Matrix Multiplication

Authors

Chandykunju Alex - s200113
Santiago Gutiérrez Orta - s200140
Carlos Marcos Torrejón - s202464
Raúl Ortega Ochoa - s202489

02614 High-Performance Computing Jan 21

January 9, 2021

Contents

Part	studyno.	Name
I	s200140	Santiago Gutierrez Orta
II	s202489	Raúl Ortega Ochoa
III	s202464	Carlos Marcos Torrejón
IV	s200113	Chandykunju Alex

I Introduction

This report is the part of DTU course on High performance Computing 02614. In this report we will be dealing with the underlying basic principle of high performance matrix multiplication. To attain the best performance of matrix multiplication we need to analyse and understand how operation must be carried out in the high performance machine.In this assignment there is four major part which are:

- Native Implementation and correctness checking with CBLAS.
- Loop Nesting Permutation test.
- Performance Analysis
- Loop Blocking

I.I Experimental Hardware Specification

```
1  Architecture:            x86_64
2  CPU op-mode(s):          32-bit, 64-bit
3  Byte Order:              Little Endian
4  CPU(s):                  24
5  On-line CPU(s) list:     0-23
6  Thread(s) per core:      1
7  Core(s) per socket:      12
8  Socket(s):               2
9  NUMA node(s):            2
10 Vendor ID:               GenuineIntel
11 CPU family:              6
12 Model:                   79
13 Model name:              Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
14 Stepping:                1
15 CPU MHz:                 2200.000
16 CPU max MHz:             2900.0000
17 CPU min MHz:             1200.0000
18 BogoMIPS:                4389.79
19 Virtualization:          VT-x
20 L1d cache:               32K
21 L1i cache:               32K
22 L2 cache:                256K
23 L3 cache:                30720K
24 NUMA node0 CPU(s):       0-11
25 NUMA node1 CPU(s):       12-23
```

II The Assignment

II.I Native Implementation and correctness checking with CBLAS

On this section, a native matrix multiplication function is implemented. This function is computed using the summation of an element-wise multiplication between a column of the first matrix and a row of the second one as it is shown below.

$$c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{kj}$$

Where c_{ij} are the elements of the output matrix C_{mxn}, a_{ik} the elements of the first matrix A_{mxk} and b_{kj} from the second matrix B_{kxn}.

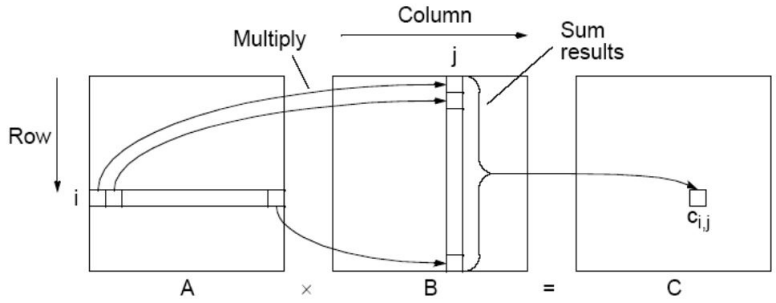

Figure 1: Sum element-wise column-row multiplication

This elements are obtained by doing the above operation inside a three nested for loops, each of them running through a different matrix dimension (m ,k or n). On this project the order choice for the loops in the native function was m -> n -> k.

```
1  /* Original permutation for matrix multiplication (mnk)*/
2  void matmult_nat(int M, int N, int K, double **A, double **B, double **C){
3  int k, n, m;
4      /* matrix multiplication */
5      for (m = 0; m < M; m++) {
6          for (n = 0; n < N; n++) {
7                          for (k = 0; k < K; k++) {
8                                  C[m][n] += A[m][k] * B[k][n];
9                          }
10         }
11     }
12 }
```

In order to compare the performance of the implemented function, it was compared with the DGEMM() function from the CBLAS[?] library.

```
1 /* Native CBLAS implementation of matrix multiplication */
2 void matmult_lib(int M, int N, int K, double **A, double **B, double **C) {
3         double alpha = 1.0, beta = 0.0;
4         cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,M,N,K, alpha, A[0], K, B[0], N, beta, C
              [0], N);
5 }
```

The CBLAs functions are highly optimized by specialist in the field, becoming the standard building blocks for performing basic vector and matrix operations. Therefore, its matrix matrix multiplication function can be considered as the ideal performance. The contrast between both functions can be visualize in the figure below.

