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 its 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 [6].

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, models are designed 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 optimisations of the algorithm.

II. Hardware Platform Characterisation

In order to completely understand the performance of an algorithm, we first need to be fully aware of the resources offered by the hardware, so that it becomes possible to develop an analysis of the bottlenecks, limiting the performance of a certain kernel.

Since we used both our team's main laptop and a SeARCH cluster 662 node, we will perform an analysis into the hardware provided by both platforms, plotting the values into a Roofline Model.

i. Team's Laptop Hardware Analysis

The team's main laptop is a Dell XPS 15 9560 released in February 2017 (release tag 29Y07H2). It packs an Intel Core i7 7700HQ Kaby Lake processor and 8GiB (2 slotsx4GiB) of DDR4-2400MHz RAM. A more detailed view of the specifications and features implemented in the chip can be found in table 1.

In order to estimate the peak floating point performance (**PFP**) for single precision using the hardware specifications, it was considered that the width of the SIMD registers is 256 bits, hence 8 floats, and the throughput is one instruction per clock cycle. Also, because it is an architecture with FMA, we are able to perform fused multiply-adds. Multiplying that by

Intel® Core™ i7-7700HQ Processor	
Code Name	Kaby Lake
Essentials	
Vertical Segment	Mobile
Processor Number	i7-7700HQ
Performance	
# of Cores	4
# of Threads	8
Processor Base Frequency	2.80 GHz
Max Turbo Frequency	3.80 GHz
Bus Speed	8 GT/s DMI
Memory Specifications	
Level 1 cache size	4 x 32 KiB 8-way set associative instruction caches
	4 x 32 KiB 8-way set associative data caches
Level 2 cache size	4 x 256 KiB 4-way set associative caches
Level 3 cache size	6 MiB 12-way set associative shared cache
Max Memory Size	64 GiB
Memory Types	DDR4-2400, LPDDR3-2133, DDR3L-1600
Max # of Memory Channels	2
Max Memory Bandwidth	37.5 GiB/s
Advanced Technologies	
Intel® 64 ‡	Yes
Instruction Set	64-bit
Instruction Set Extensions	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2

Table 1: *Team's laptop specifications [1] [3].*

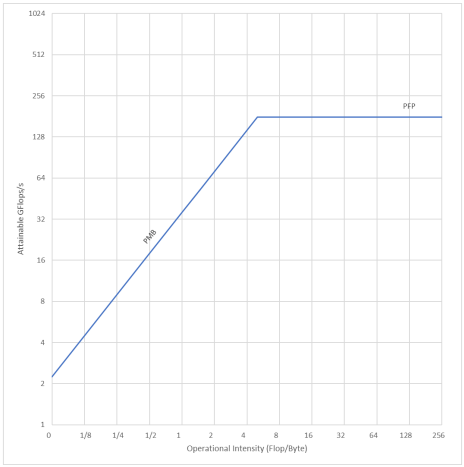

Figure 1: *Roofline Model for our team's laptop.*

the number of cores and the clock rate we came up with the following:

$$\begin{aligned} PFP &= \#cores \times frequency \times \frac{SIMD\,width}{SIMD\,throughput} \times FMA \\ &= 4 \times 2.8 \times \frac{8}{1} \times 2 \\ &= 179.2\,GFlops \end{aligned} \tag{1}$$

Since this is an hardware limit, we can plot an horizontal line illustrating that the actual floating point performance can be no higher than the horizontal line.

Next, to measure the maximum streaming bandwidth, we used the STREAM benchmark obtaining a value of 36 GiB/s for the peak memory bandwidth (**PMB**).

Finally, we are able to calculate the performance of the kernel as seen in equation 2.

$$performance = min(PFP, OP \times PMB) \tag{2}$$

Figure 1 shows the plotted lines. These lines set an upper bound on performance of a kernel depending on it's operational intensity [6].

i.1 Addition of Ceilings

Although the above values provide us with an upper bound to performance, this was obtained considering some optimisations are performed. In order to better understand which of them should be implemented first, we added the following ceilings ranking them from bottom to top, regarding the matrix multiplication kernel:

- The lowest ceiling comes from using only one of the cores provided by the machine. There so, depending only on the clock rate being the single parameter in the equation. Thus, the first optimisation that could be applied is thread-level-parallelism (**TLP**), taking full advantage of the cores presented by the hardware.
- Second comes from exploiting vectorial instructions, that decrease the CPE of the kernel, given that we are able to perform the same operation over multiple data simultaneously.
- The third appears from using FMA, to double the number of multiply-addition operations per cycle, because we are able to perform fused multiply-add operations.
- When it comes to memory ceilings, we were able to obtain a value of 16 GiB/s for the peak mem-

ory bandwidth, using the STREAM benchmark compiled without any software optimisations.

