Work Assignment Report

University of Minho

Master in Informatics Engineering

Distributed Parallel Computing

Advanced Architectures

Carlos Rafael Cruz Antunes a67711
Nuno André da Silva Oliveira a67649

Table of Contents

Table of Figures

Abstract

On this work assignment we characterized in full the testing environment, and created a roofline model for the computer in which the code was developed and performance was tested.

It was created a single-threaded function in C that computes the dot product of two square matrices (in single precision and no block optimization). To this function was added performance counters using PAPI to measure the cache misses, total number of instructions, RAM accesses, number of floating point operations, GFLOP/s and operational intensity.

To this function were applied several changes to study its behavior and performance. Throughout this report this changes will be explained and its results compared. We transposed the matrixes accessed by column to improve memory affinity increasing the performance of the algorithm. Finally the algorithm was altered to permit vectorization. In this case only a partial vectorization was possible and there was no performance.

The main difficulties were the calculation of operational intensity's values, mainly due to the reduced sizes chosen for the matrixes, which generates a very low number of RAM accesses or even zero accesses, and so a very high and unstable operational intensity value. Another difficulty was the ceilings addition to the roofline, mainly because they would seem too high for the values obtained.

Introduction

The objective of this work assignment was to study a matrix multiplication algorithm and the impact on performance when certain aspects were changed. The tests to which this algorithm was subjected were focused on loop order, memory affinity and vectorization. In this particular case the loop order studied was j-i-k.

Testing Environment

The computer used in the current work assignment was an MSI GE60 2QE 1026XPT Apache Pro laptop. This computer main features are displayed on Hardware Characterization chapter, such as the description of its processor and cache, and some information about the RAM. The software used is described on Software Characterization.

Hardware Characterization

- Manufacturer: MSI
- Model: GE60 2QE 1026XPT (Apache Pro)
- Motherboard: MSI MS-16GF
- CPU chip
  - Manufacturer: Intel
  - Model: Core i7-4720HQ
  - Reference: [1]
  - Number of Cores: 4
  - Number of Threads: 8
  - Base Frequency: 2.6 GHz
  - Maximum Turbo Frequency: 3.6 GHz
  - Instruction Set: 64-bit
  - Architecture: Haswell
  - 16 DP FLOPs/cycle
  - Theoretical Peak Performance (DP): 166 GFLOPS
  - Benchmark Peak Performance (DP): 128 GFLOPS
- Cache
  - L1 Data Cache Size: 4 x 32 KB
  - L1 Instructions Cache Size: 4 x 32 KB
  - L2 Unified Cache Size: 4 x 256 KB
  - L3 Unified Cache Size: 6144 KB
- RAM
  - Manufacturer: Kingston
  - Size: 8 GB
  - DDR3L
  - 1600 MHz
  - Latency: 11 clock cycles
  - Maximum Memory Bandwidth: 13 GB/s

Both theoretical and benchmark achieved values are for double precision. The formula used to obtain the theoretical value was: $FLOPS = \#cores \times Average\ Frequency \times Operations/Cycle$ [2]. Which in this case is the same as: $FLOPS = 4 \times 2\,600\,000 \times 16$, as this processor can execute 16 double precision floating point operations per clock cycle. The result of this equation is the theoretical value of 166.4 GFLOPS. The theoretical value for single precision floating point operations per second is 332.8 GFLOPS. Converting the benchmark value to single precision results in 256 GFLOPS.

The majority of the values presented were gathered from the Intel page [1] about the CPU and from HWiNFO64 (version 5.10) [3] and Speccy [4] which are system information tools.

Software Characterization

The operative system used throughout the assignment was Ubuntu 15.10. The C code compiler used was GCC version 5.2.1 (the flags used were `-O3 -fno-tree-vectorize -lpapi`). PAPI 5.4.1 [5] was used for all performance measurements presented on this report. The benchmark used to calculate the peak performance was "Linpack" [6] (the output file can be found under Linpack Output (DP)). The benchmark used to calculate the Maximum Memory Bandwidth was "Bandwidth" version 1.1 [7].

Roofline Model

The roofline model for the i7-4720HQ processor is represented on FIGURE 1, where we can also see the roofline for Xeon E5650 (the 431 Search cluster node). This roofline was achieved using the values of the peak floating point performance and the maximum memory bandwidth.

