IN-MEMORY PARALLEL PROCESSING FOR MATRIX TRANSPOSE

Mihaela Malita, Ph.D.*
Associate Professor, Department of Mathematics and Computer Science, Rivier University
and
Gheorghe M. Ştefan**
Professor, National University of Science and Technology Politehnica Bucharest, Romania

Abstract

Efficient implementation of parallel algorithms for matrix transposition requires careful consideration of data distribution, synchronization, communication overheads, and load balancing to achieve optimal performance in-memory parallel processing. We propose an algorithm and its implementation for execution on a p-cell parallel in-memory cellular processor (CP). Two versions of CP are considered, one with size in O(p) and another with size in O(p×log p).

Keywords: *parallelism, heterogenous systems, in-memory processing, linear algebra, matrix transpose*

1 Introduction

The transpose of an $m \times m$ matrix is performed with the help of a single-core computing system using a very simple algorithm that is executed in $O(m^2)$ time. Accelerating the execution in a heterogeneous computing system (HOST & ACCELERATOR) makes sense theoretically only if the matrix is already loaded in the accelerator as a result of a previous calculation or it is there for a calculation which follows. Otherwise, a gain in execution time cannot be achieved if it is necessary to transfer the matrix from the HOST memory to the ACCELERATOR and transfer the result back to the HOST memory. The transfer process is carried out in $O(m^2)$ time as well as the transpose under the direct control of the HOST. Even if the transpose operation is accelerated very strongly in the ACCELERATOR, we are dealing with a parallel algorithm that requires a very low *operational intensity* ($I = W/Q$ is the ratio between the work W, the number of operations performed by a given application, and the memory traffic Q, the number of bytes of memory transfers incurred during the execution of the application). For this reason, our approach does not refer to distributed systems, and in the case of parallel systems, we restrict the investigation for the application of the transpose on the matrices already loaded in the (one-chip) accelerator system.

The next section is a short presentation of work already published in literature. The third section introduces the structure and the architecture of the in-memory system we use. The algorithm we propose is presented in the 4-th section. The next section refers to the implementation. The last section provides evaluations and final comments.

2 State of the Art

In [1] is introduced a recursive algorithm which is very elegant, but needs work in $O(m^2 log\ m)$ for a $m \times m$ matrix. The recursive algorithm involves dividing an $m \times m$ matrix into four $m/2 \times m/2$ matrices and switching the upper right matrix by the lower left one. The process is repeated with all the $m/2 \times m/2$ matrices, and so on until 2×2 matrices are reached. This algorithm involves moving each of the matrix

ISSN 1559-9388 (online version), ISSN 1559-9396 (CD-ROM version).

components several times. More precisely, *log*-times half of the number of components of the matrix are moved. This fact leads to the parallel execution time which is higher than that executed sequentially by a single-core system which is in $O(m^2)$.

The same algorithm is used in [4] with emphasis on optimizing the communication cost between processors. In [5] is solved the problem of communication in the specific interconnection networks of distributed systems. In [6] the problem is solved on parallel distributed systems, where the communication between processors is the main issue to be addressed. The mesh parallel organization is investigated in [7]. The implementation of the multi-core system is investigated in [8]. A multi-thread approach is used in [10].

A linear time execution algorithm is presented in [9]. The algorithm uses operations on pseudo diagonals and rotation operations performed in constant time. This algorithm is similar to the one we propose. It is applicable in a general-purpose multi-core system but with specific hardware resources.

3 Our In-Memory Parallel Computing System

Four The von Neumann bottleneck effect can be mitigated by a tighter interleaving between the memory and processing units. The increasingly frequent projects of in-memory computing systems are based on this observation. The structure and architecture of our system will be briefly described below, but only as much as it is necessary to understand how the implementation of the matrix transpose algorithm that we propose works (more details in [11] [14] [13] [15].

