Performance Measuring and Code Profiling of Matrix Multiplication

Jose Silva
University of Minho
Braga, Portugal
Email: a74576@alunos.uminho.pt

Bruno Ferreira
University of Minho
Braga, Portugal
Email: a74155@alunos.uminho.pt

***Abstract*—The aim of this project is to use matrix multiplication as an example algorithm to show how several changes can be made to improve the performance on the studied platform, the University of Minho cluster SeARCH, specifically node 662. For such changes to be made (i) a fully characterization of the hardware available and it's limitations is needed, (ii) a study on the metrics used to measure performance followed by (iii) the code profiling and performance analyses on the different versions with discussion of the results obtained ending up with (iv) a brief conclusion on the work done. We concluded that it is of extreme importance to study the work environment in order to obtain the most out of our code by the use of all cores and vector instructions, always having in mind to keep a good memory usage due to its latency compared to the processor, being essential to block the matrix in order to get a better reuse of the data close to the processor, improving performance.**

I. Introduction

Researchers try to model and simulate real-life problems computationally, and in most cases such problems require a high amount of resources. A performance engineer must be able to study a given problem and identify the changes he can perform in order to obtain a better use of resources on the work platform at his disposal improving the overall performance. This paper will focus on matrix multiplication, a central operation in many numerical algorithms and a potentially time consuming operation. First of all it is necessary to start by identifying and characterize the hardware used, in order to reveal potential bottlenecks. To do so we will use a widely accepted model, the roofline model. Afterwards we will study some implementations of the dot product matrix multiplication in squared matrices with changes in the index order, also taking into account changes in matrices size, which is an important factor. The results obtained will be presented and justified taking into account the architecture used to run our code, with the help of the low level hardware performance counters PAPI, which provides a more specific view of the algorithm behavior helping us find the major points to improve. This work will end with a many-core co-processor (Intel Knights Corner) version of the algorithm and a conclusion on the work done.

II. Hardware Specification

Before the analysis of the algorithm, it is necessary to obtain the maximum information about the hardware we are going to use so we can identify the potential performance bottlenecks on the computing platforms at our disposal. The available hardware used for this work is the team main laptop, the quad-core Asus K450JN-WX008H that comes with an Intel 4700HQ and a SeARCH 662 node equipped with a dual dodeca-core Intel Xeon E5-2695v2. The information for our personal computer was obtained through Intel website and a Intel Haswell Architecture Review website, while the specifications of the SeARCH node came from the Intel Website and a PU info website[6].

TABLE I. Team laptop specifications

Processor	
Manufacture	Intel Corporation
Model	Intel Core I7 4700HQ
Code Name	Haswell
# Cores	4
#Threads	8
Base Frequency	2.4 GHz
Turbo Frequency	3.4 GHz
Peak FP performance	153,6 GFlops/s
Memory	
Cache L1	32 KB Intructions cache 32 KB Data cache
Cache L2	256 KB
Cache L3	6 MB
Memory Bandwidth	25.6 GB/s
Memory Latency	msec
Main Memory	8GB (2 x 4GB) DDR3L 1600 MHz SDRAM
Memory Channels	2

III.Roofline Model

The Roofline Model offers insights into the factors affecting the performance of computer systems, by quantifying the influence of the system bottleneck. By putting together in a 2-dimensional the graph peak floating point performance, the operational intensity and the memory performance we obtain a relation between the processor and traffic between main memory and the processor. This relation is made by combining peak floating point performance (Y-axis) and operational intensity (X-axis), on the same graph. Peak floating point performance, and the attainable GFlops/sec can be obtained calculating the following formulas:

Peak FP $=$ Number of Processors $\times$ Number of Cores $\times$ Clock Frequency $\times$ SIMD width $\times$ FMA $\times$ CPI

Attainable GFlops/sec $=$ Min(Peak FP, Peak Memory Bandwidth $\times$ Operational Intensity)

For this formula on both systems we assumed an optimal CPI of 1 with the number of lanes on the functional units being equal to the average cycles needed to execute one instruction. The peak memory bandwidth was obtained using **stream benchmarking**[2].

