Matrix-matrix Multiplication Optimization, Performace Measuring and Code Profiling.

Costa, A. Sérgio - A78296 and Resende, José - A77486

Abstract— **The objective of this project was to make use of an HPC environment to its full potential by using, as an example, the matrix-matrix multiplication problem for square matrices and its hardware-specific optimization, profiling and measuring. In this report, we will be observing multiple implementations that have several changes in order to improve their performance in the 662 machines from Minho University SeARCH Cluster. These changes instead of mere speculations are achieved by analyzing the hardware architecture on which the program is going to run on and the results taken from the hardware counter made available by the PAPI framework (like floating point operation and multi-level cache accesses), as well by simple code profiling throughout the implementations. Besides this, there were also strict considerations with the size of the inputs in correlation with the hardware characteristics (cache size), changes in index order and the use of simple algebraic techniques like transposing the matrices for better use of the hardware characteristics both on CPU and on GPU.**

I. INTRODUCTION

In a research environment is common to find computer intensive problems that require a well-optimized implementation to be solved in a timely manner otherwise it would be impossible to solve them. It's our job as a performance engineer to be able to study a certain problem in order to take advantage of every resource available in a machine. With our knowledge of optimization techniques and the inner working of a modern high-performance computer we can achieve better overall performance. With this paper, our primary objective is to study the matrix-matrix multiplication and catch any bottleneck that can decrease the overall performance and try to idealize a solution or a workaround to decrease the execution time by taking advantage of all the resources that are available to us and increase the overall performance of this crucial operation. Initially to understand the environment in which these benchmarks and tests were made on we will be using the *roofline model* to characterize the hardware and understand its characteristics and possible bottleneck and limitations. Then we will be studying how does the size of the square matrices, rearranging the access index and transposing the matrices and different implementations of this operation will affect the overall performance. This results will be validated and explained with the help of both the roofline model idealized before and the hardware counter extracted by the *PAPI* framework. This will also provide useful information for finding what we can improve in the next iteration of the implementation. Finally, we will compare an implementation for a Many-core processor and GPU with an *Intel Knight Landing* and a *Nvida K20* and draw some conclusions.

II. HARDWARE SPECIFICATION

Firstly we will analyze the hardware used in the making of this study, this will reveal any bottlenecks or limitations of the hardware as well as potential reasons for lost performance. We used a 2017 Macbook Pro 13 with an Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz and a 662-node with a Intel Xeon Processor E5-2695 v2 @ 2.40GHz. For our computer we got the information from the Intel, Apple and cpuworld website and the stream benchmark, and we used wikipedia to get the ram latency. For the 662 node we used the search6 hardware nodes page and the Intel website to get the specifications.

TABLE I
TEAM LAPTOP SPECIFICATIONS

Processor	
Manufactor	Intel corporation
Model	Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
Code Name	Kaby Lake
# Cores	2
# Threads	4
Base Frequency	2.30 GHz
Turbo Frequency	3.60 GHz
Peak FP performance	147.2 GFlops/s
Memory	
Cache L1	32KiB Data 32KiB Intructions
Cache L2	256KiB
Cache L3	4MB
Memory Bandwidth	19.728 GB/s
Memory Latency	0.469 ns
Main Memory	8 GB
Memory channels	2

III. ROOFLINE MODEL

With the roofline model we got a full characterization of our machine limits in terms of Flops and memory performance. With this we build a two dimensional graph where the X measures the operational intensity (floating point operations (FLOPS) per byte) and the Y measures the peak floating point performance.

$$\text{Peak GFlops} = \text{Number of processors} \times \text{Number of cores} \times \text{Clock frequency} \times \text{SIMD width} \times \text{FMA} \times \text{CPI}$$

For this formula we assumed an optimal CPI of 1.

The Roofline is obtained by applying simple bound and bottleneck analysis. In this formulation of the Roofline model, there are only two parameters, the peak performance and the peak bandwidth of the specific architecture, and one variable, the arithmetic intensity. The attainable GFLops/sec is given by the minimum of those values.

