Measuring performance and code profiling of matrix multiplication algorithms in an HPC environment

Afonso, João - *A71874,* Fernandes, Gabriel - *A71492*

***Abstract*—In non-ordinary computing tasks, performance is crucial. As computer systems and architectures evolved towards powerful multi-core machines, many engineers had simply overlooked it and did not adapt to this new paradigm. In this report, we will prove through several techniques how every slight change may improve our code performance using matrix multiplication as our testing algorithm. Instead of blindly speculating about how much performance gain can we achieve, first of all, we combine the limits and parallel paradigms in a intuitive visual model called roofline to guide us.**

To get a better apprehension of our algorithm, we use a framework to access hardware counters called PAPI. Through PAPI, we can get information about total floating point operations, cache accesses and clock cycles.

We will use SeARCH 652 node to perform our study. Using some linear algebra knowledge, we implemented several ways of multiplying matrices based on index order and transposing them when needed. The dimension of the matrices is also relevant to draw some interesting conclusions, so, we tested with multiple sizes. Then we compare their performance and justify our results based on factors like architecture specifications, since we tested our algorithms both on a single CPU-core and on a GPU device (Tesla K20).

I. Introduction

Within an scientific environment, in research, lots of extremely complex resource demanding algorithms are executed. Therefore, every inch of our available resources must be exploited in order to get the maximum out of it. It is our job, to know and apply the techniques of how to take advantage of our computational resources. The purpose of our study, is to dissect the matrix multiplication algorithm and apply the several techniques that could improve the algorithm performance and to identify potential bottlenecks and how to deal with them. First we need to get a full panorama of the main limitations of our current setup, and to do so we plotted a roofline model. This model, through the main specifications of the machine, give us the limits of the computation and memory bandwidth. The top limit is the *roof* and limits below are called *ceilings*, which provide us what path we should lead to change our code in order to get more performance. To understand the behaviour of our algorithm we use PAPI. It provide us information about lots of hardware counters. These counters give us a global view of how our program is using the hardware, thus we know what we should change in it to gain performance. Our study will only use squared matrices in the matrix multiplication. Finally, we will compare the CPU to GPU and draw some conclusions of the work done.

II. Hardware specifications of the platforms used

First of all, to start our study, we need to know well our machines' specifications and their limitations. To do so, we rely on a solid model that has been valid and accepted in the community for a long time, which is the roofline model and we will explain it later on. So, our machines are the ASUS ZenBook 3 [1], equipped with an Intel Core i7-7500U dual-core with 2.70 GHz (more info in Table I), and a SeARCH [5] 652-node equipped with a dual Intel Xeon Processor E5-2670v2 deca-core with 2.50 GHz.

To get our information about the specifications of our team laptop, we used a software called *Speccy* [6], the manufacturer website [1], Intel website [2].

In order to get the specifications of the cluster nodes we used, we used the SeARCH specifications page [4] and the Intel website [3]. We can see it in Appendix A.

TABLE I. Laptop specifications

Processor	
Manufacturer	Intel Corporation
Specification	Intel Core i7-7500U
Code name	Kaby Lake
Clock Frequency	2.70 GHz
Intel Turbo Boost	3.50 GHz
# Cores	2
# Threads	4
Peak FP Performance	86.4 GFLOPS/sec
Memory	
Cache L1	32 KB I + 32 KB D per core
Cache L2	256 KB per core
Cache L3	4 MB
Memory access bandwidth	25.28 GB/s
RAM Memory	16 GB LPDDR3 2133MHz SDRAM
RAM Latency	35 ns

III. Roofline Model

The roofline model, was used to get a full characterisation of our machine limits in terms of both memory and FLOPS performance. It consists in a two-dimensional graph where the X-axis measures the operational intensity (floating point operations (FLOPS) per byte) and Y-axis is the attainable

performance measured in FLOPS per second, also, we can know how much the memory bandwidth limits us.

In our study, we used single-precision floating point operations to calculate our peak performance. It can be measured by calculating:

Peak performance = # of cores per CPU $\times$ # of CPUs $\times$ CPU clock frequency $\times$ FMA $\times$ SIMD Width $\times$ CPI

All the other ceilings in the graphs below can be easily obtained by removing the components of the previous formula, one by one until only the clock frequency and CPI are left.