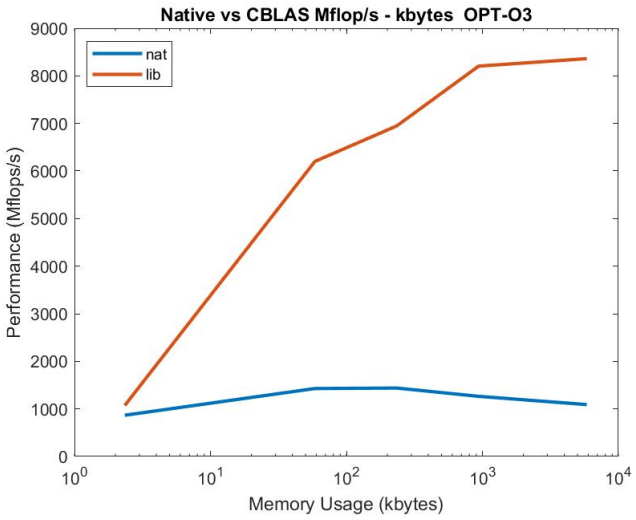

Figure 2: Performance measured (MFlops/s) respect to memory usage (KB).

It can be clearly observed the gap between the performance with respect to the memory usage that describe the different functions. This gap is the range of improve that our rough function may have.

II.II Loop Nesting Permutation test

Loop interchange is one general optimization technique. In nested loops, the order in which we access the loops can affect the performance as some programming languages are generally faster accessing rows than columns (C) while other are faster accessing columns than rows (FORTRAN), for example.

The matrix multiplication routine has 3 nested loops, so there are 3! = 6 different combinations of the order of loops. In the previous section the native implementation used *mnk* combination and had already splitted the loop initializing the C matrix, which now allows the loops to be interchanged. For example the combination *mkn*:

```
1  void matmult_mkn(int M, int N, int K, double **A, double **B, double **C) {
2  int k, n, m;
3
4  /* fill C matrix */
5  for (m = 0; m < M; m++) {
6      for (n = 0; n < N; n++) {
7          C[m][n] = 0;
8      }
9  }
10 /* matrix multiplication */
11 for (m = 0; m < M; m++) {
12     for (k = 0; k < K; k++) {
13         for (n = 0; n < N; n++) {
14             C[m][n] += A[m][k] * B[k][n];
15         }
16     }
17 }
18 }
```

To test which permutation performs best, we compute matrix multiplication with the 6 permutations for a range of matrices of different sizes. In the figures below the results are shown for the **gcc** compiler using the optimizers **Ofast, O3** and **Oo** (without optimizer), where the performance (Mflops/s) is shown as a function of the memory usage (kbytes).

