Matrix-Matrix Multiplication Algorithm

Identification and Characterization of Performance Bottlenecks on a Computing Platform through Code Profiling

António Gonçalves
University of Minho
a85516@alunos.uminho.pt

Gonçalo Esteves
University of Minho
a85731@alunos.uminho.pt

Abstract

The main objective of this project is to study and analyze different implementations of a Matrix Multiplication algorithm, applying further optimizations and even modifying deeply the initial strategy, in order to take full advantage of all of the hardware's potential. We will be using the 662 nodes from SeARCH Cluster and even explore the NVidia Kepler accelerator.

I. Introduction

In order to be more effective in all kinds of applications, technology is constantly changing. Sometimes, these changes imply the appearance of new architectures, being developed or not for a specific use. However, the increasing variety of architectures also increases the variety of possible solutions to solve the same problem.

As we want to exemplify with this paper, it comes to the programmer to know exactly how to change the code in order to make it more effective in each architecture, acknowledging as well all the possible optimizations that can be used and which ones are the best for each case.

For that we will start by analyzing our team's laptop, as well as the node used for the rest of the paper. After that, we will introduce different implementations of the matrix multiplication algorithm, studying how modifications in the executions order and in the size of each matrix can affect the over-all performance. Finally we will compare the result obtained with and without some optimizations.

II. Hardware Characterization

The laptop used by our team is a Fujitsu Lifebook A556/G, with an Intel Core i5-6200U processor and 12 GiB of RAM (a 4GiB Kingston SODIMM DDR4 Sncrono 2133 MHz RAM and a 8GiB Samsung SODIMM DDR4 Sncrono 2133 MHz RAM). Despite the usage of two different RAMs, the memory latency isn't damaged, since the CAS Latency and the data rate of both RAMs are the same.[3][1]

Intel Core i5-6200U	
Architecture	Skylake
Performance	
# of Cores	2
# of Threads	4
Peak FP Performance	73.6 GFlops
Processor Frequency	2.30 GHz
Memory Specifications	
Level 1 cache size	64 KiB
Level 2 cache size	512 KiB
Level 3 cache size	3 MiB
RAM Memory Size	4 GiB (DDR4 2133 MHz) + 8 GiB (DDR4 2133 MHz)
RAM Latency	14.06 ns
# of Memory Channels	2
Max Memory Bandwidth	34.1 GiB
Memory Bus	64-bit
Advanced Technologies	
Instruction Set	64-bit
Instruction Set Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2

Table 1: *Team's laptop processor specifications*

A similar table can be found in the Appendix, with the specifications for the Cluster's 662 nodes proces-

sors, the Intel Xeon Processor E5-2695V2.[2][4][5].

In order to determine the peak floating point performance, we used the following equation:[6]

$$PeakFPP = \#CPU \times \#cores \times ClockFrequency \times FMA \times CPI \times SIMD \quad (1)$$

Since our team's laptop processor features 2 cores, the possibility of using FMA (which enables the usage of fused multiply-adds) and SIMD registers are 256-bits long, the peak FP performance is **73.6 GHz**, considering the CPI to be 1.

When calculating the peak FP performance of one of the Cluster's 662 Nodes, we have to consider three main differences. First, each node as 2 processors, so the number of CPU's is going to be 2. Second, each processor as 12 cores. Third, the processors don't have FMA support. Thanks to this, we can assume that the peak FP performance is going to be **460.8 GHz**

III. Roofline Model

The graphs represented by the Roofline Models relates arithmetic intensity with floating-point and memory performance. With this, we are able to obtain a better understanding of the hardware upper-bounds and how to make a program reach the computing platform's peak performance.

