Performance Measuring and Code Profiling of Matrix Multiplication

University of Minho

Adriana Meireles & Shahzod Yusupov

A82582 A82617

Email:a82582@alunos.uminho.pt Email:a82617@alunos.uminho.pt

***Abstract*— The main goal of this project is to know how to take adavntage of the full potencial of an HPC environment, using the matrix multiplication alghoritms as a case of study.**

Throughout this work there will be different implementations of this algorithm in order to improve the performance, using the 662 node from the SeARCH cluster at University of Minho. It will be made several changes such as a study of the metrics used to make all the measurements, code profiling, the size of the inputs in correlation with the hardware characteristics (cache size) and its performance analysis on the different versions and respective conclusions.

I. Introduction

Overall performance is determined by the complex interaction of source code, the compiler used,the architecture and microarchitecture on which the code is run. Because of this, performance is hard to estimate, understand, and optimize. This is particularly true for numerical or mathematical functions that often are the bottleneck in applications from scientific computing, machine learning, multimedia processing, and other domains. As a performance engineers it is our job to be able to study a specific problem in order to take advantage of every resource available in the machine to improve the overall performance.

In this paper we will focus specifically on matrix multiplication, that, as mentioned before, can be a central operation in many numeric alghoritms and potencially a time consuming operation. To understand the hardware in our disposal there will be some characterizations along the paper and its associated roofline model. We will study how some changes like the size of the matrices, simple rearrengements of the loops and some optimizations like blocking can impact the overall performance of our program. Every measurement and result will be explained taking into account the architecture used to run our code, with the help of the Performance API (PAPI).

At the end we will also study the behavior of the program in a many-core processor Intel Knights Landing and in a Nvidia GPU Kepler20 and take some conclusions about the work done and the results obtained.

II. Matrix Dot-Product Algorithm Analysis

Matrix dot-product algorithm consists in the computation of C = A x B, being A, B and C three squared matrices where each line/column has N elements.

A. Basic Implementations

This study focus on the implementation of three basic versions of this algorithm, without optimizations, where the only difference is the loops rearrengement (IJK, IKJ, JKI) wich are identified by their indexes I, J and K.

At high level the three versions are quite similar. If addition is associative, then each version computes an identical result. Each version performs $O(n^3)$ total operations and an identical number of adds and multiplies. Each of the n^2 elements of A and B is read n times. Each of the n^2 elements of C is computed by summing n values. However, if we analyze the behavior of the innermost loop iterations, we find that there are differences in the number of accesses and the locality.

1)**IJK**: This is the version used in the conventional matrix multiplication where each element of matrix C is computed with a line of matrix A and

a column of matrix B. Notice that the matrix B column accesses have an impact in performance, so to reduce this negative impact, is created a version where matrix B is transposed.

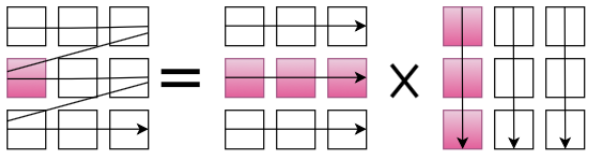

Fig. 1. IJK iteration

2)**IKJ**: For this version, the computation is done by using an element of matrix A and a line of matrix B and storing the result in matrix C, iterating the line. All accesses are row-wise so there is no need to tranpose any matrix.

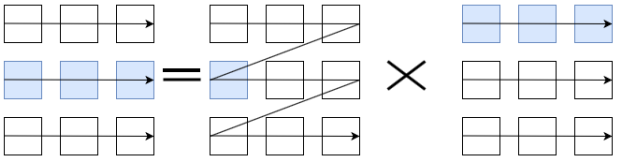

Fig. 2. IKJ iteration

3)**JKI**: Each element of matrix C is computed using a line of matrix A and a column of matrix B, but all accesses are column-wise, reason why the matrices need to be transposed.