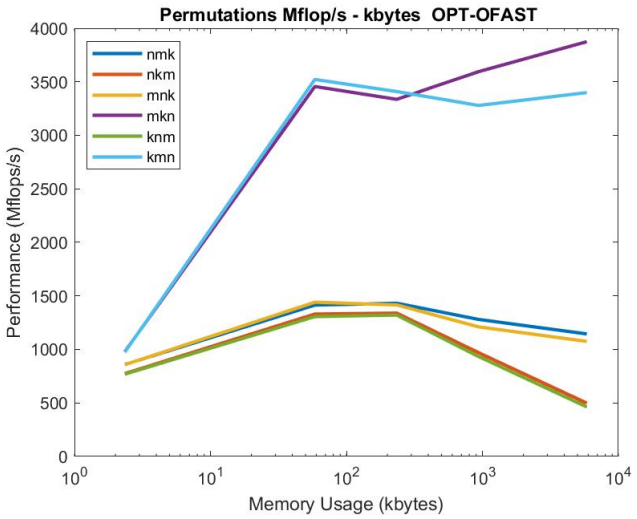

Figure 3: Loop order permutation test with GCC compiler, Ofast optimizer

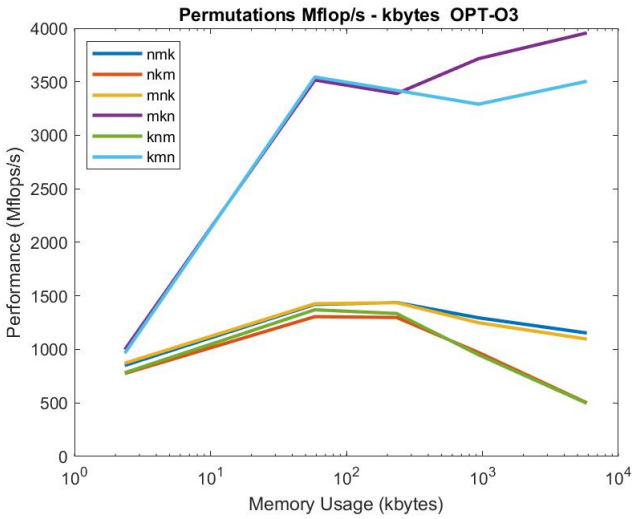

Figure 4: Loop order permutation test with GCC compiler, O3 optimizer

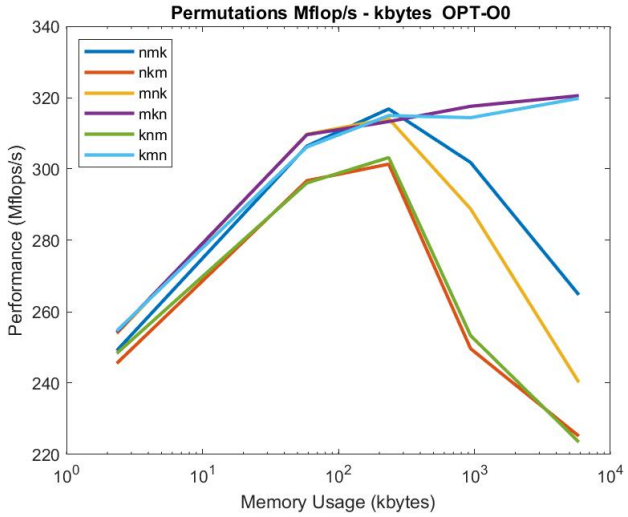

Figure 5: Loop order permutation test with GCC compiler without optmizer

Even though the scale change between the three figures above, the results are quite similar. Obtaining as optimal loop order the permutation *mkn*.

Shown below is presented a comparison of the best permutation with the CBLAS. This allow us to visualize how our algorithm has improved with respect to the high quality function.

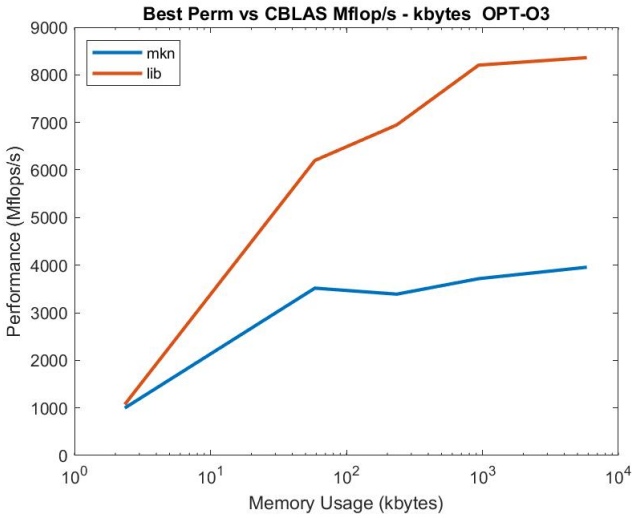

Figure 6: Comparison of performance with best permutation *mkn* vs CBLAS. GCC compiler O3 optimizer.