On our computer, featuring *FMA3* and *AVX2.0* with SIMD instructions of 256 bits **(8 floats)**, the Peak FP performance is given by the following calculation: $153,6$ GFLOPS $= 1 \times 4 \times 2,4 \times 8 \times 2 \times 1$. The other ceilings were calculated removing features on the formula above and consequently the value.

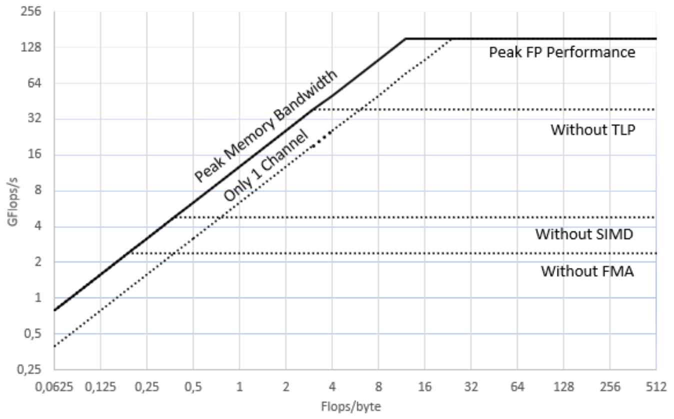

Fig. 1. Team Laptop Roofline Model

For the 662 node on SeARCH, who features *AVX* implementing SIMD instructions for 256 bits registers (8 floats), but without FMA, the Peak FP performance was $460,8$ GFLOPS $= 2 \times 12 \times 2,4 \times 8 \times 1$.

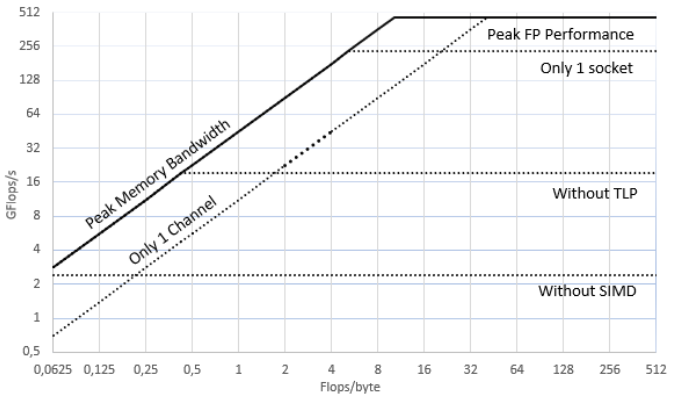

Fig. 2. 662 Node Roofline Model

For last we compared the roofs from both roofline models and as expected the cluster node obtains a much bigger GFLOPS performance, due to the high number of cores in each processing unit.

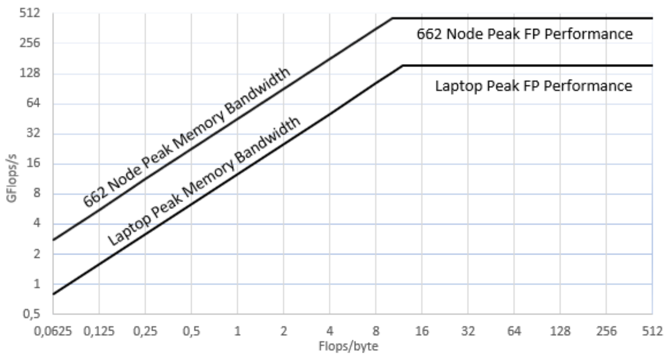

Fig. 3. Laptop and 662 Node Comparison Roofline Model

IV. PAPI PERFORMANCE COUNTERS

To obtain more data about the behavior of our algorithm, the PAPI platform (version 5.5.0) was used. This application allows us to monitor data from a big quantity of hardware counters from the processor. We needed to apply some filtering on those counters available on 662 node so we could obtain only the relevant information to our case study. From all counters listed with *papi_avail* the following table presents the ones used:

TABLE II. PAPI COUNTERS SELECTED

Counter	**Description**
PAPI_LD_INS	Load instructions
PAPI_TOT_INS	Instructions completed
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

With these native counters we can induce 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

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

A. Basic Implementations

After the initial study of the system, the work was started with the implementation of three basic versions of the dot-product function without optimization's, only with loop re-ordering, the default **IJK** iteration, **IKJ** and **JKI**. Each of these versions reveals changes in the access pattern of the matrices.

As we can see in Figure 1, in the IJK version a value of the matrix **C** is computed with a line of the matrix **A** and a column of the matrix **B**. The matrix **B** column accesses reveal a loss in performance so we created a version where this matrix is transposed to correct this access bottleneck.

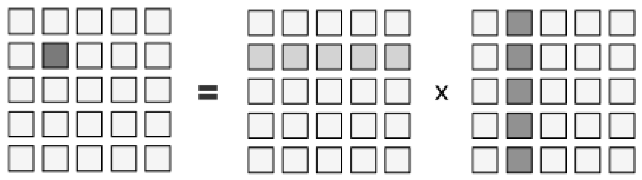

Fig. 4. IJK iteration

For the IKJ version, the result matrix **C** will store the results iterating the line, using an element of the matrix **A** and a line from matrix **B**. As all accesses are row-wise we did not transpose any of the 3 matrices.

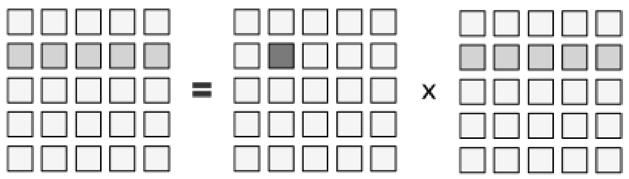

Fig. 5. IKJ iteration

As for the JKI version, the result matrix **C** will store the results iterating the column, using a column from the matrix **A** and an element from matrix **B**. As all matrices are iterated column-wise, we created a version with matrix **A** and **B** being transposed before the multiplication parte of the algorithm, with **C** being transposed in the final.

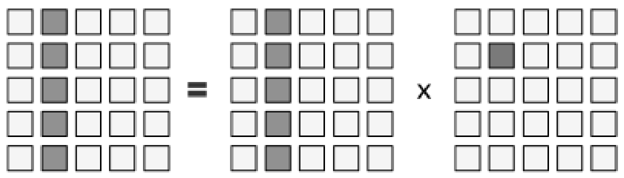

Fig. 6. JKI iteration

B. Experimental Setup

The tunings performed in this paper are targeted for the Intel Xeon E5-2695v2 dodeca-core processor. All binaries are compiled with *GCC* (version 4.9.0). Between measures all caches were clean to avoid data already in cache influencing some results, so all values are calculated from a cold cache.

The measuring technique used to get the execution timings 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, choose 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

An important factor to take into account after getting the memory system specifications is the size of the data sets to be used in the tests to perform. The sizes were chosen in order to fit the 3 matrices of the algorithm on a given cache level, using the following formula:

$$size^2 \times Float\ size \times 3\ Matrices \leq Cache\ level\ size$$

from which we obtained the following results:

TABLE III. MATRIX SIZES

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

Only smaller matrix sizes which are multiples of 16 were considered, factor based on alignment purpose with the 64 bytes cache line size, corresponding to 16 float elements.

D. Execution time measurements

For the implementations explained before several execution timings were calculated combining with the various matrix sizes.

As we can see in table IV, despite the implementation the size **32x32** does not allow a big difference between the values due to its reduced size. Even so we can see that **JKI** implementation obtains the worst result for this size.