At last, we ended up with the following Roofline Model, for our team's laptop.

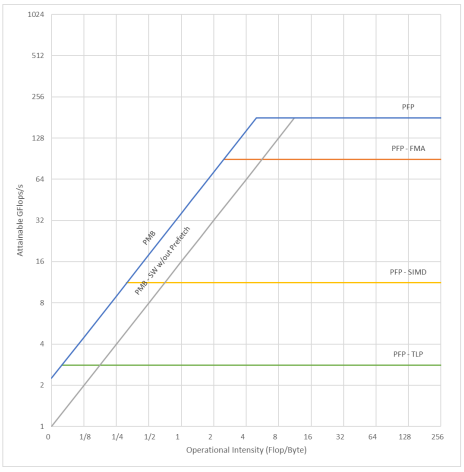

Figure 2: *Roofline Model with ceilings for our team's laptop.*

ii. SeARCH's Node Hardware Analysis

The SeARCH's 662 nodes are dual processor machines, each an Intel Xeon Processor E5-2695 v2 Ivy Bridge [5]. The more specific hardware features regarding the chip are in table 2.

With this in mind, we can estimate the peak floating point performance for single precision using the same formula as for our team's laptop as seen below in equation 3. The main differences occur in the number of processors, because the laptop is a single processor, and the cluster node has no support for FMA.

$$\begin{aligned} PFP &= \#processors \times \#cores \times frequency \times \frac{SIMD\,width}{SIMD\,throughput} \\ &= 2 \times 12 \times 2.4 \times \frac{8}{1} \\ &= 460.8\,GFlops \end{aligned} \tag{3}$$

Finally, we adapted the STREAM benchmark to run efficiently on the Xeon E5-2695v2, obtaining a peak memory bandwidth of 43 GiB/s.

Intel Xeon Processor E5-2695 v2	
Code Name	Ivy Bridge EP
Essentials	
Vertical Segment	Server
Processor Number	E5-2695V2
Performance	
# of Cores	12
# of Threads	24
Processor Base Frequency	2.40 GHz
Max Turbo Frequency	3.20 GHz
Bus Speed	8 GT/s QPI
Memory Specifications	
Level 1 cache size	12 x 32 KiB 8-way set associative instruction caches
	12 x 32 KiB 8-way set associative data caches
Level 2 cache size	12 x 256 KiB 8-way set associative caches
Level 3 cache size	30 MiB 20-way set associative shared cache
Memory Types	DDR3 800/1066/1333/1600/1866
Max # of Memory Channels	4
Max Memory Bandwidth	59.7 GiB/s
Advanced Technologies	
Intel® 64 ‡	Yes
Instruction Set	64-bit
Instruction Set Extensions	Intel® AVX

Table 2: *Intel Xeon Processor E5-2695 v2 detailed specifications [2] [4].*

Following this, we were able to plot the Roofline Model as shown in figure 3.

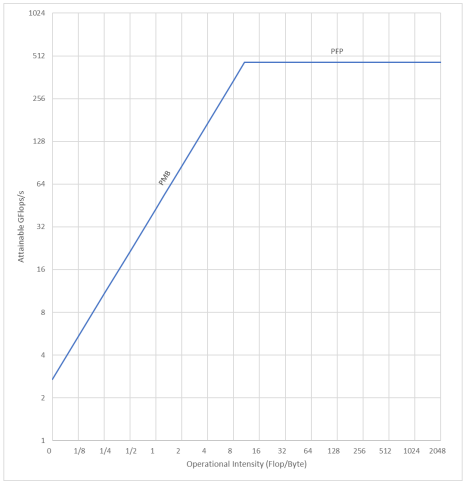

Figure 3: *Roofline Model for the Intel Xeon E5-2695v2 processor.*

iii. Comparison

As we can conclude the metrics regarding the performance analysis of the hardware are related to the resources it can provide. Table 13 in appendix i shows

the main differences between the two architectures analysed above, concerning their specifications.

Also, plotting both Roofline Models next to each other in figure 4 helps to understand that the cluster node has almost 3 times more computation power than our laptop, for CPU bounded applications.