The roofline for the same i7 processor (using only one core) with its respective ceilings is represented on **Error! Reference source not found.** where each dashed line is a memory or CPU ceiling and the triangles are the result to the different changes applied to the original matrix multiplication code.

The first ceiling (purple) represents the peak performance when there is no instruction level parallelism, witch in this case applies because each iteration of the matrix multiplication depends on the previous one being completed.

The second ceiling (blue) represents the peak performance when there is no Fused Multiply-Add operations, which were not studied in this work assignment. The third implies a code without vectorization which also could not be achieved. The forth ceiling limits the memory bandwidth and represents its value when only using one memory channel, as only one core is being used.

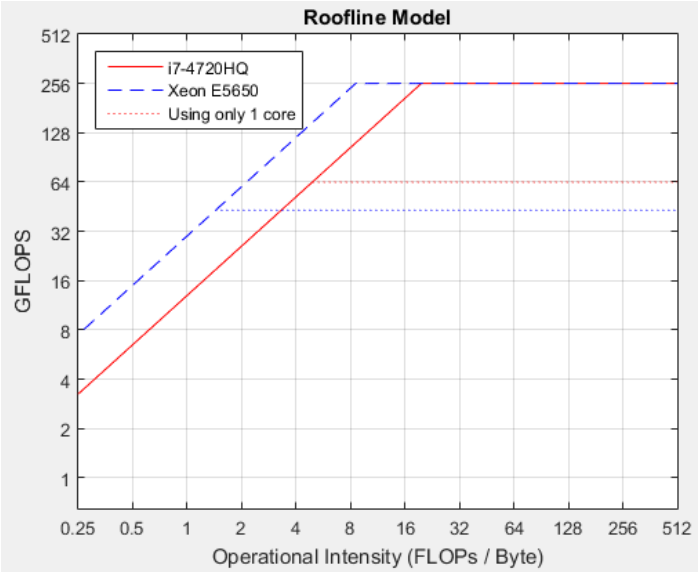

FIGURE 2 - ROOFLINE MODEL

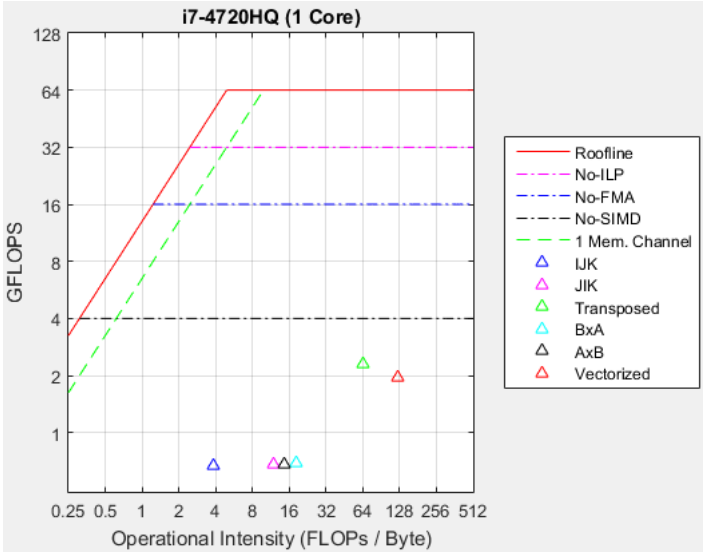

FIGURE 2 - ROOFLINE WITH CEILINGS

Matrix Multiplication

The sizes chosen for the matrix multiplication were: 40x40, 120x120, 600x600 and 1000x1000, so that they would fit in different levels of cache or even RAM (last case). The matrix multiplication was originally calculated by tree nested "for" cycles shown below with the order i-j-k (**Error! Reference source not found.**, blue triangle). Later the order was changed to j-i-k, which was given to our group for study (**Error! Reference source not found.**, purple triangle). One of the tests conducted consisted in replacing matrix B for a matrix with only ones (**Error! Reference source not found.**, cyan and black triangles).

```
for ( i …) for ( j …) for ( k …)
      result[i][j] += a[i][k] * b[k][j];
```

For this tests the results were very similar, all had a performance of approximately 0.6 GFLOP/s. The reason behind such low performance was the matrixes' orientation, in our case of study, matrix B and R is column major, which is very memory inefficient as a block is fetched (a part of a matrix line) and only one value of the several values of said block is used by the processor (represented on FIGURE 4). To solve this problem we transposed matrix B and R and adjusted the function in order to use it as row major (represented on FIGURE 5), and there was an increase on the obtained GFLOP/s (**Error! Reference source not found.**, green triangle). In this case not only was measured the matrix multiplication but also both of the transpose operations.

