MATRIX MULTIPLICATION

Filipe Mota
Universidade do Minho
Braga, Portugal

Abstract. The purpose of this project is to use matrix multiplication as an example algorithm on the studied platform cluster SeARCH, specifically the node 662.

We will make a fully characterizations of the hardware available and its limitations, a study of the metrics used to measure performance, the performance of the different versions of the dot-product algorithm discussing the obtained results and a conclusion of the work done.

Introduction

Researchers and business companies simulate real-life problems computationally, and though there is saved a lot of money doing this tests through Central Processing Units (CPU's), these require a high amount of resources, other companies need their software with faster response, there is where an performance engineer comes in, he must be able to analyze the code and identify what can be done in order to obtain a better use of the resources, this can be applied to a specific problem which requires a lot of resources so it is optimized to a determined architecture or a simpler one, which requires optimization to normal computers and mobile devices

On this paper we will focus matrix multiplications, which are the base of products of some big companies like google or optimization algorithms from facebook.

The first step will be to identify and characterize the hardware used, in order to reveal potential bottlenecks, in order to do so it will be analyzed in the widely used roofline model. The second step is to study and analyze the results of the different implementation of the dot product matrix multiplication in squared matrices with different index orders, taking also into account the different sizes of the matrices. Given the inoperability of the cluster, the code was runned in my computer, a quadcore that will be characterized later in this article with the help of the low level hardware performance counters Performance Application Programming Interface (PAPI), that gives us a more detailed view of the algorithm behaviour.

Hardware Specifications

Before starting the analysis of the different versions of the matrix multiplication it is required that we obtain all the available information about the hardware that will run the code and identify potential performances bottlenecks. The hardware characterization of gather from intel website

Table Ⅰ. Team Laptop specifications

Processor	
Manufacture	Intel
Model	Intel core i7 7700HQ
Code Name	Kaby lake
# Cores	4
# Threads	8
Base Frequency	2,8
Turbo Frequency	3,8
Peak FP performance	179,2
Memory	
Cache L1	32 KB Instruction Cache 32 KM Data Cache
Cache L2	256 KB
Cache L3	6 MB
MEmory Bandwidth	34,1
Memory Latency	Msec
Main Memory	8 gb DDR3L 2600 MHz SDRAM
Memory Channels	2

Roofline Model

The Roofline Model allows us to see the the factors affecting performance of computer systems, by quantifying the system bottleneck.

By putting together a 2-dimensional graph peak floating point performance the operational intensity and the memory performance we get the relation between the traffic of the processor and main memory. This relation consists in the combing peak floating point performance (Y) and operational intensity (X).

Peak floating performance and the attainable GFlops/sec can be obtained by the following formula:

Peak FP = Number of Processors x Number of Cores x Clock Frequency x SIMD width x FMA x CPI

Attainable GFlops/sec = Min (Peak FP, Peak Memory Bandwidth x Operational Intensity).

In both systems we used an optimal CPI of 1 with the number of lanes of functional units being equal to the average cycles needed to execute one instruction. The peak memory bandwidth was obtained using stream benchmarking

The pc used for the project and to compare to the SeARCH, features FMA3 and AVX 2.0 with SIMD instructions of 256 bits. The Peak FP performance is given by the following calculations 179,2 GFlops = 1 x 4 x 2,8 x 8 x 2 x 1.

