Advanced Architectures Assignment

Ricardo Neves A78764

21 January 2019

Abstract - The main objective for this practical assignment was to develop the overall knowledge of the content learned in the Curricular Unit of Advanced Architectures. Although, the main focus of this work was on study the performance bottlenecks on a computing platform and / or on the code profiling and analyze the performance of the algorithms.

Contents

1 Introduction

In this document, the group decided to use two different machines: my personal laptop and a Cluster SeARCH's 662 node, in Universidade do Minho.

In addiction, I will show, compare and explain all the results I got from the tests in both of this computing platforms.

In the end of this assignment, there is an appendix with some relevant information, important to complete the content of some points of this project.

Regarding PAPI, despite several attempts, I could not get results for the counters tested on my personal PC.

Since the SeARCH Cluster was not available for the tests, I will not be able to compare the estimated values with the actual values.

2 Team Main Laptop

Here is the full characterization of the personal laptop used for the assignment:

Manufacturer - ASUS
Model - K550J
CPU - Intel i7-4720HQ
Main memory size - 7863MB
Number of cores - 4 cores (8 threads)
Base frequency - 2.6 GHz
Peak FP Performance - 83.2 GFlops/s
Cache - L1d:32KB, L1i:32KB, L2:256KB, L3:6MB
Memory access bandwidth - 25.6 GB/s

This data was obtained from the Intel's official website and *cpu-world.com*, a very complete database of CPUs. All this information was confirmed by the Unix command "lscpu", that displays information about the CPU architecture in the terminal.

3 Cluster SeARCH's 662 node

Here is the characterization of the Cluster SeARCH's 662 node:

CPU - Intel E5-2695v2
Number of cores - 12 cores (24 threads)
Base frequency - 2.4 GHz
Peak FP Performance - 230.4 GFlops/s
Cache - L1d:32KB, L1i:32KB, L2:256KB, L3:30MB
Memory access bandwidth - 59.7 GB/s

This data was obtained from *search6.di.uminho.pt/wordpress/?page_id=55* and *cpu-world.com.*

4 Roofline Model

As I already said before, I will compare the results from two different machines: the personal laptop and a Cluster node. Next, I built a performance evaluation model called **Roofline Model**, as it provides insights on both the implementation and inherent performance limitations.

Peak Performance = Max Clock Frequency * FMA * SIMD * CPU.

Maximum Memory Bandwidth = Peak Performance / Operational Intensity.
Peak Performance = Maximum Memory Bandwidth * Operational Intensity.

Peak Performance of team main laptop = 2.6GHz * 2 * 2 * 8 threads.
Peak Performance of Cluster 662 node = 2.4GHz * 2 * 2 * 24 threads.

With this results, the group built the next graph, that show the performance of the two used machines:

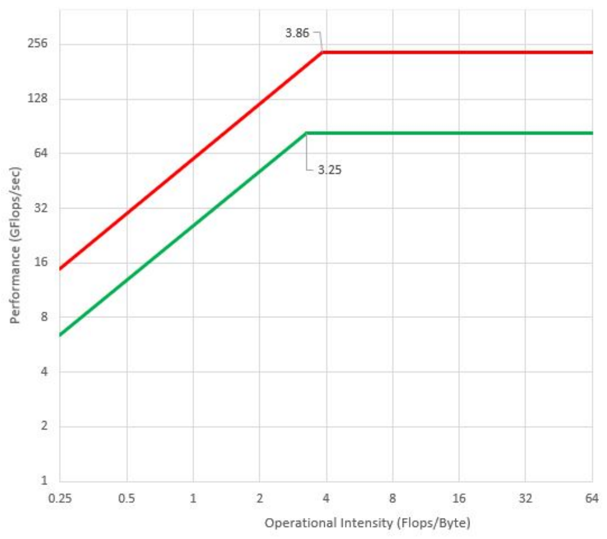


Figure 1: Roofline Model

Red line - 662 node of cluster SeARCH.
Green line - Personal laptop.

The Roofline Model of the 662 node of the Cluster, with the ceilings the group decided that were the most relevant can be found in the appendix.

5 PAPI Performance Counters

In computer science, Performance Application Programming Interface (PAPI) is a portable interface (in the form of a library) to hardware performance counters on modern microprocessors. It is being widely used to collect low level performance metrics (e.g. instruction counts, clock cycles, cache misses) of computer systems running UNIX/Linux operating systems, useful to measure a code's performance and quality.

In the first place, we loaded the papi 5.5.0 module to the 662 node. Then, we used the command "*papi_avail*" to get the name and information of all the counters available. From all the counters, we chose the ones who were more relevant for the assignment.

Name - Description
PAPI_L1_DCM - Level 1 data cache misses
PAPI_L2_DCM - Level 2 data cache misses
PAPI_L1_ICM - Level 1 instructions cache misses
PAPI_L2_ICM - Level 2 instructions cache misses
PAPI_L1_TCM - Level 1 cache misses
PAPI_L2_TCM - Level 2 cache misses
PAPI_L3_TCM - Level 3 cache misses
PAPI_L1_TCA - Level 1 total cache accesses
PAPI_L2_TCA - Level 2 total cache accesses
PAPI_L3_TCA - Level 3 total cache accesses
PAPI_TOT_INS - Instructions completed
PAPI_LD_INS - Load instructions
PAPI_SR_INS - Store instructions
PAPI_FP_INS - Floating point instructions
PAPI_FP_OPS - Floating point operations
PAPI_LD_INS - Total of DRAM load instructions
PAPI_SR_INS - Total of DRAM store instructions
PAPI_TOT_CYC - Total cycles

6 i-j-k Matrix Multiplication

In the first place, I wrote a single-threaded C function that computes the product of 2 squared matrices (with N rows and N columns), in single precision and with no block optimization.

