Advanced Architectures

Analysis of various Matrix Multiplication Algorithms and Optimizations

Eduardo Conceição
University of Minho
A83870@alunos.uminho.pt

Rui Oliveira
University of Minho
A83610@alunos.uminho.pt

Abstract

The main purpose of this work is to study the performance of a simple matrix multiplication algorithm and its optimizations, implemented in C++ and C extensions, such as CUDA. We will then analyze the various bottlenecks for the different optimizations to understand how they relate to the Roofline Model, in order to conclude the upper bound of the algorithm's performance.

I. Introduction

Evolution in computer architectures in recent decades has led to a great diversity in computational systems. Counter-intuitively, most system follow the same basic principles in design, such as the use of caches, pipelining, superscalar instruction issue and out-of-order execution [5].

This uniformity came to a head when multicore and manycore architectures were introduced. Now, manufacturers follow their own design philosophy. Now, to take full advantage of a system, a programmer must understand its intricacies. To this effect, models have been created to relate the performance of an implementation to the system in question; models like the Roofline Model [5].

In this paper we will relate the square matrix multiplication algorithm to a specific system, so that we are able to identify the various performance bottlenecks the implementations have, and what can be done in order to overcome them.

II. Hardware Description

The first step in our analysis is to understand the characteristics of the hardware we have at our disposal, as different hardware will provide different advantages and limitations.

In this section we will fully describe both the hardware used in the team's main laptop and in the SeARCH6 Cluster node used for the assignment.

i. Team's Main Hardware

We used an OMEN by HP - 15-ce012np, equipped with an Intel Core i7-7700HQ Kaby Lake co-processor and 16 GB(2slots*8GB) of DDR4-2400Mhz SDRAM [4]. A more detailed view of the features and specifications implemented in the chip can be found in table 1.

In order to estimate the peak floating point performance(PFP) for single precision, we assumed the width of the SIMD registers as 256 bits, hence 16 floats[2], and the throughput as one instruction per clock cycle. Also, since this architecture supports FMA, we are able to perform fused multiply-adds. Multiplying that by the number of cores and the clock

rate, we came up with the following (proof in the appendix):

$PFP_{teamlaptop} = 179.2GFlops$

Since this is a hardware limitation, 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 value provided by the manufacturer, obtaining a value of 36 GB/s for the peak memory bandwidth (PMB).

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

$performance = min(PFP, OP \times PMB)$ (1)

Table 1: *Team's laptop specifications*

Intel Core i7-7700HQ	
Performance	
Number of cores	4
Number of threads	8
Processor Base Frequency	2.8 GHz
Max Turbo Frequency	3.6 GHz
Memory Specifications	
L1 cache size	4 x 32 KB 8-way associative instruction caches
	4 x 32 KB 8-way associative data caches
L2 cache size	4 x 256 KB 4-way set associative caches
L3 cache size	6 MB 12-way set associative cache
Max Memory Size	64 GB
Max Memory Bandwidth	34.1 GB/s
Supported memory	DDR3L-1600, LPDDR3-2133, DDR4-2400
Memory Channels	2
Advanced Technologies	
Extensions	SSE4.1 + SSE4.2 / Streaming SIMD Extensions AVX2 (Advanced Vector Extensions 2.0)

So, the Roofline graph for this system is:

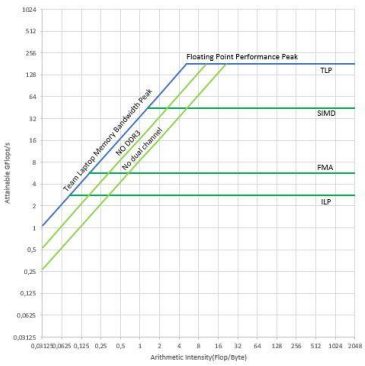

Figure 1: *Roofline Graph for the Team's Laptop*

Here, we have to note the following:

- The first roof comes from the use of Instruction Level Parallelism;
- The second one from the use of Fused Multiply Add;
- The third roof from the use of SIMD;
- The final roof from the use of Thread Level Parallelism, giving the highest acheivable performance.

ii. The SeARCH Cluster's r662 Node

The SeARCH's 662[6] nodes are dual processor machines, each an Intel Xeon Processor E5-2695 v2[1]. The more specific hardware features regarding the package are in table 2.

Table 2: *Intel Xeon Processor E5-2695 v2 characterisation*

Intel Xeon Processor E5-2695 v2	
Performance	
Number of cores	12
Number of threads	24
Processor Base Frequency	2.4 GHz
Max Turbo Frequency	3.2 GHz
Memory Specifications	
L1 cache size	12 x 32 KB 8-way associative instruction caches
	12 x 32 KB 8-way associative data caches
L2 cache size	12 x 256 KB 4-way set associative caches
L3 cache size	30 MB 20-way set associative cache
Max Memory Size	64 GB
Max Memory Bandwidth	59.7 GB/s
Supported memory	DDR3 800/1066/1333/1600/1866
Memory Channels	4
Advanced Technologies	
Extensions	SSE4.1 + SSE4.2 / Streaming SIMD Extensions AVX2 (Advanced Vector Extensions 2.0)

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 2. The main differences occur in the number of processors, because the team laptop uses a single processor, and the cluster node has no support for FMA[3].

$PFP_{clusternode} = 460.8GFlops$ (2)

Once again assuming the max memory bandwidth the manufacturer assures, we assumed the PMB to be 43 GB/s.

As such, when compared to the team laptop's graph, the cluster node's is the following:

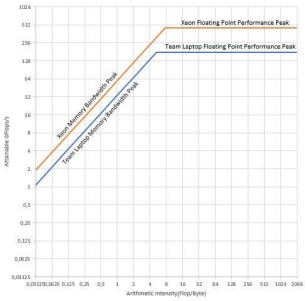

Figure 2: *Roofline Graph for the Team's Laptop compared to the node's*

III. PAPI Performance Counters

The Performance API (or PAPI) provides a considerably sized group of hardware counters that allow the program to measure several metrics related to cache accesses, misses, total instructions, etc., in order to establish patterns in the algorithm's behaviour.

Out of the many counters available in the SeARCH6 Cluster's r662 node, we chose the following:

- **PAPI_L2_DCR**: L2 Data Cache Reads
- **PAPI_L3_DCR**: L3 Data Cache Reads
- **PAPI_L2_TCM**: L2 Total Cache Misses
- **PAPI_L3_TCM**: L3 Total Cache Misses
- **PAPI_TOT_INS**: Total Instructions
- **PAPI_FP_OPS**: Total Floating Point Operations
- **PAPI_LD_INS**: Total number of Load Instructions

Using these counters, we will be able to measure certain aspects of the program, such as miss rates and accesses to RAM, that will aid in the characterization of the algorithm's performance.

IV. Implementation of the Multiplication Algorithm

i. First Implementations

The first implementation of the algorithm follows the classic structure of matrix multiplication: using three nested loops, i, j and k, where i iterates over the resulting matrix's rows and j the columns, we iterate on the first input matrix row-wise and on the second one column-wise, accumulating the sum of all elements in the result matrix, which is also iterated row-wise.

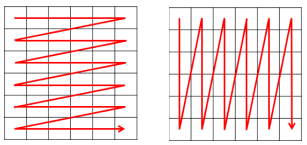

Figure 3: *Visualization of Row-wise accesses (left) versus Column-wise accesses (right)*

The column-wise access pattern to the second input matrix is very inefficient, and presents itself as a vectorization blocker.

Two other variations of the algorithm were also implemented, using other orders for the loops: i-k-j and j-k-i. In j-k-i, we have, not one, but two column-wise access patterns, which prove to be most inefficient. However, in i-k-j, the problem we faced in the first implementation is solved, and this one proves to be the most efficient one.

Nonetheless, we strive to improve the other two implementations. To do this, we can **transpose** matrices, in order to remove column-wise accesses and turn them to row-wise accesses. We applied this to the second input matrix in i-j-k and to all three matrices in j-k-i.

ii. Dataset Sizes

In order to test the limitations of this algorithm, we used four different dataset sizes: three which fit entirely in their respective level of cache an a fourth one that doesn't fit in cache at all.

As all tests were performed in the Cluster node, the cache sizes used to calculate these values were the ones described in section II.ii. In order to calculate the size of the matrices, we need to take into account three square matrices of single precision floating point numbers, each having N elements per line. So:

$cache\_size \geq N^2 \times sizeof(float) \times 3, where N = 2^k$

Taking this inequality into consideration, we used the following values for N, for each cache level:

- L1: $N = 32$
- L2: $N = 128$
- L3: $N = 1024$
- Does not fit in cache: $N = 2048$

We then ran a few tests with these values in mind.

iii. Test Results

Tests were all conducted in the SeARCH Cluster, in the described node (r662), and, for each dataset, the test were ran eight times, and the best results were picked using a k-best strategy, with a 5% tolerance.

The results, in milliseconds (ms), were the following:

	32x32	128x128	1024x1024	2048x2048
IJK	0.005	0.405	3087.577	25142.245
IKJ	0.005	0.405	911.196	7411.031
JKI	0.005	0.405	8802.939	129467.31
IJK Tr.	0.005	0.315	236.964	2247.194
JKI Tr.	0.007	0.456	902.194	27318.38

Table 3: *Test Results for the first implementations*

As one can notice, the effects of optimizations are not as noticeable with the smaller datasets, but, in the larger ones, the difference is remarkable, with both i-j-k and j-k-i implementations having a considerably superior performance when using the transposed matrices.

iv. RAM Accesses

Another metric that we will use to evaluate the performance of the implementations is the number of RAM accesses each one makes. To this effect, each algorithm will have different patterns of access to RAM, based on the pattern of access to the matrix elements.

Node r662 is able to fetch 16 single precision floating point elements from memory at once, as 64 bytes is the size of a cache line. To get the best use out of a single access, we should use as many of these 16 values as possible. To do this, we have to use an access pattern that will access contiguous positions in memory iteration after iteration (**row wise**). Each value of a given line of a matrix should be stored in memory in contiguous positions, hence accessing values in the same line from iteration to iteration should use values in the same cache line. So, for each access we make, we load 16 useful values, and, for N^2 elements, we make $\frac{N^2}{16}$ accesses to memory.

In contrast, a **column wise** access pattern, we are likely to use only one value from a cache line, because accesses are not to contiguous positions in memory, as each iteration will use a value from different lines. That being said, the number of accesses for N^2 elements is N^2. We leave as an annex the formulae to calculate the number of RAM accesses.

We can calculate the number of accesses using the PAPI counter **PAPI_L3_TCM**, as L3 cache misses correspond to accesses to RAM (formula in annex).

This will give us the ratio in relation to the entirety of the executed instructions. Another way to calculate this is in relation to load/store instructions, but we have decided to check the ration in relation to all of the instructions used.

The ratios obtained can be seen in the following table:

	32x32	128x128	1024x1024	2048x2048
IJK	0.00015	0.0000008	0.00001	0.0012
IKJ	0.00004	0.0000006	0.000006	0.0002
JKI	0.00003	0.0000006	0.0009	0.002
IJK Tr.	0.0003	0.0000007	0.00007	0.001
JKI Tr.	0.00005	0.0000001	0.00003	0.00006

Table 4: *RAM Accesses p/ Instruction*

As was expected, bigger dataset sizes imply a bigger ratio of RAM accesses, as do worse memory access

patterns.

As for the number of bytes transfered from RAM, we know that, when memory is transfered from cache it is an entire line (64 bytes), and that is done in all cache misses at the third level of cache. As such:

	32x32	128x128	1024x1024	2048x2048
IJK	376	272	8186680	5325199920
IKJ	96	186	2863767	934529070
JKI	92	151	53801554	1.01511E+11
IJK Tr.	731	146	11660450	1289771609
JKI Tr.	179	298	15727540	2122894947

Table 5: *Bytes Transferred from RAM (Avg.)*

v. Floating Point Operations

To estimate the number of floating point operations, we take into account three for loops, each iterating from 0 to N, and, in each iteration of the innermost loop, we have a sum and a multiplication. Thus:

$FPOperations = N^3 \times 2$

From which we derive:

32x32	2^{16}
128x128	2^{22}
1024x1024	2^{31}
2048x2048	2^{34}

Table 6: *Estimated Floating Point Operations*

Using the PAPI counter **PAPI_FP_OPS**, we were able to determine the real values for each size. Do note that we were not able to obtain the values for the two smaller datasets.

Using the various metrics gathered thus far, we can now compare the performance of the implementations in the Roofline graph (the relationship between the x and y coordinates in the graph are in appendix).

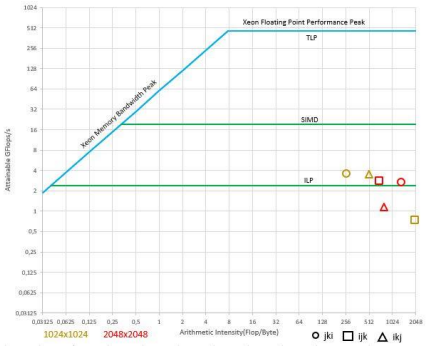

Figure 4: *Implementations in relation to the Roofline Graph*

vi. Miss Rates

In order to calculate the local miss rates, we use the formulae found in the appendix.

Taking those into account, we arrived at the following table, for i-j-k, which we considered the most interesting one to evaluate, as it is the one that had greater gains in terms of execution time when the matrix transpose is applied.

		32x32	128x128	1024x1024	2048x2048
IJK	L1	1.7%	12%	52%	52%
	L2	15%	0.4%	95%	95%
	L3	16%	0.4%	0.1%	0.9%
IJK Tr.	L1	1.5%	16%	36%	44%
	L2	16.1%	0.5%	81%	64%
	L3	6.04%	0.2%	0.06%	0.006%

Table 7: *Local Miss Rates*

As was to be expected, for bigger datasets, miss rates in higher level caches are higher. Also of note is the fact that the version with the matrix transpose, for bigger datasets, has smaller miss rates in cache levels 2 and 3, which was expected because of the better access patterns.

Another interesting point is that the smallest dataset, which supposedly fits entirely in cache level 1, has non null miss rates in higher levels. This is probably due to the fact that we have initially, after the matrices have been initialized, a cold cache, which

makes it so that loading the matrices from RAM is required, justifying the miss rates higher than zero.

vii. Overall Algorithm Behaviour

When it comes to boundary analysis on the matrix multiplication algorithm, we can state that, for every element that is added to the matrices, that is, going from N to N+1, we are performing $6N^2 + 6N + 2$ more calculations. This certainly puts some strain on the CPU, but a polynomial evolution is not the most penalizing point.

When it comes to memory, the biggest concerns are accesses to RAM. If we accept a model where we have no cache, and accesses to RAM are direct, for the three patterns of memory access (Column wise, Row wise and Mixed), we find that their respective growth is not larger than the growth of computational overhead (proof found in annex). This does not mean, however, that this family of algorithms are not memory bound, as RAM, especially DRAM, is much slower than processing units. When we make such a statement, we are assuming infinite bandwidth, which is not a logical assumption and, as such, while the PU may be able to handle the computational overhead increase, memory bandwidth is certainly not as capable, and would show deterioration faster than the computational power.

However, taking into consideration the area of the Roofline graph the non-optimized versions of the algorithm fall in, it would suggest they would benefit more from computational optimizations[5], which we will also apply later. Thus, the algorithm and its first implementations are **CPU bound**.

viii. Optimizations

viii.1 Blocking

Blocking improves efficiency as it directly explores the spacial locality principle. This principle dictates that, if a value is used, it's immediate neighbours have a high chance of also being used in the near future. Matrix blocking introduces an optimization to the basic matrix multiplication algorithm because we only use a part of the matrix at once, called a tile, which allows all the values that will be fetched from RAM at once to be used efficiently, namely by using them once and never again, in theory. We can explore this further, by defining a good tile size. As stated previously, a line of cache contains 16 floats, and these will all be pulled from the cache at once. By making the block size 16, we ensure that an entire line of cache is processed thoroughly and will not be needed again. We can also, of course use multiples of 16 for this purpose, though we only focused on 16.

We implemented blocking alone on i-j-k, while applying both blocking and the matrix transpose to both i-j-k (the one that has showed most gains thus far) and to j-k-i.

	32x32	128x128	1024x1024	2048x2048
IJK	0.005	0.405	3087.577	25142.245
JKI	0.005	0.405	8802.939	129467.31
IJK Bl.	0.064	4.195	2737.183	27318.38
IJK Bl. Tr.	0.005	0.330	1062.191	10560.728
JKI Bl. Tr.	0.037	2.352	1493.128	15083.021

Table 8: *Exec. Time for the Blocking Optimization (ms)*

The results show that there was, in general, no improvement, having occurred slowdown in some cases. This is strange, and we do not know exactly why this happened. It may be due to the fact that we used an additional check in each for loop in order to guarantee that values outside of the allowed scope are not accessed, which may have impeded some compiler optimizations.

viii.2 Vectorization

We applied explicit vectorization to i-j-k with the use of the transpose and blocking, using *pragma ivdep*.

	No Vectorization		With Vectorization	
	32x32	128x128	32x32	128x128
IKJ	0.005	0.405	0.005	0.738
IJK Tr.	0.005	0.315	0.005	0.329
JKI Tr.	0.007	0.456	0.007	0.466
IJK Bl. Tr.	0.005	0.330	0.007	0.338
JKI Bl. Tr.	0.0	2.352	0.006	0.257

Table 9: *Execution Time with Vectorization*

Gains were minute and relatively insignificant, which is something the compiler itself stated was likely to happen.

viii.3 Vectorization and OpenMP

As the r662 node has dual *Xeon* 12-core processors, in order to not use hyperthreading, we used 24 threads in total, dividing the workload equally amongst them.

	OMP	No Vectorization
IJK Bl. Tr.	313,288	10560,728
JKI Bl. Tr.	460,269	15083,021

Table 10: *Open MP execution time (ms)*

This version had very significant gains, superlinear ones even. We achieved this possibly due to the use of several L1 and L2 caches, as each core has their own, which improves memory management (and also lends credence to the point that these implementations are memory bound).

viii.4 CUDA

The final optimization we applied was the use of a device, a Kepler GPU, as it is the kind of accelerator available in the r662 node, implemented using the C extension CUDA, with blocking and taking advantage of shared memory. The results, using once again the biggest dataset, are the following:

	Comp.	DT to Device	DT from device
IJK Bl.	84.919	11.533	8.873

Table 11: *CUDA Execution Time (ms)*

These results are leaps and bounds better than even the optimization using OpenMP. Also, communication to the device took longer than from the device, which is justified by the fact that we are transferring two matrices to the device, in opposition to the single one transferred from the device.

V. Conclusion

In conclusion, the the classic matrix multiplication algorithm stands to gain much in terms of performance when applying these optimizations. From simple modifications such as the application of the transpose matrix, to the use of C extensions like OpenMP and CUDA, we were able to understand the gains obtained from the use of such optimizations, and how we could relate these optimizations with the specif hardware used.

VI. References

[1] *Intel Xeon E5-2670 v2 specifications.* URL: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%5C%20E5-2670%5C%20v2.html.

[2] *Ivy Bridge Processors.* URL: https://www.nas.nasa.gov/hecc/support/kb/Ivy-Bridge-Processors%5C_445.html.

[3] James Reinders Jim Jeffers. *Intel Xeon Processors, Science Direct.* URL: https://www.sciencedirect.com/topics/computer-science/intel-xeon-processor.

[4] *OMEN by HP 15-ce012np specifications.* URL: https://www.cpu-world.com/CPUs/Core%5C_i7/Intel-Core%5C%20i7%5C%20i7-7700HQ.html.

[5] David A. Patterson Samuel W. Williams Andrew Waterman. "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures". In: (2008).

[6] *SeARCH 6 Cluster hardware description.* URL: http://search6.di.uminho.pt/wordpress/?page%5C_id=55.

VII. Appendix

i. Peak Floating Point Performance

$PFP_{teamlaptop} = \#cores \times frequency \times \frac{SIMDwidth}{SIMDthroughput} \times FMA = 4 \times 2.8 \times \frac{256}{32} \times 2 = 179.2GFlops$

ii. RAM Accesses per Instruction

$RAMAccessesperInstruction = \frac{L3TCM}{TOTINS}$

iii. Miss Rate Formulae

- $L1MissRate = \frac{L2DCR}{LDINS}$
- $L2MissRate = \frac{L3DCR}{L2DCR}$
- $L3MissRate = \frac{L3TCM}{L2TCM}$

iv. FP Operations

	Est. FP op.	IJK	IKJ	JKI	IJK Tr.	JKI Tr.
32x32	65536	-	-	-	-	-
128x128	4194304	-	-	-	-	-
1024x1024	2147483648	8588486654	3267237595	10146061437	1269307749	2314692502
2048x2048	17179869184	67314995188	25966117683	97329187911	1217034314	18760325345

Table 12: *Obtained Floating Point Operations*

v. Coordinates for each point in the graph

$alg\_coord_{size} = (\frac{FP\_OPS/10^9}{T_{exec}/10^3}, \frac{FP\_OPS}{N^2 \times sizeof(float) \times 3})$