In our loop order, the position to be written to in the inner cycle is always the same, making vectorization impossible because there cannot be multiple instructions storing their data to the same spot in memory at the same time. To solve this problem we had to create an auxiliary array to store all the values of the multiplications and then add all the results and store them in the resultant matrix (the flags used were: `-O3 -lpapi -march=native -fopt-info-optimized`). By doing this we lost the multiply-add balance, lost the possibility of using the FMA instruction set, created another for loop and allocated one array with the size of one matrix line, degrading the performance of the optimization to the point where it is better without it (**Error! Reference source not found.**, red triangle).

All of this tests followed the k-best scheme with k=3 with 5% tolerance. All test were executed 8 times.

Conclusion

We concluded that the loop order studied was very inefficient, even more than the original i-j-k, because it leads to column major matrixes. Of all the tests to which the algorithm was subjected only the vectorization and the transposition had a positive impact on performance, and even so, the vectorization proved to be not very efficient in this loop order. This poor vectorization performance was due to a need to increase the algorithm's complexity in order to be able to vectorize (this new complexity also increased the number of floating point operations).

The main difficulties were the calculation of operational intensity's values, mainly due to the reduced sizes chosen for the matrixes, which generates a very low number of RAM accesses or even zero accesses, and so a very high and unstable operational intensity value. Another difficulty was the ceilings addition to the roofline, mainly because they would seem too high for the values obtained.

References

[1] "Intel® Core™ i7-4720HQ Processor," Intel, [Online]. Available: http://ark.intel.com/products/78934/Intel-Core-i7-4720HQ-Processor-6M-Cache-up-to-3_60-GHz.

[2] S. W. Williams, A. Waterman and D. A. Patterson, Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures, Berkeley: University of California, 2008.

[3] M. Malík, "HWiNFO," [Online]. Available: http://www.hwinfo.com/.

[4] "Speccy," Piriform, [Online]. Available: https://www.piriform.com/speccy.

[5] Innovative Computing Laboratory, "PAPI," University of Tennessee, 29 November 2015. [Online]. Available: http://icl.cs.utk.edu/papi/.

[6] J. Dongarra, "LINPACK," [Online]. Available: http://www.netlib.org/linpack/.

[7] Z. T. Smith, "Bandwidth," 2015. [Online]. Available: http://zsmith.co/bandwidth.html.

Annexes

Linpack Output (DP)

```
Current date/time: Wed Nov 25 19:30:19 2015

CPU frequency:    2.593 GHz
Number of CPUs: 1
Number of cores: 4
Number of threads: 8

Parameters are set to:

Number of tests: 1
Number of equations to solve (problem size) : 30000
Leading dimension of array                  : 30000
Number of trials to run                     : 8
Data alignment value (in Kbytes)            : 4

Maximum memory requested that can be used=7200604096, at the size=30000

================== Timing linear equation system solver =================

Size   LDA    Align. Time(s)    GFlops   Residual     Residual(norm)
Check
30000  30000  4      146.730    122.6867 8.668107e-10 3.416977e-02   pass
30000  30000  4      144.283    124.7675 8.668107e-10 3.416977e-02   pass
30000  30000  4      140.143    128.4533 8.668107e-10 3.416977e-02   pass
30000  30000  4      139.828    128.7423 8.668107e-10 3.416977e-02   pass
30000  30000  4      140.608    128.0286 8.668107e-10 3.416977e-02   pass
30000  30000  4      144.802    124.3198 8.668107e-10 3.416977e-02   pass
30000  30000  4      163.525    110.0860 8.668107e-10 3.416977e-02   pass
30000  30000  4      150.497    119.6156 8.668107e-10 3.416977e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
30000  30000  4       123.3375 128.7423

Residual checks PASSED

End of tests
```

Acronyms

Acronym	Meaning
API	Application Program Interface
CPU	Central Processing Unit
DDR3L	Double Data Rate Type 3 Low Voltage
DP	Double Precision
FLOPS	Floating Point Operations per Second
FMA	Fused Multiply-Add
GCC	GNU Compiler Collection
ILP	Instruction Level Parallelism
PAPI	Performance API
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SP	Single Precision

HWiNFO Report

A report generated by HWiNFO about the computer hardware can be found in the "HWiNFO report.txt" file.