Unfortunately, we weren't able to determine the peak memory bandwidth, necessary for making a more precise Roofline model, so we considered the max memory bandwidth of the processors and the peak memory bandwidth to be the same. With this information, plus the peak FP performance values, we are able to determine the attainable GFlops:

$$AttainableGFlops = min(PeakFPP, PeakMemoryBW \times ArithmeticIntensity) \quad (2)$$

The obtained roofline model for both systems follows:

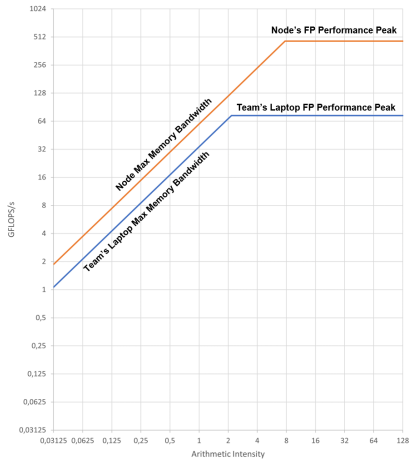

Figure 1: *Roofline Model for both systems*

i. Addition of Ceilings

The addition of ceilings is a good way of providing even more useful information in a Roofline model. Ceilings resemble the hardware's bottlenecks, being the optimizations relative to the lower ones the easier to implement by the programmer or the compiler.

With this in mind, we modified the Roofline model of our team's laptop, in order to add the ceilings.

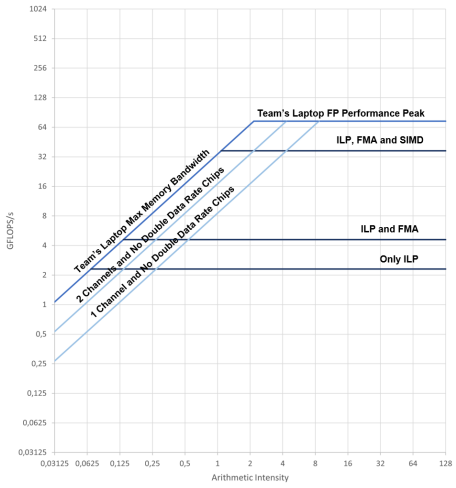

Figure 2: *Roofline Model with the Additional Ceilings*

i.1 Computational Ceilings

The highest computational ceiling resembles the usage of all the hardware's capabilities, so it's value is the same as the peak FP performance.

In the next ceiling, we considered the utilization of only one of the system's cores, so this ceiling is going to be placed 2 times lower then the first one.

Next, we discarded the usage of the AVX2 implementation and the SIMD instructions. Since our kernel is the multiplication of single precision floating point matrices, with SIMD we were able to execute 8 operations per cycle. With this in mind, this ceiling is placed 8 times lower than the previous one.

In the final ceiling, we considered the utilization of only Instruction Level Parallelism, discarding the usage of the FMA capability that the system provided. Since FMA allowed us to execute one addition and one multiplication in each clock cycle, we placed this ceiling 2 times lower than the previous one.

i.2 Memory Ceilings

Once again, the highest ceiling resembles the usage of all the hardware's capabilities, in this case, the memory's capabilities.

In the next ceiling, we discarded the optimizations provided by the double rate RAM chips used in the laptop. With this, the chip is capable of transferring data on the rising and the falling edges of the clock signal. Considering the absence of this optimization, this ceiling is placed 2 times lower than the first one.

The last ceiling resembles the non-existence of multiple memory channels. Since our team's laptop has 2 memory channels, the final ceiling is placed 2 times lower than the previous one.

IV. PAPI Counters

In order to have a better understanding of how the algorithm used works we will be using PAPI (version 5.5.0). This API has a large amount of counters used to store the number of times the program used an event.

For this work we used the following counters:

- **PAPI_L2_DCR** Level 2 Data Cache Reads
- **PAPI_L3_DCR** Level 3 Data Cache Reads
- **PAPI_L2_TCM** Level 2 Total Cache Misses
- **PAPI_L3_TCM** Level 3 Total Cache Misses
- **PAPI_LD_INS** Load Instructions executed
- **PAPI_TOT_INS** Total Instructions executed
- **PAPI_FP_OPS** Total of Floating Point Operations executed

Each of these counters will be usefull in order to analyze different aspects of the algorithm, such as miss-rates to different cache levels and the number of acesses to RAM.

