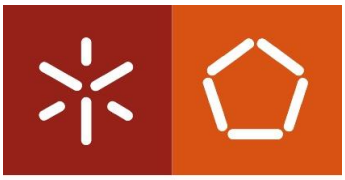

Universidade do Minho

Departamento de Informática

Advanced Architectures
Dotprod Implementations and Benchmarking

João Teixeira (A85504)
José Filipe Ferreira (A83683)

January 30, 2021

Abstract

This paper documents the development and benchmarking of different dotprod algorithms with the aim of analyzing, among others, the impact of matrix transposing, vectorization, paralllization and CUDA on the computation time

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last decades the hardware market has seen a shift towards the mass availability of multicore CPUs and the constant increase of the computing power of the GPUs. Due to this increased availability there is a growing need for performance engineers and programmers capable of exploring this constantly expanding field.

In this paper we will document the development and benchmarking of different dotprod algorithms in order to study the impact of several alterations on the basic dotprod implementation.

Firstly we will compare different sequential versions of the algorithm. Then we will introduce blocking to the algorithm and vectorize it. Finally, to compare the maximum performance of the CPU with the maximum performance of the GPU, a parallelized version of the algorithm will be developed with OpenMP and a GPU version will be developed making use of the CUDA API.

Chapter 2

Characterization of the hardware

Our team's main laptop is a Lenovo ThinkPad X260. It is powered by a i5-6300U, a dual-core hyperthreaded skylake cpu.

To get more information related to the CPU's memory hierarchy we used the command *lstopo* that displays a graphic with information related with the cache size.

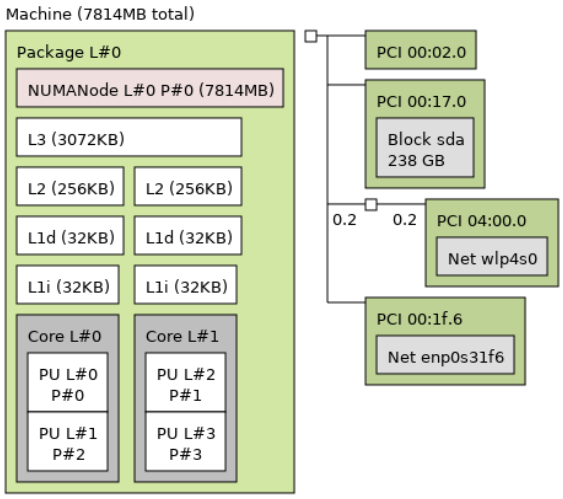

Figure 2.1: lstopo graphic

As it is visible in the picture above the CPU has 32KiB of L1 cache per core, 256KiB of L2 cache per core and 3072MiB L3 cache. It also has 7814MiB of Random Access Memory(RAM) with minimum latency of 13.750 nano seconds acording to vendor specs.

The computer used for obtaining the results during this paper was the node 662 of the search cluster at Universidade do Minho[1]. It is equipped with a dual Intel Xeon E5-2695v2 processor with 24 cores and Nvidia Kepler K20m accelerators. This CPUs have 32 KiB of L1 cache for each core, 256KiB of L2 cache, 30MiB of L3 cache and 64GiB of RAM.

[1] http://search6.di.uminho.pt

Chapter 3

Dotprod Implementation

The dotprod is a mathematical operation between two matrices (A and B) that results in another matrix (C). To obtain the position (i,j) in the matrix C we need to multiply each value of the line i in the matrix A with each value of the column j of the matrix B and add them.

When programming this behaviour it is commonly used a nested triple for loop going from 0 to the size of the matrix. Each loop having a different variable usually i, j and k. The variable i indexes the lines on the A matrix, the j indexes the columns on the B matrix and the k is the loop that iterates over the line and column.