3.1 Structure

The structure on which in-memory processing is based assumes a data memory organized as a matrix **M** of *m* lines and *p* columns, of scalars `s[ij]`, which is subject to processing by an array of *p* execution elements. Each execution element operates on the data formed in a column of *m* elements from **M**

```
M[j] = [s[1j] s[2j] ... s[ij] ... s[mj]] for j = 1,2, ..., p.
```

M is seen by the entire array of execution elements as a memory of *m* vectors

```
V[i] = [s[i1] s[i2] ... s[ij] ... s[ip]] for i = 1,2, ..., m.
```

Thus, the considered system is a cellular one in which each cell, C_i for $i = 0.1, ..., p\text{-}1$, contains an execution element and the associated memory, the vector `M[j]`. The system works like a cellular automaton, let's call it a cellular processor, CP. The functions that are executed in each clock cycle are issued by a controller called CONTR.

In this paper we will consider two versions for the array of execution units. Version 1 assumes only direct bidirectional connections between adjacent cells. Version 2 is additionally connected with a Benes-Waxman type permute network that receives from the execution units a vector of *p* components and returns another vector of *p* components that represents a permutation of the received vector. The permutation is specified by a scalar vector stored in **M**.

The size of the system in the case of version 1 is in $O(p)$, while in the case of version 2 is in $O(p \times logp)$. The latency introduced by the permute network is $(-1+2\times log_2 p)$.

The minimal structure of registers in each execution unit C_i, which we consider is:

- `a[i]`: accumulator register as element in the vector `A = [a[0] ... a[p-1]]` distributed along the cellular system
- `addr[i]`: address register as part of the vector `ADDR = [addr[0] ... addr[-1]]` distributed along the CP.

The accumulator register can be loaded with a value generated by CONTR or loaded from **M**, its content can be stored in a location of **M**. Also, the contents of the accumulator can be processed logically or arithmetically with a value generated by CONTR or received from **M**.

3.2 Architecture

For the transpose operation we are investigating, we exemplify from Instruction Set Architecture of CP the following sub-set of instructions, issued in each clock cycle by CONTR and executed by CP:

```
ADDRLD            : addr[i] <= a[i]
IXLOAD            : a[i] <= i; IX = [0,1, ..., p-1]
VLOAD(value)      : a[i] <= value
LOAD(value)       : a[i] <= mem[value]
RLOAD(value)      : a[i] <= mem[value+addr(i)]
RILOAD(value)     : a[i] <= mem[value+addr(i)]; addr[i] <= value+addr(i)
STORE(value)      : mem[value] <= a[i]
RSTORE(value)     : mem[value+addr(i)] <= a[i]
RISTORE(value)    : mem[value+addr(i)] <= a[i]; addr[i] <= value+addr(i)
REM(value)        : a[i] <= reminder(a[i]/value)
SHIFTL            : if (i=p-1) ? a[i] <= 0 : a[i] <= a[i+1]
SHIFTR            : if (i=0) ? a[i] <= 0 : a[i] <= a[i-1]
RSEND(value)      : PERMUTE <= {A, mem[addr[i]+value]} = {vector, pattern}
GET               : A $<$= PERMUTE
WAIT(value)       : a[i] <= a[i] during (value) cycles
```

In addition to these functions, there are the typical logic & arithmetic operations. The control of issuing the previously listed instructions by CONTROL to the CP is carried out by the specific functions of a mono-core processor.

4 Algorithm

To describe the proposed algorithm, we will define the concept of *pseudo diagonal* of a square matrix.

Definition 1. Let be the square matrix

$$\begin{bmatrix} v_{00} & v_{01} & \cdots & v_{0\,m-1} \\ v_{10} & v_{11} & \cdots & v_{1\,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m-1\,0} & v_{m-1\,1} & \cdots & v_{m-1\,m-1} \end{bmatrix}$$

where

$$PD(k) = [v_{(0+k)_{mod\,m}\,0}\ v_{(1+k)_{mod\,m}\,1} \cdots v_{(m-1+k)_{mod\,m}\,m-1}]$$

is the *k*-th *pseudo-diagonal* in the matrix **M**. (The 0-th pseudo-diagonal is the main diagonal.)