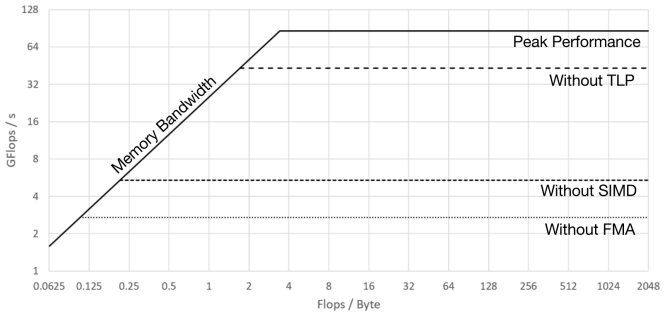

Fig. 1. Team laptop roofline model

As we can see in Table I, the laptop we chose is composed by a 2.7 GHz dual-core CPU, each one with a SIMD width of 256 bits (8 Floats) and FMA. For instance we consider the optimal CPI being one, as an approximation for the number of cycles per instruction needed to compute at maximum ILP usage. This way the laptop peak performance is about $2 \times 1 \times 2.7 \times 2 \times 8 \times 1 = 86.4$ GFLOPS. Peak performance (*roof*) and all the calculated ceilings are expressed in Figure 1.

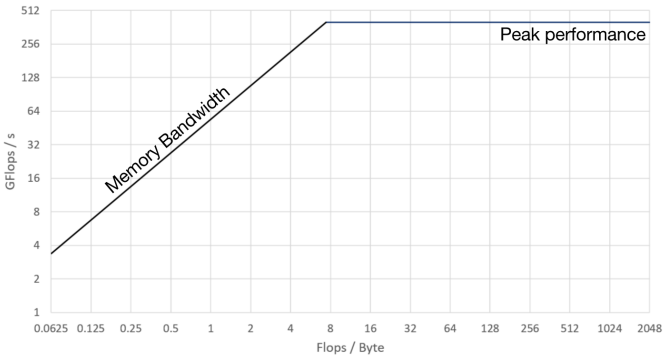

Fig. 2. Cluster 652 node roofline model

Both the cluster 652-nodes are composed with a dual-CPU deca-core with 2.5 GHz, SIMD 256 bits width, but without FMA. Using the previous formula we get our peak performance value: $10 \times 2 \times 2.5 \times 1 \times 8 \times 1 = 400$ GFLOPS, as seen in Figure 2.

As expected, the high number of CPU cores in the cluster Xeon processor largely overcome the laptop performance, although it has lower sequential performance, as seen in Figure 3.

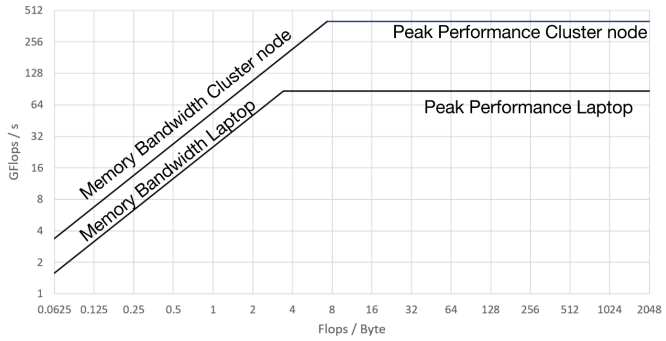

Fig. 3. Roofline model comparing CPU's from both platforms

IV. Metrics used using PAPI

First of all, to start our study which relies upon performance measuring through matrix multiplication, a tool was needed. We used PAPI (version 5.5.0), which is the swiss army knife of the performance measuring tools because it can provide us lots of relevant information. In order to simplify our study, we need to filter what are the most significant counters to our case study available in the 652 node. To see all counters available we executed the command: `papi_avail`. From that list, we used the following counters available:

TABLE II. Selected counters

Counter	Description
PAPI_LD_INS	Load instructions
PAPI_L2_DCR	Level 2 data cache reads
PAPI_L3_DCR	Level 3 data cache reads
PAPI_L3_TCM	Level 3 total cache misses
PAPI_L3_TCA	Level 3 total cache accesses
PAPI_FP_INS	Floating point instructions

To measure the elapsed time between the start and the end of our algorithms, we used the function *gettimeofday* instead of a PAPI provided counter, which returned the time in µseconds.

V. Matrix multiplication algorithm

This algorithm consists in the dot product of two squared matrices (**A** and **B**) with size **N** and storing the result in a new matrix **C**. The **A** matrix is filled with random generated numbers, and matrix **B** is just filled with ones. Let **I** identifies each row, **J** identifies each column and **K** identifies the current iteration, that is, the current element being processed.