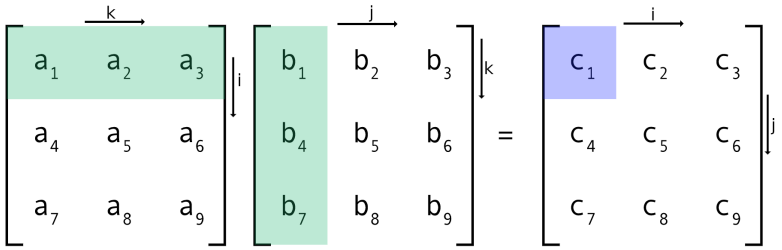

Figure 3.1: Dotprod Diagram

The first implementation of the algoritm had the nested loop in the order **i-j-k**. However, this order can be altered without having to change the content of the loop. Therefore, in order to measure differences in performance two other options were created: **i-k-j** and **j-k-i**. After canalizing all the created algorithms we realized that the way the matrices were indexed differed between implementation. In some they were accessed by column and others by line.

While accessing a matrix line by line explores cache locality and decreases cache misses, accessing a matrix collumn by collumn increases cache misses due to a lower cache locality. Therefore, in order to mitigate the penalty of accessing the matrices column by collum two more algorithms were created: **i-j-k trans** that transposes the matrix B before computing the dotprod; **j-k-i trans** that transposes the matrices A and B before calculating the dotprod and transposes the matrix C after computing.

This results in the following patterns of access depending on the algorithm:

	A	B	C
i-j-k	line	collumn	line
i-j-k trans	line	line	line
i-k-j	line	line	line
j-k-i	collumn	collumn	collumn
j-k-i trans	line	line	line

Table 3.1: Matrices access based on loop order

In order to explore even more cache locality another version of the algorithm was created that uses block optimization. In this version of the algorithm we used the access order of **i-j-k** while also transposing the matrix B in order to be able to access it line by line.

In this version, only part of a line of A, a block of B and part of a line of C is kept in cache at any given time. The line from A is multiplied by the block from B in order to populate the line in C. And after this calculation is done, another section of A is fetched that is multiplied by the same block in B in order to compute another section of a line from C. This way, the block in B is reused several times reducing the number of load instructions of the program.

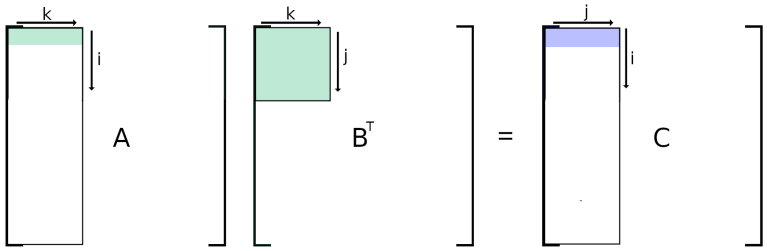

Figure 3.2: Dotprod with Block Optimization

After creating a working dotprod implementation with block optimization we decided to apply the vectorization flags to the compilation and analize the compiler output in order to check if the code was successfully vectorized. Due to a Read after write data dependency minor tweaks had to be applied to the code in order for it to vectorize successfully.

Finally this version of the dotprod with block optimization and vectorization was modified to run efficiently on all cores of a multicore device making use of OpenMP.

Due to the fact that the GPU is design to perform operations over matrices it would be relevant to compare the performance of the optimized dotprod with the performance of a algorithm designed to use all the SMX of a Kepler architecture. This algorithm uses matrix blocking and GPU shared memory in order to maximize the performance of the computation.

In this algorithm each thread calculates a position of the C matrix. Since the shared memory is shared within a given block and contains all the values needed to calculate a temporary block that will be added to the final block on the matrix C, all the threads on that given block can calculate their value accessing only the shared memory. This way the threads spend less time on the scoreboard waiting for the memory. Due to the parallel nature of the GPU all the blocks are being computed at the same time.

