sciendo

VOLUME 17, N° 4 2023
Journal of Automation, Mobile Robotics and Intelligent Systems

Matrix Transposition Algorithm Using Cache Oblivious

Submitted: 11th January 2023; accepted: 18th March 2023

Samuel Guzmán López, Adolfo Javier San Gil Santana, Jorge Alberto Cuba Alonso del Rivero, Sonia Pérez Lovelle, Humberto Díaz Pando

DOI: 10.14313/JAMRIS/4-2023/25

Abstract:
The Parallel and Distributed Computing group belonging to the Integrated Technological Research Complex (CITI). has been engaged in the creation of general-purpose components that support the processing of large volumes of information that characterize the problems involved in parallel computing.

Using the oblivious cache model, which works independently of the computer architecture, and the divide and conquer principle, an algorithm for matrix transposition is implemented to reduce the execution time of this algebraic operation. The algorithm ensures that most of the data content is loaded to the cache for fast processing, and makes the most of its stay in the cache to minimize missed reads and achieve greater speed.

The work includes conclusions and statistical tests carried out from experiments on computers with different architectures, reflecting the superiority of the algorithm that uses oblivious cache from an order of matrix determined according to the characteristics of each PC.

Keywords: *Cache oblivious, Matrix transposition, Missed readings*

1. Introduction

The Integrated Technological Research Complex (CITI) was created as a coordination project between the *Technological University of Havana (*CUJAE) and the Ministry of Interior (MININT). This entity is designed to host the most advanced technologies being worked with worldwide [1].

CITI's mission is to develop technologies to enhance the security and internal order of the country. Its vision is to be a creative, innovative and benchmark organization in human capital management. In addition, to be a reference in the applicability of the results obtained in the development of systems, technologies and innovative integrated applications, with impact on security and internal order, for which it will base its work on the integration of highly qualified professionals with talented students [1].

At CITI there are projects in which matrix transposition is intensively used, so this task was assigned to the Parallel and Distributed Computing group, which is dedicated to reduce the execution time of various algorithms by employing parallelism and recurrence techniques. This time, the technique selected by the group was the cache oblivious, a recurrent technique about which there is some literature and implementation tested and documented by other authors [2–4]. This method was used by the authors in a research work on matrix multiplication obtaining good results [5].

2. Caching Algorithms

2.1. Cache-aware Algorithms

Cache-aware algorithms take into account the hardware architecture on which they are running, mainly the cache architecture, i.e. the number of cache levels and the size of each level. They are specifically developed to perform as well as possible in the environment for which they were created.

This poses a problem when changing the environment, since if a cache-aware algorithm is executed outside the architecture for which it was designed, it will not perform well. To counteract this problem, cache oblivious algorithms were created, able work efficiently on any architecture [6].

2.2. Cache Oblivious Algorithms

Cache oblivious algorithms have a design that will always be "cache-optimal", regardless of the cache hierarchy. In 1996, the idea of realizing algorithms that do not take into account the architecture of the computer where they are executed was conceived by Charles E. Leiserson and called cache oblivious algorithms. This topic was first published in 1999 by Harald Prokop in his master's thesis at the Massachusetts Institute of Technology [4]. The use of the cache oblivious model has a wide variety of applications such as: matrix multiplication, matrix transposition, Bioinformatics (RNA secondary structure prediction), Shortest Path Algorithm with order O(n), dynamic programming of the Gaussian solution (Numerical Mathematics).

The use of the cache oblivious model aims to decrease missed reads or cache misses since these algorithms use the divide-and-conquer principle to divide the problem into small subproblems until a cache-fitting size is reached, regardless of the size of the cache.

By reducing the number of missed reads or cache misses, execution times are significantly reduced, resulting in greater efficiency.

One of the features by which it outperforms the traditional cache is self-tuning. In typical cache algorithms, the algorithms require tuning to various cache parameters that are not always available from the manufacturer and are often difficult to extract automatically which hinders code portability whereas in cache oblivious algorithms no such tuning is required, a single algorithm should work well on all machines without any modification [3, 4, 7–9].

2.3. Matrix Transposition

Matrix transposition is a fundamental operation of linear algebra and other computational primitives such as the multidimensional Fast Fourier Transform; it is also applied in numerical analysis in economics, image and graphics processing, as well as being used in cryptographic methods [10].