◊

Example 1. *In the following* 5×5 *matrix, each pseudo diagonal is emphasized by the value it contains. The first pseudo diagonal is filled-up with 1s, the second with 2s, and so on.*

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

◊

The transpose algorithm for matrix **V** of components v(i,j), into the matrix **W** of components w(i,j), for $i, j = 0,1, \ldots, m-1$ is the following:

```
/*****************************************************************************
    Algorithm for matrix transpose
        Initial matrix, V, with components: v(i,j), for i,j = 0, 1, ... m-1
        Final matrix, W, with components:   w(i,j), for i,j = 0, 1, ... m-1
        Working vector: A = [a[0] a[1] ... a[m-1]]
    Note: addition and subtract operations are modulo m operations
*****************************************************************************/
for(k=0; k<m; k=k+1) begin
  (1) A <= [v(0+k,0) v[1+k,1]...v[m-1+k,m-1] // A <= PD(k.V)
  (2) A <= [a(0-k) a(1-k) ... a(m-1-k)]      // A <= (A rotate k positions right)
  (3) for(j=0; j<m; j=j+1) w(j+k,j) <= a(j)  // PD(m-k,W) <= A
end
```

Steps (1) and (3) in the main loop of the previous algorithm are performed in CP in one clock cycles.

For the second step, there are two solutions:

- the k position rotate can be performed using shift operations in a number of cycles belonging to $O(k)$
- the k position rotate can be performed using a permute operation with a latency of $(-1+2\times log_2 p)$ clock cycles.

When $m < p$ in CP $\lfloor p/m \rfloor$ $m\times m$ matrices can be distributed stored in m vectors from **M**.

Example 2. *Let us consider the transpose of a* 4×4 *matrix:*

```
    Initial                                                          Final
                                 k = 0    k = 1    k = 2    k = 3
V:  0 1 2 3                                                      V:  0 1 2 3
    4 5 6 7     (step 1) 0 5 A F  4 9 E 3  8 D 2 7  C 1 6 B          4 5 6 7
    8 9 A B                                                          8 9 A B
    C D E F     (step 2) 0 5 A F  3 4 9 E  2 7 8 D  1 6 B C          C D E F

W:  x x x x     (step 3) 0 x x x  0 4 x x  0 4 8 x  0 4 8 C      W:  0 4 8 C
    x x x x              x 5 x x  x 5 9 x  x 5 9 D  1 5 9 D          1 5 9 D
    x x x x              x x A x  x x A E  2 x A E  2 6 A E          2 6 A E
    x x x x              x x x F  3 x x F  3 7 x F  3 7 B F          3 7 B F
```

◊

5 Implementations

The algorithm for matrix transpose is translated into the program running on CP and developed in the assembly language previously sketched.

For implementation, we will consider a number of $\lfloor p/m \rfloor$ of $m \times m$ matrices stored in m horizontal vectors in memory **M**, as follows:

```
V[1]=[a[11] a[12] ... a[1m] ... b[11] b[12] ... b[1m] ... c[11] c[12] ... c[1m] ... ]
V[2]=[a[21] a[22] ... a[2m] ... b[21] b[22] ... b[2m] ... c[21] c[22] ... c[2m] ... ]
 ...   ...   ...  ...  ...  ...  ...   ...  ...  ...  ...  ...   ...  ...  ...  ...
V[m]=[a[m1] a[m2] ... a[mm] ... b[m1] b[m2] ... b[mm] ... c[m1] c[m2] ... c[mm] ... ]
```

A, of components `a[ij]`, **B**, of components `b[ij]`, **C**, of components `c[ij]`, and so on, are stored in vectors `V[1], ..., V[m]` of the memory **M**. The program that runs on the CP does not depend on the number of matrices transposed in parallel, nor on the size m that defines them.

Depending on the number of cells, p, and the size, m, of the matrices to be transposed, the two versions for performing the rotation operation (step 2) must be analyzed independently.