Bandwidth Report

A report generated by "Bandwidth" with all the results of the benchmark can be found in the "bandwidth report.txt" file. The graph generated is represented on FIGURE 3 - MEMORY BENCHMARK RESULTS FROM BANDWIDTH 1.1.

PAPI Report

The PAPI events available can be found in "Papi_Avail.txt" file. The events used for this work assignment were: PAPI_L1_TCM, PAPI_L2_TCM, PAPI_L3_TCM, PAPI_TOT_INS.

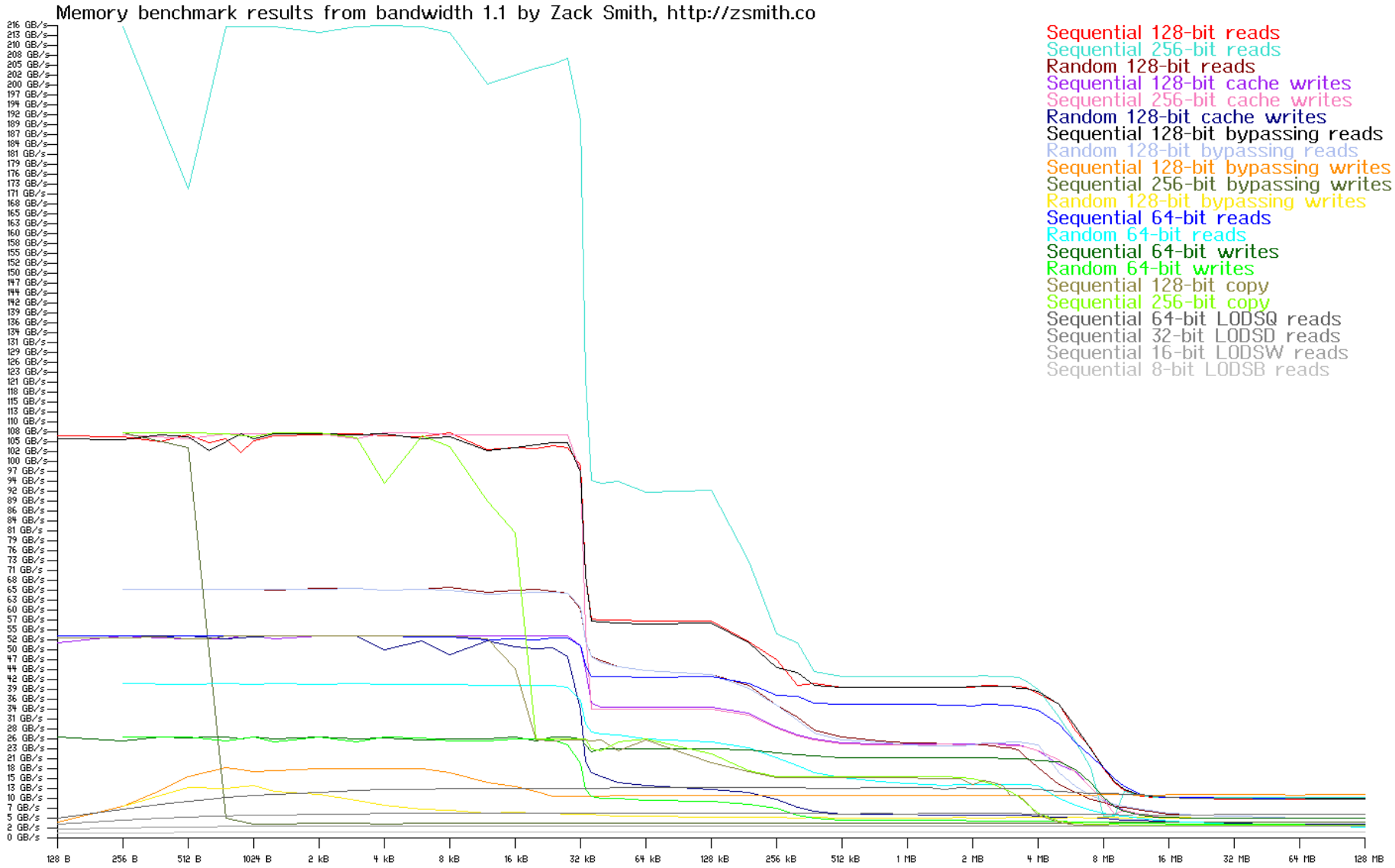

FIGURE 3 - MEMORY BENCHMARK RESULTS FROM BANDWIDTH 1.1

j-i-k

Original

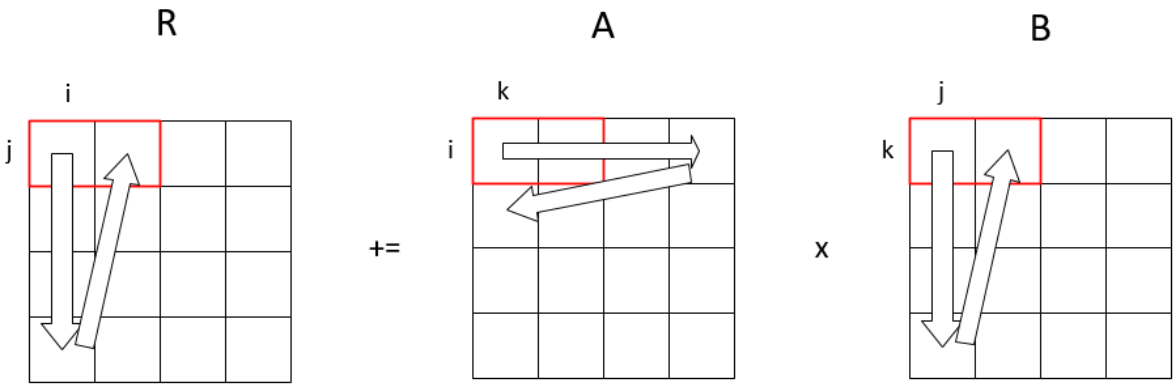

FIGURE 4 - ORIGINAL MATRIX MULTIPLICATION

B and R transposed

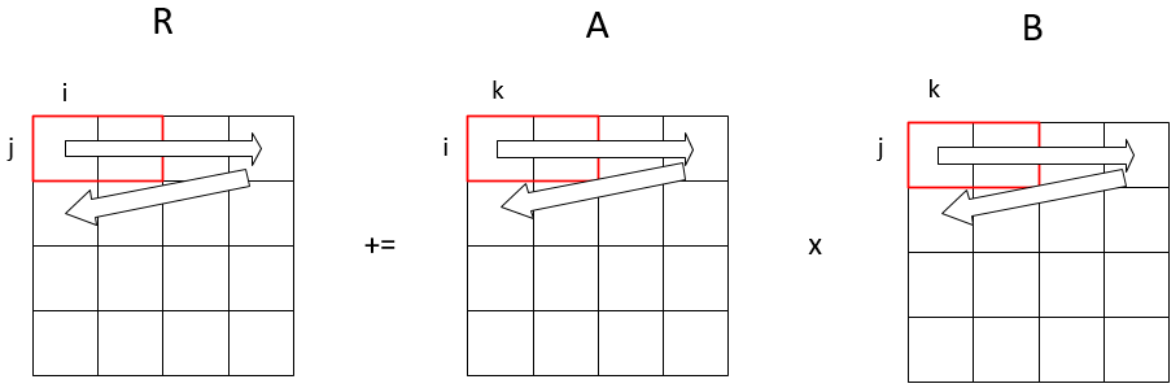

FIGURE 5 - MATRIX MULTIPLICATION WITH B AND R TRANSPOSED

"mmult" Function for vectorization

Function "mmult" changed to make vectorization possible.

```
void mmult(float a[SIZE][SIZE], float b[SIZE][SIZE], float result[SIZE][SIZE],int n)
{

	int i, j, k, l;			//Loop Counters
	float aux;			//Variable to store the sum of line
	float auxA[SIZE];		//Array to store the results of multiplications
	float btrans[SIZE][SIZE];	// Transposed Matrix b
	float restrans[SIZE][SIZE]; // Transposed Result Matrix

	// Transpose the matrix
	transpose(b, btrans);

	for (j = 0; j<n; j++) {
		for (i = 0; i<n; i++) {
			aux = 0;
			// Vectorized loop
			for (k = 0; k<n; k++) {
				// put the result in auxiliary array
				auxA[k] = a[i][k] * btrans[j][k];
			}
			for (l = 0; l<n; l++) {
				// Add all members of the array
				aux += auxA[l];
			}
			restrans[j][i] = aux;
		}
	}
	// Transpose the result matrix
	transpose(restrans, result);
}
```

Results Graph

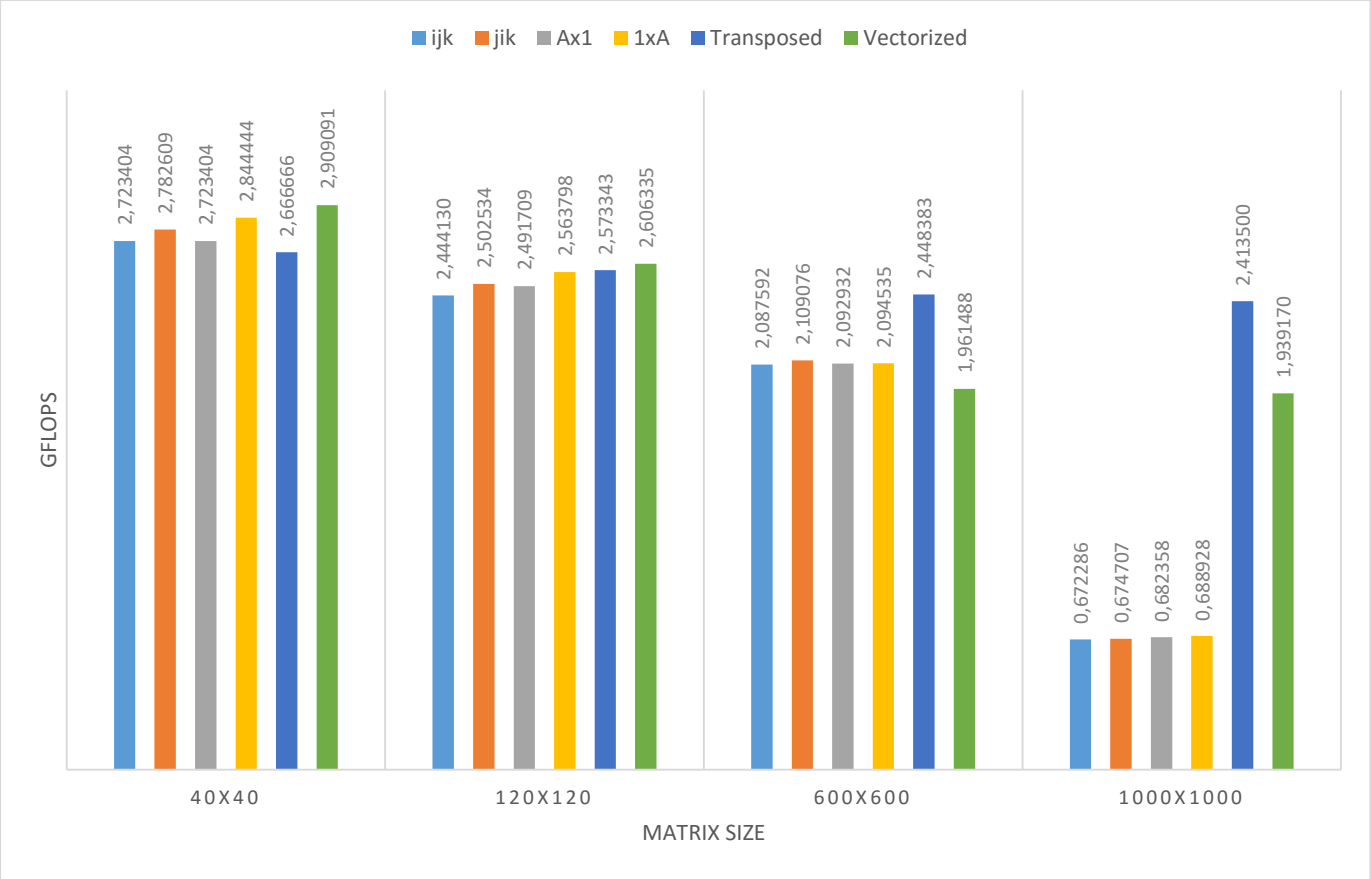

FIGURE 6 - RESULTS

The complete data can be found under “data.xlsx” file.