My A matrix was filled with random integers between 0 and 20, while our B matrix is only completed with the number 1.

With that being said, my first program consists of three ”for” loops. The first runs all the lines of the matrix A, the second one runs the columns of the matrix B, while the last ”for” is responsible for the multiplication of the right values of both matrices, saving the result in the right matrix C spot.

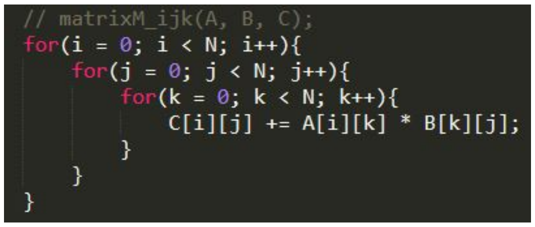

```
// matrixM_ijk(A, B, C);
for(i = 0; i < N; i++){
    for(j = 0; j < N; j++){
        for(k = 0; k < N; k++){
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Figure 2: i-j-k matrix multiplication code

In this algorithm shown above, we can see that there are four matrix accesses per inner loop. One access to matrix A, another access to matrix B and two accesses to matrix C (one to get information, another one to save it).

However, we can see that, while the matrices A and C are accessed by rows, the matrix B is accessed by columns, which turns the lookups more expensive, because the information is not saved in contiguous memory.

7 i-k-j Matrix Multiplication

To answer this problem, the group coded the following:

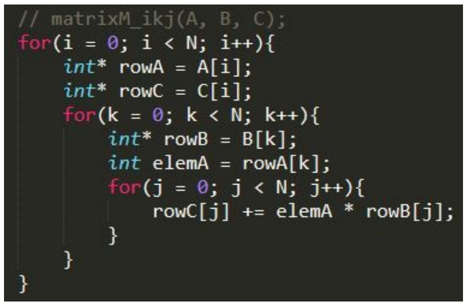

```
// matrixM_ikj(A, B, C);
for(i = 0; i < N; i++){
    int* rowA = A[i];
    int* rowC = C[i];
    for(k = 0; k < N; k++){
        int* rowB = B[k];
        int elemA = rowA[k];
        for(j = 0; j < N; j++){
            rowC[j] += elemA * rowB[j];
        }
    }
}
```

Figure 3: i-k-j matrix multiplication code

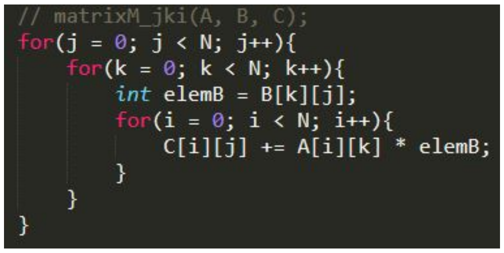
At the beginning, the matrix A row that will be used is saved in the array ”rowA”, as well as the row of matrix C, where we are going to save the result.

In the next cycle, we saved the row of the matrix B that will be used for the product. We also get to know what element of the array ”rowA” is going to be used.

In the inner loop, we don't need to access the matrices because everything is already saved in the proper variables (the data accessed is contiguous in the memory). This saves a very good amount of time, as we will see next, when the execution time measurements show.

8 j-k-i Matrix Multiplication

Next, the group coded the j-k-i matrix multiplication, without any transposed matrix:



```
// matrixM_jki(A, B, C);
for(j = 0; j < N; j++){
    for(k = 0; k < N; k++){
        int elemB = B[k][j];
        for(i = 0; i < N; i++){
            C[i][j] += A[i][k] * elemB;
        }
    }
}
```

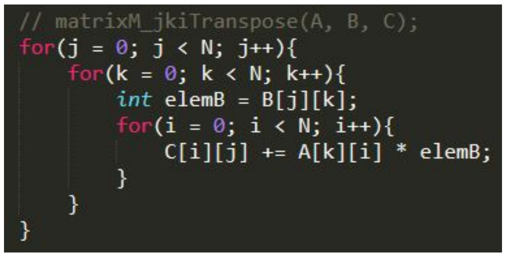
Figure 4: j-k-i matrix multiplication code

In this algorithm, we save, at the beginning, the value of the matrix B and multiply this value by the matrix A, saving the given result.

However, as we are going to see next, this implementation is not very good because the information is accessed by columns, which is not what we're looking for.

9 j-k-i Matrix Multiplication with Transposed Matrices

Now, the group decided to use the transposed matrices A and B:



```
// matrixM_jkiTranspose(A, B, C);
for(j = 0; j < N; j++){
    for(k = 0; k < N; k++){
        int elemB = B[j][k];
        for(i = 0; i < N; i++){
            C[i][j] += A[k][i] * elemB;
        }
    }
}
```

Figure 5: j-k-i matrix multiplication with transposed matrices code

Here, the matrices used were the transpose of A and the transpose of B. In the second loop, we save the element of the matrix B that will be used for the product and then, in the next "for", we multiply this by all the right elements of the matrix A.

Now, the values are accessed by rows, and the performance difference between this two last algorithms will be shown next.

10 Size of Matrices

Each square matrix is filled with a certain number of Ints. One Int has a storage size of 4 bytes.

Now, with this information, I am going to calculate the ideal size of the 3 matrices used in the algorithm, using only one core of the 662 node of the Cluster (by mistake, the KiB metrics were used instead of KB):

10.1 L1 Cache

The L1 cache of the used node has a size of 32KiB, as I already mentioned in the beginning of the document.

Now, this means we can have 32 000B / 4B = 8000 Ints divided for the 3 matrices. This means each matrix will be composed by 8 000 / 3 = 2666.66 Ints. If we calculate the square root of this number, we will obtain the size of the 3 matrices. sqrt(2666.66) = 51.64 Ints. Of course, the size must be an Int number, so we will say that each matrix will have the ideal size of 51 Ints (51 * 51 = 2 601 elements).

After all this calculations, I took in consideration the fact that the size of a cache line is 64B. So, with this 64B, we can work with 64B / 4B = 16 Ints. This leaves us with the conclusion that the number of Ints in a matrix should be a multiple of 16, which 2601 clearly is not one (2601 / 16 = 162.56).

So, we need to find the closest number to 51 (and below) that is a multiple of 16. The number we are looking for is 48. As we can see, **48 * 48** = 2304 Ints per matrix, which is a multiple of 16.

10.2 L2 Cache

The L2 cache has a size of 256KiB. Doing all the calculations we did for the L1 cache:

256 000B / 4B = 64 000 Ints.
64 000 / 3 = 21 333.33 Ints.
sqrt(21 333.33) = 146 Ints.

The maximum size of a matrix is 146 * 146 Ints.
The ideal size is **144 * 144** Ints.

10.3 L3 Cache

The L3 cache of the 662 node has a size of 30720KiB. Again, we redo all the calculations, with the correct values:

30 720 000B / 4B = 7 680 000 Ints.
7680000 / 3 = 2560000 Ints.
sqrt(256 0000) = 1600 Ints.

The maximum (and ideal size, because 1600 is multiple of 16) is **1600 * 1600** Ints, for the 662 node of the Cluster SeARCH.