5.1 Shift-based rotation solution

The matrix transpose algorithm is translated into the program developed in assembly language outlined in subsection 3.2. The macros use instructions of the type briefly presented in subsection 3.2.

```
/*****************************************************************************
    Program for matrix transpose
    Initial: matrices are stored in the first m vectors of M
    Final: the result
*****************************************************************************/
macro 1: compute in each cell reminder(index/m) = column index in each matrix
macro 2: expand initially loaded matrices to emphasize pseudo diagonals
macro 3: read pseudo diagonals & load them as lines
macro 4: rotate right each line according to its index k
macro 5: expand the resulting matrix to emphasize pseudo diagonals of the result
macro 6: compose the matrix by restoring the pseudo diagonals in their positions
```

Example 3. In order to simplify the understanding of the simulation result, the same matrix (the one used in the previous example), is initially loaded $\lfloor p/m \rfloor$ times into m vectors of **M** for $m = 4$ and $p = 16$, as follows:

```
Initial:
vect[0] = [0  1  2  3     0  1  2  3     0  1  2  3     0  1  2  3]
vect[1] = [4  5  6  7     4  5  6  7     4  5  6  7     4  5  6  7]
vect[2] = [8  9  a  b     8  9  a  b     8  9  a  b     8  9  a  b]
vect[3] = [c  d  e  f     c  d  e  f     c  d  e  f     c  d  e  f]
```

The program will calculate the transposes of the four initial matrices in parallel.

The indexes of columns of the $\lfloor p/m \rfloor$ *matrices are computed starting from the index vector* `IX = [0 1 ... p-1]`, *with the following result:*

```
macro 1:
vect[15] = [0  1  2  3     0  1  2  3     0  1  2  3     0  1  2  3]
```

The vector `[15]` *is loaded as ADDR to be used for accessing pseudo diagonals in* **M**.

By copying the first m–1 lines of each matrix immediately after the initial vectors (using the suggestion offered by Sarrus' rule used in the calculation of determinants), the pseudo diagonals are easy emphasized to be read using RILOAD instructions with ADDR initialized with the content of `vect[15]`*:*

```
macro 2:
vect[0] =  [0  1  2  3    0  1  2  3    0  1  2  3    0  1  2  3]
vect[1] =  [4  5  6  7    4  5  6  7    4  5  6  7    4  5  6  7]
vect[2] =  [8  9  a  b    8  9  a  b    8  9  a  b    8  9  a  b]
vect[3] =  [c  d  e  f    c  d  e  f    c  d  e  f    c  d  e  f]
vect[4] =  [0  1  2  3    0  1  2  3    0  1  2  3    0  1  2  3]
vect[5] =  [4  5  6  7    4  5  6  7    4  5  6  7    4  5  6  7]
vect[6] =  [8  9  a  b    8  9  a  b    8  9  a  b    8  9  a  b]
```

The pseudo diagonals are selected from `vect[0], ..., vect[6]` *and loaded in* **M** *as follows:*

```
macro 3:
vect[16] = [0  5  a  f    0  5  a  f    0  5  a  f    0  5  a  f]
vect[17] = [4  9  e  3    4  9  e  3    4  9  e  3    4  9  e  3]
vect[18] = [8  d  2  7    8  d  2  7    8  d  2  7    8  d  2  7]
vect[19] = [c  1  6  b    c  1  6  b    c  1  6  b    c  1  6  b]
```

Right rotate operations are applied k times according to the index of line:

```
macro 4:
vect[16] = [0  5  a  f    0  5  a  f    0  5  a  f    0  5  a  f]
vect[17] = [3  4  9  e    3  4  9  e    3  4  9  e    3  4  9  e]
vect[18] = [2  7  8  d    2  7  8  d    2  7  8  d    2  7  8  d]
vect[19] = [1  6  b  c    1  6  b  c    1  6  b  c    1  6  b  c]
```

Vector `vect[16]` *is rotated 0 times,* `vect[17]` *is rotated 3 times,* `vect[18]` *is rotated 2 times, and* `vect[19]` *is rotated once. The operation is performed based on SHIFTL and SHIFTR instructions in* $m^2+21m-20 \in O(m^2)$ *number of cycles.*