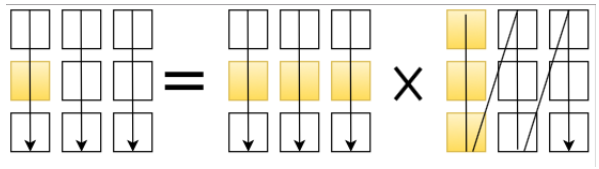

Fig. 3. JKI iteration

4)**IJKTransposed**: This version is similar to IJK's version, the only difference is that the matrix B was transposed making the all accesses row-wise, as we can see in the figure 4

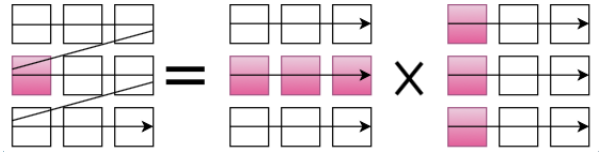

Fig. 4. IJK Transposed iteration

5)**JKI Transposed**: As mentioned before in the IJK algorithm all accesses on matrices are column-wise, so all of them were transposed. In this version the formula is the same as JKI but all accesses are now row-wise, reducing the negative impact on performance.

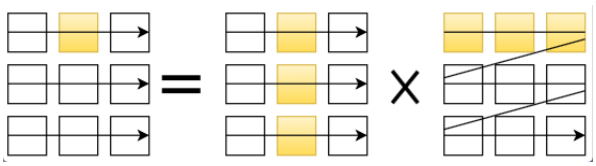

Fig. 5. JKI Transposed iteration

B. Experimental Setup

The machine used to do the measurements is a 662 node from the SeARCH cluster at University of Minho, that is equipped with a Intel Xeon E5-2695 v2 12 processors. All binaries are compiled with GCC(version 4.9.0) and between measures, all caches were clean so the values are calculated from a cold cache.

To measure time the technique we used was the K-Best scheme with K=3, a 5% tolerance and at most 8 executions, where we measure five executions of the dot-product implementation, chose the best three and the best needed to had a 5% tolerance to the other two values. If this does not happen the process is repeated a maximum of eight times.

C. Data-set sizes

After the analysis of the implementations,we used the following formula to calculate the size of the matrices so it can fit in each cache level:

$size^2 * 3 * 4 \leq$ *Cache Level size*

- **3**: number of matrices
- **4**: float size(4 bytes)
- **Ki**:1024 bytes

1) L1
size:32KiB
N*N*3*4 $\leq$ 32768 $\rightarrow N < 52$
2) L2
size:256KiB
N*N*3*4 $\leq$ 262144 $\rightarrow N < 148$
3) L3
size:30720KiB
N*N*3*4 $\leq$ 31457280 $\rightarrow N < 1619$

4) RAM
size:122880KiB
N*N*3*4 $\leq$ 125829120 $\rightarrow N > 3238$

After we calculated the limit of each cache level, the considered sizes were:

TABLE I
MATRIX SIZES

Cache Level	Size
Cache L1	32x32
Cache L2	128x128
Cache L3	1024x1024
Main Memory	3248x3248

D. Execution time measurements

For the implementations explained before, the running time was mesured for the different matrix sizes.

TABLE II
DOT-PRODUCT TIME MEASUREMENTS IN MILISECONDS

	32x32	128x128	1024x1024	3248x3248
IJK	0.179	11.818	9709.139	57956.168
IJK Transposed	0.175	10.800	5440.142	43515.565
IKJ	0.173	10.774	5433.377	43377.254
JKI	0.183	11.751	16416.400	81939.947
JKI Transposed	0.182	11.200	9478.924	55193.784

For the implementation 32x32 we can't derive solid conclusions because the size is reduced. However, with JKI it's obtained worst performance.

With matrix size 128x128 the values are still similar due to the small data-set size, but we can see that JKI is the worst implementation as for the smallest size.

For the 1024x1024 data-set size, the differences between the several implementations in execution times grows and we can clearly see that JKI implementation is the worst due to the column-wise access with a time of 16416.400 milliseconds, very different from its transpose implementation (changes the accesses to row-wise) who takes 9478.924 milliseconds to execute. On the other hand, IKJ is the best implementation due to both A and B matrices are iterated row-wise.

For last, the 2048x2048 data-set size, the same pattern is followed about JKI as the worst implementation and about IKJ best performance.

III. ALGORITHM BEHAVIOR ANALYSIS

A. Main Memory behavior

In order to analyze the RAM behavior we decided to estimate RAM accesses per instruction and the number of bytes transferred to/from the RAM. The algorithm contains 3 nested cycles where each one iterates size times resulting in $size^3$ iterations.On each iteration of the algorithm one element of A, B and C are accessed.