II.III Performance Analysis

For the next part we will use Solaris Studio Analyzer to study the CPU and Caches performance . Tables 1, 2 and 3 contain the results obtained for matrix multiplication of dimensions $(m, n, k) = (100, 100, 100)$ and 1000 maximum iterations for the specified permutations using three different optimizers.

Excl. Total CPU sec.	Incl. Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 Cache Hits	Excl. L2 Cache Misses	Name
38.807	38.807	132422670367	169653056	163251053	3003006	<Total>
6.645	6.645	22022063256	60819019	57618018	2002004	matmult_nkm
6.635	6.635	22022063243	35211011	38412012	1001002	matmult_knm
6.394	6.394	22032073293	35211014	32010012	0	matmult_mnk
6.384	6.384	22142183619	16005005	12804005	0	matmult_kmn
6.384	6.384	22102143478	6402002	9603002	0	matmult_mkn
6.354	6.354	22102143478	16005005	12804004	0	matmult_nmk

Table 1: Performance analysis with no optimizer.

Excl. Total CPU sec.	Incl. Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 Cache Hits	Excl. L2 Cache Misses	Name
6.935	6.935	17067099004	512160163	508959161	2002004	<Total>
1.461	1.461	4884894046	121638038	118437041	1001003	matmult_knm
1.461	1.481	4844853915	131241042	134442041	0	matmult_nkm
1.401	1.401	3003008615	12804002	9603001	1001001	matmult_nmk
1.391	1.391	2852858178	118437043	118437042	0	matmult_mnk
0.580	0.590	760762177	64020019	64020017	0	matmult_mkn
0.570	0.590	720722073	64020019	64020019	0	matmult_kmn

Table 2: Performance analysis for O3 optimizer.

Excl. Total CPU sec.	Incl. Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 Cache Hits	Excl. L2 Cache Misses	Name
6.955	6.955	17067099019	508959162	508959159	1001002	<Total>
1.501	1.501	4844853910	134442041	131241040	1001002	matmult_nkm
1.481	1.481	4884894058	118437038	118437039	0	matmult_knm
1.391	1.401	2852858185	118437043	118437040	0	matmult_mnk
1.391	1.391	3003008613	9603001	12804000	0	matmult_nmk
0.590	0.590	720722065	64020021	64020021	0	matmult_kmn
0.590	0.590	760762188	64020018	64020019	0	matmult_mkn

Table 3: Performance analysis for Ofast optimizer.

Comparing the results in Tables 1, 2, 3 we observe a great increase in the total L1 D-cache hits as we implement faster optimizers, but at the expense of higher L1 D-cache Misses as well. This indicates that while we are considerably increasing the flow of data into the microprocesors' CPU it is not directly leading to better results at a higher speed. If we attend to the different permutations routines implemented for the matrix operations, we can see that the *mkn* permutation operation -which provided one of the noticeably best performances among the others, provides one of the best balances Hits/Miss with the three used optimizers. It maintains a high level of L1 D-cache and L2 Cache hits while keeping L1 D-cache and L2 Cache Misses low compared with the other routines.

II.IV Loop Blocking or Loop Tiling

Loop blocking is another general optimization technique that helps improve memory access (spatial locality) and data re-use (temporal locality). Loop blocking reduces the number of times we load the same data, by performing a block of operations using the same data at a time, thus reducing the times it has to be loaded.

In *Section 2* we concluded the optimal order for the nested loops is the combination *mkn* after comparing its performance with the other 5 permutations over a range of different matrices sizes, so now the Loop blocking optimization will be used to find the optimal block size for the *mkn* permutation.