The lines are loaded as pseudo diagonals over a space of $(2m-1)$ *vectors initialized with zeros:*

```
macro 5:
vect[8]  = [0  0  0  0    0  0  0  0    0  0  0  0    0  0  0  0]
vect[9]  = [1  5  0  0    1  5  0  0    1  5  0  0    1  5  0  0]
vect[10] = [2  6  a  0    2  6  a  0    2  6  a  0    2  6  a  0]
vect[11] = [3  7  b  f    3  7  b  f    3  7  b  f    3  7  b  f]
vect[12] = [0  4  8  c    0  4  8  c    0  4  8  c    0  4  8  c]
vect[13] = [0  0  9  d    0  0  9  d    0  0  9  d    0  0  9  d]
vect[14] = [0  0  0  e    0  0  0  e    0  0  0  e    0  0  0  e]
```

The final form is obtained by composing the resulting matrix by restoring the pseudo diagonals in their positions:

```
macro 6:
vect[8]  = [0  4  8  c    0  4  8  c    0  4  8  c    0  4  8  c]
vect[9]  = [1  5  9  d    1  5  9  d    1  5  9  d    1  5  9  d]
vect[10] = [2  6  a  e    2  6  a  e    2  6  a  e    2  6  a  e]
vect[11] = [3  7  b  f    3  7  b  f    3  7  b  f    3  7  b  f]
```

◊

The operation that consumes the largest number of cycles is the rotation, because it is performed by combining shifting operations in order to obtain rotations on sub-vectors of any size, depending on the side of the considered matrices.

5.2 Permute-based rotation solution

If the system contains a permute network (for details see [3] [2]), the section `macro 4` of the previously described algorithm is executed faster. The program is designed and optimized in two steps.

5.2.1 Permute-based rotation limited by the latency of the permute network

The sequence of operations for rotation involving the permute network is the following:

```
/*****************************************************************************
    Rotate using permute network with O(log p) latency
*****************************************************************************/
    RLOAD(k);                // select the line to be rotated
    RSEND(6m-1);             // load permute network with the permute vector
    WAIT(-1+2Log_2(p));      // wait the propagation through permute network
    GET;                     // ACC <= PERMUTE_OUTPUT
    RILOAD(1);               // store the rotated lines back in memory
```

The execution time for a rotate operation results $3+2\times log_2 p \in O(log\ p)$, the entire macro being executed in number of cycles in $O(m\times log\ p)$. The acceleration obtained using the permute network is limited by the latency it introduces. Indeed, for each permutation a number of $(-1+2\times log_2 p)$ must be spent for waiting the result of the permutation operation.

5.2.2 Permute-based rotation avoiding the latency of the permute network

Instead of staying idle while waiting for the result of the permute network, we can choose to insert into the permute network a number equal to $(-1+2\times log_2 p)$ lines from matrix **M**. Meanwhile, usually after one clock cycle, the result of the first permutation is accessible at the output of the permute network. Then, the results of the initiated permutations will be available rhythmically. This approach allows us to avoid the effect of the latency introduced by the permute network. The loop repeated $\lceil m/(-1+2\times log_2 p)\rceil$ times is the following:

```
/*****************************************************************************
    Rotate avoiding permute network O(log p) latency
*****************************************************************************/
// INSERT in permute network (-1+log_2 p) lines to be rotated
    RILOAD(1);                // select the line to be rotated
    RSEND(2m-1);              // load permute network with the permute vector
    RILOAD(1);
    RSEND(2m-1);
        ...
    RILOAD(1);
    RSEND(2m-1);
    RILOAD(-m/(-1+log p));    // restore address register
// EXTRACT (-1+log_2 p) lines from the output of the permute network
    GET                       // ACC <= PERMUTE_OUTPUT
    RISTORE(1);               // store the rotated lines back in memory
    GET
    RISTORE(1);
        ...
    GET
    RISTORE(1);
```

The execution time for rotate operations for all the m lines of matrices is $4m+const \in O(m)$.

6 Evaluations

We will evaluate the performances for the two implementations – the one with and without the switching network – and in two different situations, with the matrices in **M** or with the matrices in the system's external memory.

In the situation where the matrices to be transposed result from a calculation made in the accelerator, it is no longer necessary to load them from the system's external memory. And if the transposition result is used in a subsequent immediate calculation, the transfer time in which the matrices are loaded from and after transposition stored in the system memory, outside the accelerator, will not be taken into account.