Attainable GFlops/sec = min(peak performance, peak bandwidth $\times$ arithmetic intensity)

The peak bandwith was obtained using **stream benchmarking** for both, computer and cluster.

The team's computer features **FMA3** and **AVX2** with SIMD of 256 bits (8 floats), the Peak Performance is obtained by the following calculation: $2 \times 2 \times 2.3 \times 8 \times 2 \times 1$ = 147,2 GFlops. The ceilings below were calculated by removing features from the formula. It's also important to notice that the computer's memory has 2 channels.

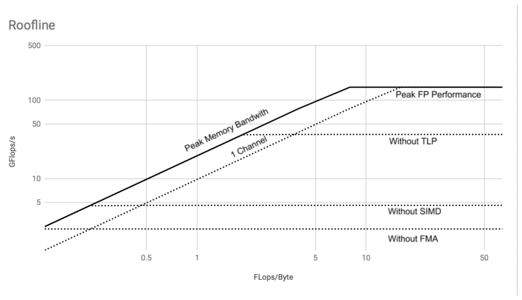

Fig. 1. Team Computer's Roofline

The cluster features **AVX2** with SIMD of 256 bits (8 floats), the Peak Performance is obtained by the following calculation: $2 \times 12 \times 2.4 \times 8 \times 1 \times 1$ = 460,8 GFlops. It's also important to notice that the cluster doesn't feature **FMA**. The ceilings below were calculated by removing features from the formula.

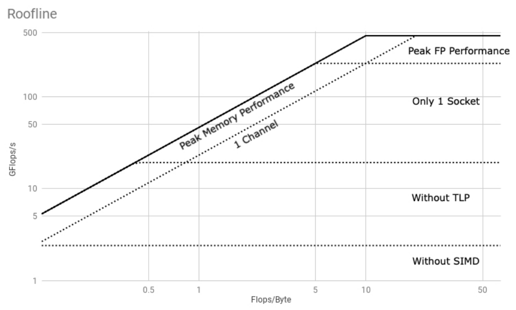

Fig. 2. Cluster's Roofline

After building both roofline models, the obvious next step was to build a roofline comparison between them. The result was the expected, the attainable GFlops/sec was much higher for the cluster, mainly because of the higher number of cores in each processing unity.

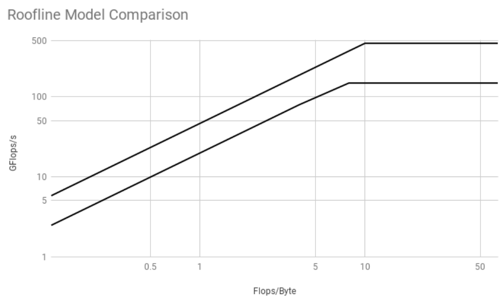

Fig. 3. Roofline Comparison

IV. METRICS

To justify the results and to have a better understanding of what is going on during each algorithm, the team decided to use PAPI. PAPI provides a simple interface for the user, and it allows the use of performance counters found in most major microprocessors, such as **total cache misses** and **total cache accesses**.

Furthermore, it was timed the duration of each implementation for the various matrix's sizes.

The chosen metrics were:

- **Time**: gettimeofday (microsec precision)
- **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
- **RAM accesses**: PAPI_L3_TCM
- **Flops**: PAPI_FP_INS

Unfortunately, all the miss rates were miss calculated and, when it was noticed, it was impossible to repeat the tests, because the cluster suffered some problems.

V. MATRIX DOT-PRODUCT ALGORITHM ANALYSIS

Matrix product is an operation that produces a matrix from the algebraic multiplication of two matrices. Matrix dot-product Algorithm consists of the computation of that algorithm: **C = A** $\times$ **B**, although all the matrices are squared and have the same size.

$$c_{\mathbf{ij}} = \sum_{n=0}^{size-1} a_{\mathbf{ik}} b_{\mathbf{kj}}, 0 \leq i \leq size-1, 0 \leq j \leq size-1$$

The work started with the implementation of 3 basic algorithms. These algorithms don't implement any optimization, the only difference between them is the loop reordering **IJK**, **IKJ**, and **JKI**. For each algorithm, there is a loop re-order, as it was said before, which will modify the elements iteration, consequently modifying how memory accesses are done.