The L3 cache of my laptop has a size of 30720KiB. Again, we redo all the calculations, with the correct values:

6 144 000B / 4B = 1 536 000 Ints.
1 536 000 / 3 = 512 000 Ints.
sqrt(512 000) = 715 Ints.

The maximum size of a matrix for the laptop is 715*715 Ints.
The ideal size is **704*704** Ints.

10.4 External RAM

If the size of the matrix doesn't fit in any of the caches mentioned above, the CPU need to load it from the external RAM.

With that being said, we just need to find a size greater than 1600 * 1600, that is also multiple of 16. For example, the matrix with size **2000 * 2000** Ints fits all the requirements mentioned.

11 Execution Time Measurement

11.1 Team Laptop

For each function coded and input size, the group did three time measurements in the main laptop. The results obtained are available in the appendix.

With the data from the tables, I built the following graph, that shows the relation between the input size and the execution time of the 3 algorithms.

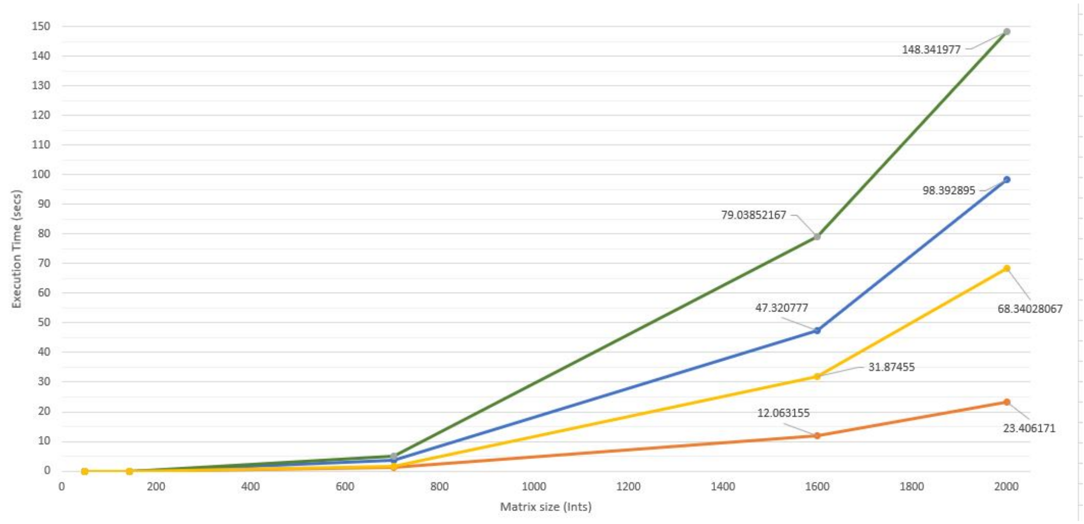


Figure 6: Execution time graph in the team laptop

Blue line - i-j-k function.
Red line - i-k-j function.
Green line - j-k-i function.
Yellow line - j-k-i with transposed matrices function.

11.2 Cluster SeARCH's 662 node

For each function coded and input size, the group did three time measurements in the 662 node of the Cluster. The tables with the results can also be seen in the appendix.

With the data from the tables down below, I built the following graph, that shows the relation between the input size and the execution time of the 3 algorithms.

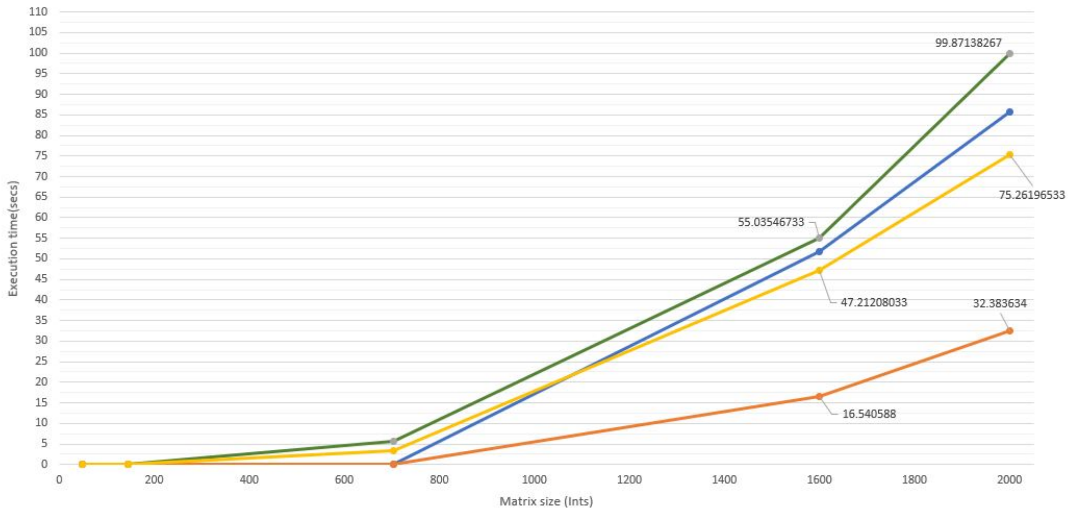


Figure 7: Execution time graph in 662 node

Blue line - i-j-k function.
Red line - i-k-j function.
Green line - j-k-i function.
Yellow line - j-k-i with transposed matrices function.

As we can see in the results, the ikj implementation was the faster one.

We also prove that the jki implementation is faster if it is using the transposed matrices, as both of the matrices are accessed by rows in this case. The jki implementation is the one that takes the most time to complete the process, due to the access being executed by columns.

For smaller matrices (48*48 and 144*144) the execution time for every implementation is almost the same, where the result is given almost instantly.

12 RAM accesses per instruction and bytes to/from RAM

The next step is to estimate the number of RAM accesses per instruction and the number of bytes transferred to/from the RAM.

Next to the estimation of this number, I am going to confirm the results with PAPI readings.