V. Matrix Dot-Product Algorithm

Before explaining how we implemented this algorithm, it is important to understand what it does. The result matrix will be obtained by multiplying the other two ($C = A \times B$). For this, the three used matrices are square and have the same side size N.

i. First Implementations

That being said, there are two ways to read and calculate the values of the matrices: **Column-wise** and **Row-Wise**. As we will see, the performance of the algorithm will vary depending on how these accesses are made.

To multiply and store the result we will use three cycles (i,j,k), in which the two inner cycles represent the rows and the columns of the matrices (i and j, respectively).

With this, we started with 3 basic implementations, being them **IJK**, **IKJ** and **JKI**, which we will now analyze.

Firstly **IJK**, where each element of the result matrix is calculated using a line of the matrix A and a column of matrix B.

After that we have **IKJ**. For this implementation all accesses are made row-wise, and as we will see in the analyze, this will be the best out of all non-optimized options.

Finally we have **JKI**, where we will use one column of each of the matrices A and B.

Since it is expected that, each time the accesses are made column-wise, there is a lower performance, we implemented a optimized version of **IJK** and **JKI**, using transposed matrices. These two new implementations, that we will call **IJK Transposed** and **JKI Transposed**, will access all the information needed from both matrices row-wise.

ii. Size of the Matrices

In order to better analyze this problem, we used different sizes N for the three matrices. With this we wanted to fit all values needed to calculate C, as well as C itself, in the same cache level, in order to minimize the communication between levels.

After consulting the information about node 662[2], we know the precise size of each cache level. Since we have a total of 3 matrices, each one with N * N elements, being them floats, to store all the information needed we will need to have at least:

$$3 \times N \times N \times 4 \text{ bytes}$$

Which gives us:

1. L1 Cache: Size: 32KiB per core

   $N \times N \times 3 \times 4 < 32 \times 1024 \iff N < 52$

   We will consider N to be 32, because it's the greatest base power 2 size that guarantees that all matrices fit in the L1 cache.

2. L2 Cache: Size: 256KiB per core

   $32 \times 1024 < N \times N \times 3 \times 4 < 256 \times 1024 \iff 52 < N < 148$

   Using the same logic, our N will be 128.

3. L3 Cache: Size: 30720KiB

   $256 \times 1024 < N \times N \times 3 \times 4 < 30720 \times 1024 \iff 148 < N < 1619$

   Which gives us a N of 1024.

4. External RAM:

   $30720 \times 1024 < N \times N \times 3 \times 4 \iff 1619 < N$

   And finally a N of 2048.

Accessed Hierarchy Level	Size
L1 Cache	32 x 32
L2 Cache	128 x 128
L3 Cache	1024 x 1024
External RAM	2048 x 2048

Table 2: *Matrix sizes*

That being said, the size for each cache level are in table 2.

With later analyze of results we noticed that even using a size of 2048×2048 the number of accesses to the external RAM were not as high as expected. This happens because even though it is a higher number than what we can store without using the RAM, it is still not high enough to make the algorithm access it a significant number of times.

iii. Time Measurements

Firstly we measured the execution time for each of the implementations. These measurements were the result of getting the average of the 3 best times, with a 5% tolerance, from a total of 8 executions.

	32 x 32	**128 x 128**	**1024 x 1024**	**2048 x 2048**
IJK	0.090317	2.98923	11947.1	33342.9
IJK Transposed	0.099489	3.10544	1397.0	10190.4
IKJ	0.097937	3.29642	**1263.4**	**9917.2**
JKI	0.113218	3.64967	15770.6	74002.3
JKI Transposed	**0.089449**	**2.95239**	1387.9	10921.1

Table 3: *Execution Time for each Size/Implementation (ms)*

As expected, **JKI** has the worst result out of all implementations without transposed matrices. As we said before, this happens because all accesses that are made will be column-wise for all matrices. For the opposite reason, **IKJ** is the fastest, since accesses are all made row-wise.