For this case, the execution time when using shifting operation for rotate, expressed in number of clock cycles and confirmed by the running simulation, is:

$$m^2 + 37m + 8(log_2 p - \lfloor log_2 m \rfloor - 1) - 6 \in O(m^2)$$

(The number p is usually a power of 2.) Using the permute network for rotations, the execution time is:

$$22m + 8(log_2 p - \lfloor log_2 m \rfloor) + 8 \in O(m)$$

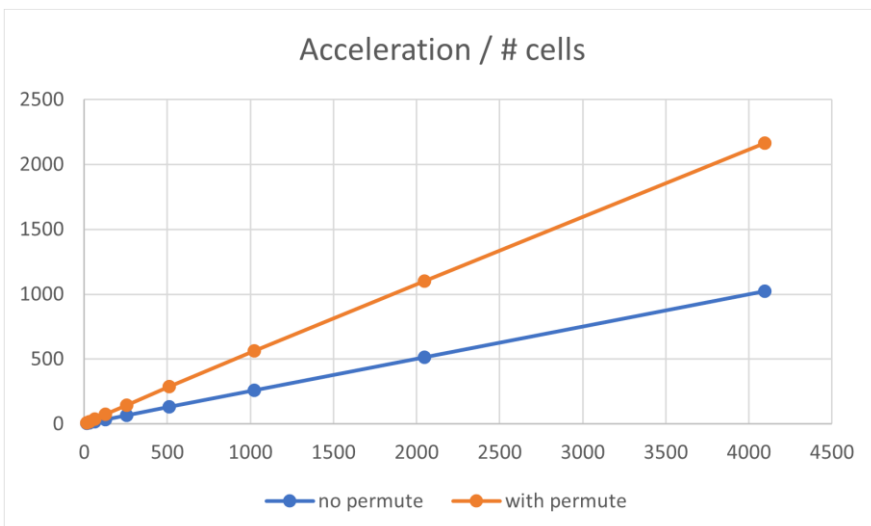

Figure 1: Acceleration for 16×16 matrices without the transfer time.

Because the execution time on a system with p cells, with size in $O(p)$, without a permute network belongs to $O(m^2)$, the acceleration belongs to

$$\alpha^p(m) \in O(\lfloor p/m \rfloor) = O(p/m)$$

The execution time on a system with p cells, with size in $O(p \times \log p)$, and permute network belongs to $O(m)$, thus acceleration belongs to

$$\alpha^p_{permute}(m) \in O(m \times \lfloor p/m \rfloor) = O(p)$$

In Figure 1 we consider the acceleration for 16 ×16 matrices transposed in accelerators with p cells, p = 16, 32, …, 4096. The acceleration is evaluated using a program running on a mono-core x86. The number of cycles for each of the $m \times m$ elements of the matrix is 14 (according to Appendix A and B).

If we have to take into consideration the transfer time to and from the accelerator, we will have to add to the execution time a number of $m \times p$ clock cycles, because the interface with the external system allows the transfer of two cycles per cycle. Because the operational intensity for the transposition operation is very low, the effect of the transfer time is significant. Figure 2 shows the result of the evaluations, confirmed by the measurements.

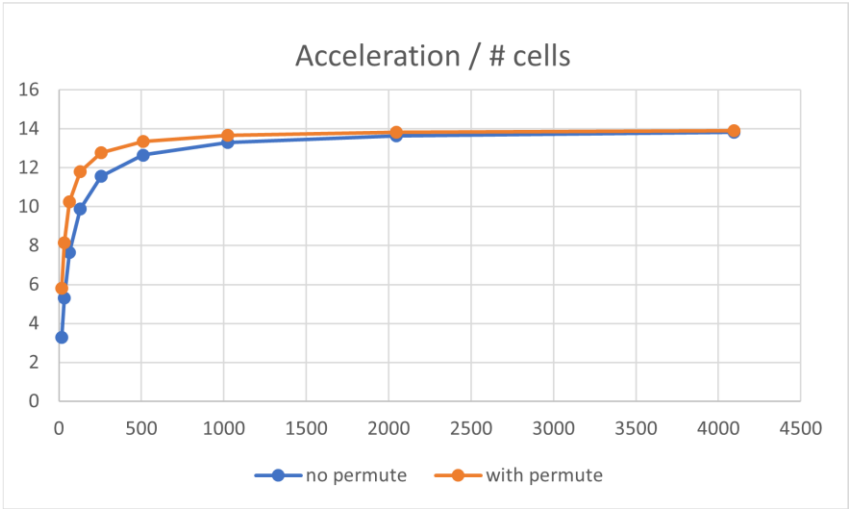

Figure 2: Speedup for 16 ×16 matrix considering transfer time.