For the first data-set, with size 48*48, there will be 48*48*48 = 110592 instructions, because of the 3 loops in the code. Per instruction, there will be 4 accesses to the memory, one for each of the matrices A and B, and two accesses for the matrix C. However, one of this accesses will be a store, when we save the new result of C[i][j].

By this means, there will be 110 592 stores and 110 592*3 = 331 776 loads.

In addition, we can see that, for each inner iteration, we have one load and two FP operations (the product and the sum), so will have a ratio of 0.5 RAM accesses/instruction. This applies to all the algorithms, and all the data sets used.

Remember that the cache line size is 64B, so 64*8*110 592 = 56 623 104 bytes is the estimated value for the transfer to/from the RAM.

Data-set	**Estimated**	**Real Value**	**Estimated**	**Real Value**
48	0.5	-	56 623 104	-
144	0.5	-	1 528 823 808	-
704	0.5	-	178 643 795 968	-
1600	0.5	-	2 097 152 000 000	-
2000	0.5	-	4 096 000 000 000	-

13 FP operations executed

Per inner loop, there is two FP operations: the product of the value of A and the value of B, and the sum of this result with the last value of C.

With this, we can say that the total FP operations is: FP operations per iteration * # of iterations.

Data-set	**Estimated**	**Real value**
48	221 184	–
144	5 971 968	–
704	697 827 328	–
1600	8 192 000 000	–
2000	16 000 000 000	–

14 Miss Rate in j-k-i Algorithm

To get the Miss Rates, it is necessary to calculate the division between the number of level cache misses and the number of total number of level cache accesses.

However, as I already said before in the Introduction, I could not get PAPI to get the right values of the counters.

So, the Miss Rates were not calculated.

15 Block Optimization

This algorithm is responsible for dividing the matrices in smaller blocks of the same size.

After all the calculations in the variable "temp", the results are combined into the final matrix C.

I decided to run some tests on this implementation, in order to get the execution time of those. Here is the average of the results taken.

From a Block Size of 8:
704*704 data-set - 1.10 secs;
1600*1600 data-set - 12.8 secs;
2000*2000 data-set - 24.9 secs.

From a Block Size of 16:
704*704 data-set - 1.02 secs;
1600*1600 data-set - 11.6 secs;
2000*2000 data-set - 22.8 secs.

16 Vectorization Code

Here, we turned the variable "temp" into a vector with 8 positions, due to the chosen block size being also 8.

Compiling with the command *gcc -O2 -ftree-vectorize -fopt-info-vec matrixM.c*, the compiler was able to vectorize only the last of the nested loops.

17 Parallelization with OpenMP