Going more in detail for each size, we can see that there are almost no differences between implementations for smaller data sizes, such as 32×32, or 128×128. A possible explanation for this is that even though there are column wise accesses, since we used a hot cache, all matrices are stored in the L1 or L2 cache, not needing to acess the RAM. Even though

we can see that **JKI** is the slowest, **IKJ**, which is supposedly the fastest of all implementations, is just a bit slower than **IJK**.

As we get to larger data sizes, the differences in results get clearer. For a size of 1024×1024 we can see that, as expected, **IKJ** is the fastest, followed by the two transposed implementations, the **IJK** and finally, the slowest one, **JKI**. Even though the transposed implementations are accessing the values row wise, these have the added time of transposing the matrices, making them just a bit slower than **IKJ**.

Finally, for a size of 2048×2048, the conclusions are the same as stated above.

VI. Algorithm Analysis

In this section we will estimate and analyze the behavior of the implementations, focusing on the RAM accesses and total of *bytes* transferred, the number of floating point operations and the global miss rates of the different levels of cache.

i. Ram Accesses

Firstly, since our implementation does not clear the cache after generating each of the random matrices, the number of RAM accesses is lower than expected. This happens because after creating the matrices, some of their information will be stored in cache, reducing the total number of needed accesses to RAM, or even making it so that all the data is stored directly in the cache (something that we noticed for smaller data sizes).

Now, if we consider that the cache is cleared after generating the matrices, and assuming that all information is stored in RAM, since the node we use allows us to load 64 bytes each time the RAM is accessed, we will get a total of 16 float values from it, which will then be stored in cache. Because the algorithm implementation that we use is not always the same, we need estimate the number of accesses that each one of them will do:

1. **Column-wise accesses**: elements are accessed column by column. This way, every time we advance in the algorithm and need a new value we will need to access the RAM again. This makes it so that we need to access the RAM a total of $N \times N$ times, being N the side of the matrix.
2. **Row-wise accesses**: As stated before, we will be able to fetch a total of 16 floats for each RAM access, which will decrease the total number of accesses needed to complete the program. We will then need $N \times (N/16)$ accesses, since each time we accesses the RAM to get a value, the next 15 values from that line will also be loaded.

That being said, we can assume the following:

1. For the IJK implementation, we will read one matrix Column-wise, and the other Row-wise. So the total number or accesses to the RAM will be:

$$N \times (N + \frac{N}{16}) \tag{3}$$

2. For the JKI implementation, both matrices are read Column-wise, so the number of accesses to complete the program will be:

$$2 \times N \times N \tag{4}$$

3. Finally, for all the other versions, since both matrices will be read Row-wise, the number of accesses to the RAM will be:

$$2 \times N \times (\frac{N}{16}) \tag{5}$$

In order to get these values, we used some of the PAPI counters mentioned before, being them **PAPI_L3_TCM** and **PAPI_TOT_INS**. The reason between the two gave us the number of RAM accesses per instruction, and multiplying the first one for 64 (the number of bytes read in each access) gave us the total of bytes read.

As we said before, since we didn't clear the cache, the number of accesses to the RAM for smaller sizes is actually zero. This happens because the values are still kept in cache, even after the creation of the matrices. Even though these number are a bit off

of what we expected, we can still see that **JKI** has the highest number of *bytes* transferred, since when we read a value we do not use the remaining 15 that come in the same line. One possible explanation for the number of *bytes* transferred in the transposed implementations being a bit smaller than the **IKJ** is that when we transpose the matrices some information might stay in the lower cache levels.

