# Matrix-matrix Multiplication, Performance Measuring and Code Profiling

Universidade do Minho
Lucas Pereira & Nuno Silva
A68547 A78156

***Abstract*— The main objective of this paper is to demonstrate the various changes that can be applied to improve performance. For this, we make use of the matrix multiplication algorithm, and the SeARCH cluster of the University of Minho, more concretely of node 662. Therefore, in this report, we will observe several implementations that will have several differences in order to improve the performance in the SeARCH cluster. All of these implementations are based on an analysis of the hardware architecture against which the program will run and the results obtained by the counter provided by the PAPI framework, for example, floating point operations and multi-level cache access. In addition, considerations were made with input sizes, which were correlated with hardware characteristics, cache size, and index order changes. Also making use of simple algebraic operations such as transposing arrays for better usage of the hardware in question.**

## I. Introduction

In the research world, it is common to encounter intense computational problems that require an enormous amount of resources. A performance engineer may be able to study a given problem and identify how changes are needed to make the best use of available resources. In this article we will focus on matrix multiplication, and capture any bottleneck that causes a decrease in overall performance and devise a solution to reduce the execution time, to increase the performance of this multiplication. Initially, for a piece of more in-depth knowledge about this challenge and the resources that are available, the use of a *roofline model* to understand generally how different algorithms behave and possible bottlenecks. Subsequently, the impact of different sizes of a matrix, the exchange of its indexes and the use of matrices that affect an overall performance is studied. The results displayed must be presented and explained, taking into account the resources used and with the help of PAPI framework counters. This will present useful information for finding or improving performance.

## II. Hardware Specification

First of all, we carry out an analysis of the available hardware, it will indicate any bottleneck or limitations of the hardware as well as possible reasons for a loss of performance. We use node 662 with an *Intel Xeon Processor E5-2695-V2*. The specifications for this node were obtained directly from the cluster and the processing unit website[1] [2] [3].

Cluster Specification

| **Processor** | |
|---|---|
| **Codename** | Ivy Bridge |
| **Model** | Intel Xeon E52695v2 |
| **# Cores** | 12 |
| **# Threads** | 24 |
| **Base Frequency** | 2.4 GHz |
| **Turbo Frequency** | 3.2 GHz |

| **Memory** | |
|---|---|
| **Cache L1** | 32 KiB for Data<br>32 KiB for Instructions |
| **Cache L2** | 256KiB |
| **Cache L3** | 30MiB |
| **Max Memory Bandwidth** | 59.7 GiB/s |
| **Main Memory** | 64 GiB |

## III. Roofline Model

The *roofline model*[5] offers a more in-depth and specific look at the factors that affect the performance of a system, by quantifying the influence of a bottleneck on the system. This look is transmitted to us in the form of a two-dimensional graph where the X-axis shows the measurements of floating-point per byte operations and Y-axis measures the peak floating-point performance.

Peak FP = Number of Processes x Number of Cores x Clock Frequency x SIMD width x FMA x CPI

Attainable GFlops/sec = Min(Peak FP, Peak Memory Bandwidth x Operational Intensity)

The roofline was obtained by applying simple bound and bottleneck analysis. The peak bandwidth was obtained using **Stream Benchmark** [4].

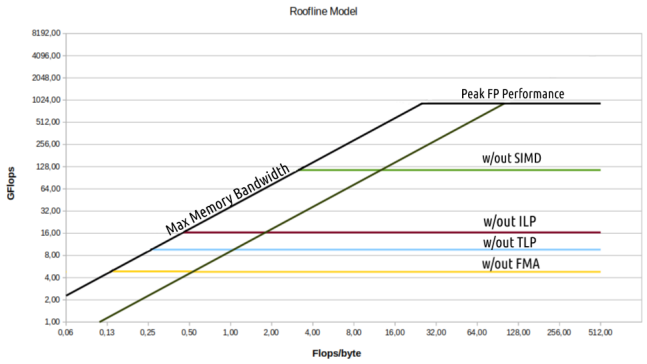


Fig. 1: Node 662 Roofline

For the cluster, who features AVX implementing SIMD of 256 bits (8 floats), the peak performance is obtained by the following calculation: 2 x 12 x 2.4 x 8 x 2 = 921.6 GFlops

## IV. PAPI Performance Counters

For a deeper knowledge on the algorithm behavior, we used the PAPI platform (version 5.4.1). This application, allow us the use of performance counters found in major microprocessors.