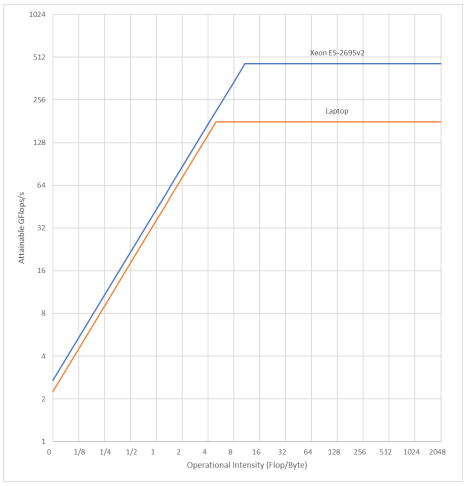

Figure 4: *Intel Core i7-7700HQ vs Intel Xeon E5-2695v2 processor Roofline Model.*

III. 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 are crucial when we are trying to achieve the maximum performance, on superscalar architectures.

IV. 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 access 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 matrices are stored row-wisely in memory.

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 were defined, based on equation 4, 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 3.

$$CacheSize \geqslant size^2 \times sizeof(float) \times 3 \qquad (4)$$

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 3: *Data set sizes.*

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.

For each execution a validation test was performed to ensure that 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.

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 4.

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

	32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
IJK	0.037	2.82	2972.84	23635.89
IJK Transposed	0.037	2.30	1139.97	9067.64
IKJ	0.033	1.91	970.51	7732.51
JKI	0.046	7.10	17431.63	80917.90
JKI Transposed	0.036	1.98	993.18	7851.90

Table 4: *First implementations execution time.*

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. This is due to the fact that the **32x32** matrices fit in the first level of cache, there so the data is not replaced/invalidated.

Moving to the **128x128** matrices, 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 a 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.

V. 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, to obtain the values for each implementation we designed the formulas in appendix ii.

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	1.80E−03	1.10E−04	3.90E−05	1.99E−05
IJK Transposed	1.57E−03	1.42E−04	4.18E−05	1.87E−05
IKJ	1.35E−03	1.33E−04	1.72E−05	8.84E−06
JKI	2.15E−03	7.40E−05	8.06E−06	5.19E−04
JKI Transposed	2.16E−03	1.64E−04	6.39E−05	5.22E−05

Table 5: *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	7.47E+00	3.31E+01	1.23E+04	4.92E+04
IJK Transposed	6.56E+00	3.73E+01	5.62E+03	2.01E+04
IKJ	5.56E+00	3.49E+01	2.32E+03	9.52E+03
JKI	9.16E+00	4.05E+01	1.23E+04	4.01E+06
JKI Transposed	8.91E+00	4.31E+01	8.60E+03	5.61E+04

Table 6: *Data transferred to/from the RAM (in KiB).*

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 and the hardware has no FMA support, 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 and implementations, giving the following results:

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	6.63E+04	4.82E+06	5.05E+09	3.97E+10
IJK Transposed	6.67E+04	4.21E+06	2.15E+09	1.72E+10
IKJ	6.58E+04	4.20E+06	2.16E+09	1.72E+10
JKI	6.80E+04	8.75E+06	2.44E+10	1.24E+11
JKI Transposed	6.59E+04	4.20E+06	2.15E+09	1.72E+10

Table 7: *Number of floating point operations executed.*

With the number of single precision floating point operations and the number of bytes to transferred to/from the main memory, we are able to obtain a value for the operational intensity. Also, having the time measurements for each size and implementation, we can plot the attainable floating point performance (see appendix iii). With this in mind, we plotted the obtained values in the Roofline Model in figure 5.

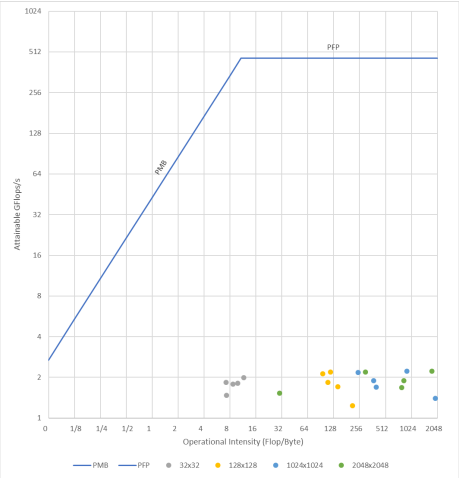

Figure 5: *Plotting of the obtained values.*

As we expected, because the values for the performance obtained for the data sets (except for the smallest) are bellow the flat part of the roof, the kernel performance is compute bound.

