Performance Measuring and Code Profiling of Matrix Multiplication

Frederico Pinto
University of Minho
Braga, Portugal
Email: a73639@alunos.uminho.pt

Sara Pereira
University of Minho
Braga, Portugal
Email: a73700@alunos.uminho.pt

***Abstract*—The main goal of this project is to reveal the importance of knowing how to take advantage of the hardware ate our disposal in order to increase the performance of our programs. The algorithm that will be used is dot-product multiplication on University of Minho cluster SeARCH, specifically node 662. Later, we will make a full characterization of the hardware available and it's limitations. To complement, it will be made a study of the metrics used to make all the measurements and code profiling and its performance analysis on that platform.**

I. Introduction

This paper will focus on matrix multiplication, specifically the dot-product multiplication which can be a very time consuming operation if one does not have the knowledge to benefit from the hardware at his disposal or even take advantage of all the optimizations that can be made to increase performance. However, first it is necessary to start by characterize the available hardware and study it's potential bottlenecks with the help of a widely accepted model, the roofline model. At the same time, we have to consider factors as the type of accesses made to matrices as well as the matrices size. The results will be exposed and justified with the help of the low level hardware performance counters PAPI, these counters are useful to help us find the points which need improvement. To finish, the work will end by making the analysis of the results obtained with our code running on Nvidia Kepler, the micro architecture GPU and on Knights Landing Xeon Phi co-processor.

II. Hardware Specification

On this section we are going to identify certain performance bottlenecks on the hardware at our disposal. So, our team laptop is a dual core Macbook Pro Retina 13 with a Intel Core i5 (I5-4278U) processor. At the same time, we have available a SeARCH 662 node equipped with a dual dodeca-core Intel Xeon E5-2695v2. The informations about our team's computer were obtained through the Intel website. The SeARCH specifications also came from the Intel website.

TABLE I. Team Computer Specifications

Processor	
Manufacture	Intel Corporation
Model	Intel Core i5 (I5-4278U)
Code Name	Haswell
# Cores	2
#Threads	4
Base Frequency	2.6 GHz
Turbo Frequency	3.1
Peak FP performance	83,2 GFlops/s
Memory	
Cache L1	32KB Instruction cache 32KB Data cache
Cache L2	256KB per core
Cache L3	3MB
Memory Bandwidth	25.6GB
Memory Latency	msec
Main Memory	8GB DDR3L SDRAM 1600 MHz
Memory Channels	2

III. Roofline Model

The Roofline model presents us with a quantification of the influence of the system's bottlenecks and suggest which optimizations to perform. The proposed model ties together floating-point performance, operational intensity, and memory performance together in a two-dimensional graph. The Y-axis is attainable floating-point performance and X-axis is operational intensity. We use operational intensity because we want to measure traffic between caches and RAM. 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)

We assumed the number of lanes on the functional units are equal to the average cycles needed to execute one instruction, which gives us a optimal CPU of 1. By using **stream benchmarking**[1] we were able to obtain the peak performance.

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:

Macbook Pro :: FMA3(2) and AVX2.0 SIMD(8 Floats) 256bits

Macbook Pro :: $1 \times 2 \times 2.6 \times 8 \times 2 \times 1$

The other ceilings were calculated removing features on the formula above and consequently the value.

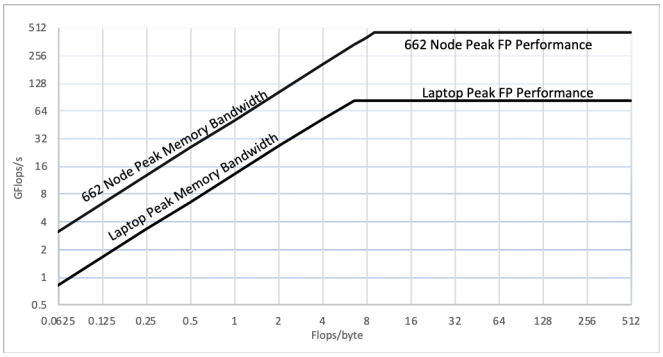


Fig. 3. Comparison Roofline Model between our team's laptop and 662 node

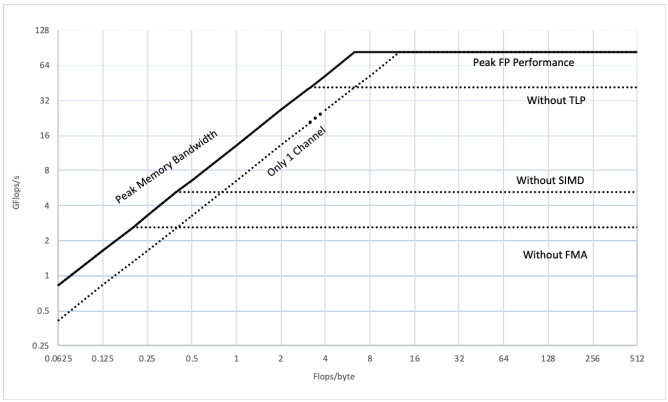


Fig. 1. Team Computer Roofline Model

Secondly, 662 node on SeARCH features *AVX* implementing SIMD instructions for 256 bits registers, but **without FMA**, the Peak FP performance was:

Peak FP performance $= 2 \times 12 \times 2,4 \times 8 \times 1 = 460,8$ GFLOPS

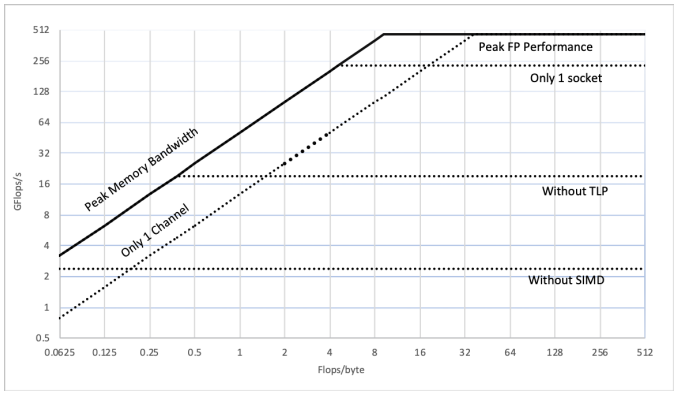


Fig. 2. 662 node Roofline Model

Lastly, we thought it would male sense to compare both rooflines between our team's laptop and the SeARCH node. As we expected, the 662 node presents a superior GFLOPS performance due to the larger quantity of cores in each processor unit.

IV. PAPI PERFORMANCE COUNTERS

The platform used to learn more about matrix multiplication algorithm was PAPI (version 5.5.0). This application allow us to take advantage of some of the processor's counters so we can monitor our programs. By using the **papi_avail** command we can filter just the counters available on 662 node and select all the essential information to our case study. The following table presents the counters picked:

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

Through the use of these native counters we are capable to calculate some values like the number of hits and misses on a certain cache level, RAM accesses, number of reads and 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

To begin the study, after all the analysis done to gain knowledge about the rooflines of 662 node and our team's laptop, we started by implementing three different approaches in what concerns matrix multiplication. Those were basic versions of the dot-product multiplication without any kind of optimizations, just with loop reordering. First, the default version **IJK**, then **IKJ** and, for last, **JKI**. As we can see in the figures below, the IJK multiplication uses, to obtain matrix C, a line of matrix A and a **column of matrix B**. However, accessing a matrix by column reveals to be very harmful to the performance. To correct that we transposed the matrix **B** so the accesses can be made row-wise.

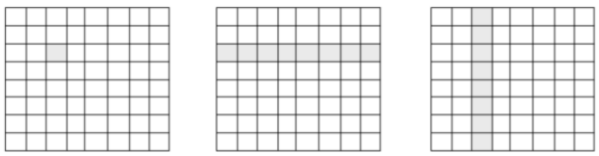


Fig. 4. Matrix **C** = A x B, **IJK multiplication**

Secondly, the IKJ multiplication uses one line of the matrix A and one element of matrix B, so there are no column-wise accesses and, consequently, no need to transpose any of the matrices.

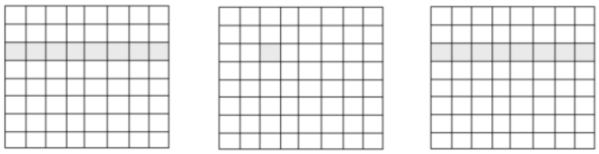


Fig. 5. Matrix **C** = A x B, **IKJ multiplication**

Lastly, the JKI multiplication, where a column of the matrix A is multiplied by one element of the matrix B resulting on filling the results on matrix C by column. As all the accesses are made column-wise we transposed A and B, and then, after the calculations, transposed the final matrix C.

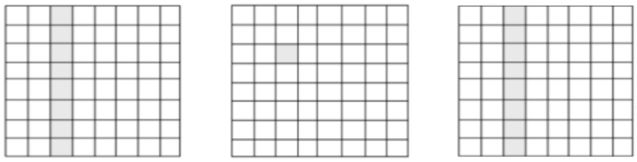


Fig. 6. A x B = **C**, **JKI multiplication**

B. Experimental Setup

The computer system used was the Intel Xeon E5-2695v2. All binaries are compiled with *GCC* (version 4.9.0). Between measures, all caches were clean so the data already cached does not influence the 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, 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

In order to calculate the sizes of the matrices to fit in each cache level, the following formula was used:

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

The sizes chosen take into account the cache line size which is 64 bytes or 16 float elements, therefore only sizes multiple of 16 were used considered the cache alignment. The following results were obtained:

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

D. Execution time measurements

In this phase of the paper, we have calculated several execution times for the different matrix sizes.