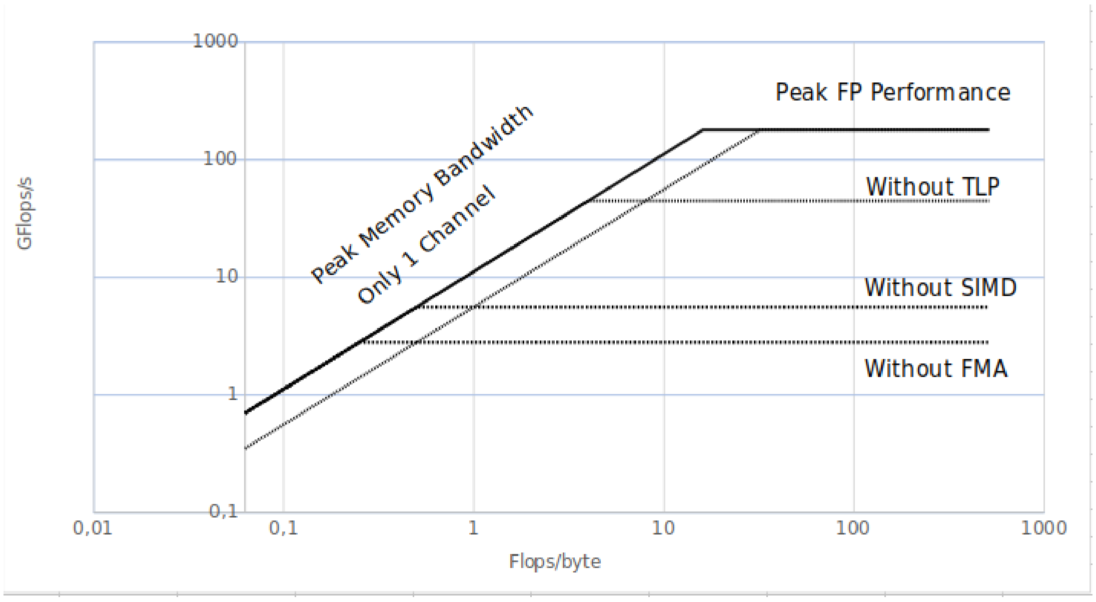

Fig. 1. Team Laptop Roofline Model

For the 662 node of the SeARCH, who features AVX implementing SIMD instruction of 256 bits, but without FMA, the peak performances is given by 460,8 = 2 x 12 x 2,4 x 8 x1

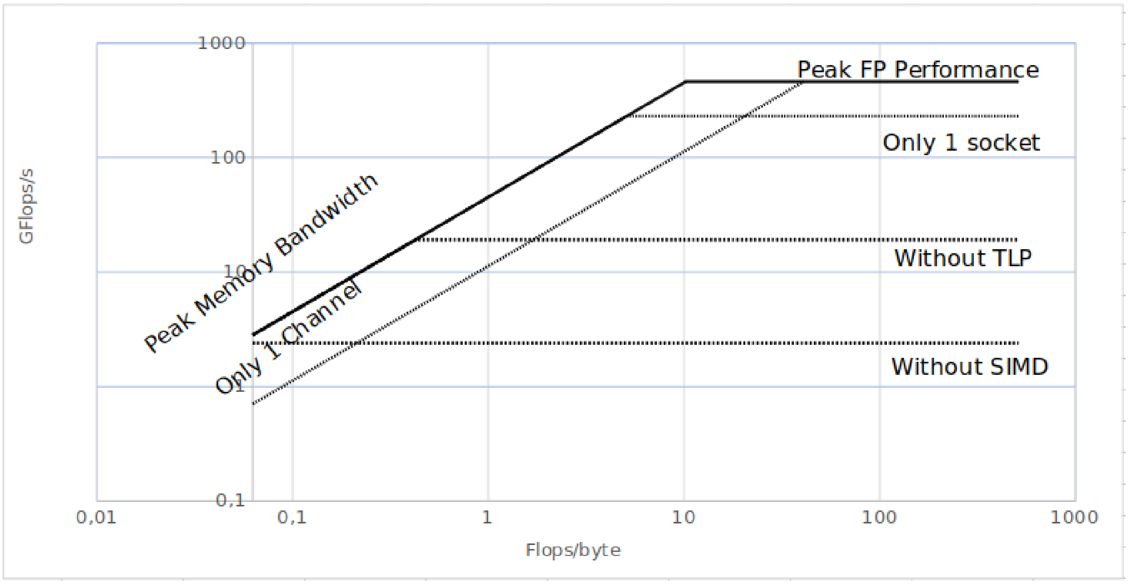

Fig. 2. Node Roofline Model

As expected the the cluster node obtains a much higher GFLOPS performance due to the number of cores in each processing unit.

