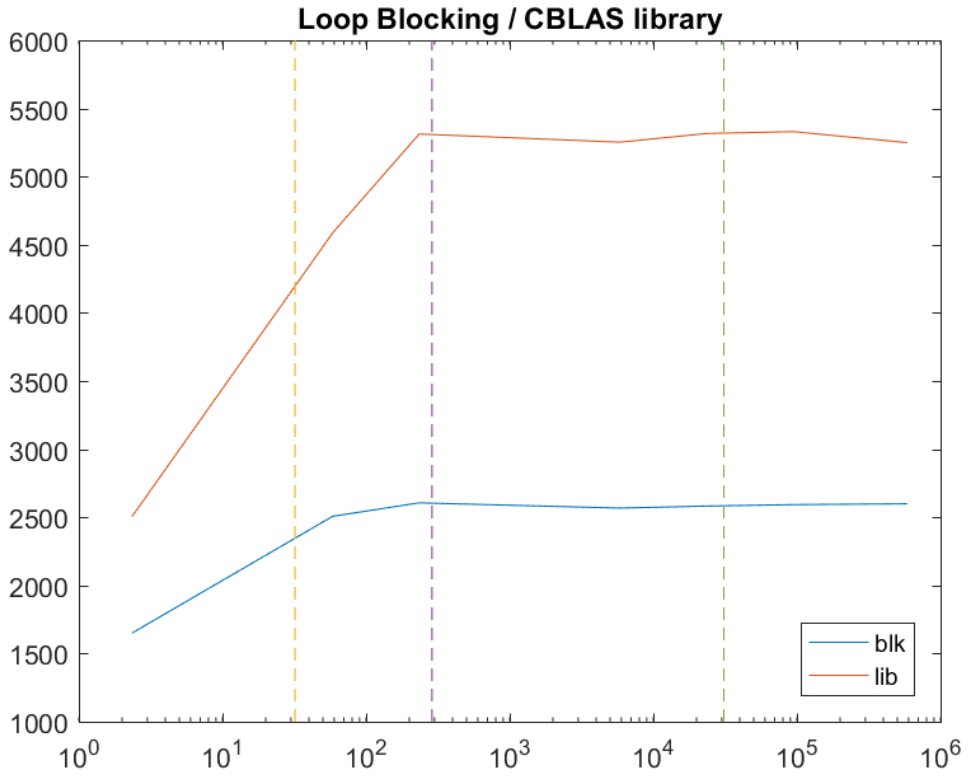


Figure 6.2: Comparison between the library and the blocking implementations, built with GCC.

As we can see in 6.1, there's a great fall in performance in the native implementation, compared to the blocking implementation, when we deal with data greater than the L3 cache size. This is due to a better usage of the cache available on the machine.

It is seen in figure 6.2 that as expected the library performs better than our loop blocking implementation since it has been properly optimized for many years.

The SUN studio compiler (not reported here), performs some internal loop blocking operations, thus a native matrix-matrix multiplication implementation has not a so steep fall as the one compiled with GCC. To see an effective fall in the program compiled in SUN we need to manage really huge amount of data.

Hereafter we present a comparison of different block sizes, in order to analyze which block size produces the highest MFLOPS rate.

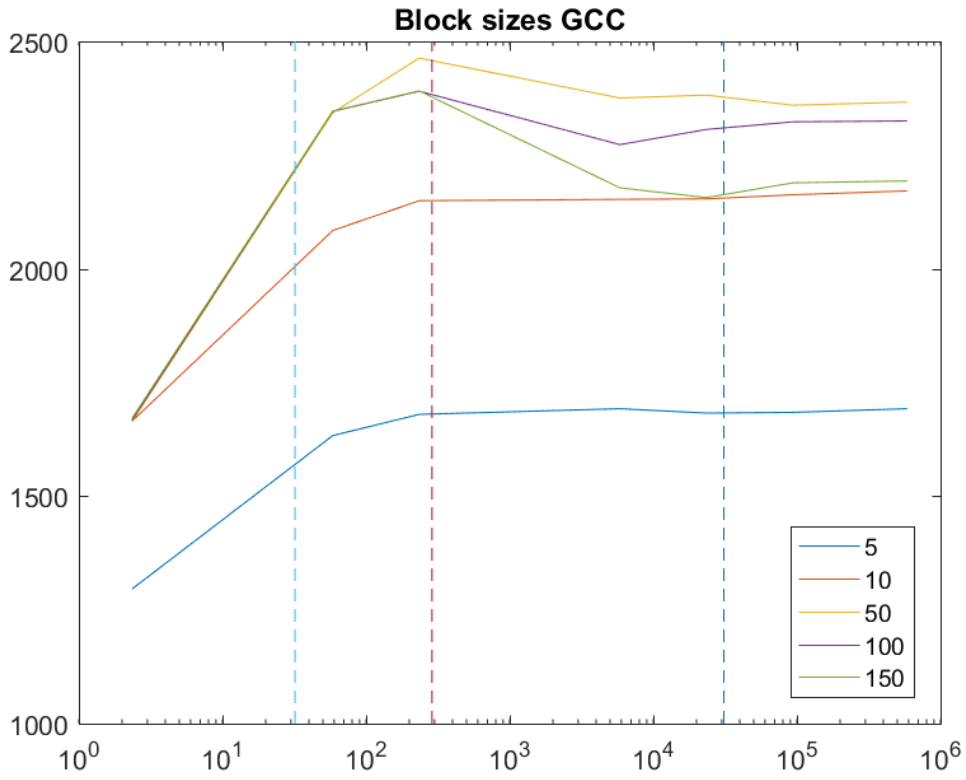


Figure 6.3: Comparison on different block sizes in an executable built with GCC.