TABLE IV. TIME MEASUREMENTS FOR PRIMARY IMPLEMENTATIONS (MICROSECONDS)

	32x32	128x128	1024x1024	2048x2048
IJK	51	2808	7565020	14723532
IKJ	22	1100	413869	3323369
JKI	44	2558	17082664	30395535
IJK Transposed	45	2821	4672333	14704946
JKI Transposed	42	2572	15544507	30411022

As we can see on the table, sometimes due to the small volume of data, transposing a matrix does not improve the execution time, which happens on **128x128** dataset. At the same time, we noticed that for small datasets **JKI** implementation has better performance than **IJK**. Theoretically this would not take place because that execution has two matrices, which accesses are column-wise instead of only one like **IJK**. However, because of the cache line size and the small matrix size, this theoretical part has a minor effect on executions.

As expected, for bigger datasets the best performance between the two algorithms that need matrix transposition belongs to **IJK**. In general, we calculate better execution times for the all matrices accesses column-wise algorithm, **IKJ**.

VI. ALGORITHM BEHAVIOR ANALYSIS

A. Main Memory Behavior

After analyzing execution times for dot-product matrix multiplication, we decide to estimate RAM accesses per instruction in order to examine the RAM behavior. At the same time, we also estimated the number of bytes transferred to/from RAM. To perform this calculations we based on the fact that the algorithms we operate with contain 3 nested cycles and each one iterates **size** times. All these repetitions result in $size^3$ iterations where one element of the 3 matrices is accessed.

When the matrices are accessed row-wise we assume that to every 16 elements there is one RAM access because of the size of the cache line that can hold 16 float elements. However, when the matrices access is column-wise we assume that we are towards the worst case scenario where every element accessed will be on RAM, which might not be true.

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

	32x32	128x128	1024x1024	2048x2048
IJK	0.000508	0.000105	0.000023	0.000064
IKJ	0.000803	0.000144	0.000013	0.0000171
JKI	0.000516	0.000102	0.000021	0.000097
IJK Transposed	0.000490	0.000071	0.000020	0.000079
JKI Transposed	0.000479	0.000006	0.000015	0.000102

To calculate the RAM traffic we need to multiply the **level 3 cache misses**, which is equivalent to the RAM accesses, by 64, the number of bytes that each access moves to cache. The results are presented bellow.

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

	32x32	**128x128**	**1024x1024**	**2048x2048**
IJK	8.56	111	12108,45	34132,37
IKJ	8.19	76,25	3237,86	43303,75
JKI	9,75	120,75	12430,02	58850,10
IJK Transposed	8.38	75,38	10598,33	42481,11
JKI Transposed	9,5	72,5	9300,31	61400,83

B. Floating Point Performance

Since 662 node does not feature FMA, we know that for the dot-product matrix multiplication 2 operations will be executed every iteration, one multiplication and one accumulation. Thus, to calculate floating point operations common to all implementations we use the formula:

$$\text{FP Operations} = 2 \times size^3$$

At the same time, we calculate the number of floating point operations for the different data sizes used in our matrices and builted the following table:

TABLE VII. FLOATING POINT OPERATIONS

	Floating Point Operations
32x32	65536
128x128	4194304
1024x1024	2147483648
2048x2048	17179869184

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_L2_TCM / PAPI_L1_DCM + PAPI_L1_ICM

L3 miss rate = PAPI_L3_TCM / PAPI_L3_TCA

And create a table with calculations for the different cache levels and different datasets:

TABLE VIII. MISS RATE (%) TABLE FOR **IJK** IMPLEMENTATION

		32x32	**128x128**	**1024x1024**	**2048x2048**
IJK	**MR L1**	0,25	2,48	52,32	64,72
	MR L2	41,40	1,93	4,42	37,99
	MR L3	94,78	34,99	0,24	0,07
IJK Transposed	**MR L1**	0,19	2,48	52,33	58,91
	MR L2	57,88	1,50	4,42	25,50
	MR l3	95,04	26,77	0,21	0,08

As we can prove based on the results of table VIII, small datasets, for the algorithm without matrix transposition, fit entirely on L1 cache so the miss rates for this sizes are very low. For **128x128** matrices, we expect the miss rate to be smaller for L1 and L2 cache and bigger for L3. For the two bigger datasets, we noticed opposite rates, this is, the miss rates are larger for L1 and L2 cache and very close to 0 for L3 cache. This succeed because of the huge dimensions, which makes the data to be mostly stored on L3 resulting on hits almost every time.

