Matrix-matrix Multiplication

Characterisation and Analysis of the Performance On a Computing Platform

João Alves
University of Minho
a77070@alunos.uminho.pt

Filipe Silva
University of Minho
a77284@alunos.uminho.pt

Abstract

The goal of this project is to study the Roofline Model methodology, on the characterization of the performance bottlenecks, code profiling and it's performance analysis, applying it to the matrix multiplication algorithm. The computing platform we worked with was a multicore processor, of the Universidade do Minho cluster SeARCH, more precisely the SeARCH 662 nodes.

I. Introduction

Nowadays, nearly almost desktop and server computers follow the same design. That is, besides the variations on the instruction sets, all of them use caches, pipelining, superscalar instruction issue, and out-of-order execution [1].

However, switching to multicore means that processors are a lot more diverse in terms of architecture and implementations, depending on the manufacturers which often ofter multiple products differing mainly on the number of cores.

So, it becomes the job of the programmer to better understand the different aspects of the platform's architecture. Therefore, were designed models to provide insights into the performance of the applications developed.

In this paper, we will focus on the performance of the matrix multiplication algorithm on a computing platform. Starting by developing a model to characterise the hardware environment, analyse bottlenecks and further study on the optimizations of the algorithm.

II. PAPI Performance Counters

The PAPI project successfully implements a cross-platform interface of performance hardware counters, that may provide information on how to reduce computational bottlenecks. However, in order to get relevant data from the API to our case study, we had to select a few of these events. So, to list all the available counters on the 662 nodes (PAPI version 5.5.0), we used *papi_ avail* and chose the following:

- **PAPI_LD_INS** Load instructions;
- **PAPI_L1_DCM** Level 1 data cache misses;
- **PAPI_L3_DCR** Level 3 data cache reads;
- **PAPI_L3_TCM** Level 3 cache misses;
- **PAPI_L3_TCA** Level 3 total cache accesses;
- **PAPI_FP_OPS** Floating point operations;
- **PAPI_VEC_SP** Single precision vector/SIMD instructions.

Counting these events allows us to induce certain values for important metrics, that affect the algorithm's performance. These are going to be discussed in later sections, but some of them include the miss rate for a certain level of cache, the number of RAM accesses per instruction, the floating point operations executed and the number of SIMD instructions, which is crucial when we are trying to achieve the maximum performance, on superscalar architectures.

III. Matrix Dot-Product Algorithm Analysis

Following the analysis and the characterisation of the hardware environment, the case study that we will be applying these modelling concepts consists of the dot product of two square matrices, where:

$$C = A \times B$$

Being **A** and **B** two square matrices of size $\mathbf{N} \times \mathbf{N}$ and **C** the result of the dot product between them.

i. First Implementations

The first implementations of the algorithm followed the standard three nested loop for the indexes i, j and k, where i represents the rows and j represent the columns of the matrices.

The main changes occur when the order of the indexes is changed, producing different accesses patterns to the elements of the matrices.

However, when the accesses are by column, there will be a negative impact on the performance, because the memory accesses are performed row-wise, during the execution of the algorithm.

This applies to the **IJK** version since the result stored in the **C** matrix is computed with a line of the matrix **A** and a column of the matrix **B**. To address this issue, we transposed the matrix **B** to correct the access, now being row-wise.

This also occurs for the **JKI** version of the computation, since the result is iterated column-wise. So, as seen above, we changed the solution to perform the transposition of the matrices **A** and **B**, before the multiplication part of it, and **C** in the end.

ii. Input Data Structure

In order to analyse the code execution on a single core of the multicore *662* node, four different data set sizes where defined, based on equation 1, to fit on each cache level. It was also taken into consideration choosing powers of two, from the range of values obtained by that equation. The results can be found in table 1.