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	0.000	0.000	0.00000007642861	0.00027002095715
IJK Transposed	0.000	0.000	0.00000002311285	0.00007055172240
IKJ	0.000	0.000	0.00000003708873	0.00006078009251
JKI	0.000	0.000	0.00000011177507	0.00062417191016
JKI Transposed	0.000	0.000	**0.00000002069876**	**0.00004570592803**

Table 4: *RAM accesses per instruction*

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	0.000	0.000	42048	1188000832
IJK Transposed	0.000	0.000	11136	271721600
IKJ	0.000	0.000	17856	233997504
JKI	0.000	0.000	76864	3432598784
JKI Transposed	0.000	0.000	**9984**	**176129216**

Table 5: *Bytes transferred to/from RAM*

ii. Floating Point Operations

Now, we will estimate the number of floating point operations for each implementation. In theory, it should be $N^3 \times 2$, since each inner cycle iteration has 2 operations (an addition and a multiplication), and there is no FMA support. In order to measure the exact number we used **PAPI_FP_OPS**, getting the following results:

	32 x 32	128 x 128	1024 x 1024	2048 x 2048
Theorical Number	65536	4194304	2147483648	17179869184
IJK	**65753**	**4367912**	13268233388	57065261934
IJK Transposed	72073	4734245	2412286890	19184128384
IKJ	66093	4370514	**2193490597**	**17455724925**
JKI	66210	4388513	22009104946	71719500065
JKI Transposed	67016	4399664	2204069977	17525048340

Table 6: *Number of floating point operations executed*

As we can see, the number of FPs is really close to the expected for sizes of 32 and 128 for all algorithms. As we go to bigger sizes, **IKJ** and **IJK**/**JKI Transposed** still have normal values, but that does not apply to normal **IJK** and **JKI**. We think this might happen because the counter might be assuming that the memory attributions done to store floats in cache are floating point operations, which should not be true.

iii. Plotting the results in the Roofline Model

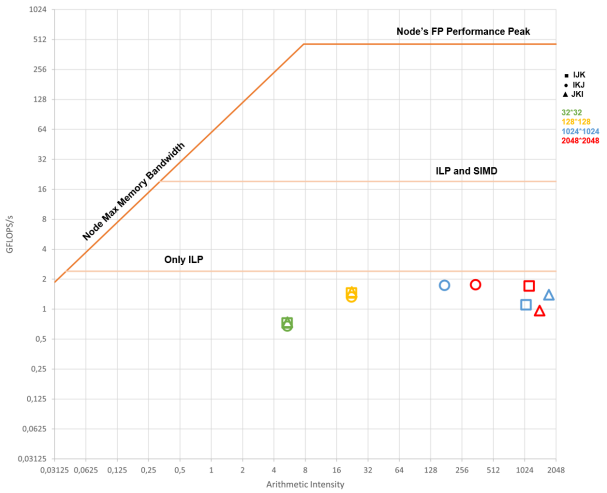

Figure 3: *Plotting of the obtained values.*

In this case study, it is correct to assume that the problem presented to us is mostly a compute bounded problem. We can conclude this by observing the roofline model above, and verifying that the plotted values are in the right side of the ridge point.[6]

iv. Cache Behaviour

Finally, we will analyze how the different algorithms behave when they need to fetch the information to the different levels of cache, especially for the transposed implementations. For this we will use some PAPI counters, in order to calculate the exact number of misses for each level. We used the following formulas:

$$L1MissRate = \frac{PAPI\_L2\_DCR}{PAPI\_LD\_INS} \times 100$$
$$L2MissRate = \frac{PAPI\_L3\_DCR}{PAPI\_L2\_DCR} \times 100$$
$$L3MissRate = \frac{PAPI\_L3\_TCM}{PAPI\_L2\_TCM} \times 100$$