Looking at the single core peak floating point performance, all these values are limited by the lack of vectorization. This being the next optimisation to be performed, in order to surpass that ceiling.

iii. Cache Behaviour

At last, 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:

		32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
Normal	L1 MR(%)	0.32	50.84	50.05	50.06
	L2 MR(%)	47.28	0.21	99.97	99.65
	L3 MR(%)	94.86	10.83	0.02	0.01
Transposed	L1 MR(%)	0.31	3.25	3.18	3.15
	L2 MR(%)	41.95	0.62	3.15	2.80
	L3 MR(%)	93.48	71.46	4.13	1.94

Table 8: *Miss rates of the **IJK** implementation.*

Like we foresaw, transposing the matrices improves the re-usage of the values loaded from the main memory. Although in the smaller data sets the miss rates are still high, because of the cost of transposing the matrices, the execution times are lower compared with the version that accesses the values column-wise. Proving that the algorithm benefited from this change.

VI. Optimisations

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 elements, before the need to access the main memory, thus improving temporal locality.

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.

Applying this technique to the single threaded code, that also benefited from transposing the matrices, and for the bigger size of input, we got the results in table 9.

	2048 x 2048
IJK Transposed	9173
IKJ	9175
JKI Transposed	9182

Table 9: *Execution times (in ms) of the implementations optimized with blocking and transposition.*

ii. Vectorization

Since the machine we used (*662 SeARCH*) featured AVX instruction set, another optimisation that would improve the single core performance of the algorithms is vectorization through SIMD. With that in mind the team decided to switch to the Intel C++ Compiler (**ICPC**) in order to better analyse the vecotrization reports and instruct the compiler to vectorize in case it is needed.

Following that we measured the execution times for the two smaller data sets, getting the subsequent results in table 10.

	32x32	**128x128**
IJK Transposed	0.009	0.590
IKJ	0.009	0.560
JKI Transposed	0.010	0.609

Table 10: *Execution times (in ms) of the vectorized algorithms.*

iii. Multi-Core

After analysing and optimizing the performance of the single-threaded code, it was time to take full advantage of the resources provided by the multicore devices of the 662 nodes.

With that in mind, we adapted the vectorized code of the dot product to run efficiently on the 24 cores presented by the hardware, using **OpenMP** without Hyper Threading.

Next, we measured the execution time of the implementations, applying the same scheme as before, for the larger data set only, giving the following results in table 11.

	2048x2048
IJK Transposed	298.885
IKJ	223.530
JKI Transposed	325.403

Table 11: *Execution time (in ms) of the vectorized algorithms adapted to run in all cores of the multi-core devices.*

VII. Intel Knight's Landing

In order to complement the results collected before, we also adapted the code developed for the multicore 662 nodes, to run in the Intel Knights Landing many-core server. Meaning that we selected the best performer implementation (index order i-k-j) and increased the number of threads to run in the all of the 64 cores provided by the machine, although we also tested the same implementation only with 2 threads per core of the **KNL**.

The results regarding the execution time were taken and put in table 12, where we can compare with other hardware platforms.

VIII. GPU Kepler

At last, we modified the same function seen above to run in all SMX of a GPU Kepler (installed on a 662 node), rewriting the code with the CUDA paradigm in mind. This time, each thread computes one element of the result matrix,

After compiling and measuring the execution times, the results obtained can be seen in table 12.

	Xeon E5-2695v2	**Knight's Landing**		**GPU Kepler**		
	24 Threads	**64 Threads**	**128 Threads**	**Transfer to Device**	**Computation**	**Transfer to Host**
IKJ	223.53	290.43	164.38	14.26	140.35	9.92

Table 12: *Execution times (in ms) of the same implementation in multiple platforms.*

IX. Conclusions

Switching to a multicore architecture of computing systems, the programmers confront a more diverse range of hardware capabilities. So, it is important to fully understand and analyse each resource that can be provided, in order to squeeze the maximum performance of a kernel.

In this paper, we showed how important it is to fully characterise the hardware platform environment, developing the Roofline Model, in order to get insights into the performance of a given kernel.

This allied with the PAPI profiling API, allowed us to develop strategies to overcome the bottlenecks and

optimise the performance of the matrix multiplication kernel.

These approaches include correcting the use of cache, taking advantage of SIMD instruction sets and adapting the code to run efficiently on all cores.

References

[1] CPU-World. *Intel Core i7-7700HQ Mobile processor*. URL: http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%5C%20i7%5C%20i7-7700HQ.html.

[2] CPU-World. *Intel® Xeon® Processor E5-2695 v2*. URL: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%5C%20E5-2695%5C%20v2.html.

[3] Intel. *Intel® Core™ i7-7700HQ Processor*. URL: https://ark.intel.com/products/97185/Intel-Core-i7-7700HQ-Processor-6M-Cache-up-to-3-80-GHz-.