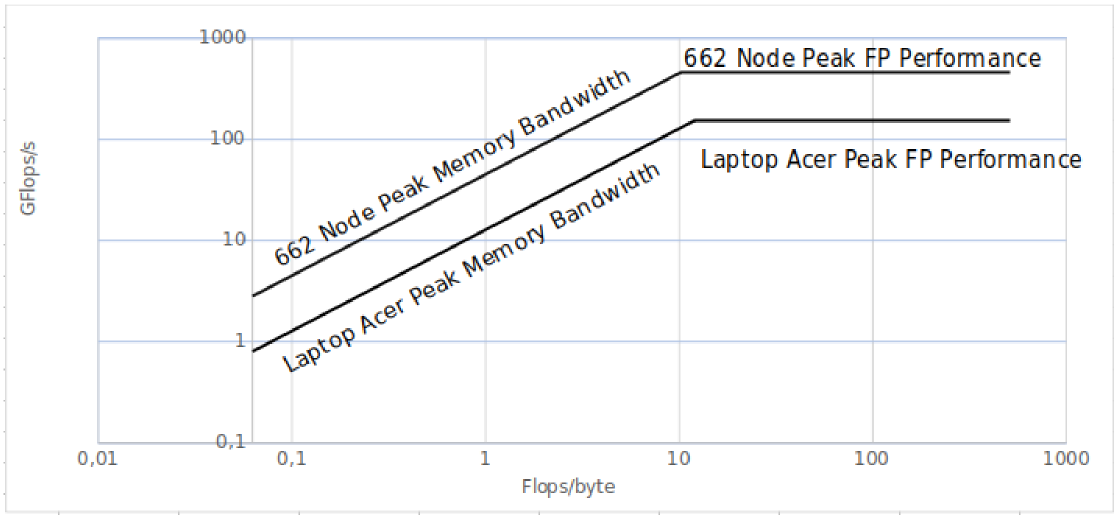

Fig. 3. Laptop and 662 Node Comparison Roofline Model

PAPI

In order to gather more information about our algorithm behaviour, it was used the PAPI platform, version 5.6.1. This application allows us to monitor many hardware counters from the processor. In this project we only use a few:

- 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 acesses;
- PAPI_TOT_INS : Instructions performed;
- PAPI_LD_INS : Instructions loaded;

With the counters we can infer some values like Random Access Memory (RAM) accesses per instruction or the miss rate of a cache level.

ANALYSIS OF THE MATRIX DOT PRODUCT

The matrix dot-product algorithm consists in calculation of the **C = A x B,** being three squared matrices.

This project started with the implementation of three versions of the dot-product function without optimization, **IJK, IKJ** and **JKI**.

In the IJK version a value is computed and stored in the matrix **C**, being this value the computed line of matrix A with the column of matrix B.

In the IKJ will be stored a line, using an element of the matrix A, and a line from matrix B.

As for the JKI, in the matrix C will be stored the result of a line of the matrix A with an element of matrix B

Because of the accesses to the elements row by row are a bottleneck, we will transpose the matrix that will be accessed row by row, before the multiplication occurs, they are the IJK and JKI versions.

The data-set size used to perform were selected in order to fit the given cache level using the formula:

$$size\ x\ size\ x\ float\ size\ x\ 3\ matrix \leq cache\ level\ size$$

Table Ⅱ. Matrix Sizes

Memory level	Size	cache level size / memory size
Cache L1	32 x 32	12 / 32 KB
Cache L2	128 x 128	192 / 256 KB
Cache L3	512 x 512	3 / 6 MB
Main Memory	2048 x 2048	48 / 6 MB

Only value multiple of 16 were considered because each cache line has a size of 64 bytes, corresponding to 16 float elements.

In the 2048 size there will be used the 6 MB of the cache and 42 MB of the main memory, we will see some very interesting results further ahead.

MEASURES

In this paper the technique used to measured the execution times was the K-Best squeme with K=3, a 5% tolerance and at most 8 executions, the first measurement of our versions was their execution time in milliseconds:

Table Ⅲ. Time Measurements

	32 x 32	128 x 128	512 x 512	2048 x 2048
IJK	0,036	2,129	148	38 363
IKJ	0,028	1,235	77	5 076
JKI	0,039	1,765	207	93 127
IJK Transposed	0,034	2,046	141	9 359
JKI Transposed	0,034	1,299	79	5 257

As we can see in the table the version **JKI** has the worst times, and the **IKJ** the best.

For the **32** data set the difference for all versions is very little because of the reduced size. Comparing the **IJK** and **JKI** we can see the the **JKI** starts with better performance in the smaller data sizes and a much worst performance in higher data sizes, this is due to the column wise access while the **IKJ** is the fastest because of the row wise access, however in theirs transpose the **JKI** not only achieves gains in the order of almost 18 times in the **2048x2048**, but also is faster then the **IJK** transpose. Also, if we take a closer look, we notice the ratio between the **IJK** and **KJI** in the **512** size, is 71% slower, but in the **2048** is 41% slower, this is because of the cache, while in the **512** data set has the capacity to hold 2 matrices, in the **2048** data set we are using 6 MB of RAM and 42 MB of the main memory, the impact in the performance is more substantial worst due to the constant accesses to the main memory.

MAIN MEMORY

In order to analyze the RAM behavior it was decided to estimate the RAM accesses per instruction and the number of bytes transferred to/from RAM.

As the algorithm has 3 nested cycles in each iteration there will be one access to A, B and C resulting in $size^3$ iteration.