A. Explaining data sets

After knowing the memory specifications of the machine used, we wanted to know how large the matrices should be to fit all three matrices (**A**, **B** and **C**) in every memory layer. To do so, first we calculate the height/width of each matrix:

$$N = \sqrt{\frac{\text{Cache level size (bytes)}}{(\#\text{ Matrices} \times \text{Float size (bytes)})}}$$

For each cache level, to fit entirely the 3 matrices, we get, at most, the following dimensions:

- L1 Cache: **52x52**
- L2 Cache: **147x147**
- L3 Cache: **1478x1478**

For alignment purposes the total matrix elements must be a multiple of 16 (16 floats have the size of 64 bytes which is the cache line size). Therefore we defined the ensuing matrix dimensions as we can see below:

- L1 Cache: **32x32**
- L2 Cache: **128x128**
- L3 Cache: **1024x1024**
- RAM: **2048x2048**

B. Basic implementations

Hereupon, at first, we built three different classical dot product implementations besides the classical one with the index order **IJK**, which are the **IKJ** and **JKI**.

For the **IJK**, we compute the multiplication between each line of matrix A by each column of matrix B and store the result in a element of the matrix C as we can see in Figure 4.

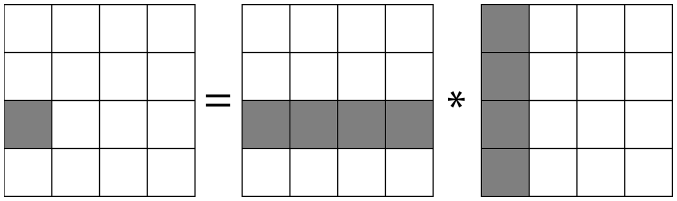

Fig. 4. I-J-K

For the **IKJ** matrix, we will compute partial sums in matrix C using an element from matrix A and one line from matrix B. All matrices are computed row-wise as we can see in Figure 5.

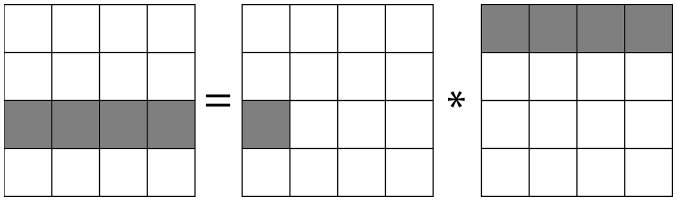

Fig. 5. I-K-J

For the **IKJ** matrix, we will compute partial sums in matrix C using an element from matrix A and one column from matrix B. All matrices are computed column-wise as we can see in Figure 6.

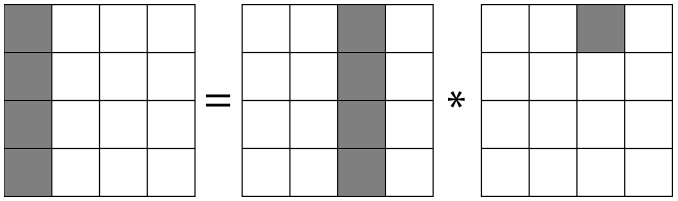

Fig. 6. J-K-I

C. Time measuring

In order to get our total execution times, we measured it 8 times, with 5% tolerance and we applied the 3-best algorithm.

As we can see in Table III for the smallest data set (32×32), we cannot see any substantial difference among them due to being too small, still we can say that the **IJK** implementation is the least suitable to the smallest data set and it takes 0.048 milliseconds to perform the matrix multiplication.

For the 128×128 data set the worst implementation is the **JKI**, the rest of it are very similar, and it takes approximately 6.7 milliseconds, the best ones take approximately 2.6 milliseconds. The reason why that implementation is the worst is because it is column-wise, so the next element is not the next element in memory, we are not exploiting spatial locality.

For the 1024×1024 data set the **JKI** implementation also is the worst for the same reason explained before, taking approximately 11 seconds to perform matrix multiplication. The **IJK-transpose** implementation, where we perform a transpose of matrix **B** before computing the matrix multiplication, is the best one taking approximately 1 second.

For the 2048×2048 data set, the **JKI** implementation is still the worst, taking 94 seconds to run. The best one is the **IJK-transpose** taking 8 seconds to run.