[4] Intel. *Intel® Xeon® Processor E5-2695 v2*. URL: https://ark.intel.com/products/75281/Intel-Xeon-Processor-E5-2695-v2-30M-Cache-2-40-GHz-.

[5] Services and Advanced Research Computing with HTC/HPC clusters. *Cluster Nodes*. URL: http://search6.di.uminho.pt/wordpress/?page_id=55.

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

A. Appendix

i.

Intel® Core™ i7-7700HQ Processor		**Intel Xeon Processor E5-2695 v2**	
Code Name	Kaby Lake	Code Name	Ivy Bridge EP
Essentials		**Essentials**	
Vertical Segment	Mobile	Vertical Segment	Server
Processor Number	i7-7700HQ	Processor Number	E5-2695V2
Performance		Performance	
# of Cores	4	# of Cores	12
# of Threads	8	# of Threads	24
Processor Base Frequency	2.80 GHz	Processor Base Frequency	2.40 GHz
Max Turbo Frequency	3.80 GHz	Max Turbo Frequency	3.20 GHz
Bus Speed	8 GT/s DMI	Bus Speed	8 GT/s QPI
Memory Specifications		**Memory Specifications**	
Level 1 cache size	4 x 32 KiB 8-way set associative instruction caches	Level 1 cache size	12 x 32 KiB 8-way set associative instruction caches
	4 x 32 KiB 8-way set associative data caches		12 x 32 KiB 8-way set associative data caches
Level 2 cache size	4 x 256 KiB 4-way set associative caches	Level 2 cache size	12 x 256 KiB 8-way set associative caches
Level 3 cache size	6 MiB 12-way set associative shared cache	Level 3 cache size	30 MiB 20-way set associative shared cache
Memory Types	DDR4-2400, LPDDR3-2133, DDR3L-1600	Memory Types	DDR3 800/1066/1333/1600/1866
Max # of Memory Channels	2	Max # of Memory Channels	4
Max Memory Bandwidth	37.5 GiB/s	Max Memory Bandwidth	59.7 GiB/s
Advanced Technologies		**Advanced Technologies**	
Intel® 64 ‡	Yes	Intel® 64 ‡	Yes
Instruction Set	64-bit	Instruction Set	64-bit
Instruction Set Extensions	Intel® AVX	Instruction Set Extensions	Intel® AVX
FMA Support	v3	FMA Support	n/a

Table 13: *Intel core i7-7700HQ vs Intel Xeon E5-2965v2 specifications.*

ii. Estimates of Load Instructions of Matrix Multiplication

The next formulas were derived with the intent to estimate the number of accesses to the main memory in the dot product matrix multiplication.

$$\mathbf{IJK} = \frac{N^2}{L} + N^2 + \frac{N^2}{L} = N^2 + \frac{2N^2}{L} \tag{5}$$

$$\mathbf{IJKTr} = 2N^2 + \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L} = 2N^2 + \frac{3N^2}{L} \tag{6}$$

$$\mathbf{IKJ} = \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L} = \frac{3N^2}{L} \tag{7}$$

$$\mathbf{JKI} = N^2 + N^2 + N^2 = 3N^2 \tag{8}$$

$$\mathbf{JKITr} = 2N^2 + 2N^2 + 2N^2 + \frac{N^2}{L} + \frac{N^2}{L} + \frac{N^2}{L} = 6N^2 + \frac{3N^2}{L} \tag{9}$$

iii. Matrix Multiplication Kernel PAPI Readings

With these counters, we were able to derive the subsequent formulas so to get insights on metrics relevant to the performance of the kernel.

$$\textbf{L1 MR} = \frac{PAPI\_L1\_DCM}{PAPI\_LD\_INS} \tag{10}$$

$$\textbf{L2 MR} = \frac{PAPI\_L2\_TCM}{PAPI\_L1\_DCM} \tag{11}$$

$$\textbf{L3 MR} = \frac{PAPI\_L3\_TCM}{PAPI\_L2\_DCM} \tag{12}$$

$$\#\textbf{RAM Accesses} = PAPI\_L3\_TCM \tag{13}$$

$$\#\textbf{Bytes TransferredRAM} = \#RAMAccesses \times 64 \tag{14}$$

$$\#\textbf{FP Operations} = PAPI\_FP\_OPS \tag{15}$$

$$\textbf{OpIntensity} = \frac{\#FPOperations}{\#BytesTransferred_{RAM}} \tag{16}$$

$$\textbf{Performance} = \frac{\#FPOperations}{ExecutionTime \times 10^5} \tag{17}$$