Papi Counters Selected

| Counters | Description |
|---|---|
| **PAPI_LD_INS** | Load Instructions |
| **PAPI_L2_DCR** | Level 2 data cache read |
| **PAPI_L3_DCR** | Level 3 data cache read |
| **PAPI_L3_TCM** | Level 3 total cache misses |
| **PAPI_L3_TCA** | Level 3 total cache accesses |

Using these native counters we can obtain values like *miss rate* for a certain cache level, ram accesses per instruction or the number of bytes the processor transferred from/to RAM.

## V. Matrix dot-product Algorithm Analysis

A product between matrices is an operation that generates a result matrix. Matrix dot-product algorithm consists of computing an algorithm in the format C = A x B, where A, B and C matrices are square.

After an initial analysis of the system, the work was started with the implementation of three versions of the dot-product function, without optimizations using only index reordering, **IJK**, **IKJ** and **JKI**, with the **IJK** version being the *default*. Each of these versions has differences in the access pattern of the matrix.

### A. Implementations

*1) IJK:* For each element in line, I of A multiplies for each element in column J of B. As we can see, C and A accesses are row-wise, although B are column-wise. After observing this, it was created a similar algorithm but using B transposed matrix.

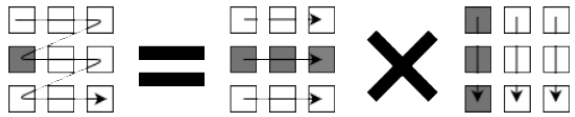

Fig. 2: IJK Algorithm

*2) IKJ:* In this algorithm, for each, A element multiplies it to a K line of B, and adds it to each element in line I of C. As we can see, this algorithm is already row-wise, and for that reason we did not created an algorithm using transposed any of the 3 matrixes

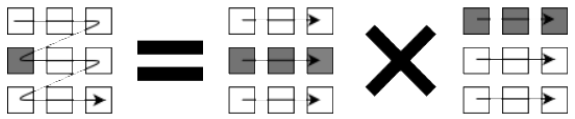

Fig. 3: IKJ Algorithm

*3) JKI:* For each B element, multiplies it for a K column of A and adds the result to each column J of C. As we can tell, all-access are column-wise, after this observation we created a similar algorithm using all matrixes transposed.

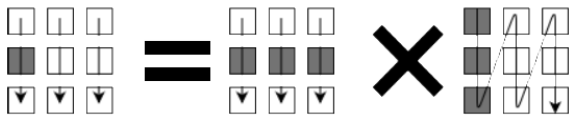

Fig. 4: JKI Algorithm

*4) IJK Transposed:* In this algorithm, we will iterate through each element of line J of matrix B, in spite of iterating through each element of J columns of matrix B. Since B was transposed, it will B row-wise, making all access row-wise as well.

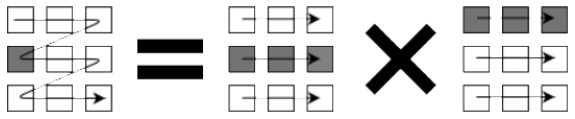

Fig. 5: IJK Transposed Algorithm

*5) JKI Transposed:* As previously mentioned, in this algorithm, we transpose all the matrices involved, so that the accesses are per line. And so to be able to take advantage of the spatial and temporal location.

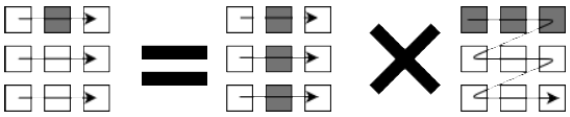

Fig. 6: JKI Transposed Algorithm

### B. Experimental Setup

The implemented optimizations were designed according to the processor that we will use, Intel Xeon E52695V2, with 12 cores. All generated files were compiled with GCC (version 5.3.0). Among all measurements, all caches have been cleared, in order to avoid data in the cache that influences some results. We followed the K-Best scheme technique for measurements of execution times, with a K = 3, and a tolerance of 5%, which means that after carrying 8 tests, we choose the 3 best from them and all must not differ 5% in result.