As stated before,matrix multiplications requires 3 nested loops. For loop block optimization now an extra loop is added for every existing loop, resulting in a total of 6 nested for loops. As seen in the following *matmult_blk* function.

```
1  /* Matrix multiplications using batches (for best permutation mkn)*/
2  void matmult_blk(int M, int N, int K, double **A, double **B, double **C, int bs)
3  {
4      int m0, n0, k0, m, k, n;
5     /* case batch is too large */
6     bs = fmax(1, fmin(bs, K));
7
8      /* Fill in C matrix */
9      for (m0 = 0; m0 < M; m0++) {
10         for (n0 = 0; n0 < N; n0++) {
11             C[m0][n0] = 0;
12         }
13     }
14     /* Matrix multiplication with batches */
15    for (m0 = 0; m0 < M; m0 += bs) {
16       for (k0 = 0; k0 < K; k0 += bs) {
17          for (n0 = 0; n0 < N; n0 += bs) {
18             for (m = m0; m < fmin(m0 + bs, M); m++) {
19                for (k = k0; k < fmin(k0 + bs, K); k++) {
20                   for (n = n0; n < fmin(n0 + bs, N); n++)
21                      C[m][n] += A[m][k] * B[k][n];
22                }
23             }
24          }
25       }
26    }
27 }
```

To test which block size performs best, we compute matrix multiplication for a range of block sizes. As stated previously, the use of a faster optimizer leads to higher compilation speed at expenses of the reliability of results. There is a big gap in performance between O3 and no use of optimizer, while the difference between *O3* and *Ofast* is relatively small in comparison. In addition, considering the performance analysis of the caches and results obtained for different permutations in section 3, we have decided to use O3 as the optimal optimizer option among the three explored for mkn permutation. When adding loop blocking to the mkn permutation operation we see that the performance is lower (Fig. 7) than the one seen for the same permutation in Figure 4. This is an unexpected result since blocking should lead to a better usage of the cache available in the machine as can be seen in Table 4 using, 25 blocksize in $(m, n, k) = (100, 100, 100)$ matrix operation. The blocked version presents a considerably higher amount of L1 D-cache Hits and less L1 D-cache misses along with higher CPU usage.

Excl. Total CPU sec.	Incl. Total CPU sec.	Excl. L1 D-cache Hits	Excl. L1 D-cache Misses	Excl. L2 Cache Hits	Excl. L2 Cache Misses	Name
0.580	0.590	760762177	64020019	64020017	0	matmult_mkn
1.491	1.501	3093098877	3201005	3201001	0	matmult_blk

Table 4: Performance analysis for O3 optimizer.

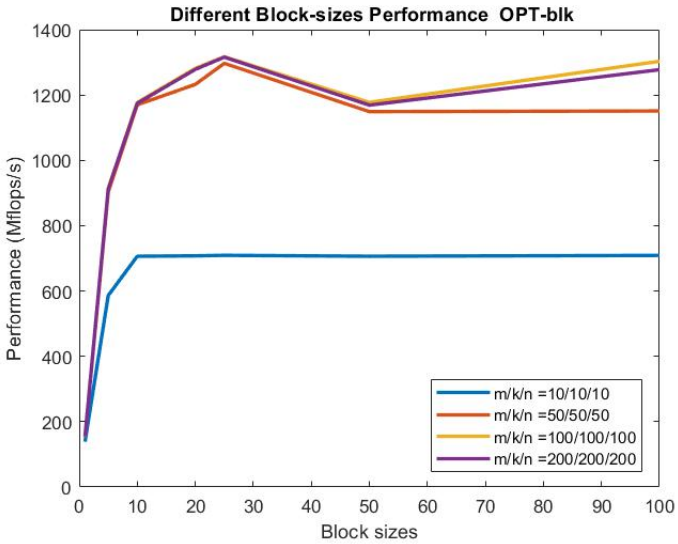

Figure 7: Loop blocking Optimizing test with GCC compiler, O3 optimizer

We used loop blocking for a total of 4 operations with different matrix sizes as stated in Figure 7. For the smaller matrices the performance flattens for block sizes bigger than the matrix size itself due to the restrictions specified in the code above, adding robustness to

the code. The bigger matrices operations show a local maximum at 25 blocksize, an optimal blocking size. Bigger matrices operations might have maximums at higher blocksizes.