When we access the matrix row-wise, we only need to access RAM every 16 elements because of the size of the cache line that can hold 16 float elements. However, when we access the matrix column-wise we assume that the worst case scenario where every element accessed will be on RAM, which might not be true. For the transposing matrix each element accessed represents one RAM access.

As previously stated, to calculate theoretical main memory accesses of a matrix we used the following formulas:

- **Row-wise access**:$size$ x $\frac{size}{cache\_line\_width}$
- **Column-wise access**: $size$ x $size$
- **Transposing access**:$size^2$ per iteration x 2 values loaded per iteration

The following results were obtained by using PAPI counters,**PAPI_L3_TCM** e **PAPI_TOT_INS**:

TABLE III
RAM ACCESSES/ INSTRUCTION

	32x32	128x128	1024x1024	3248x3248
IJK	0.000953	0.000146	0.000031	0.000173
IJK Transposed	0.000729	0.000124	0.000012	0.000090
IKJ	0.000506	0.000115	0.000006	0.000051
JKI	0.000977	0.000161	0.000033	0.000243
JKI Transposed	0.000684	0.000123	0.000029	0.000155

To calculate the RAM traffic we need to multiply the **level 3 cache misses** (RAM Accesses) by 64, which is the number of bytes that each access moves to cache. The results are featured below:

TABLE IV
DATA TRAFFIC FROM/TO RAM IN KBYTES

	32x32	128x128	1024x1024	3248x3248
IJK	14	138	17183	759216
IJK Transposed	12	134	6003	347137
IKJ	12	116	3110	195798
JKI	14	175	19631	1068746
JKI Transposed	10	155	17986	850758

B. Floating Point Performance

To calculate floating point operations to all implementations we use the formula:

$FP\ Operations = 2 * size^3$

The results on the table below are the floating point operations for each data-set sizes:

TABLE V
FLOATING POINT OPERATIONS

	FP Operations
32x32	65536
128x128	4194304
1024x1024	2147483648
3248x3248	68529577984

We plotted the achieve performance on the Roofline Model. To calculate GFlops/sec and Flops/byte we used the following formulas respectively:

Flops/byte: $\frac{FPOperations}{DATA\_TRAFFIC\_FROM/TORAM}$

GFlops/sec: $\frac{FPOperations}{(Texec(s))}$ x 10^{-9}

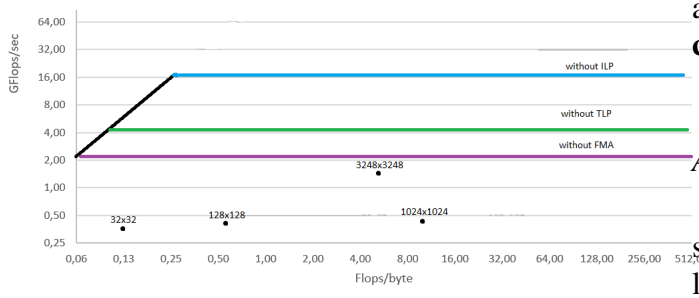

Fig. 6. IKJ Performance

C. Cache behavior

To understand the cache behavior we calculated the miss rate percentage on memory reads for each cache level with PAPI counters. So following formulas:

- **L1 miss rate**: PAPI L2 DCR / PAPI LD INS
- **L2 miss rate**: PAPI L3 DCR / PAPI L2 DCR
- **L3 miss rate**:PAPI L3 TCM / PAPI L3 TCA

TABLE VI
MISS RATE FOR IJK IMPLEMENTATION

		32x32	128x128	1024x1024	3248x3248
IJK	L1	0.33%	2.49%	60.55%	72.70%
	L2	45.21%	2.61%	3.66%	51.92%
	L3	98.90%	54.84%	0.23%	0.19%
IJK Transposed	L1	0.38%	4.24%	3.27%	3.22%
	L2	89.16%	3.83%	8.64%	6.10%
	L3	97.05%	21.36%	1.47%	8.98%

As we can see the results of the table above, small data-sets, for the algorithm without matrix transposition, fit entirely on L1 cache so the miss rates for this sizes are very low unlike L2 and L3 that are very high.For the size (128*128) the cache L3 as a high miss rate and the cache L2 as a low miss rate value because the matrix fits completely in cache L2.