This seemingly innocuous permutation problem lacks both temporal and spatial locality and is therefore difficult to implement efficiently for matrices with a large volume of data. In fact, there is no temporal locality to exploit, since each element of the matrix is accessed at most once [10].

As far as spatial locality is concerned, the matrix element swaps (i, j) and (j, i) implicit in the transpose semantics, when translated into memory addresses using canonical row-major or column-major ordering, equals the memory localities ni+j and nj+i. Depending on the values of i and j, these may be close or far apart in terms of cache sets or memory pages. Careful scheduling of these swap operations is required to gain any advantage from these multiword cache lines [10].

Explicit transposition of an array into memory can often be avoided by accessing the same data in a different order. For example, software libraries for linear algebra, such as BLAS, generally provide options to specify that certain matrices should be interpreted in transposed order to avoid the need for data movement [10].

Describing the algebraic operation as such, a transposed matrix is the result of rearranging the original matrix by exchanging rows for columns in a new matrix (see Figures 1 and 2).

In other words, the transposed matrix is the action of selecting rows from the original matrix and rewriting them as columns in the new matrix.

Examples:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^t = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^t = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Figure 1. Example of transposition of a square matrix and another of order 2x3 (taken from [11])

$$\begin{bmatrix} 0 & 0 & 4 \\ 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 3 & 2 \\ 0 & 2 & 3 \\ 0 & 3 & 4 \\ 3 & 3 & 1 \end{bmatrix}^t = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 3 & 2 & 3 & 3 \\ 4 & 4 & 0 & 2 & 3 & 4 & 1 \end{bmatrix}$$

Figure 2. Example of transposition of a matrix of order 3x7 (taken from [11])

2.4. Transposition of Block Matrices

The manipulation of matrices with a large number of rows and columns involves big problems, even when they are handled with a computer. Therefore, it is often interesting to know how to decompose a problem using large matrices into smaller problems, i.e., using smaller matrices [11].

The possibility of decomposing a matrix into smaller matrices has many applications in communications, electronics, solving systems of equations, taking advantage of the vector structure of some computers, and so on. And, especially, it gives the possibility of writing a matrix in a more compact way [11].

The blocks are obtained by drawing imaginary vertical and horizontal lines between the elements of the matrix. Their dimension depends on the size of the cache blocks and aims to store as much information as possible.

3. Algorithm Implementation

3.1. Description of Operation

The fundamental idea is to reduce the transpose of a matrix to the transpose of small submatrices. This is achieved by dividing the matrices in a half along their largest dimension until only one matrix transpose that fits in the cache needs to be carried out (in theory, one could further divide the matrices down to a base case of size 1×1, but in practice a larger base case is used, e.g., 16×16, in order to amortize the overhead of calling recursive subroutines) [12].

3.2. Description of the Implementation

In section 2, all the theoretical foundations that support the implementation of a matrix transposition algorithm using the cache oblivious model were presented. Algorithm 1, adapted from the one found in https://es.stackoverflow.com, was used.

This algorithm has four integers and a pointer as parameters, of which the first and third are fundamental to divide the original matrix into small submatrices. The second and fourth refer to the number of rows and columns respectively, while the pointer refers to the result matrix.