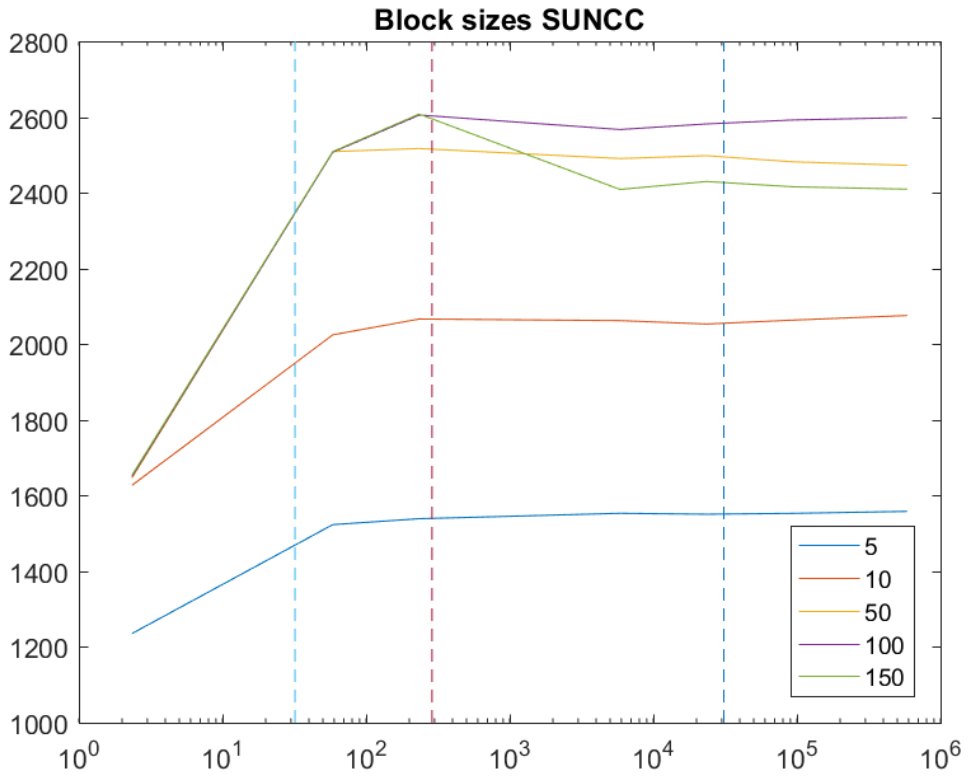


Figure 6.4: Comparison on different block sizes in an executable built with the SUN compiler.

As we can see from the two plots, the block size that better fits the cache for the executable built with GCC is the one with 50, with a processing rate at around 2500 MFLOPS.
Instead, the SunCC executable performs a little better using a block size of 100, and it's able to run at an highest rate, approximately 2600 MFLOPS.

By loading specific blocks into memory and by working only on them as if they were sub-matrix, we can increase the number of cache-hits and reduce the number of cache-misses, which means an increase in performances: this is known as the *spatial locality* principle.

7 | Performance analysis

In order to study the performance of the routines we used the software Solaris Studio Analyzer for both the loop interchange and the loop blocking. We specifically compared the number of cache hits and cache misses for problems of different sizes.

Since matrices in C are stored in row-major order any loop interchange causes variances in the number of Cache misses and Cache hits and thus in the global performance of the routines. We again saw how the the ratio between hits and misses for the MKN routine is the largest, which means that the CPU makes better use of the memory that is accessed every time, see table 7.2.

We have gathered the number of cache misses and cache hits for different block sizes in table 7.1. As we said the loop blocking technique tries to keep the memory needed every time close to the CPU, and thus create spatial locality, by dividing the total size in chunks of size similar to the cache.

Block size	L1 hits	L1 misses	L2 hits	L2 misses
5	33249970097	3159996	0	2000009
10	29329973633	37919895	31599910	6000020
50	25129977406	259119265	255959272	2000007
100	25279977299	183279481	173799505	7000022
150	24849977656	107439697	18959947	87000264

Table 7.1: Analysis of the hits and misses with different block sizes (executable built with SunCC).

Combination	L1 hits	L1 misses	L2 hits	L2 misses
nkm	764679311805	74879146741	69924280848	4954014866

Table 7.2: Analysis of the hits and misses of a *nkm* for-cycle permutation.