When a matriz access pattern is row-wise it was considered we were accessing RAM every 16 iterations to get more values, because the cache line can hold 16 float elements (64 bytes), when it is access by column-wise, it was assumes the worst scenario, where each element accessed will be on RAM, however this will not always happen, because of data sets like the 512, that can hold 2 matrices in the L3 cache, (we can see in the table 1 the impact, the **IJK** takes 148 ms, the **JKI** takes 77 ms (almost half) while the worst one **JKI** takes 207 ms (almost 3 times the **JKI**)

- **Row-wise matrix** = $size\ x\ size\ /\ cache\ line\ width$
- **Column-wise matrix** = $size\ x\ size$
- **Transposing matrix** = $size^2\ iterations\ x\ values\ loaded\ per\ iteration$

The values of RAM accesses per instructions were calculated using PAPI counters **PAPI_L3_TCM** and **PAPI_TOT_INS** giving the following results:

Table IV. Ram Accesses / Instructions

	32 x 32	128 x 128	512 x 512	2048 x 2048
IJK	0,001 279	0,000 287	0,000 072	0,009 428
IKJ	0,001 343	0,000 293	0,000 099	0,010 408
JKI	0,000 608	0,000 236	0,000 121	0,011 192
IJK Transposed	0,001 573	0,000 323	0,000 137	0,009 887
JKI Transposed	0,001 063	0,000 315	0,000 134	0,009 561

To calculate the RAM traffic we just need to multiply the level 3 cache misses (PAPI_L3_TCM) by 64, which is the number of bytes that each access moves to cache:

Table V. Data Transferred from/to RAM (KBytes)

	32 x 32	128 x 128	512 x 512	2048 x 2048
IJK	24,8	333	6 756	50 513
IKJ	24,2	305	7 623	42 921
JKI	27,5	257	11 278	43 448
IJK Transposed	24,3	318	6 399	39 798
JKI Transposed	20	274	9 736	37 716

FLOATING POINT PERFORMANCE

The CPU that is being used for this project features FMA, which means that we will be performing one floating point operation on each iteration. This way we get FP Operations = size^3:

Table VI. Floating Point Operations

Size	FP Operation
32 x 32	32768
128 x 128	2097152
512 x 512	134217728
2048 x 2048	8589934592

Both algorithms benefited from being transpose, it was decided to use the one who beneficiated the most, the **JKI**.

To get information about the miss rate percentage on memory reads for each cache level, we use 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 Ⅶ. Miss Rate for the **JKI** implementation

	Cache	32 x 32	128 x 128	512 x 512	2048 x 2048
Normal	L1	0,11	5,67	39,98	47,21
	L2	55,43	3,23	3,97	78,96
	L3	69,29	11,41	0,60	1,81
Transposed	L1	0,10	0,095	0,25	0,28
	L2	26,74	1,14	17,18	7,33
	L3	63,78	51,92	0,72	56,87

For the 32 data set, L2 and L3 miss rates are high because the data fits completely on L1 cache. The second data set, 128 as expected has the higher miss rate on L3 cache and the smallest on the L2 since data completely fit the L2 cache.

Despite cost of transposing the matrix which makes the miss rate worst in some cases, it provides a row-wise access to data, improving the reuse of cache and better performance

OPTIMIZATIONS

Blocking

The first optimization applied was blocking in order to get a better cache use. With this technique we divide the matrices in squared blocks and those blocks are calculated independently.

The data for matrices A and B will be needed by the several blocks created, resulting in a bigger data replication however having the blocks allows to keep those values in cache, reusing them, which reduces the main memory access that have high latency, resulting in a better performance

Multi-core and Vectorization

Until now the tests were performed in a single core, not taking the advantage of the full power of the computer, it was used OpenMP to split the matrix blocks by the 8 cores of the Acer laptop.

With vectorization we can apply one instruction over multiple elements stored in vector registers, this processor has AVX2 which means has the ability to run 256 bits instructions giving a better performance.

Table Ⅷ. Blocking Implementations of the **JKI** (Msec)

	Block size	32 x 32	128 x 128	512 x 512	2048 x 2048
JKI	n/a	0,039	1,765	207,56	93 127
JKI Transposed	n/a	0,034	1,299	79,12	5 257
Blocking	16	0,041	1,725	99,87	6 406
	32	0,042	1,830	111,24	7 389
Block + Vectorization	16	0,005	0,233	16,49	3 153
	32	0,003	0,121	8,74	1 328
Block + openmp	16			29,74	2 054
	32			28,63	1 578

As we can see the 3 optimizations had very satisfactory performance results except the blocking for the lowers data sizes (weren't performed any tests with openmp, because isn't possible to escalate with 8 threads such sizes)

As explained earlier, with blocking the different blocks created will have all the same data of the matrices, so we get a better performance with the 16 block size (for the blocking only), because the higher data replication is justified by the less accesses to the main memory, while with 32 block size there are much more accesses to the memory, it's also patent that blocking benefits the most when there is access to the main memory. With vectorization is the opposite, as the vectorization has the capacity to run 256 bits instructions, there is no need for so many blocks. The best result were with vectorization for all data sets.

With openmp, we can see the 512 data set with 8 threads limits the performance, however we get a very positive performance for the 2048 data set.

Conclusion

The matrix multiplication it's an algorithm where multiple tunings can be performed so the overall performance improves, some tuning might seem good when though, but the scenario rapidly shifts when tested.

References

[1] Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-00086), 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