Table 1. Computer characteristics

Characteristics	PC1	PC2	PC3	PC4	PC5	PC6	PC7
Processor	Intel(R) Core (TM) i3-5020U CPU @ 2.2GHz 2.2GHz	Intel(R) Celeron (R) CPU G3900 @ 2.8GHz 2.8GHz	Intel(R) Celeron (R) CPU G1840 @ 2.8GHz 2.8GHz	Intel(R) Core (TM) i3-7130U CPU @ 2.7GHz 2.7GHz	Intel(R) Core (TM) i3-4130 CPU @ 3.4GHz	Intel (R) Core (TM) i7-1165G7 @ 2.8GHz 2.8GHz	Intel (R) Pentium (R) CPU G4560 @ 3.50GHz 3.50GHz
RAM	4,00GB Single-Channel DDR3 @798MHz	4.00GB (2.95GB utilizable) DDR4-2133	2,00GB Single-Channel DDR3 @665MHz	8.00GB (7.95GB utilizable) DDR4-2400	8.00 GB DDR3	16.2GB (15.8GB utilizable) DDR4-3200	8.00GB (7.95GB utilizable) DDR4-2400
Type of system	Windows 64 bits	Windows 64 bits	Windows 64 bits	Windows 64 bits	Linux 64 bits	Windows 64 bits	Windows 64 bits
Motherboard	ASUSTek COMPUTER INC.X540LA	Gigabyte Technology Co., Ltd. B85M-DS3H	Gigabyte Technology Co., Ltd. B85M-DS3H	Dell Inc. 02DG7R (U3E1) Versión A00	Gigabyte B85M-DS3H	HP Spectre 14-EA	Gigabyte Ga-H110m-S2h
Cache L1 (instructions)	64KB.	64KB	64KB	64KB.	64KB.	128KB	64KB
Cache L1 (data)	64KB.	64K	64K	64KB.	64KB.	192KB	64KB
Cache L2	512KB	512KB	512KB	512KB	512KB	5MB	512KB
Cache L3	3MB	2MB	2MB	3MB	3MB	12MB	3MB

```
void cachetranspose(int rb, int re, int
cb, int ce, Matrix* T)
{int r = re - rb, c = ce - cb;
 if (r <= 16 && c <= 16) {
  for (int i = rb; i < re; i++) {
   for (int j = cb; j < ce; j++) {
   T->data[j*rows+i]=data[i*columns+j];}}}
   else if (r >= c) {
     cachetranspose(rb,rb+(r/2),cb,ce,T);
     cachetranspose(rb+(r/2),re,cb,ce,T);}
 else {
  cachetranspose(rb,re,cb,cb+(c/2),T);
  cachetranspose(rb,re,cb+(c/2),ce,T);}}
```

Algorithm 1. Recursive matrix transposition algorithm using cache oblivious

4. Experiments

For the development of the experiments it was necessary a previous study of several algorithms (traditional, blocks and blocks_for_squared_matrices) to establish a comparison with those using the cache oblivious model (for square and non-square orders). These experiments consist of running each algorithm 5 times on orders with different characteristics (see Table 1). From the results obtained, a statistical analysis is performed to determine whether the algorithms using the cache oblivious model are superior (in terms of execution time and missed reads) to those that do not use this model.

4.1. Results

In this section we present diagrams showing the average execution time and the missed readings (the latter only in PC5), for each of the algorithms analyzed.

In those cases, where non-squared matrices were tested, these were filled with zeros in order to use the blocks_for_squared_matrices algorithm for the corresponding comparisons.

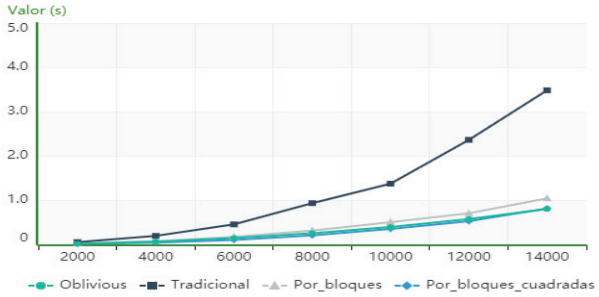

Figure 3. Behavior of the algorithms in PC1 for square orders

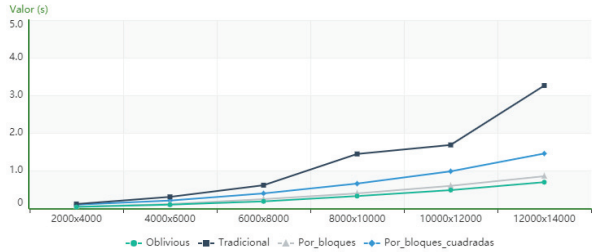

Figure 4. Behavior of the algorithms in PC1 for non-square orders

As can be seen in Figure 3, for the computer identified as PC1, the blocks_for_squared_matrices algorithm is faster than the rest of those analyzed for orders lower than 14000, from which the algorithm using cache oblivious starts to be superior.