In the following graph it can be observed the performance with respect to the memory used for each block size tested.

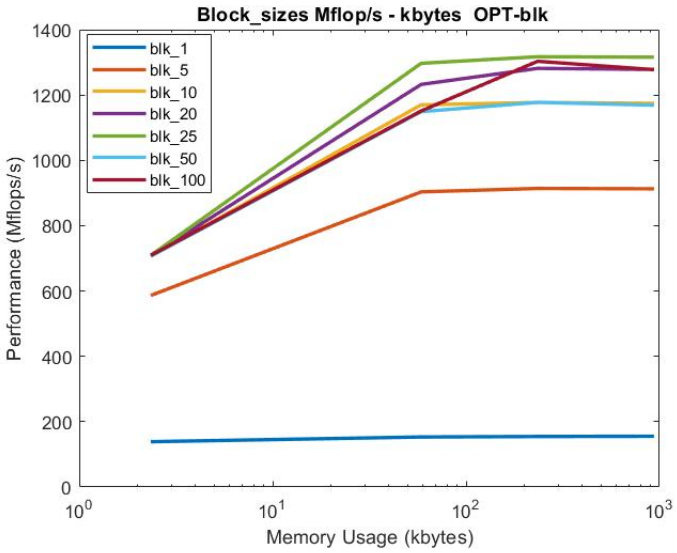

Figure 8: Performance comparison WRT. block size

This graph is meant to give a notion of how the performance improve from block size=1. It is not used as a result, since, the test was carried out adding more loops to do the blocking computation automatically. Therefore, increasing the computation and downgrading the performance of the algorithm. However, it can be helpful to visualize the improvement between the optimal set-up with block size 25 and the standard set-up with block size 1.

Even though the blocking algorithm was implemented correctly with respect to the theory. The optimization was poorly as it can be inferred from the following figure.

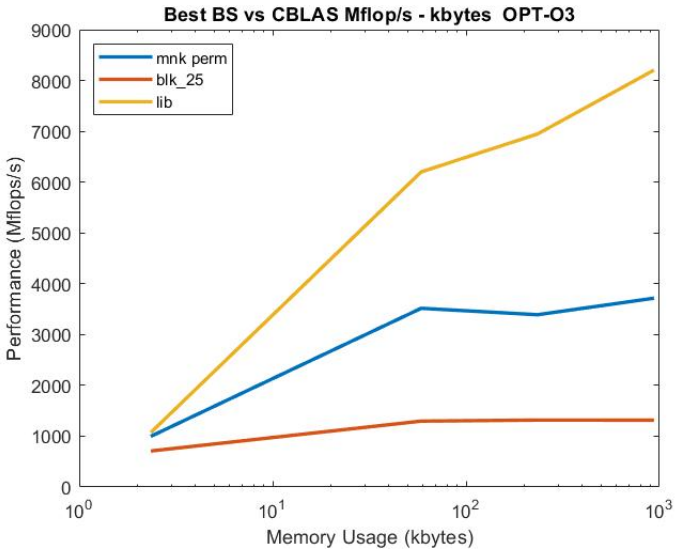

Figure 9: Performance comparison CBLAS, mnk perm and Block size 25

III Conclusion

At the beginning of the project we started with a rough matrix-matrix multiplication function with a low performance *matmult_nat()*. This performance was compared with the high efficiency subroutine *dgemm()* from CBLAS library. Showing us that there was a big room for improvement as can be seen in *figure 2*.

The first step in order to optimize our code was test with different loop order permutations. We could observe that having an inner n-loop improved the performance greatly (*figures 3, 4, 5*). That is due to the memory access pattern, with the inner n- loop we satisfy the row-major storage pattern used on C and C++ which are the programming languages that we used. Thus, indexing contiguous memory locations, for contiguous row elements. We further checked this result by analysing the cache performances for each of the coded permutations.

As a final optimization of the *mkn* permutation we implemented loop blocking and found the optimal blocksize to be around 25 for matrices with sizes on the order of 100s (*figure 7*).

List of Figures

List of Tables