$$CacheSize \geqslant size^2 \times sizeof(float) \times 3 \quad (1)$$

Fitting Level of The Hierarchy	Size
L1 Cache	32 x 32
L2 Cache	128 x 128
L3 Cache	1024 x 1024
External RAM	2048 x 2048

Table 1: *Data Set sizes.*

For each execution of an algorithm was performed a validation test, to ensure the implementation was correct. This was accomplished by verifying if all resulting columns of the product $\mathbf{A} \times \mathbf{B}$ had the same values. However, when measuring the execution time, this step wasn't taken into count.

iii. Execution Time Measurements

Having implemented all the of the solutions discussed above and selected the sizes for the input data sets, we measured the execution time, for each of the algorithms.

So to have results more reproducible, we applied a k-best scheme with $K = 3$ and a 5% tolerance, every 8 executions. The results obtained for each of the dot product implementations and its respective input size can be seen in table 2.

Looking at the results, we perceived that the **JKI** implementation performed the worst for all sizes of the matrices, justifying the previous statements about the inefficiency of the column-wise memory accesses. Although the differences are more perceptible for larger sizes of the matrices, even for the **32x32** it got the worst results, like we predicted. Also, we can recognise that the **IKJ** implementation performed the best

	32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
IJK	0.037	2.820	2972.836	23635.89
IJK Transposed	0.037	2.30	1139.97	9067.64
IKJ	0.033	1.91	970.51	7732.51
JKI	0.046	7.096	17431.63	80917.90
JKI Transposed	0.036	1.980	993.18	7851.90

Table 2: *First implementations execution time.*

for all of the sizes, although the differences are more perceptible for larger input matrices.

Regarding the **32x32** matrix size, there is not much of a difference between all of the execution times, even so, we verify what was stated above.

For the **128x128** sizes, only the **JKI** implementation stands out for being so much higher than the others. Because this still is a smaller size, the differences between the remaining, aren't as noticeable as well.

Now for the larger size of **1024x1024**, the contrast between the solutions that used column-wise access from the ones that implemented it row-wise is more perceptible. Verifying that the **IJK** and **JKI** benefited from transposing the matrices.

Finally, for the bigger dimension of **2048x2048**, the conclusions stated above are even more evident.

IV. Analysis of the Behaviour of the Algorithms

After measuring the execution times, we analysed the performance of the main memory accesses, the number of floating point operations and the cache behaviour, for each of the dot product implementations. These are going to be discussed in the next subsections.

i. RAM Behaviour

Considering that the 662 nodes have a 64 byte bus width, we conclude that every access to the RAM retrieves 16 floats to the cache.

Since the algorithm is implemented with three nested loops, each iterating N times and every iteration reaches an element of A, B and C, when the access pattern is row-wise, we estimate one RAM access per 16 iterations, per matrix. That is considering that a line of cache holds 16 floats.

However, this does not apply to the access pattern being column-wise so, for the purpose of estimating, we assumed that for every load of a value from a column, we access the RAM.

Taking these into consideration, we developed the following formulas to estimate the number of RAM accesses of a matrix, for each pattern:

$$\textbf{transposing a matrix} = 2N^2$$

$$\textbf{row-wise} = \frac{N^2}{L}$$

$$\textbf{column-wise} = N^2$$

Where $N \times N$ is the size of the matrix and L is the number of floats that fit in a line of cache. Finally, the values for each implementation we designed the following formulas:

$$\textbf{IJK} = \frac{N^2}{L} + N^2 + \frac{N^2}{L}$$

$$\textbf{IJKTr} = 2N^2 + \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L}$$

$$\textbf{IKJ} = \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L}$$

$$\textbf{JKI} = N^2 + N^2 + N^2$$

$$\textbf{JKITr} = 2N^2 + 2N^2 + 2N^2 + \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L}$$

Having estimated these values, these were then obtained using the PAPI counters **PAPI_L3_TCM** and **PAPI_TOT_INS**, giving us the following results:

	32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
IJK	0,001803311	0,0001098	3,8972E-05	1,98544E-05
IJK Transposed	0,001573423	0,000141863	4,18087E-05	1,87154E-05
IKJ	0,001353602	0,000132935	1,71935E-05	8,84091E-06
JKI	0,002153034	7,40155E-05	8,05869E-06	0,0005191
JKI Transposed	0,002160826	0,000164294	6,38714E-05	5,21549E-05

Table 3: *RAM accesses per instruction.*

Finally, in order to measure the number of bytes transferred to and from the main memory we multiplied the each of the results obtained above by 64 since it is the number of bytes transferred on every RAM access. Doing this, we obtained the next results:

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	7648	33856	12598528	50425248
IJK Transposed	6720	38240	5756736	20594112
IKJ	5696	35744	2372224	9749216
JKI	9376	41472	12598080	4109140608
JKI Transposed	9120	44160	8804032	57449952

Table 4: *Data transferred to/from the RAM.*

		32 x 32	128 x 128	1024 x 1024	2048 x 2048
Normal	L1 MR(%)	0,319675799	50,84354022	50,05148565	50,06053531
	L2 MR(%)	47,28158812	0,212728108	99,96520818	99,65526762
	L3 MR(%)	94,86503453	10,82904941	0,018310724	0,009194196
Transposed	L1 MR(%)	0,309776054	3,248650954	3,18084901	3,15465252
	L2 MR(%)	41,95045334	0,616594482	3,146355414	2,798500911
	L3 MR(%)	93,47826087	71,45725973	4,125660755	1,939631658

Table 6: *Miss rates of the **IJK** implementation.*

ii. Floating Point Operations Analysis

Next, in the behaviour analysis of the implementations, we tried to estimate the number of floating point operations executed during each dot product implementation.

Since every algorithm does two operations every iteration, we predict that all implementations are going to execute $2 \times N^3$ operations. To verify our estimates, we tested this for the various sizes of input, giving the following results:

Size	FP Operations
32 x 32	66267
128 x 128	4817848
1024 x 1024	5051118689
2048 x 2048	39683603954

Table 5: *Number of floating point operations executed.*

iii. Cache Behaviour

Finally, in order to have some insights into the improvements of transposing a matrix on the cache behaviour, we determined the miss rate for each cache level on implementations with and without transposing the matrices. We used the following PAPI counters and formulas for each level:

$$\textbf{L1 MR} = PAPI\_L1\_DCM / PAPI\_LD\_INS$$

$$\textbf{L2 MR} = PAPI\_L2\_TCM / PAPI\_L1\_DCM$$

$$\textbf{L3 MR} = PAPI\_L3\_TCM / PAPI\_L2\_DCM$$

V. Optimizations

After analysing the previous algorithms, it is time to optimise the performance of them by taking full advantage of the resources available.

We were able to see that interchanging the loops made the code access the data in the order in which they are stored. This way we improved the cache performance without affecting the number of instructions executed.

Finally, there are other optimisations we can make to enhance the execution of the algorithm, that are going to be discussed in the next sections.

i. Blocking

In order to maximise the accesses to the data loaded into the cache, before it is replaced, we applied a technique known as blocking. This optimisation consists of, instead of operating on entire rows or columns of the matrices, dividing them into squared submatrices and operating over them. This will allow us to reuse more of the elements, before the need to access the main memory.

With better use of the cache, we expect the execution time to decrease for the data sets that were requiring more accesses to the RAM.

ii. Vectorization

Since the machine we used (*662 SeARCH*) featured AVX the vectorization could make the code faster so, after Blocking, we added vectorization to the code, since the code wasn't vectorized with the compiler flags a flag was added to the code to force the code vectorization by the compiler.

References

[1] Andrew Waterman Samuel Williams and David Patterson. “Roofline: AnInsightfulVisual Performance Modelfor Floating-Point Programsand Multicore Architectures”. In: (2008).