A. Implementations

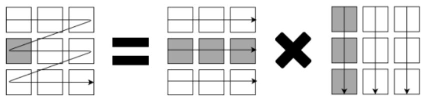

Fig. 4. IJK Algorithm

1) IJK: This is the default implementation. For each C element $c_{\mathbf{ij}}$, it multiplies each element in line **i** of A matrix, with each corresponding element in column **j** of B matrix. As can be seen in the picture above, C and A accesses are row-wise, although B accesses are column-wise. After observing this, it was created an ikj algorithm with a transposed B matrix.

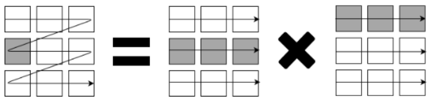

Fig. 5. IKJ Algorithm

2) IKJ: For each A element $a_{\mathbf{ik}}$, it multiplies that element with each element in line **k** of B matrix, and adds the result to each element in line **i** of C matrix. As it can be seen in the picture above, all accesses are row-wise.

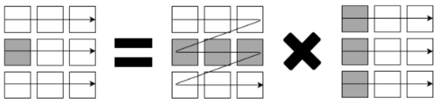

Fig. 6. JKI Algorithm

3) JKI: For each B element $b_{\mathbf{kj}}$, it multiplies that element with each element in column **k** of A matrix, and adds the result to each element in column **j** of C matrix. As can be seen in the picture above, all accesses are column-wise. After observing this, it was created a jki algorithm with all matrices transposed.

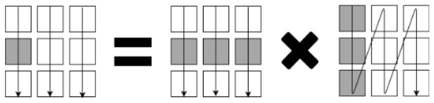

Fig. 7. IJK Transposed Algorithm

4) IJK Transposed: The formula is the same as the **ijk algorithm**, but it will iterate through each element of line **j** of matrix B, in spite of iterating through each element of column **j** of matrix B, since B was transposed, making all accesses row-wise. This makes all accesses row-wise as it can be seen in the picture above.

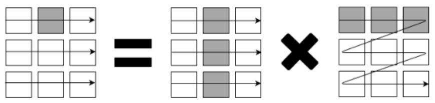

Fig. 8. JKI Transposed Algorithm

5) JKI Transposed: The formula is the same as the **jki algorithm**, but it will iterate through line elements, instead of iterating column-wise in all matrices. This makes all accesses row-wise as it can be seen in the picture above.

B. Data Sets

After analyzing all the algorithms, it was necessary to define the correct sizes for the matrices. This is an important decision since it will decide where the elements are in memory.

The decision of how many data sets should be tested was easy, one for each memory level.

Important information for the calculations:

- **FLOAT SIZE:** 4bytes
- **NUMVER OF MATRICES:** 3

1) L1:

SIZE: 32KB

$$\frac{32}{4} = 8000 Floats$$

$$\frac{8000}{3} = 2666 Floatspermatrix$$

$$\sqrt{2666} = 51.6333225737$$

$$\boxed{\mathbf{51} \times \mathbf{51}} \text{ (max matrix size)}$$

2) L2:

SIZE: 256KB

$$\frac{256}{4} = 64000 Floats$$

$$\frac{64000}{3} = 21333 Floatspermatrix$$

$$\sqrt{21333} = 146.058207575$$

$$\boxed{\mathbf{146} \times \mathbf{146}} \text{ (max matrix size)}$$

3) L3:

SIZE: 30MB

$$\frac{30}{4} = 7.5 M Floats$$

$$\frac{7.5M}{3} = 2.5 M Floatspermatrix$$

$$\sqrt{2.5M} = 1581.13883008$$

$$\boxed{\textbf{1581} \times \textbf{1581}} \text{ (max matrix size)}$$

After calculating all these maximum sizes, it was necessary to decide the sizes. The sizes decided were:

- **L1 size:** 32×32
- **L2 size:** 128×128
- **L3 size:** 1024×1024
- **RAM size:** 2048×2048

VI. ALGORITHM BEHAVIOR ANAYSIS

A. *Execution time*

From table 3 we can observe the different execution time for the defined datasets that use different Overall the JKI implementation was the slowest implementation, this can be given for its poor cache optimization because of the multiple column iterations, and the IKJ was the overall fastest implementation, this can be justified by the row iteration off all the matrix and its faster than the transpose implementations because of the lack of wasted time on transposing the matrices that need it.