Here, with the purpose of running the code with the less time possible, I used the *pragma omp parallel* to share the matrices through the various threads of the processor.

In order to do this, I set the "for" variables private and gave the information of how many threads would I be using.

Next, we see the *Speed-Up* graph of this implementation. I used the bars graph because I thought it would be the best way to show and compare all the Speed-Ups and Data-Sets available:

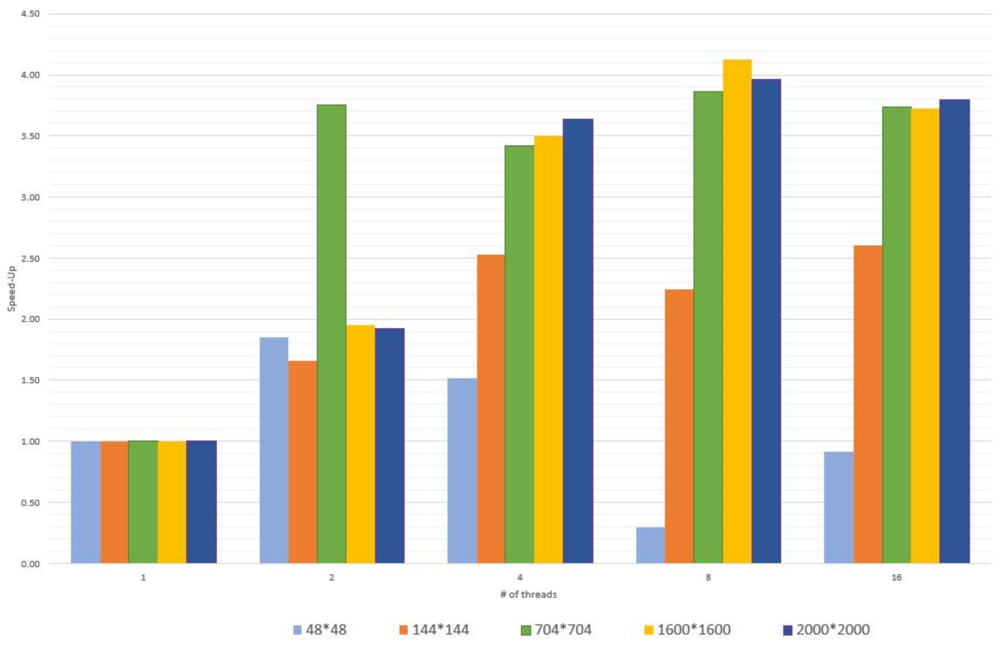


Figure 8: Speed-Up for the parallel code

We recognize, from the graph above, that 48*48 and 144*144 data-sets don't scale because the matrices are too small to get full advantage from the OpenMP implementation.

However, we see too that the other greater data-sets scale until the 8 threads. After this, the execution time starts to increase, which is not good.

So, to take full advantage of the OpenMP, 8 threads should be used because it has the lower execution time and, of course, the better performance.

18 Conclusions

In order to make a final analysis of all the results obtained so far, I can affirm that the best computational performance was obtained in the parallel algorithm of ijk, with 8 threads. This result can be obtained by fixing a previously chosen data set, and comparing all the algorithms that iterated over it. Therefore, for this work of multiplication of matrices (bigger or smaller), the group calls the use of the version parallelized with OpenMP of 8 threads.

In conclusion, I have addressed various aspects requested in the statement given.

However, it failed to analyse some points requested by the professor, as well as trying to improve the ones that weren't explained or studied so well.

This practical assignment helped me to comprehend much more and with better detail about computing platforms' performance, cache, the PAPI tool and other studied contents.

19 Appendix

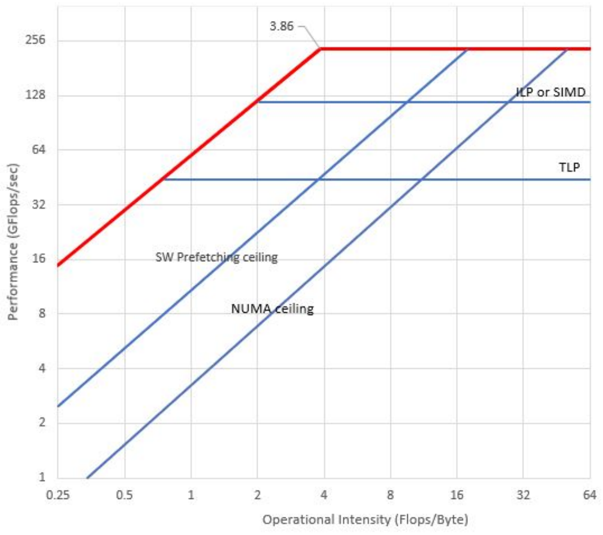
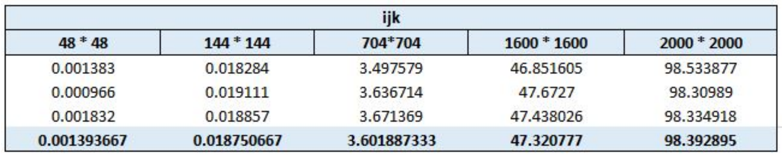
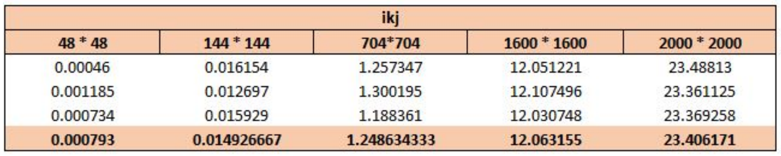


Figure 9: Roofline Model of 662 node with ceilings



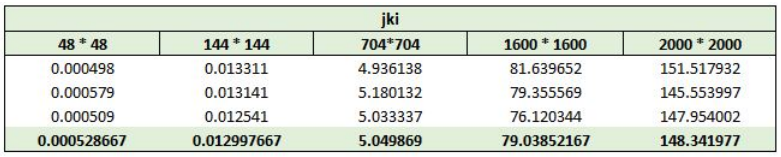
ijk				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.001383	0.018284	3.497579	46.851605	98.533877
0.000966	0.019111	3.636714	47.6727	98.30989
0.001832	0.018857	3.671369	47.438026	98.334918
0.001393667	**0.018750667**	**3.601887333**	**47.320777**	**98.392895**

Figure 10: Execution times for i-j-k function in the team laptop



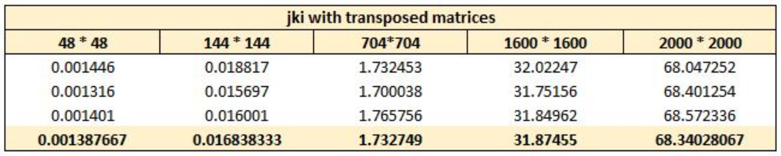
ikj				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.00046	0.016154	1.257347	12.051221	23.48813
0.001185	0.012697	1.300195	12.107496	23.361125
0.000734	0.015929	1.188361	12.030748	23.369258