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 **5211.14** milliseconds, very different from it's transpose implementation (changes the accesses to row-wise) who takes **750.28** milliseconds to execute.

For last, the **2048x2048** data-set size, we still got **JKI** as the worst implementation, about **6** times slower than it's transposed implementation.

TABLE IV. TIME MEASUREMENTS FOR PRIMARY IMPLEMENTATIONS (MSEC)

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	0.029	2.121	1462.07	18815.36
IKJ	0.025	1.507	726.026	5785.90
JKI	0.036	2.246	5211.14	35815.04
IJK Transposed	0.026	1.995	1074.59	8622.05
JKI Transposed	0.025	1.547	750.28	5905.12

VI. ALGORITHM BEHAVIOR ANALYSIS

A. Main Memory Behavior

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 the access pattern of one matrix is row-wise we considered that we are only accessing RAM every 16 iterations to get more values because the cache line can hold 16 float elements. When the access pattern of one matrix is column-wise, we assumed a worst case scenario where every element accessed will be on RAM, which might not happen, has for some data-sets cache has the capacity to hold the 3 matrices. For the transposing part of the algorithm the RAM accesses were taken into account, where each element accessed represents one RAM access.

Taking these factors into account we considered the following formulas for calculating the main memory accesses of one matrix depending on its access pattern:

- **Row-Wise matrix** $= size \times \frac{size}{cache\ line\ width}$
- **Column-Wise matrix** $= size \times size$
- **Transposing matrix** $= size^2$ iterations $\times$ 2 values loaded per iteration

This values were then calculated with PAPI, using the counters **PAPI_L3_TCM** and **PAPI_TOT_INS**, obtaining the following results:

TABLE V. RAM ACCESSES / INSTRUCTION

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	0.001009	0.000193	0.000023	0.000012
IKJ	0.001284	0.000213	0.000025	0.000013
JKI	0.000823	0.000156	0.000018	0.000009
IJK Transposed	0.001241	0.000215	0.000026	0.000013
JKI Transposed	0.001007	0.000209	0.000025	0.000018

To calculate the RAM traffic we simply 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 presented bellow in **KBytes**.

TABLE VI. DATA TRANSFERRED FROM/TO RAM (KBYTES)

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	16	203	12392	49518
IKJ	18	196	11541	48431
JKI	17	205	12392	47419
IJK Transpose	18	199	12210	48861
JKI Transpose	15	195	11797	68827

B. Floating Point Performance

For the floating point performance we know 662 node does not feature FMA, which means we will perform two floating point operations on each iteration (one multiplication and one accumulation). This way we can obtain floating point operations for matrix-dot product common to all implementations using the formula:

FP Operations $= 2 * size^3$

We tested this for the several data-set sizes and ended up with the results on the table below.

TABLE VII. FLOATING POINT OPERATIONS

	FP Operations
32 x 32	65536
128 x 128	4194304
1024 x 1024	2147483648
2048 x 2048	17179869180

We plotted the achieved performance on the Roofline Model ordering by size starting with 32x32 until 2048x2048 as we will show on the following graph.

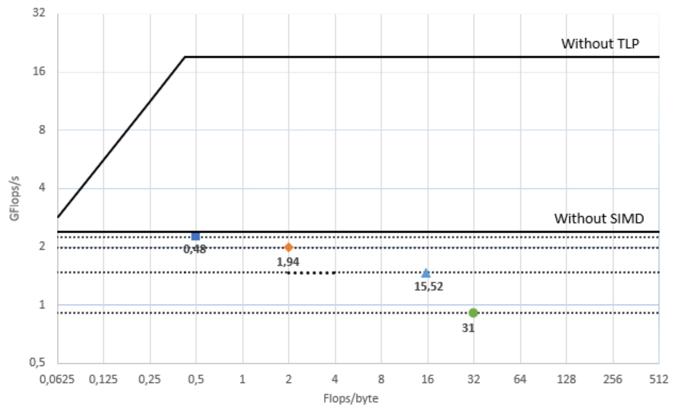

Fig. 7. Plotted achieved performance on roofline model for IJK iteration

C. Cache Behavior

To get information on the cache behavior we calculated the miss rate percentage on memory reads for each cache level with PAPI counters. To calculate the miss rate we used the 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 VIII. MISS-RATES TABLE FROM **IJK** IMPLEMENTATION