Chapter 4

Dotprod Benchmarking

In order to properly benchmark the different algorithms created and compare them we choose to collect data with PAPI and *sys/time.h*. The full list of counters available in the used system is on the Appendix B. From this list we choose to use the following PAPI counters:

`PAPI_L1_TCM, PAPI_L2_TCM, PAPI_L3_TCM, PAPI_LD_INS, PAPI_SR_INS, PAPI_TOT_INS`

With this set of counters we can validate the theory that different algorithms have different cache misses and memory access and also identify possible bottlenecks related to cache misses.

To also benchmark the impact of different sizes of matrices we choose four different matrix sizes. The smaller one fits in the L1 cache; the second smallest fits in the L2 cache; the second largest fits in the L3 cache and finally the biggest one requires significant access to RAM. Based on the CPU used for benchmarking the following table was created were the formula to calculate the max size is:

$$N^2 * 3 * sizeof(float) = cache\_size$$

	L1 data	L2 data	L3 data	RAM
cache size	32768B	262144B	31457280B	>31457280B
max size	52	147	1619	>1619
used size	40	120	1500	4000
estimated FP op's	128000	3456000	6750000000	$1.28*10^{11}$
estimated RAM transfers	19200B	172800B	27000000B	192000000B

Table 4.1: Matrix sizes used for benchmarking

For the tests, the matrix A was filled with random numbers and the matrix B was filled with ones. This way, AxB results in a matrix where all column have the same value and BxA results in a matrix where all lines have the same value. This property was used to assert if all the variations of the algorithm were producing the correct results. After each test the cache was cleared to ensure all the tests used a cold cache. To reduce the outliers, the time measurements were performed 8 times using a K-best scheme with K=3 and a tolerance of 5%.

	40x40	120x120	1500x1500	4000x4000
i-j-k	66	1733	4239493	177553349
i-j-k trans	94	1698	3223331	71675215
i-k-j	71	1718	3194044	61695968
j-k-i	72	1744	8596300	324751067
j-k-i trans	*69*	*1375*	*2273077*	*45375529*

Table 4.2: Execution time for non-blocking sequential algorithms (time in μs)

The results achieved for the **j-k-i trans** were not the expected results. We expected that the **j-k-i trans** should be slower than **i-j-k** due to the extra cost of transposing more matrices. However this is not what was observed on the results. After canalizing the compiler output we realized that the only function that vectorized was that one. Therefore, all the tables have the line relative to this algorithm in italic.

For the 40x40 matrix the benefit for the computation that comes from transposing the matrix is not enough to outweigh the cost of actually transposing it. For bigger matrices the difference in times becomes even more noticeable resulting in results one order of magnitude faster for the 4000x4000 matrix.

	40x40	120x120	1500x1500	4000x4000
i-j-k	.001321	.000057	.000582	.781009
i-j-k trans	.001911	.000105	.000308	.084568
i-k-j	.027150	.007252	.000259	.030864
j-k-i	.036501	.008403	.001570	.785903
j-k-i trans	*.034159*	*.008555*	*.000902*	*.053824*

Table 4.3: RAM access per total instructions (%)

		40x40	120x120	1500x1500	4000x4000
i-j-k	L1	.2360	2,1902	35.5386	34.4700
	L2	.0762	.0237	2.3420	34.4019
	L3	.0025	.0001	.0016	2.0842
i-j-k trans	L1	.1865	2.0801	2.0981	2.1194
	L2	.0902	.0294	.0532	.7009
	L3	.0664	.0196	.0015	.2712
j-k-i	L1	.1411	1.9620	50.0075	50.1961
	L2	.0873	.0273	9.6299	50.1126
	L3	.0709	.0173	.0031	1.5741
j-k-i trans	*L1*	*.6037*	*7.9182*	*8.4272*	*8.5532*
	L2	*.3874*	*.1204*	*.2507*	*6.4841*
	L3	*.3270*	*.0831*	*.0087*	*2.9880*

Table 4.4: Global miss rate for algoritms that benefited from transposing

Analysing this table we can confirm the theory that transposing the matrix in order to access it line by line has, in fact, the result of reducing cache misses. The outlier in this case is the **j-k-i trans** that actually increased the global miss rate. After carefully analysing the results we realized that the cache misses for a given cache remained mostly the same between **j-k-i** and **j-k-i trans**, what greatly decreased was the number of load and store instructions. Since the formula used for calculating the global miss rate divides the number of cache misses for a given cache level by the total number of memory access, reducing the number of memory access increases the global miss rate.

	i-j-k trans	i-j-k trans + block	speedup
4000x4000	71675215	64385275	1.1132

Table 4.5: Time comparison of block optimisation (time in μ)

	i-j-k trans	i-j-k trans + block + vec	speedup
40x40	94	71	1.3239
120x120	1698	1523	1.1149

Table 4.6: Time comparison of block optimization with vectorization (time in μ)

	i-j-k trans	i-j-k trans + block + vec + OpenMP	speedup
4000x4000	71675215	2407551	29.7710

Table 4.7: Time comparison of block optimization with vectorization and OpenMP (time in μ)

The blocking of the algorithm gratly reduces the time of the algorithm. Simply adding blocking we achieved a speedup of 1.1132. Adding vectorization we were able to achieve a speedup of 1.3239 for the smallest matrix. Finnaly, adding parallelism with OpenMP to the vectorized code yielded a speedup of 29.7710.

	Computation	Host to Device	Device to Host	Total
120x120	195	33	9	640
512x512	1390	524	154	4206

Table 4.8: Analysis of CUDA times (time in μs)

	120x120	Speedup
CUDA	640	1.0000
i-j-k trans + block + vec	1523	2.3796
i-j-k trans	1698	2.6531

Table 4.9: Time comparison of CUDA (time in μs)

Chapter 5

Conclusions

After the analysis of the results collected during the elaboration of this report, we can conclude that memory access optimizations can greatly impact an algorithm performance, reducing cache misses, therefore, reducing the data wait time of the processor. We can also conclude that GPU processing can be a lot faster to process matrices than a CPU, due to its inherently parallel architecture.

Appendix A

Roofline of the 662 node

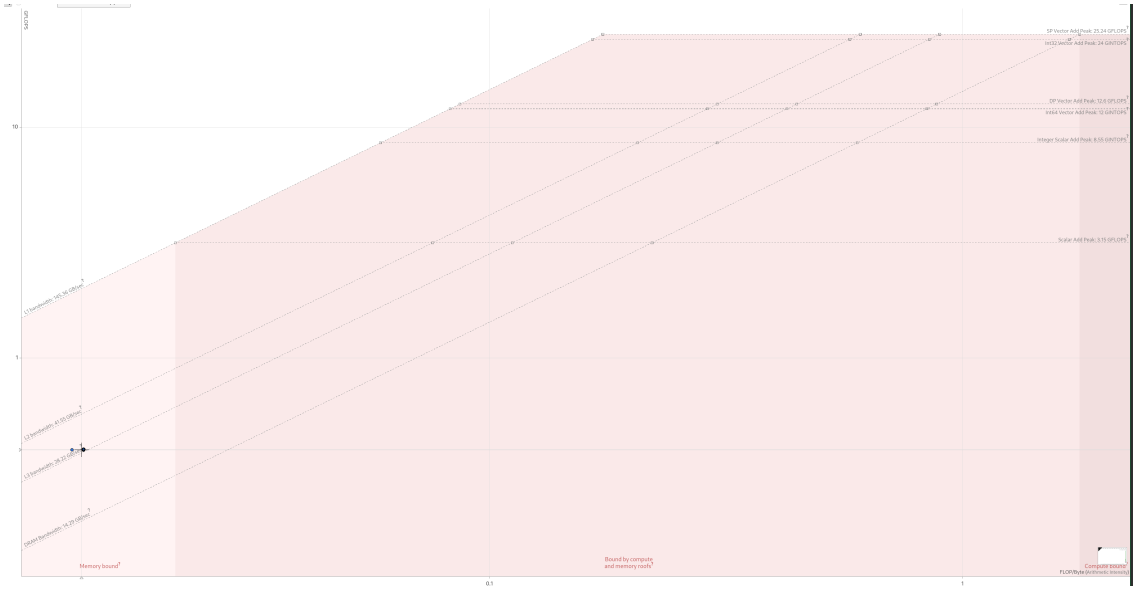

Figure A.1: roofline on the 662 node

Appendix B

Available papi counters

```
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.5.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz (62)
CPU Revision             : 4.000000
CPUID Info               : Family: 6  Model: 62  Stepping: 4
CPU Max Megahertz        : 2401
CPU Min Megahertz        : 1200
Hdw Threads per core     : 2
Cores per Socket         : 12
Sockets                  : 2
NUMA Nodes               : 2
CPUs per Node            : 24
Total CPUs               : 48
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------

================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  No   Level 3 cache misses
PAPI_TLB_DM  0x80000014  Yes  Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  No   Instruction translation lookaside buffer misses
PAPI_L1_LDM  0x80000017  No   Level 1 load misses
PAPI_L1_STM  0x80000018  No   Level 1 store misses
PAPI_L2_STM  0x8000001a  No   Level 2 store misses
PAPI_STL_ICY 0x80000025  No   Cycles with no instruction issue
PAPI_BR_UCN  0x8000002a  Yes  Unconditional branch instructions
PAPI_BR_CN   0x8000002b  No   Conditional branch instructions
PAPI_BR_TKN  0x8000002c  Yes  Conditional branch instructions taken
PAPI_BR_NTK  0x8000002d  No   Conditional branch instructions not taken
PAPI_BR_MSP  0x8000002e  No   Conditional branch instructions mispredicted
PAPI_BR_PRC  0x8000002f  Yes  Conditional branch instructions correctly predicted
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_FP_INS  0x80000034  Yes  Floating point instructions
PAPI_LD_INS  0x80000035  No   Load instructions
PAPI_SR_INS  0x80000036  No   Store instructions
PAPI_BR_INS  0x80000037  No   Branch instructions
```

```
PAPI_TOT_CYC 0x8000003b  No   Total cycles
PAPI_L2_DCH  0x8000003f  Yes  Level 2 data cache hits
PAPI_L2_DCA  0x80000041  No   Level 2 data cache accesses
PAPI_L3_DCA  0x80000042  Yes  Level 3 data cache accesses
PAPI_L2_DCR  0x80000044  No   Level 2 data cache reads
PAPI_L3_DCR  0x80000045  No   Level 3 data cache reads
PAPI_L2_DCW  0x80000047  No   Level 2 data cache writes
PAPI_L3_DCW  0x80000048  No   Level 3 data cache writes
PAPI_L2_ICH  0x8000004a  No   Level 2 instruction cache hits
PAPI_L2_ICA  0x8000004d  No   Level 2 instruction cache accesses
PAPI_L3_ICA  0x8000004e  No   Level 3 instruction cache accesses
PAPI_L2_ICR  0x80000050  No   Level 2 instruction cache reads
PAPI_L3_ICR  0x80000051  No   Level 3 instruction cache reads
PAPI_L2_TCA  0x80000059  Yes  Level 2 total cache accesses
PAPI_L3_TCA  0x8000005a  No   Level 3 total cache accesses
PAPI_L2_TCR  0x8000005c  Yes  Level 2 total cache reads
PAPI_L3_TCR  0x8000005d  Yes  Level 3 total cache reads
PAPI_L2_TCW  0x8000005f  No   Level 2 total cache writes
PAPI_L3_TCW  0x80000060  No   Level 3 total cache writes
PAPI_FDV_INS 0x80000063  No   Floating point divide instructions
PAPI_FP_OPS  0x80000066  Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes  Floating point operations; optimized to count scaled single precisi
PAPI_DP_OPS  0x80000068  Yes  Floating point operations; optimized to count scaled double precisi
PAPI_VEC_SP  0x80000069  Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  No   Reference clock cycles
-------------------------------------------------------------------------
Of 50 available events, 17 are derived.

avail.c                                      PASSED
```