		32 x 32	128 x 128	1024 x 1024	2048 x 2048
IJK	L1 MR(%)	0.551	2.360	50.685	44.547
	L2 MR(%)	48.663	4.008	4.764	12.232
	L3 MR(%)	0.000	0.000	0.001	1.322
IJK Transposed	L1 MR(%)	0.395	4.185	3.244	3.195
	L2 MR(%)	51.927	3.558	8.663	6.006
	L3 MR(%)	0.000	0.000	0.003	13.144
JKI	L1 MR(%)	0.336	2.191	52.828	50.172
	L2 MR(%)	52.491	3.469	10.779	61.637
	L3 MR(%)	0.000	0.000	0.166	0.404
JKI Transposed	L1 MR(%)	0.517	2.865	2.209	2.161
	L2 MR(%)	48.909	8.663	8.754	6.005
	L3 MR(%)	0.000	0.000	0.003	8.455

Table 7: *Miss rates for each cache level*

The use of transposed matrices in the **IJK** and **JKI** implementations proved to have a positive impact on both cases. Transposing all the matrices that were accessed column-wise made possible a row-wise access to them. The most obvious advantage of this was the reduction of the execution times, making them be similar to the ones obtained by the **IKJ** implementation. The cause to this was the fact that reading the matrices row-wise made possible to store more usefull information in cache. Thanks to this, the cache misses, especially for the L1 cache, decreased a lot, making so that there weren't needed as many acesses to the RAM, decreasing the total of bytes transferred to/from the RAM, and subsequently the time needed to read all values. Also, it is important to notice that the number of floating point operations has decreased too. All of this made so that the negative impact of generating the transpose matrices would be less significant when compared to the benefict that the use of transpose matrices brings.

VII. Optimisations

Now that we finished analyzing the normal implementations we can start using new optimisations in order to have a even better performance. We will be covering the use of blocking, vectorization, a implementation based on multi-core devices and finally a CUDA implementation.

i. Blocking

The block optimization is efficient thanks to the locality optimization it grants. The principle of locality it's a property which says that when a process acesses a set of memory locations, it's probable that it will access that same set again in a short period of time.

The dot-product is a problem that fits well with this idea, since we will acess the same rows and columns of the matrices multiple times. When well used, the block optimization can guarantee that the rows and columns that we need repeatedly will stay on the cache, making the acess to the information quicker.

In this particular case, and as we previously said, the 662 nodes can read from RAM 16 floats at a time. With this, we can assume that we must work with blocks of size 16, in order to take the most possible advantage of all the values that we read from RAM at a time.

	No Blocking	**Blocking**
IJK Transposed	10190.4	6620.77
IKJ	9917.2	5902.36
JKI Transposed	10921.1	5844.08

Table 8: *Execution times (ms) of the optimized implementations, for a size of 2048* $\times$2048.

ii. Vectorization

In order to vectorize the code, we used the *#pragma ivdep* clause in the inner loop of the 3 implementations that used blocking. With this, we compel the compiler to ignore all the assumed vector dependencies, since it is safe ignore those assumed dependencies.

	Not Vectorized	**Vectorized**
IJK Transposed	0.099489	0.0226498
IKJ	0.097937	0.0178814
JKI Transposed	0.089449	0.0219345

Table 9: *Execution times(ms) for a size of* 32×32

	Not Vectorized	**Vectorized**
IJK Transposed	3.10544	1.096970
IKJ	3.29642	0.787973
JKI Transposed	2.95239	0.851075

Table 10: *Execution times(ms) for a size of* 128×128

iii. Multi-Core

The multi-core devices of the 662 node has 24 cores each. With this information, we can assume that in an **OpenMP** implementation, the best way to assure a good workload distribution is by using 24 threads, each one resembling one of the node's cores. Despite this, we made tests for 2, 4, 8, 16 and 24 threads, in order to guarantee that. Also, by using the *#pragma omp simd* clause in the inner loop of all implementations, we can make it so that multiple iterations of the loop can be executed concurrently using SIMD instructions.