0.000793	**0.014926667**	**1.248634333**	**12.063155**	**23.406171**

Figure 11: Execution times for i-k-j function in the team laptop



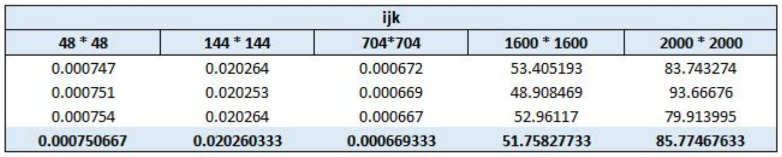
jki				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.000498	0.013311	4.936138	81.639652	151.517932
0.000579	0.013141	5.180132	79.355569	145.553997
0.000509	0.012541	5.033337	76.120344	147.954002
0.000528667	**0.012997667**	**5.049869**	**79.03852167**	**148.341977**

Figure 12: Execution times for j-k-i function in the team laptop



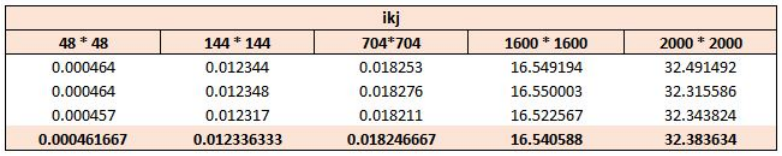
jki with transposed matrices				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.001446	0.018817	1.732453	32.02247	68.047252
0.001316	0.015697	1.700038	31.75156	68.401254
0.001401	0.016001	1.765756	31.84962	68.572336
0.001387667	**0.016838333**	**1.732749**	**31.87455**	**68.34028067**

Figure 13: Execution times for j-k-i with transposed matrices function in the team laptop



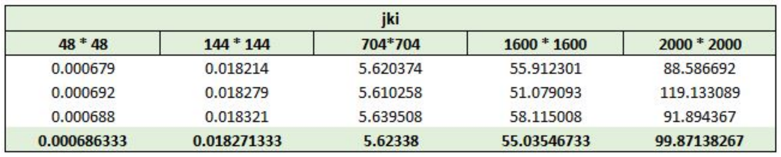
ijk				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.000747	0.020264	0.000672	53.405193	83.743274
0.000751	0.020253	0.000669	48.908469	93.66676
0.000754	0.020264	0.000667	52.96117	79.913995
0.000750667	**0.020260333**	**0.000669333**	**51.75827733**	**85.77467633**

Figure 14: Execution times for i-j-k function in 662 node



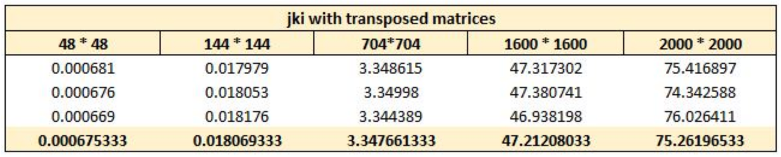
ikj				
48 * 48	**144 * 144**	**704*704**	**1600 * 1600**	**2000 * 2000**
0.000464	0.012344	0.018253	16.549194	32.491492
0.000464	0.012348	0.018276	16.550003	32.315586
0.000457	0.012317	0.018211	16.522567	32.343824
0.000461667	**0.012336333**	**0.018246667**	**16.540588**	**32.383634**

Figure 15: Execution times for i-k-j function in 662 node



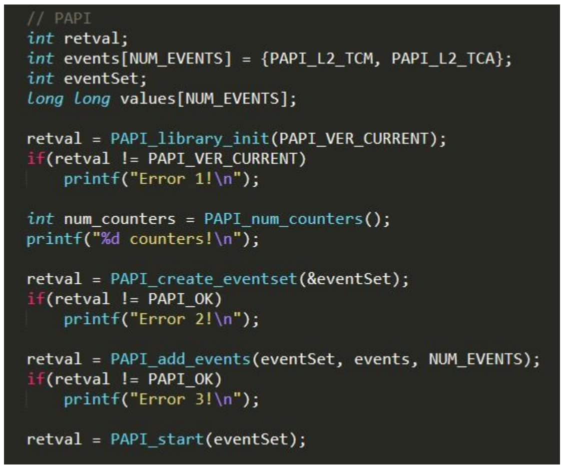
jki				
48 * 48	144 * 144	704*704	1600 * 1600	2000 * 2000
0.000679	0.018214	5.620374	55.912301	88.586692
0.000692	0.018279	5.610258	51.079093	119.133089
0.000688	0.018321	5.639508	58.115008	91.894367
0.000686333	**0.018271333**	**5.62338**	**55.03546733**	**99.87138267**

Figure 16: Execution times for j-k-i function in 662 node



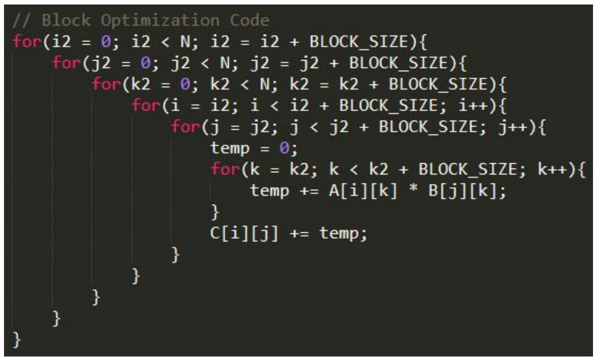
jki with transposed matrices				
48 * 48	144 * 144	704*704	1600 * 1600	2000 * 2000
0.000681	0.017979	3.348615	47.317302	75.416897
0.000676	0.018053	3.34998	47.380741	74.342588
0.000669	0.018176	3.344389	46.938198	76.026411
0.000675333	**0.018069333**	**3.347661333**	**47.21208033**	**75.26196533**

Figure 17: Execution times for j-k-i with transposed matrices function in 662 node



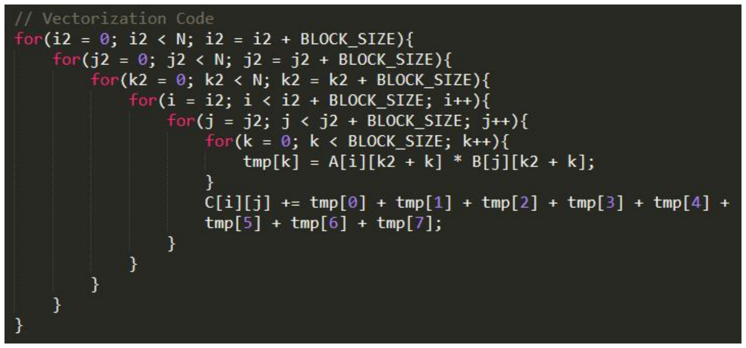
```
// PAPI
int retval;
int events[NUM_EVENTS] = {PAPI_L2_TCM, PAPI_L2_TCA};
int eventSet;
long long values[NUM_EVENTS];