### C. Dataset Sizes

An important factor to decide, after obtaining the system specifications, is the size of the matrices that will later be used for performance tests. The sizes chosen took into account, the sizes of the caches. On which we use the following formula:

$$Matrices\_Size^2 * Float\_Size * Number\_Matrices \leq Cache\_Level\_Size$$

Matrices Sizes

| Memory Level | Size |
|---|---|
| **Cache L1** | 32 x 32 |
| **Cache L2** | 128 x 128 |
| **Cache L3** | 1024 x 1024 |
| **Main Memory** | 2048 x 2048 |

As we can see in the table above, only sizes multiples of 16 were considered, based on alignment purpose with the 64 byte cache.

### D. Execution Time Measurements

The next table shows the differences between the execution times and the influence of the matrix sizes on them. In general, the **JKI** implementation has the worst times, this may be due to its poor cache optimization. On the other hand, the **IKJ** implementation presents the best results, this due to taking advantage of the spatial and temporal locality. It also presents better results in relation to the algorithms that use transposed matrices, this due to not using them, thus avoiding a waste of time.

Execution Time

| Time Measures | | 32 x 32 (milliseconds) | 128 x 128 (milliseconds) | 1024 x 1024 (seconds) | 2048 x 2048 (minutes) |
|---|---|---|---|---|---|
| **IJK** | **Normal** | 0.25109 | 14.79968 | 9.385265 | 1.278071 |
| | **Transposed** | 0.24339 | 14.86301 | 7.45494 | 1.031743 |
| **IKJ** | **Normal** | 0.25096 | 14.65726 | 7.448425 | 1.028054 |
| **JKI** | **Normal** | 0.25505 | 15.01334 | 15.712192 | 2.903596 |
| | **Transposed** | 0.25949 | 14.81051 | 7.462713 | 1.032205 |

## VI. Algorithm Behavior Analysis

### A. Main Memory Behavior

To analyze the behaviour of the RAM, we decided to estimate the accesses per instruction to it and the number of bytes transferred to/from the RAM. With these metrics, we have a better idea that implementations that use column-wise iterations have more memory access than row-wise iterations. Because these allow better use of the cache and in turn faster without having huge miss penalty in accessing the RAM.

RAM Accesses per Instruction

| RAM ACCESSES per INSTRUCTION | | 32 x 32 | 128 x 128 | 1024 x 1024 | 2048 x 2048 |
|---|---|---|---|---|---|
| **IJK** | **Normal** | 2.15E-06 | 3.255E-06 | 3.13323E-06 | 2.09383E-05 |
| | **Transposed** | 2.109E-06 | 3.063E-06 | 1.16803E-06 | 6.54647E-07 |
| **IKJ** | **Normal** | 2.097E-06 | 2.518E-06 | 5.96591E-07 | 8.61085E-07 |
| **JKI** | **Normal** | 2.353E-06 | 4.138E-06 | 3.38108E-06 | 9.9375E-06 |
| | **Transposed** | 2.108E-06 | 3.597E-06 | 1.36665E-06 | 1.48296E-06 |

To calculate the traffic in RAM, we simply multiply the **l3 cache misses** (RAM Accesses) by 64, which is the number of bytes that each access moves to the cache.

RAM Data Transfer

| RAM DATA TRANS | | 32 x 32 (KiB) | 128 x 128 (KiB) | 1024 x 1024 (MiB) | 2048 x 2048 (MiB) |
|---|---|---|---|---|---|
| **IJK** | **Normal** | 32.5 | 72.3125 | 11.6135625 | 618.627875 |
| | **Transposed** | 31.875 | 68.125 | 4.3313125 | 19.346 |
| **IKJ** | **Normal** | 31.6875 | 55.9375 | 2.2113125 | 25.441 |
| **JKI** | **Normal** | 35.5625 | 91.9375 | 12.53225 | 293.6061875 |
| | **Transposed** | 31.875 | 80.1875 | 5.0723125 | 43.843625 |

### B. Floating Point Performance

We can obtain floating point operations for matrix-dot product using the formula:

FP Operations = 2 * matrix-size

We plotted then the achieved result on the Roofline model.