Despite the cost of transposing the matrix which makes miss rate worst in some cases, it provides an row-wise access to data, which will improve reuse of data on cache, improving the execution time.

In summary, after the analysis of memory usage and operations of the dot-product multiplication together with the identification of different bottlenecks, we realize that this implementation takes most of the execution time on floating point operations than on readings/writings which makes it a **cpu-bound** algorithm.

IV. OPTIMIZATIONS

A. Block optimization

Simple rearrangements of the loops can increase spatial locality, but observe that even with good loop nestings, the time per loop iteration increases with increasing array size. What is happening is that as the array size increases, the temporal locality decreases, and the cache experiences an increasing number of capacity misses. To fix this, we can use a general technique called **blocking**.

The general idea of blocking is to organize the data structures in a program into large chunks called blocks. The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Blocking a matrix multiply routine works by partitioning the matrices into submatrices and then

exploiting the mathematical fact that these submatrices can be manipulated just like scalars.

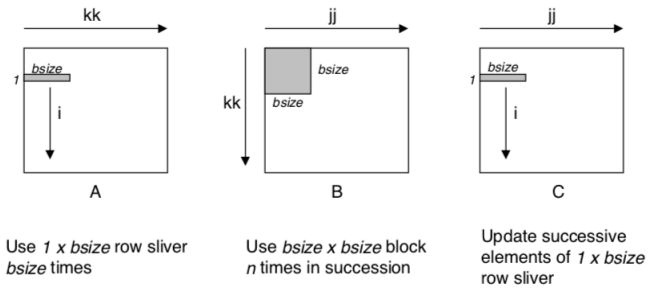

Fig. 7. IJK Transposed iteration

B. Block Optimization and Multi-threading

Using blocking the performance of the program improves a lot, but there is still many room to improve. Multi-threading alows to run our program in parallel using OpenMP, spliting the matrix blocks by the 24 cores of the cpu in the 662 node.

C. Block optimization, Multi-threading and Vectorization

Combining the previous techniques with vectorization, wich alows to apply one instruction over multiple elements stored on vector registers, we can achieve the best overall performance. The node 662 features AVX2, being able to run 256 bit instructions (8 elements).

D. Results

TABLE VII
BLOCKING IMPLEMENTATIONS EXECUTION TIME IN MILISECONDS

	32x32	128x128	1024x1024	3248x3248
Blocking	0.078	5.012	1335.561	139223
Block+Vec	0.074	4.017	-	-
Block+Vec+OMP	-	-	-	109213

E. Many-core processor

Intel Knights Landing was the many-core processor used for this purpose, changing the number of threads and the size of data sets. We can observe that in the first three data sets, using 64 threads shows better performance while in the bigger matrix using 256 threads is the best choice to obtain better results.

TABLE VIII
TIME OF KNL IN MILISECONDS

	32x32	128x128	1024x1024	3248x3248
32	1,313	95,007	54575,422	1533417,241
64	1,251	88,215	54029,713	1612224,608
128	1,328	90,724	54656,817	1613424,52
256	1,561	98,956	53860,615	1479027,755

F. GPU

We develop an implementation that takes the advantage of the shared memory in a block so there was performance improvement that if we used an easier implementation because of the better optimized memory accesses. We obtained the following results:

TABLE IX
TIME MEASUREMENTS USING CUDA IN MILISECONDS

	32x32	128x128	1024x1024	3248x3248
CUDA	282.61	280.87	293.46	350.19

V. CONCLUSIONS

Despite all the problems with cluster and not normal results the aim of understanding a computer system using different optimizations of an algorithm was accomplished. Various optimizations were tested with different hardware components in order to overall performance improves. Although there was improvements, however not as much as it was expected.

VI. APPENDIX

```
void matrixMult_block_Vec(float** A, float** B, float** C, int SIZE) {
  int i = 0, j = 0, jj = 0, kk = 0;
  float x[16];
  int b_SIZE = 16;


  for (jj = 0; jj < SIZE; jj += b_SIZE) {
    for (kk = 0; kk < SIZE; kk += b_SIZE) {
      for (i = 0; i < SIZE; i++) {
        for (j = jj; j < ((jj + b_SIZE) > SIZE ? SIZE : (jj + b_SIZE)); j++) {
          x[0] = A[i][kk] * B[kk][j];
          x[1] = A[i][kk+1] * B[kk+1][j];
          x[2] = A[i][kk+2] * B[kk+2][j];
          x[3] = A[i][kk+3] * B[kk+3][j];
          x[4] = A[i][kk+4] * B[kk+4][j];
          x[5] = A[i][kk+5] * B[kk+5][j];
          x[6] = A[i][kk+6] * B[kk+6][j];
          x[7] = A[i][kk+7] * B[kk+7][j];
          x[8] = A[i][kk+8] * B[kk+8][j];
          x[9] = A[i][kk+9] * B[kk+9][j];
          x[10] = A[i][kk+10] * B[kk+10][j];
          x[11] = A[i][kk+11] * B[kk+11][j];
          x[12] = A[i][kk+12] * B[kk+12][j];
          x[13] = A[i][kk+13] * B[kk+13][j];
          x[14] = A[i][kk+14] * B[kk+14][j];
          x[15] = A[i][kk+15] * B[kk+15][j];


          C[i][j] += x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7]
          + x[8] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[15];
        }
      }
    }
  }
}
```

Fig. 8. Vectorized Algorithm

```
void matrixMult_blockVecOMP (float **A, float **B, float **C, int SIZE) {
  int i = 0, j = 0, jj = 0, kk = 0;
  float x[16];
  int b_SIZE = 16;

#pragma omp parallel for private(i, j, jj, kk, x)
  for (jj = 0; jj < SIZE; jj += b_SIZE) {
    for (kk = 0; kk < SIZE; kk += b_SIZE) {
      for (i = 0; i < SIZE; i++) {

        #pragma omp simd
        for (j = jj; j < ((jj + b_SIZE) > SIZE ? SIZE : (jj + b_SIZE)); j++) {
          x[0] = A[i][kk] * B[kk][j];
          x[1] = A[i][kk+1] * B[kk+1][j];
          x[2] = A[i][kk+2] * B[kk+2][j];
          x[3] = A[i][kk+3] * B[kk+3][j];
          x[4] = A[i][kk+4] * B[kk+4][j];
          x[5] = A[i][kk+5] * B[kk+5][j];
          x[6] = A[i][kk+6] * B[kk+6][j];
          x[7] = A[i][kk+7] * B[kk+7][j];
          x[8] = A[i][kk+8] * B[kk+8][j];
          x[9] = A[i][kk+9] * B[kk+9][j];
          x[10] = A[i][kk+10] * B[kk+10][j];
          x[11] = A[i][kk+11] * B[kk+11][j];
          x[12] = A[i][kk+12] * B[kk+12][j];
          x[13] = A[i][kk+13] * B[kk+13][j];
          x[14] = A[i][kk+14] * B[kk+14][j];
          x[15] = A[i][kk+15] * B[kk+15][j];


          C[i][j] += x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7]
          + x[8] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[15];
        }
      }
    }
  }
}
```

Fig. 9. Multithreading Algorithm

```
void cuda (float*A, float*B, float*C, int N) {
  float *DA, *DB, *DC;

   int b_SIZE = 16;
  size_t size = N * N * sizeof(float);

  cudaMalloc(&DA, size);
  cudaMalloc(&DB, size);

  cudaMemcpy(DA, A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(DB, B, size, cudaMemcpyHostToDevice);

  cudaMalloc(&DC, size);

  dim3 dimBlock(b_SIZE,b_SIZE);
  dim3 dimGrid((N + dimBlock.x -1) / dimBlock.x, (N + dimBlock.y -1 )/ dimBlock.y);

  cudaBlockKernel<<<dimGrid, dimBlock>>>(DA, DB, DC, N);

  cudaDeviceSynchronize();

  cudaMemcpy(C, DC, size, cudaMemcpyDeviceToHost);

  cudaFree(DA);
  cudaFree(DB);
  cudaFree(DC);
}
```

Fig. 10. Cuda Algorithm

REFERENCES

[1] Computer Systems: A Programmer's Perspective
[2] http://gec.di.uminho.pt/miei/cpd/aa/code_optim.pdf
[3] http://gec.di.uminho.pt/miei/cpd/aa/papi.pdf
[4] http://gec.di.uminho.pt/Discip/MaisAC/CS-APP_Bryant/csapp.ch6.pdf
[5] Roofline:http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/ispass-2013_177.pdf