Figure 4 shows that for the computer identified as PC1, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

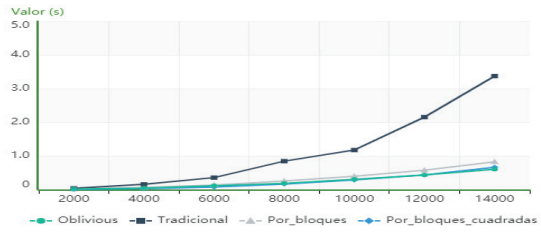

Figure 5. Behavior of the algorithms in PC2 for square orders

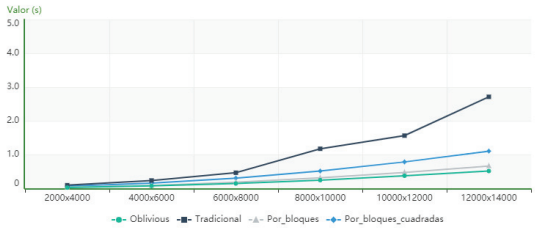

Figure 6. Behavior of the algorithms in PC2 for non-square orders

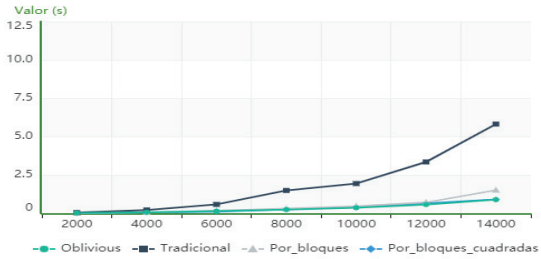

Figure 7. Behavior of the algorithms in PC3 for square orders

It is evident from Figure 5 that, for the computer described as PC2, the algorithm using cache oblivious is superior in terms of execution time to the traditional and block algorithms for all orders, while the blocks_for_squared_matrices algorithm has lower or similar times to the one using cache oblivious up to order 10000, from which the cache oblivious algorithm presents lower values.

Figure 6 shows that, for the computer identified as PC2, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

Figure 7 shows that, for the computer identified as PC3, the algorithm using cache oblivious is superior in terms of execution time to the traditional and block algorithms for all orders, while the blocks_for_squared_matrices algorithm has lower execution times than the one using cache oblivious up to order 6000. Between 8000 and 12000 the results are similar and from the latter, the cache oblivious algorithm starts to have lower values.

Figure 8 shows that, for the computer identified as PC3, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

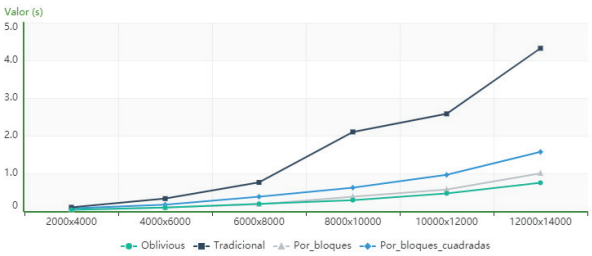

Figure 8. Behavior of the algorithms in PC3 for non-square orders

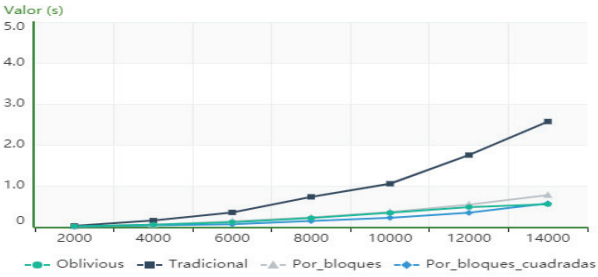

Figure 9. Behavior of the algorithms in PC4 for square orders

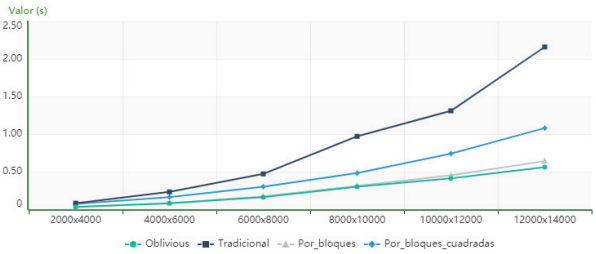

Figure 10. Behavior of the algorithms in PC4 for non-square orders

Figure 9 shows that, for the computer identified as PC4, the algorithm using cache oblivious is superior in terms of execution time to the traditional and block algorithms for all orders, while the blocks_for_squared_matrices algorithm has lower execution times than the one using cache oblivious until order 12000, when they start to have similar values.

Figure 10 shows that, for the computer identified as PC4, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_square_matrices algorithms for all the orders analyzed.

Figure 11 shows that, for the computer identified as PC5, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

Figure 12 shows that, for the computer identified as PC5, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

It is evident in Figure 13 that, for the computer identified as PC6, the algorithm using cache

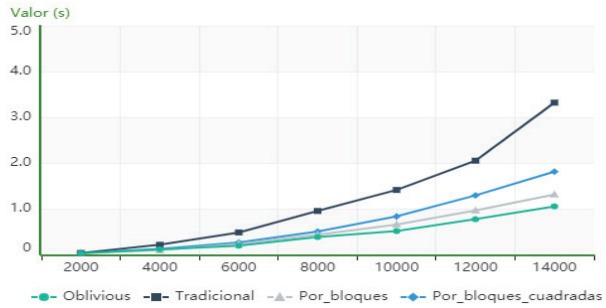

Figure 11. Behavior of the algorithms in PC5 for square orders

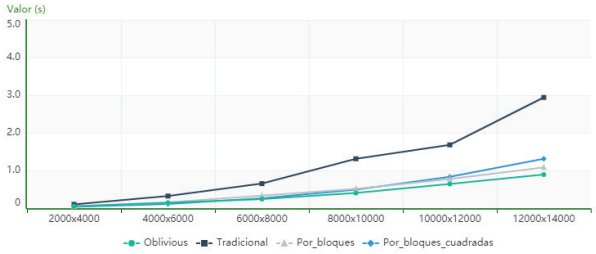

Figure 12. Behavior of the algorithms in PC5 for non-square orders

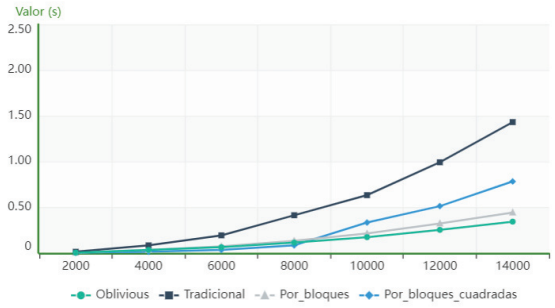

Figure 13. Behavior of the algorithms in PC6 for square orders

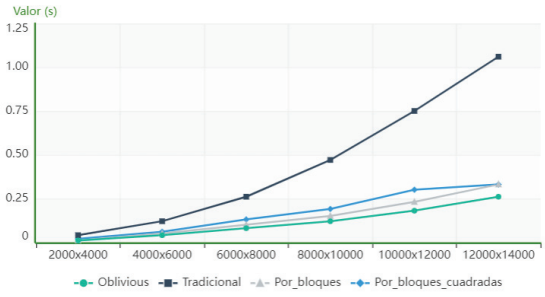

Figure 14. Behavior of the algorithms in PC6 for non-square orders

oblivious is superior in terms of execution time to the traditional algorithms, by blocks and blocks_for_squared_matrices, starting from the order 10000x 10000.

Figure 14 shows that, for the computer identified as PC6, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block

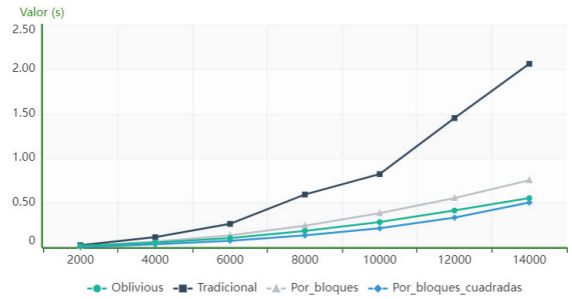

Figure 15. Behavior of the algorithms in PC7 for square orders

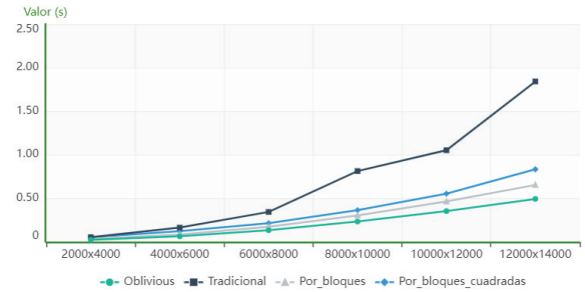

Figure 16. Behavior of the algorithms in PC7 for non-square orders

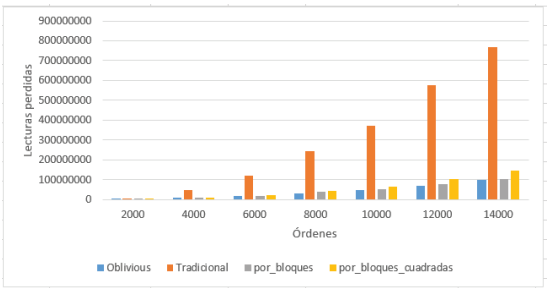

Figure 17. Performance of the algorithms in terms of missed readings on PC5 for square orders

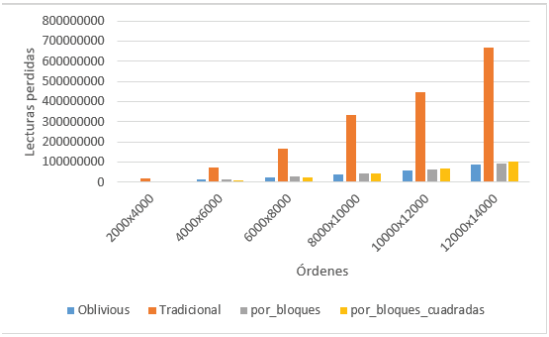

Figure 18. Performance of the algorithms in terms of missed readings on PC5 for non-square orders

and block_for_squared_matrices algorithms for all the orders analyzed.

Figure 15 shows that, for the computer identified as PC7, the blocks_for_squared_matrices algorithm is superior to the others analyzed and it can be observed that the algorithm using cache oblivious obtains a certain parity from the order 14000x14000.

Table 2. Results obtained in PC1 for square orders

Algorithms	traditional		blocks		blocks_for_squared_matrices	
	p-value	Analysis	p-value	Analysis	p-value	Analysis
10000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$1 > \alpha$	Does not reject H_0
12000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$1 > \alpha$	Does not reject H_0
14000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0

Table 3. Results obtained in PC1 for non-squared orders

Algorithms	traditional		blocks		blocks_for_squared_matrices	
	p-value	Analysis	p-value	Analysis	p-value	Analysis
8000x 10000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
10000x 12000	$0.02895 < \alpha$	Rejects H_0	$0.02895 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
12000x 14000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0

Table 4. Results obtained in PC5 for missed readings in square orders

Algorithms	traditional		blocks		blocks_for_squared_matrices	
	p-value	Analysis	p-value	Analysis	p-value	Analysis
10000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
12000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
14000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0

Table 5. Results obtained in PC5 for missing readings in non-square orders

Algorithms	traditional		blocks		blocks_for_squared_matrices	
	p-value	Analysis	p-value	Analysis	p-value	Analysis
10000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
12000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0
14000	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0	$0.03125 < \alpha$	Rejects H_0

Figure 16 shows that, for the computer identified as PC7, the algorithm using cache oblivious is superior in terms of execution time to the traditional, block and block_for_squared_matrices algorithms for all the orders analyzed.

4.1.1. Missed readings

The PAPI (Performance Application Programming Interface) library, developed at the University of Tennessee, was used to account for missed reads. Its main purpose is to provide access to the PMCs (Performance Monitoring Counter) of a diverse collection of modern processors [13]. PAPI provides an abstraction layer that allows developers to access PMCs. Instead, the developer uses calls to the PAPI API (Application Programing Interface), making the code portable, i.e. it can be used on any architecture supported by the library without modifying access to PMCs [14].

The missing readings were counted on the PC5 computer, which has a Linux operating system because the library used (PAPI) has not provided new updates since the XP version of Windows.

Figure 17 shows that, for the computer identified as PC5, the algorithm that uses cache oblivious has the lowest number of missed readings.

Figure 18 shows that, for the computer identified as PC5, the algorithm using cache oblivious has fewer missed readings.

5. Statistical Analysis

The Wilcoxon nonparametric test was used for statistical analysis. It was selected since it was proven that the data do not follow a normal distribution and due to the small sample size. It is expected that, when the test is run, it will return a $p < \propto$ value, if this occurs H_0 is rejected and it is concluded that the execution time of the cache oblivious algorithm is less than that of the traditional algorithm.

Several signed rank tests were applied when the samples were paired, one for each of the last three orders of the algorithms on each computer described.

The following are the results obtained on PC1 in terms of execution time and on PC5 in terms of missed readings:

It is evident in the results of Table 2 that the blocks_for_squared_matrices algorithm is faster than the rest of the analyzed algorithms for orders lower than 14000, from which the algorithm using cache oblivious starts to be superior.

The results in Table 3 show the superiority in terms of execution time of the algorithm using the cache oblivious model for all the orders analyzed.

Table 4 shows the superiority in terms of missed readings of the algorithm using the cache oblivious model for all orders analyzed.

Table 5 shows the superiority in terms of missed readings of the algorithm using the cache oblivious model for all orders analyzed.

The test was performed with the statistical software R. After the test it was demonstrated that the matrix transposition algorithm using the cache oblivious, depending on the architecture of the computer where it was used and from a certain order, will be better than the other algorithms analyzed.

6. Conclusion

Under the computational conditions used for the experiments:

1) On a computer with a Windows operating system, in the matrix transposition operation, for square order matrices it is not feasible to employ the algorithm using the cache oblivious model for an order less than 14000 x 14000.
2) Regardless of the computer architecture, it was shown that from order 6000 x 8000 for non-square order matrices, the matrix transposition algorithm using cache oblivious is faster than the rest of the algorithms studied.
3) The blocks_for_squared_matrices algorithm has a lower performance when used for non-square matrices since these must be completed with zeros until their order is square and therefore the algorithm increases its execution time.
4) For large volumes of information, the execution time is in direct correspondence to the missed readings.
5) Algorithms that use the cache oblivious model for large volumes of information have fewer missed readings than the rest.

AUTHORS

Samuel Guzmán López* – Technological University of Havana José Antonio Echeverría, Cuba, e-mail: samuguzmanlopez97@gmail.com.

Adolfo Javier San Gil Santana – Technological University of Havana José Antonio Echeverría, Cuba, e-mail: asang@ceis.cujae.edu.cu.

Jorge Alberto Cuba Alonso del Rivero – Technological University of Havana José Antonio Echeverría, Cuba, e-mail: jcuba@ceis.cujae.edu.cu.

Sonia Pérez Lovelle – Technological University of Havana José Antonio Echeverría, Cuba, e-mail: sperezl@ceis.cujae.edu.cu.

Humberto Díaz Pando – Technological University of Havana José Antonio Echeverría, Cuba, e-mail: hdiazp@ceis.cujae.edu.cu.

*Corresponding author

References

[1] www.portal.citi.cu. (accessed 4/6/2019).

[2] C. Mayer. “Cache oblivious matrix operations using Peano curves,” Department of Computer Science Technische University Munchen, Germany, 2006.

[3] M. Frigo, Leiserson, H. Prokop, and Ramachandran. “Cache Oblivious Algorithms”, MIT Laboratory for Computer Science, Cambridge, USA, 1999.

[4] H. Prokop. “Cache-Oblivious Algorithms,” Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology., Massachusetts 1999.

[5] A. J. San Gil Santana, S. Guzmán López, and J. A. Cuba Alonso del Rivero. “Algoritmo de multiplicación de matrices utilizando cach inconsciente y curva de Peano,” *XVIII Convención y Feria internacional Informática 2020,* 2020.

[6] T. M. Chilimbi. “Cache Conscious Data Structues Desing and Implementation,” University Of Wisconsin 1999.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-Oblivious Algorithms,” *ACM Transactions on Algorithms,* 2012.

[8] Ritika. “Cache-aware and cache-oblivious algorithms,” Master of Engineering Computer science and engineering, Thapar University Patiala 2011.

[9] S. Neeraj and S. Sandeeep. “E?cient cache oblivious algorithms for randomized divide-and-conquer on the multicore model,” 2018.

[10] S. Chatterjee and S. Sen. “Cache Efficient Matrix Transposition,” Department of Computer Sciense, University of North Carolina Chapel Hill, NC 27599-3175, USA – Indian Institute of Technology New Delhi 110016, India, 2005.

[11] M. Palacios. “Matrices,” Departamento de Matemática Aplicada Universidad de Zaragoza, 2018.

[12] D. Tsifakis, P. Alistair, Rendell, and P. E. Strazdins. “Cache Oblivious Matrix Transposition: Simulation and Experiment,” Department of Computer Science, Australian National University Canberra, Australian, 2004.

[13] V. M. Weaver et al.. “PAPI 5: Measuring Power, Energy and the Cloud,” *International- Symposium on Performace Analysis of Systems and Software*, 2013.

[14] P. J. Mucci, S. Browne, C. Deane, and G. HO. “PAPI: A Portable Interface to Hadware Performance Counters,” University of Tennessee, Knoxville, Tennessee, 1999.