TABLE I: FP Operations per Size

| Matrix Size | FP Operation |
|---|---|
| **32x32** | 65536 |
| **128x128** | 4194304 |
| **1024x1024** | 2147483648 |
| **2048x2048** | 17179869184 |

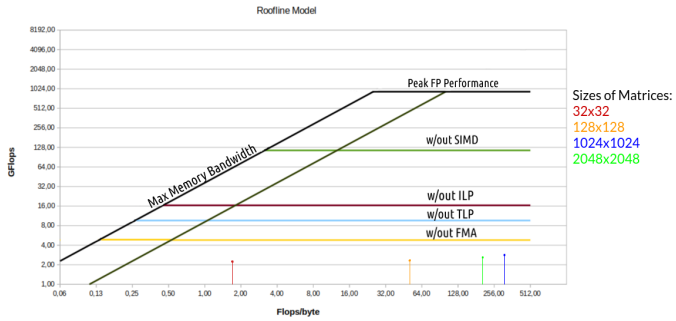


Fig. 7: Node 662 Roofline

### C. Cache Behavior

We make use of miss rate to analyse cache behavior. Miss rates for each cache level were calculated following the formulas below:

**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

| CACHE MISS RATE | | | 32x32 | 128x128 | 1024x1024 | 2048x2048 |
|---|---|---|---|---|---|---|
| ijk | normal | l1 | 0,0016 | 0,0839657413 | 5,451802573822 | 6,0059030669 |
| | | l2 | 71,664168 | 1,5670013705 | 5,072564416601 | 14,648691403 |
| | | l3 | 0,4627129 | 1,0037681136 | 0,36139912714 | 0,1123831691 |
| | transposed | l1 | 0,001574 | 0,0809932069 | 0,263596392743 | 0,2644844167 |
| | | l2 | 74,416018 | 1,7642810717 | 7,203439712625 | 7,7451787631 |
| | | l3 | 0,4499196 | 0,8456015083 | 1,379943422377 | 1,3040680743 |
| ikj | normal | l1 | 0,0016957 | 0,0805186965 | 0,261645918196 | 0,2631141226 |
| | | l2 | 68,172589 | 1,4803590162 | 7,362505107581 | 6,7578548621 |
| | | l3 | 0,4350938 | 0,8607217561 | 0,921520699884 | 0,8174596958 |
| jki | normal | l1 | 0,0016718 | 0,0827029203 | 10,41898297229 | 11,281699937 |
| | | l2 | 78,899083 | 1,1895862915 | 8,064599883637 | 73,447828952 |
| | | l3 | 0,4992412 | 1,1925216341 | 0,132747775275 | 0,0315454896 |
| | transposed | l1 | 0,0016357 | 0,1721880446 | 0,267842703907 | 0,2677675258 |
| | | l2 | 72,149745 | 1,7156793038 | 7,177656701904 | 7,7413660474 |
| | | l3 | 0,455899 | 1,0215475257 | 1,648939944143 | 2,3698893137 |

Fig. 8: Miss Rates

## VII. Optimizations

After the implementation of a first solution, and subsequent analysis of it, we concluded that we could optimize it in order to make the best use of resources. So in this section, we will talk about the optimized solutions designed, such as Blocking, Vectorization, among others.

### A. Blocking

One optimization that can be performed for a better cache use, is *Blocking*. In this technique, the matrices are divided into blocks of size sixteen and calculate the matrix multiplication algorithm for each block independently.

This will generate more iterations due to the same data for matrices **A** and **B** will be needed by several blocks. But allows to keep all those values in cache, reuse them, and improve execution time by reducing memory accesses.

| Time Measures | 32 x 32 (milliseconds) | 128 x 128 (milliseconds) | 1024 x 1024 (seconds) | 2048 x 2048 (seconds) |
|---|---|---|---|---|
| **Blocking** | 0.173641 | 11.009317 | 4.514164769 | 35.353734873 |

## VIII. Conclusions

The matrix multiplication its an algorithm its a highly prominent for tunning so that the overall performance can be improved. Although the many tests and various optimizations used, we conclude that it's very hard to predict the overall systems performance.

## References


[1] http://search6.di.uminho.pt/wordpress/?page_id=55

[2] https://software.intel.com/sites/default/files/article/393195/intel-xeon-phi-core-micro-architecture.pdf

[3] https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html

[4] https://www.cs.virginia.edu/stream/

[5] Williams, S., Waterman, A. and Patterson, D. (2008). Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures.

[6] Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., You, H. and Zhou, M. (n.d.). Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters.