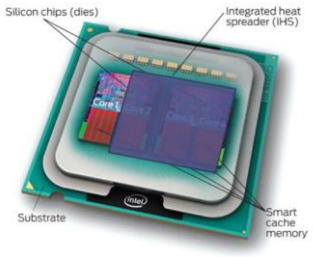
Faculty of Engineering - University of Porto

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

CPD project 1

"Performance evaluation of a single core"



Project CPD 22/23
(L.EC028: Parallel and Distributed Computing)

Bachelor's Degree in Informatics and Computing Engineering

Practical classes:

Professor António Castro

Students (Class 4 Group 11):

Marcos Aires up202006888@fe.up.pt

Pedro Balazeiro up202005097@fe.up.pt

Rúben Viana up202005108@fe.up.pt

Index

Introduction

With this project, we pretend to conduct a study which objective is to analyze the effect of the memory hierarchy on the processor performance (on a single core) when accessing large amounts of data. To achieve this goal, we will use several matrix multiplication algorithms and collect information according to different metrics to understand how these different algorithms may have different effects in the performance of the CPU. We will implement these matrix multiplication algorithms in two distinct programming languages: Julia and C/C++, for comparison purposes. In the case of C/C++, we will take advantage of the Performance API (PAPI) to collect relevant performance indicators of the program execution.

Problem description and algorithms explanation

To conduct this study, we will approach the problem, product of two matrices in three different ways showed below:

1. Basic Multiplication

Problem description:

The basic/standard matrix multiplication algorithm implementation in C/C++ was already given to us. We implemented the same algorithm in Julia and, for both languages, we registered the execution times for matrixes from 600x600 to 3000x3000 with increments of 400 in size of both number of lines and columns (note that only square matrices were used in the whole project).

Algorithm explanation:

The algorithm is composed by three nested for-loops: the first one iterates through the "lines" of the first matrix, the second one through the "columns" of the second matrix. Inside this loop a temporary variable initialized to 0 is used to store the sum of the corresponding elements of a row in first matrix and a column in second matrix. The third one iterates over the elements of the selected row in first matrix and the selected column in second matrix (the variable k is used as the index for the elements in this loop). So concluding, the algorithm multiplies the element in position k in the current line by the element in position k in the current column, and then adds the result to the temporary variable. When we complete the innermost loop, the value assigned is stored in the current position, defined by the line and the column, in the result matrix. The process is repeated for all rows and columns reaching the final matrix product wanted.

Complexity analysis:

The time complexity of this algorithm for matrices is O(n^3), where n is the size of the matrix, because it requires three nested loops to iterate over all elements of the matrices and do the required addition and multiplication operations.

The space complexity is O(3*n^2), because it requires store space for the two input matrices and the output matrix that are used. The space complexity could be

reduced by overwriting one of the input matrices with the result obtaining the output matrix when the algorithm is completed.

2. Line Multiplication

Problem description:

The second one was implemented using line multiplication. We implemented it in C/C++ and in Julia again and performed the same execution time tests. This time additional test for C++ with matrices from 4096x4096 to 10240x10240 and increments of size 2048 were done.

Algorithm explanation:

The algorithm is composed by three nested for-loops: the first one iterates through the "lines" of the first matrix, the second one iterates over the elements of the current line of the first matrix and the innermost matrix iterates over the "columns" of the second matrix. The variable k is now used just for the position of the elements in the first matrix lines and the variable j is now used for the position of elements in the second matrix columns. So concluding, the algorithm multiplies the element in position k of the current line in the first matrix with the element in position j of the current column in the second matrix and then adds the result to the corresponding element in the current line and column of the output matrix. The process is repeated for all rows and columns reaching the final product wanted. The difference between this algorithm and the first one is that the resulting product is directly added to the corresponding element in the output matrix instead of using a temporary variable to store the sum of the products.

Complexity analysis:

Although the order of the loops changes for line multiplication, the time and space complexity remain the same as basic multiplication, because the algorithm still performs the same number of operations and requires the same storage space.

3. Block Multiplication

Problem description:

The third one was implemented using block multiplication, but this one was only implemented in C++ and the same execution time tests from the second step (the additional ones) were done for different block sizes (128, 256 and 512).

Algorithm explanation:

The algorithm is composed by 3 nested for-loops plus 2 different nested for-loops (5 nested for loops in total): the first two iterate through the "lines" of the first matrix and "columns" of the second matrix respectively, the last two divide the matrix in blocks and do the same as line multiplication with the help of the third for-loop (the variable k and j have the same purpose and the variable i allows to go through all lines). The process is repeated for all submatrices.

Complexity analysis:

The time complexity of this algorithm in the worst case is O(n^3*bk^2), where n is the size of the matrix and bk is the block size used to divide the matrices, because it needs three nested for-loops to iterate over all submatrices (and then elements) of the two matrices and then needs two different nested for-loops to iterate through the submatrices themselves using line multiplication. The space complexity is the same as the other two because the storage space required is the same.

Note that, in all three algorithms, the matrices are represented as single dimension arrays. This is because the elements of the matrices are being accessed as if they were in a straight line.

Performance metrics

As referred in the introduction performance metrics were used to achieve our goals. To compare the two programming languages above mentioned we used the execution time of the algorithms, which is a very good performance metric because it measures the total time taken by a program to complete its task. The main objective of most programs is to complete their tasks as quickly as possible and execution time is a way to evaluate the effectiveness of developers optimization efforts, compare the different algorithm implementations (when comparing inside the language) and compiler options, libraries, built-in features for efficient memory management, … (when comparing the two languages). While using PAPI for C++ two relevant performance indicators were collected: data cache misses (for level 1 and 2) and data cache accesses (for level 2 and 3). These metrics can indicate how well a program is utilizing the memory hierarchy of the computer architecture and provide insight into how effectively it is using the data cache. The memory hierarchy includes various levels of memory with different speeds and sizes, such as registers, L1 cache, L2 cache, and main memory. The data cache is a small, fast memory that is used to store frequently accessed data. When a program accesses data that is not already in a particular level, it causes a cache miss, and the data must be fetched from a slower level of memory. This can be a slow operation resulting in a significant performance penalty. If the program is causing many cache misses, the cache is not being efficiently used or the cache size may be insufficient for the program's needs. By measuring the number of data cache accesses, we can understand how well it is using the cache. If the number of cache accesses is low, it may indicate that the cache is not being effectively used and that we are doing more accesses to the main memory, which is slower. On the other hand, if that number is high, it may indicate that the program is making good use of it and that the cache size is appropriate for the program needs.

Results and analysis

1. Basic Multiplication

C++ results:

SIZE (LINES X COLUMNS)	TIME (SECONDS)	L1 DCM	L2 DCM	L2 DCA	L3 DCA
600 X 600	0,249	244513353	38177587	218983621	38177587
1000 X 1000	1,406	1223755839	315611293	1103824500	315611293
1400 X 1400	4,707	3523280385	1624873472	3205693912	1624873472
1800 X 1800	22,814	9089426043	7530546632	8445284019	7530546632
2200 X 2200	49,163	17652220733	24012323563	16390049157	24012323563
2600 X 2600	86,434	30898381176	53111767241	28693177100	53111767241
3000 X 3000	149,017	50301812980	1,00431E+11	46595636164	1,00431E+11

Julia results:

SIZE (LINES X COLUMNS)	TIME (SECONDS)
600 X 600	0,483
1000 X 1000	2,330
1400 X 1400	7,108
1800 X 1800	23,396
2200 X 2200	48,856
2600 X 2600	86,156
3000 X 3000	144,712

Analysis:

Julia is a high-level dynamic programming language designed to be used for numerical and scientific computing while C++ is a low-level compiled language. As we can see the results for both languages using basic multiplication are very similar and the reason behind that may be because both can generate optimized machine code for the underlying hardware. So it means that the optimization level in this case is very similar.

As we can see as the size of the matrices increases the time increments drastically and that is because the number of operations increases, which results in the potential increment of the number of data cache misses due to the reasons mentioned in the performance metrics for this case. The data cache accesses also increment due to the number of operations and increase in memory used.

2. Line Multiplication

C++ results:

SIZE (LINES X COLUMNS)	TIME (SECONDS)	L1 DCM	L2 DCM	L2 DCA	L3 DCA
600 X 600	0,131	27137987	58862443	1746582	58862443
1000 X 1000	0,589	125848366	265124319	9503303	265124319
1400 X 1400	1,978	346669092	709922586	28154075	709922586
1800 X 1800	4,067	745970797	1522914038	63367204	1522914038
2200 X 2200	7,765	2077554720	2762244264	303969997	2762244264
2600 X 2600	12,759	4412574954	4480157692	757355548	4480157692
3000 X 3000	19,759	6779132112	6868811945	1197399470	6868811945

Additional C++ results:

4096 X 4096	51,106	17717778153	17439666044	5122417870	17439666044
6144 X 6144	176,046	59771270792	59722313402	17695226582	59722313402
8192 X 8192	416,299	1,41521E+11	1,38328E+11	40619683878	1,38328E+11
10240 X 10240	806,054	2,76388E+11	2,72354E+11	80742902417	2,72354E+11

Julia results:

SIZE (LINES X COLUMNS)	TIME (SECONDS)
600 X 600	0,370
1000 X 1000	1,650
1400 X 1400	4,705
1800 X 1800	10,354
2200 X 2200	19,719
2600 X 2600	32,213
3000 X 3000	47,890

Analysis:

As we can see in this method, comparing to basic multiplication the time for execution is way lower for the same matrices and that is because in line multiplication it takes advantage of the CPU caching and reduces the memory access. As we saw before in the algorithms explanation, each row of the resulting matrix is computed independently for line multiplication. That leads to much faster computation and the number of memory accesses is reduced for the same number of operations. So normally the DCM (Data Cache Misses) and the DCA (Data Cache Accesses) will be greatly reduced with this method because cache is being well used. For the same reasons in basic multiplication time execution increases abruptly with the size increment. Comparing to Julia, C++ has greater benefits from this method and that may be for different reasons: C++ code is translated to machine code before it is executed, while Julia compiles the code during runtime; C++ is low-level language and that provides direct access to the hardware and memory while Julia is a high-level language designed for ease of use and rapid development which can cause some penalty on performance; although both have libraries of well-optimized numerical that are used for matrix operations, the C++ libraries may be more optimized…

3. Block Multiplication

C++ results (block size = 128):

SIZE (LINES X COLUMNS)	TIME (SECONDS)	L1 DCM	L2 DCM	L2 DCA	L3 DCA
4096 X 4096	44,802	9934978482	32071055500	1820657776	32071055500
6144 X 6144	155,977	33515067157	1,07122E+11	6126645141	1,07122E+11
8192 X 8192	420,161	79488632101	2,54921E+11	14037805628	2,54921E+11
10240 X 10240	725,745	1,55155E+11	4,96453E+11	28333511263	4,96453E+11

C++ results (block size = 256):

SIZE (LINES X COLUMNS)	TIME (SECONDS)	L1 DCM	L2 DCM	L2 DCA	L3 DCA
4096 X 4096	41,188	9142467378	22198093230	1164194874	22198093230
6144 X 6144	148,501	30855987830	75603631782	3924926285	75603631782
8192 X 8192	476,155	73362639306	1,68169E+11	12524951606	1,68169E+11
10240 X 10240	630,478	1,42846E+11	3,44831E+11	18175746337	3,44831E+11

C++ results (block size = 512):

SIZE (LINES X COLUMNS)	TIME (SECONDS)	L1 DCM	L2 DCM	L2 DCA	L3 DCA
4096 X 4096	42,093	8759605647	19655788444	904831148	19655788444
6144 X 6144	143,182	29612639164	66376212069	2978290152	66376212069
8192 X 8192	409,074	70232091061	1,49227E+11	12815542510	1,49227E+11
10240 X 10240	636,426	1,36868E+11	3,16568E+11	12789904155	3,16568E+11

Analysis:

As we can see when the block size increases the time has tendency to decrease and the number of DCM and DCA to. Also, comparing with the results of line multiplication, the time execution, DCM and DCA are a little better (but similar). The reason behind that may lie in the fact that when using block multiplication, a smaller portion of the matrix is being accessed at any given time, reducing the number of cache misses, and increasing the probability that the required data is already present in cache. But we must be careful, although larger block sizes may mean that more data is processed in each operation, reducing the number of operations required overall and resulting in fewer cache misses (leading to better performance and decreasing the execution time), there is a limit to how large the block can be before the cache memory becomes overloaded and then the cache misses increase again. Therefore, it is important to choose an optimal block size that maximizes the cache utilization and minimizes the execution time.

Note: The L2 DCM sometimes is higher than the L1 DCM because L2 cache is larger and farther from the processor than L1 cache. When the processor accesses the L1 cache and the data is not there it goes to L2 cache, so if the data is also not there it goes to a higher level until it finds what he wants and sometimes this leads to more L2 cache misses. This happens above because the size of matrices increases and with that DCM and DCA too.

Note: The values for L2 DCM and L3 DCA are the same, the response to this may be that the L3 cache is fully inclusive of the L2 cache which means that any data stored in the L2 cache is also stored in L3 cache. So, when there is a miss in the L2 cache there is a corresponding access to the L3 cache.

Conclusions

In summary, the choice of algorithm and programming language can have a significant impact on the performance of a single core, and it is important to carefully consider these factors when developing software for performance-critical applications. We should always choose the options available that are more well-suited to the problem at hand.

References

https://moodle.up.pt/course/view.php?id=1970
https://math.stackexchange.com
https://icl.utk.edu/papi/docs
https://julialang.org
https://www.quora.com/Is-C-faster-than-Julia
https://www.hardwaretimes.com/difference-between-l1-l2-and-l3-cache-how-does-cpu-cache-work/
https://en.wikipedia.org/wiki/Cache_inclusion_policy