	32x32	128x128	1024x1024	2048x2048
ijk	0,047	2,842	1,09E7	9,37E7
ikj	0,015	0,721	3,58E5	2,85E6
jki	0,059	3,357	2,4E7	1,76E8
ijk_trans	0,052	2,780	1,3E6	1,09E7
jki_trans	0,019	0,770	3,72E5	2,9E6

TABLE II
EXECUTION TIME (MILISECONDS)

B. *Main Memory and cache behaviour*

To analyze the RAM behaviour we decided to use the PAPI to have the number of Ram accesses for every size of data set and implementation. With this metrics we can clearly understand that the implementations that use column wise iteration have more ram access than the the implementations that use a pure row wise iteration, that allows them to have a better use of the cache and therefore being faster without the huge miss penalty of ram accesses.

	32x32	128x128	1024x1024	2048x2048
ijk	120	814	1,8E5	3,6E6
ikj	115	806	5,4E4	1,08E6
jki	162	1052	1,9E5	5,37E6
ijk_trans	103	809	6,5E4	4E5
jki_trans	119	991	9,2E4	1,9E6

TABLE III
RAM ACCESSES

C. *Floating point performance*

For the floating point performance we used the recorded measures from the PAPI counters to calculate the floating point operations per iteration through out all operations using this :

$$I = \frac{\#FP}{64 * RAMACCESSES}$$

$$GFlops/sec = \frac{Flops}{TExec(s)} * 10^{-9}$$

With the values obtained we plotted them on the roofline model by implementation with the different sized data sets defined before and the result was the following :

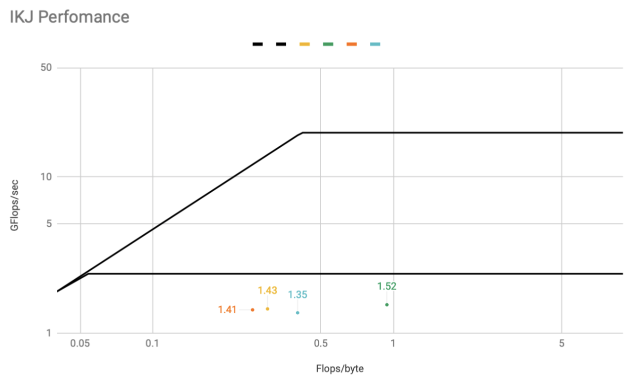

Fig. 9. IKJ Performance

- 🟨 - 32×32
- 🟩 - 128×128
- 🟧 - 1024×1024
- 🟦 - 2048×2048

VII. OPTIMIZATIONS

Henceforth the analysis of the first implementations we understood that we could optimize the implementation to better use the resources that are at our reach for a better use of the resources that are available to us.

A. *Blocking*

Using the matrix multiplication by blocks we will divide each matrix into significant chunks and apply the multiplication as independent sub-matrices. This will improve the cache usage because each one of this sub matrices can fit entirely on cache saving precious time.

	32x32	128x128	1024x1024	2048x2048
block	45	2769	1,47E6	9E6

TABLE IV
TIME OF BLOCKING OPTIMIZATION (MICROSECONDS)

B. *Vetorization*

With vectorization we were able to reduce the number of instructions because of the AVX capacity of the 662 node to run 256 bits instructions (8 elements).

	32x32	128x128	1024x1024	2048x2048
blockVec	10	619	4,81E5	4,2E6

TABLE V
TIME OF VERTORIZATION OPTIMIZATION (MICROSECONDS)

C. Multi-threading and Blocking

With blocking we have independent sub matrix that can easily be used to computed this matrices in parallel using OpenOMP to split the blocks by the 24 cores of the cpu in the 662 node.