2048×2048			
	Non Blocking	**Blocking**	**OpenMP**
IJK Transposed	10190.4	6620.77	278.510
IKJ	**9917.2**	5902.36	**247.107**
JKI Transposed	10921.1	**5844.08**	321.126

Table 11: *Execution time (ms) using **OpenMP** compared with other implementations*

iv. **CUDA** Implementation

Lastly, we modified our implementation so that it could run effectively in all SMX of a GPU Kepler. In order to achieve that, we made so that each thread would compute one of the values of the result matrix, using a blocking optimization, making the implementation the more efficient possible.

The tests were done using one of the 662 Nodes that has a NVidia Kepler accelerator.

	Data Transfers		**Computation**
	Host to Accelerator	Accelerator to Host	
IJK Transposed	14.2164	10.4772	85.0882

Table 12: *Execution time(ms) for a size of* 2048×2048

VIII. Conclusions

We believe that with this paper we showed that it is important to take into consideration the hardware and software capabilities in order to achieve the best performance possible.

Starting with an analyzes of the algorithm and all the different implementations, and following with some of the possible optimizations, we managed to compare the results using PAPI to count the number of events in order to analyze even better what was happening with the algorithm.

When the objective is to create a program that is truly efficient, we should take into consideration all factors, not only the ones related to software, but also all of the hardware's features.

We should make our programs run smoothly on the hardware they are designed to, taking advantage of the best hardware capabilities for each specific case.

References

[1] CPU-World. *Intel Core i5-6200U Processor*. URL: https://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5-6200U%20Mobile%20processor.html.

[2] CPU-World. *Intel® Xeon® Processor E5-2695 v2*. URL: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2670%20v2.html.

[3] Intel. *Intel® Core™ i5-6200U Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/88193/intel-core-i5-6200u-processor-3m-cache-up-to-2-80-ghz.html.

[4] Intel. *Intel® Xeon® Processor E5-2695 v2*. URL: https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e5-2695-v2-30m-cache-2-40-ghz.html.

[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: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures". In: (2008).

Additional references:

1. https://www.openmp.org/spec-html/5.0/openmpsu42.html
2. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html
3. https://en.wikipedia.org/wiki/Loop_nest_optimization
4. https://www.techpowerup.com/gpu-specs/tesla-k20m.c2029

A. Appendix

i. Hardware Specifications

Intel Core i5-6200U	
Architecture	Skylake
Performance	
# of Cores	2
# of Threads	4
Peak FP Performance	73.6 GFlops
Processor Frequency	2.30 GHz
Memory Specifications	
Level 1 cache size	64 KiB
Level 2 cache size	512 KiB
Level 3 cache size	3 MiB
RAM Memory Size	4 GiB (DDR4 2133 MHz) + 8 GiB (DDR4 2133 MHz)
RAM Latency	14.06 ns
# of Memory Channels	2
Max Memory Bandwidth	34.1 GiB
Memory Bus	64-bit
Advanced Technologies	
Instruction Set	64-bit
Instruction Set Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2

Table 13: *Team's laptop processor specifications*

Intel Xeon Processor E5-2695V2	
Architecture	Ivy Bridge EP
Performance	
# of Cores	12
# of Threads	24
Peak FP Performance	460.8 GFlops
Processor Frequency	2.40 GHz
Memory Specifications	
Level 1 cache size	64 KiB
Level 2 cache size	256 KiB
Level 3 cache size	30 MiB
Max RAM Memory Size	64 GiB
Max # of Memory Channels	4
Max Memory Bandwidth	59.7 GiB
Memory Bus	64-bit
Advanced Technologies	
Instruction Set	64-bit
Instruction Set Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2

Table 14: *Cluster's 662 Nodes processors specifications*

ii. Compilation Flags

Regular Compilation:
-O2 -L/share/apps/papi/5.5.0/lib -I/share/apps/papi/5.5.0/include -Wall -Wextra -fopenmp -Wno-unused-parameter

Vectorization: -O3 -Wall -Wextra -qopenmp -Wno-unused-parameter -qopt-report=3 -qopt-report-phase=vec