retval = PAPI_library_init(PAPI_VER_CURRENT);
if(retval != PAPI_VER_CURRENT)
    printf("Error 1!\n");

int num_counters = PAPI_num_counters();
printf("%d counters!\n");

retval = PAPI_create_eventset(&eventSet);
if(retval != PAPI_OK)
    printf("Error 2!\n");

retval = PAPI_add_events(eventSet, events, NUM_EVENTS);
if(retval != PAPI_OK)
    printf("Error 3!\n");

retval = PAPI_start(eventSet);
```

Figure 18: PAPI implementation



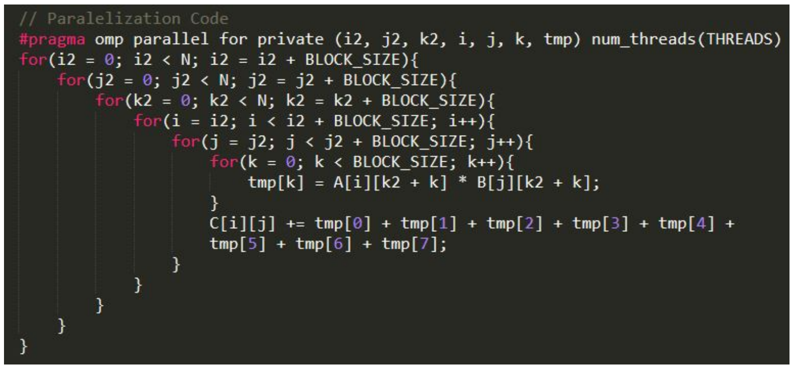
```
// Block Optimization Code
for(i2 = 0; i2 < N; i2 = i2 + BLOCK_SIZE){
    for(j2 = 0; j2 < N; j2 = j2 + BLOCK_SIZE){
        for(k2 = 0; k2 < N; k2 = k2 + BLOCK_SIZE){
            for(i = i2; i < i2 + BLOCK_SIZE; i++){
                for(j = j2; j < j2 + BLOCK_SIZE; j++){
                    temp = 0;
                    for(k = k2; k < k2 + BLOCK_SIZE; k++){
                        temp += A[i][k] * B[j][k];
                    }
                    C[i][j] += temp;
                }
            }
        }
    }
}
```

Figure 19: Block optimization code



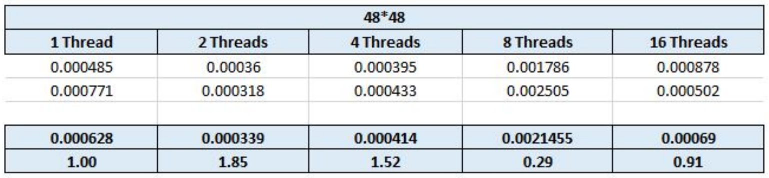
```
// Vectorization Code
for(i2 = 0; i2 < N; i2 = i2 + BLOCK_SIZE){
    for(j2 = 0; j2 < N; j2 = j2 + BLOCK_SIZE){
        for(k2 = 0; k2 < N; k2 = k2 + BLOCK_SIZE){
            for(i = i2; i < i2 + BLOCK_SIZE; i++){
                for(j = j2; j < j2 + BLOCK_SIZE; j++){
                    for(k = 0; k < BLOCK_SIZE; k++){
                        tmp[k] = A[i][k2 + k] * B[j][k2 + k];
                    }
                    C[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3] + tmp[4] +
                    tmp[5] + tmp[6] + tmp[7];
                }
            }
        }
    }
}
```

Figure 20: Vectorization code



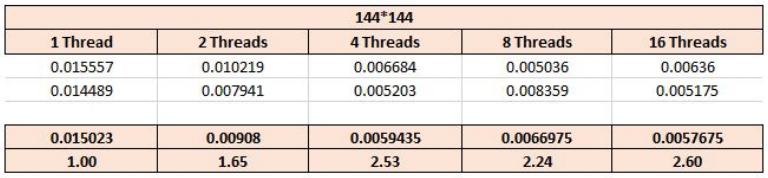
```
// Paralelization Code
#pragma omp parallel for private (i2, j2, k2, i, j, k, tmp) num_threads(THREADS)
for(i2 = 0; i2 < N; i2 = i2 + BLOCK_SIZE){
    for(j2 = 0; j2 < N; j2 = j2 + BLOCK_SIZE){
        for(k2 = 0; k2 < N; k2 = k2 + BLOCK_SIZE){
            for(i = i2; i < i2 + BLOCK_SIZE; i++){
                for(j = j2; j < j2 + BLOCK_SIZE; j++){
                    for(k = 0; k < BLOCK_SIZE; k++){
                        tmp[k] = A[i][k2 + k] * B[j][k2 + k];
                    }
                    C[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3] + tmp[4] +
                    tmp[5] + tmp[6] + tmp[7];
                }
            }
        }
    }
}
```

Figure 21: Parallelization code



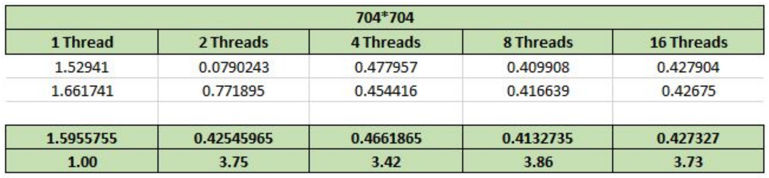
48*48				
1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
0.000485	0.00036	0.000395	0.001786	0.000878
0.000771	0.000318	0.000433	0.002505	0.000502
0.000628	**0.000339**	**0.000414**	**0.0021455**	**0.00069**
1.00	**1.85**	**1.52**	**0.29**	**0.91**

Figure 22: Execution times for 48*48 data-set with parallel code



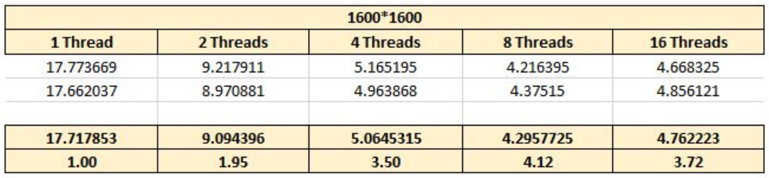
144*144				
1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
0.015557	0.010219	0.006684	0.005036	0.00636
0.014489	0.007941	0.005203	0.008359	0.005175
0.015023	**0.00908**	**0.0059435**	**0.0066975**	**0.0057675**
1.00	**1.65**	**2.53**	**2.24**	**2.60**

Figure 23: Execution times for 144*144 data-set with parallel code



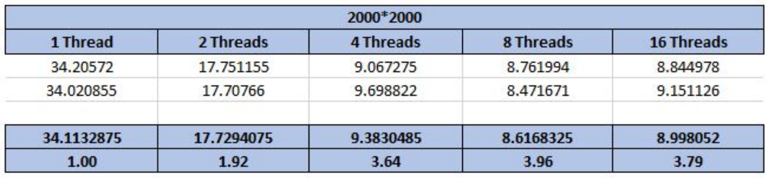
704*704				
1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1.52941	0.0790243	0.477957	0.409908	0.427904
1.661741	0.771895	0.454416	0.416639	0.42675
1.5955755	**0.42545965**	**0.4661865**	**0.4132735**	**0.427327**
1.00	**3.75**	**3.42**	**3.86**	**3.73**

Figure 24: Execution times for 704*704 data-set with parallel code



1600*1600				
1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
17.773669	9.217911	5.165195	4.216395	4.668325
17.662037	8.970881	4.963868	4.37515	4.856121
17.717853	**9.094396**	**5.0645315**	**4.2957725**	**4.762223**
1.00	**1.95**	**3.50**	**4.12**	**3.72**

Figure 25: Execution times for 1600*1600 data-set with parallel code



2000*2000				
1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
34.20572	17.751155	9.067275	8.761994	8.844978
34.020855	17.70766	9.698822	8.471671	9.151126
34.1132875	**17.7294075**	**9.3830485**	**8.6168325**	**8.998052**
1.00	**1.92**	**3.64**	**3.96**	**3.79**

Figure 26: Execution times for 2000*2000 data-set with parallel code