TABLE III. TIME MEASUREMENTS ON DIFFERENT DOT PRODUCT APPROACHES (MILLISECONDS)

	32x32	**128x128**	**1024x1024**	**2048x2048**
I-J-K	0.048	2.942	2761.80	21454.3
I-K-J	0.032	2.687	1300.82	10376.3
J-K-I	0.032	6.763	11406.1	93727.9
Transposed I-J-K	0.029	2.613	988.192	7918.43
Transposed I-K-J	-	-	-	-
Transposed J-K-I	0.033	2.623	1324.43	10809.1

The roofline graph presented in Figure 7 contains the ceilings for the cluster referred CPU, as well as the measured values for the I-J-K index order algorithm, with the initial transposition of matrix B. The plotted points appear in the graph, in the X-axis, following a crescent order from left to right according to the data set size from 32×32 to 2048×2048

We can conclude that even though we are transposing the matrix **B** before computing the dot product, thus turning it into a row-wise computation, it is still better than computing matrix **B** column-wise. This is the same reason that makes the **JKI** the worst implementation of them all.

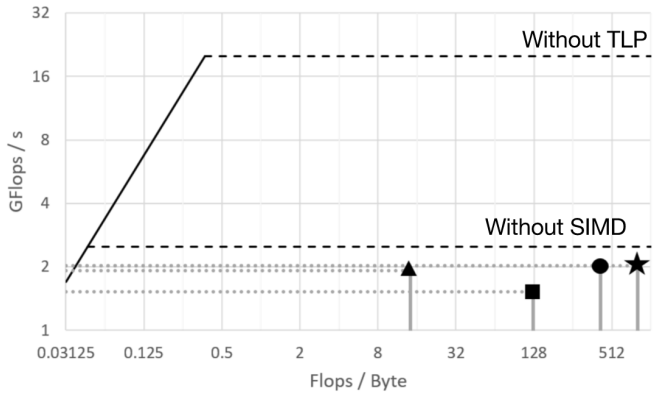

Fig. 7. Roofline model with measured performance

D. Cache behaviour analysis

To get our information about cache behaviour we did the following:

- L1 miss-rate = PAPI_L2_DCR ÷ PAPI_LD_INS
- L2 miss-rate = PAPI_L3_DCR ÷ PAPI_L2_DCR
- L2 miss-rate = PAPI_L3_TCM ÷ PAPI_L3_TCA

Despite the three generic implementations we also added some more. For instance, where we have computations that process a matrix row-wise and the another matrix is computed column-wise, we transpose the column-wise matrix in order to process both in row-wise order. When we transpose a matrix, miss-rate will be reduced since we will be crossing it row by row, but we get an overhead which corresponds to the computational cost of doing it.

TABLE IV. MISS RATE VALUES ON I-J-K WITH COLD CACHE (%)

		32x32	128x128	1024x1024	2048x2048
Standard	**mr L1**	0.759	7.601	8.234	8.259
	mr L2	38.781	0.985	2.478	2.499
	mr L3	58.420	52.064	2.785	1.931
Transposed	**mr L1**	0.404	3.242	3.156	3.142
	mr L2	31.533	0.754	3.353	2.825
	mr L3	56.456	47.812	3.459	2.193

As we can see in the Table IV, with a cold cache, we can see that for the 32×32 data set, the L2 and L3 cache misses are high because it fits completely in the L1 cache. The 128×128 data set has a high percentage of L3 cache misses because it fits completely in the L2 cache. In general, it is important to say that a transposed matrix reduces the miss rate, because the access to that matrix will be row-wise. Therefore, the next element to be computed will have a bigger chance of being in cache, since each matrix line is loaded in cache each time. If we were computing the matrix column-wise, the next element could be not in cache since it was a big leap in the matrix.

E. Blocking

Another modification to the preceding implementations, can be a matrix multiplication by blocks. This means that we will divide each matrix into significant chunks and apply the multiplication of that sub-matrices independently as shown in Figure 8. This will help us to keep these sub-matrices in cache, which will improve our computational execution times, since we can fit these sub-matrices completely in higher cache levels.

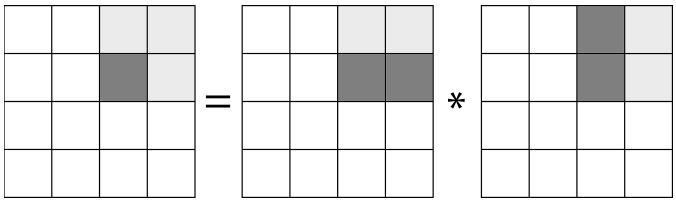

Fig. 8. I-J-K block

We can foresee that this will be able to be easily computed in parallel since, as referred before, those sub-matrices are independent from each other. Using **OMP**, we can divide the matrix by the total number of CPU-cores (we assigned one thread per CPU-core) and therefore assign to each CPU-core a sub-matrix chunk. This is a major improvement since each chunk will have at its disposal the cache of its CPU-core.