For transposed matrices algorithm, we observe that occasionally the cost of transposing increases miss rate, but as it provides row-wise accesses to data will improve the reuse of the same data, improving locality and consequently decreasing the same rate.

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.

VII. OPTIMIZATIONS

Ending the analysis for the sequential implementation of different versions of the dot-product multiplication we decided to perform some optimizations in order to take full advantage of our resources.

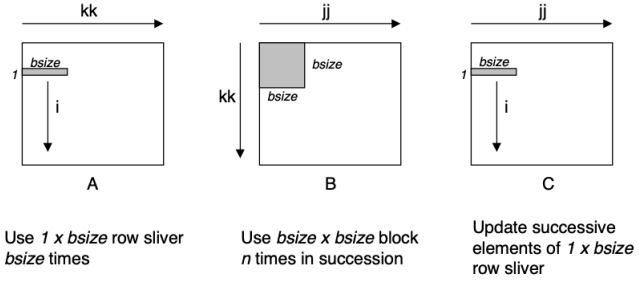
We will begin by blocking the matrix in order to make progress with data locality, take advantage with the usage of SIMD instructions by vectorize the algorithm and even build a multicore implementation by using OpenMP. With this last optimization we need to adapt our datasets so we can compare the distinct implementations fairly.

A. Blocking

With larger datasets, the temporal locality of data decreases and the cache experiences superior miss rates. So we use the **blocking** technique to get better performance of our algorithm.

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 it and loads in the next chunk, and so on. With this, we reuse some data preventing extra memory accesses by replicate data of matrices **A** and **B** and keeping those values on cache for longer and discarding the latency of transfering data to/from memory.

Fig. 7. Example of graphic interpretation of blocked matrix multiplication



Where on the innermost loop of the algorithm a **bsize** sliver of A is multiplied by a **bsize**×**bsize** block of B and accumulates into a **bsize** sliver of C, where **bsize** is an amount that fits evenly into blocks.

1) Block Optimization and Vectorization

After analising this technique, we expect that it improves our algorithm's performance, so to prove it we used the same smaller matrices as before, **32x32** and **128x128** and calculated their execution time measurements with blocks which size was 16:

TABLE IX. BLOCING IMPLEMENTATIONS EXECUTION TIMES (MICROSECONDS)

	32x32	128x128	1024x1024	2048x2048
IJK	51	2808	7565020	14723532
IJK Transposed	45	2821	4672333	14704946
IJK Blocking	46	3058	-	10481941
IJK Blocking + Vetorization	23	1242	-	-
IJK Blocking + OpenMP	-	-	122621	746883

Initially, our code was not being vectorized so we have to modify it to force the compiler to do such task, by unrolling the elements on the for loop and splitting iterations we were able to obliterate the data dependencies between loops. In fact, after analyzing the results we observed better outcomes in the execution times.

B. Multi-core

Till this moment we were running 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. As 662 node features AVX, having capacity to run 256 bits instructions, we can take advantage of it and reduce the number of instructions performed.

As we can see on table IX, for bigger datasets we are in front of significant improves in execution time.

C. Nvidia Kepler and Intel Knights Landing

For this tasks, we pretend to take maximum advantage of the performance of Intel's processor Knights Landing and of Nvidia Kepler. The first one supports until 4 threads per core with a total of 72 available cores. The second one has 2880 cuda cores, ideal for executing massive parallel operations. The expectation is that the addition of hyper-threading, with a minimum of **two threads per core** improves performance significantly, once we have a huge amount of cores to distribute work, which is useful once our algorithm is **cpu-bound**.

VIII. CONCLUSIONS

We started this paper by analyzing all the resources we had at our disposal in order to know it's limitations and to learn how far to go in what concerns taking maximum advantage of its capabilities. Secondly, we were given an algorithm to analyze meticulously and make conclusions about the optimizations to be made according to the hardware available.

The first optimization performed was accessing the matrix by blocks, which brought better results for larger datasets once the smaller ones already present good data locality. After removing all the dependencies, the compiler was able to vectorize the code and also obtained better execution times for the same algorithm. With OpenMP we shared the work between the many cores available, which was possible beacause of data replication, and obtained the fastest version for dot-product mattrix multhttps://www.overleaf.com/project/5c09a0bdee9323182cec1c92iplicat

In conclusion, to the algorithm studied several changes and improvements can be made to increase the performance. As we analyzed before, theoretically some of them would get better results by making a optimization but that does not always succeed. Overall, the results are what we expect.

REFERENCES

[1]