	32x32	128x128	1024x1024	2048x2048
blockOMP	46754	23086	2,9E5	8,2E5

TABLE VI
TIME OF BLOCKING WITH MULTI-THREADING OPTIMIZATION (MICROSECONDS)

D. Multi-threading, Vetorization and Blocking

With the combination of this techniques we can achieve the best overall performance. Has we can see by the following table there was a considerable speed up in comparison with the version without vectorization.

	32x32	128x128	1024x1024	2048x2048
blockOMPVec	39055	26723	75688	3,56E5

TABLE VII
TIME OF BLOCKING WITH MULTI-THREADING AND VETORIZATION OPTIMIZATION (MICROSECONDS)

E. Many-core processor

We used the many-core processor KNL using different number of threads and diferent sized data sets. We can interpret the result as with smaller matrices we have better performance with a smaller number of threads, while in bigger matrices we have better performance with 32x32 and 128.

	32x32	128x128	1024x1024	2048x2048
32	63120	61165	91999	3,17E5
64	65415	51990	95436	6,15E5
128	86989	82999	1,29E5	3,02E5
256	1,23E5	1,23E5	1,91E5	3,92E5

TABLE VIII
TIME OF KNL (MICROSECONDS)

F. GPU

We develop two implementations, one is the simplest implementation possible but is also slower and another that takes the advantage of the shared memory in a block. We can see a significant improvement of performance with the second one that can be justified by the better optimized memory accesses and usage within the block.

	32x32	128x128	1024x1024	2048x2048
Simple	2,66E5	2,71E5	3E5	4,74E5
Shared Memory	2,7E5	2,68E5	2,81E5	3,59E5

TABLE IX
TIME OF CUDA (MICROSECONDS)

VIII. CONCLUSIONS

Even though all the cluster's problems and all the abnormal results, the goal, which was the comprehension of a computer system using different optimizations of an algorithm, was achieved. Various optimizations, which took advantage of many computer features and were tested with different hardware components, were implemented, and the final conclusion was that it is really difficult to predict the overall system's performance. The optimizations were noticed, however sometimes not as much as it was suposed to.

IX. APPENDIX

Processor	
Manufactor	Intel corporation
Model	Intel Xeon Processor E5-2695 v2 @ 2.40GHz
Code Name	Ivy Bridge
# Cores	12
# Threads	24
Base Frequency	2.40 GHz
Turbo Frequency	3.20 GHz
Peak FP performance	460,8 GFlops/s
Memory	
Cache L1	32KiB Data 32KiB Intructions
Cache L2	256KiB
Cache L3	30MB
Memory Bandwidth	46,19 GB/s
Main Memory	64 GB

TABLE X
CLUSTER SPECIFICATIONS

REFERENCES

[1] Search node specification http://search6.di.uminho.pt/wordpress/?page_id=55
[2] 662 node CPU https://ark.intel.com/products/75281/IntelXeonProcessorE52695v2-30MCache240GHz
[3] Team laptop CPU https://ark.intel.com/products/97535/IntelCorei57360UProcessor4M-Cacheupto360GHz
[4] Team laptop page specification https://support.apple.com/kb/SP754?locale=en_US
[5] Team laptop page features http://www.cpuworld.com/CPUs/Core_i5/IntelCore%20i5%20i5-7360U.html
[6] Roofline Calculation https://en.wikipedia.org/wiki/Roofline_model
[7] Roofline http://gec.di.uminho.pt/miei/cpd/aa/Roofline.pdf
[8] Loop Re-Order http://www.math-cs.gordon.edu/courses/cps343/presentations/Matrix_Mult.pdf
[9] Memory Hierarchy http://gec.di.uminho.pt/Discip/MaisAC/CSAPP_Bryant/csapp.ch6.pdf
[10] Optimizing Program Performance http://gec.di.uminho.pt/Discip/MaisAC/CS-APP_Bryant/csapp.ch5.pdf