		32 x 32	128 x 128	1024 x 1024	2048 x 2048
	MR L1	0.241	2.192	44.310	48.092
	MR L2	39.484	0.973	5.161	27.669
Normal	**MR L3**	89.320	74.133	0.566	4.957
	MR L1	2.332	3.416	3.174	3.123
	MR L2	5.607	0.665	4.632	7.134
Transposed	**MR L3**	82.417	69.798	3.488	29.150

As we can see for the **IJK** implementation on the smallest data-set (32x32), L2 and L3 miss rates are high because data fit completely on L1 cache. The second data-set (128x128) as expected has a high L3 miss rate, but a really small L2 miss rate since data fit in L2 cache completely. 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.

VII. OPTIMIZATIONS

After the analysis of the basic implementations, we decided to obtain even more performance by taking advantage of the full power of our resources.

In this section several different implementations will be presented such as blocking the matrix for a better data locality, vectorization to take advantage of SIMD instructions, a multi-core implementation with openMP and for last a version for the Intel many-core Coprocessor.

When we change the executions to run in more than one core, we need to adapt the data-sets so that the comparisons between implementations are fair.

A. Blocking

One optimization we can perform on our code is to apply blocking in order to get a better cache use. In this technique we divide the matrices in squared blocks and calculate the matrix multiplication algorithm to each of this blocks independently.

This will generate a bigger data replication as the same data for matrices **A** and **B** will be needed by several blocks but allows to keep all those values in cache, reusing them, improving execution time by reducing main memory accesses which have a high latency.

B. Multi-core and Vetorization

Till this moment we just ran our code on a single core, not taking advantage of the full power of the machine. To change this we used OpenMP to split the matrix blocks by the **24** cores present in SeARCH 662 node. The use of vector instructions allows to apply one instruction over multiple elements stored in vector registers. The 662 SeARCH node features AVX, having capacity to run 256 bits instructions (8 elements), reducing substantially the number of instructions, improving the performance of the execution.

C. Intel Coprocessor

For this task we pretend to take the best performance of the Intel many-core co-processor Knights Corner present in 662 node. This co-processor supports until 4 threads per core, with a total of 61 cores. As this coprocessor **divides each clock cycle between two threads**, it is essential to make use of hyper threading of at least 2 threads per core in order to obtain the maximum performance of the machine, so **120 threads** were used in the execution of the tests, with 1 physical core being used as manager.

D. Results and Analysis

As we can see in the results time table the vectorization allowed to improve performance because it provides one instruction working on various elements simultaneously. Blocking made the execution times worst for the smaller datasets because they already presented a good data locality, but has the size increases so the performance with the use of this technique. The use of multiple cores improved the performance as we can see by the execution time, cause even having a bigger data replication we can count on multiple cores, each with part of the matrix calculating without dependencies part of the result matrix.

TABLE IX. BLOCKING IMPLEMENTATIONS EXECUTION TIME (MSEC)

	32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
Blocking	0.049	2.848	1519.313	11661,415
Block + Vectorisation	0.023	0.833	-	-
Block + OMP	-	-	-	943.948

VIII. CONCLUSIONS

The matrix multiplication its an algorithm where multiple tunings can be performed so that the overall performance improves. Some of those tunings look good on paper but when tested, the results are slightly different from what was expected.

REFERENCES

[1] Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

[2] McCalpin, John D., 1995: ”Memory Bandwidth and Machine Balance in Current High Performance Computers”, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.