For the 32×32 data set, the **TLP** version takes longer to compute that the simple blocking version, that time is due to thread management and other **OMP** set-up times. For the remaining data sets, the **TLP** version is faster, because we are using all of the cores available to compute each chunk. For the 2048×2048 data set, the simple blocking version takes approximately 9 seconds to compute, and the **TLP** version takes approximately 2 seconds. The vectorisation in this case made little difference. We can see the computing times in Table V

TABLE V. BLOCKING (MILLISECONDS)

	32x32	128x128	1024x1024	2048x2048
Simple blocking	0.043	2.725	1155.48	9069.13
W/ Vectorisation	0.042	2.336	1431.15	9057.89
W/ TLP (20 cores)	0.624	2.202	334.290	2073.59

F. GPU

To exploit the high performance capabilities of the GPU Tesla K20, present in the 652-node, we adapted the previous implementations to run in the GPU. As they are pleasingly parallel, we expect to have great performance improvements comparing to the traditional execution in the regular CPUs. In small data sets, it is not so evident and we lose some performance due to communication time which is bigger than the computing time. But for bigger data sets, the gain is really obvious.

The GPU can be seen as a set of grids and each grid will have a set of blocks. Each block has a set of threads. A block will share their memory among their various threads.

Hereupon, in CUDA implementations we assign each thread to a matrix element and since the GPU has lots of threads, it is really hard to stall the whole computing.

For the tiled implementation, we can assign each sub-matrix (tile) to a CUDA block, thus taking advantage of the shared memory of the CUDA block.

TABLE VI. CUDA MATRIX MULTIPLICATION (MILLISECONDS)

	32x32	128x128	1024x1024	2048x2048
Standard, Global Memory	266.561	271.208	300.067	474.101
Blocking, Shared Memory	268.462	270.861	281.840	359.777

We developed both implementations, and we saw that the tiled version was faster than the standard version. The difference between both implementations gets bigger how much the data set size increases. For the 2048×2048 data set, the standard version takes 474 milliseconds, and the tiled version takes 359 milliseconds. To put in perspective, the **IJK-transposed** implementation takes 7918 milliseconds to compute the same data set.

To sum it up, in GPU, the tiled version is faster than the classical implementation. For small data sets it is faster to execute in CPU than in GPU (due to the device initialisation and communication times) but for bigger data sets, the GPU is faster.

VI. CONCLUSIONS

The most evident conclusion we get from this study, is that we need to get a full knowledge of how much the machine is limited by both memory and computing limits. This is important because the code must be adapted to each machine in order to exploit every possible performance gain, and some techniques can be applied to certain machines where others cannot. Relatively to the algorithm, the computation must be always, if possible, row-wise to take advantage of the spatial locality. Even if we need to transpose every matrix needed, it is still profitable. And the most evident conclusion is that the GPU is faster than the CPU in the case study.

REFERENCES

[1] ASUS ZenBook 3. https://www.asus.com/Notebooks/ASUS-ZenBook-3-UX390UA/specifications/.

[2] Intel® Core™ i7-7500U Processor. https://ark.intel.com/products/95451/Intel-Core-i7-7500U-Processor-4M-Cache-up-to-3_50-GHz-.

[3] Intel® Xeon® Processor E5-2670 v2 . https://ark.intel.com/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz.

[4] SeARCH Nodes Specification. http://search6.di.uminho.pt/wordpress/?page_id=55.

[5] 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).

[6] Speccy - Fast, lightweight, advanced system information tool for your PC. https://www.piriform.com/speccy.

APPENDIX A
652-NODE HARDWARE CHARACTERISATION

Processor	
Manufacturer	Intel Corporation
Specification	Intel Xeon E5-2670v2
Code name	Ivy Bridge
Clock Frequency	2.50 GHz
Intel Turbo Boost	3.30 GHz
# Cores	10
# Threads	20
Peak FP Performance	400 GFLOPS/sec
Memory	
Cache L1	32 KB I + 32 KB D per core
Cache L2	256 KB per core
Cache L3	25 MB
Memory access bandwidth	50 GB/s
RAM Memory	64 GB