The weight of the transfer time in the total calculation time becomes dominant for a sufficiently large p, so that the acceleration is limited to a constant value, making the effect of the accelerator unimportant above a certain limit. Also, the difference between the two implementations of the rotation operation defines insignificant.

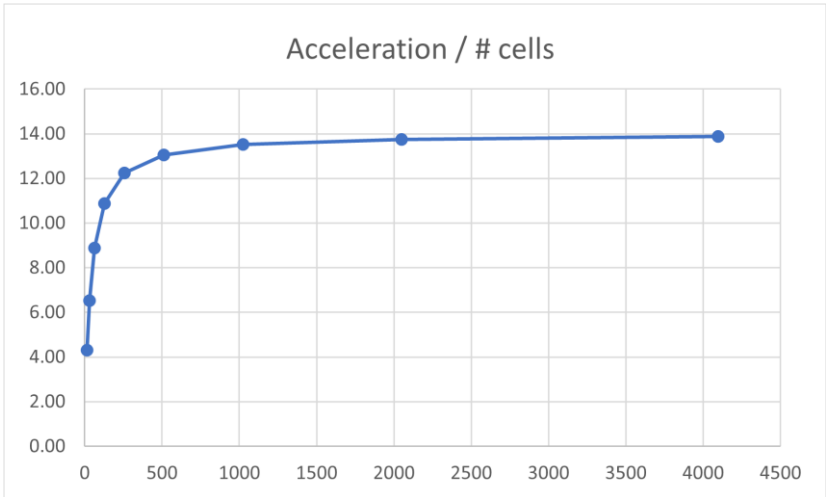

Figure 3: Acceleration for $p \times p$ matrices on p-cell accelerator without permute network.

After evaluating the transposition of many small size matrices, we will evaluate the transposition of the maximum size matrices, $p \times p$, for accelerators defined by p = 16, 32, ..., 4096. We will first consider the situation without transfer. In Figure 3 we have the results of the run in the version without the permutation network. The acceleration limitation for large matrices is observed due to the quadratic dependence on the matrix size. If we consider the contribution of the permutation network, then the acceleration represented in Figure 4 results.

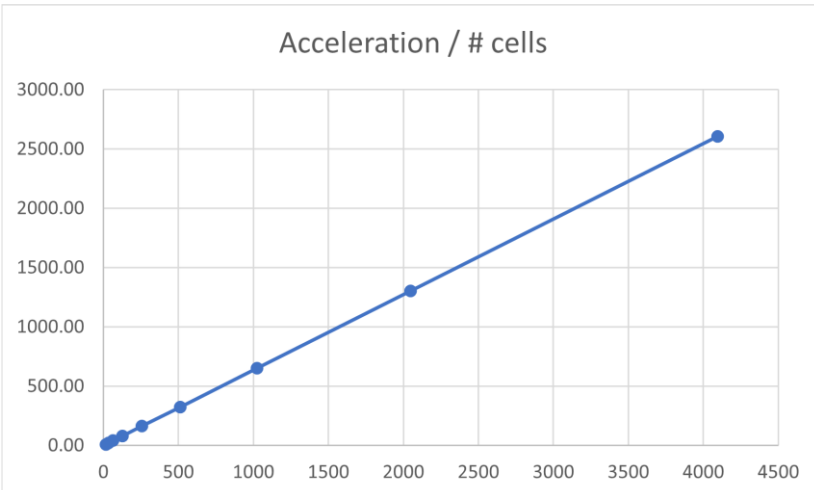

Figure 4: Acceleration for $p \times p$ matrices on p-cell accelerator with permute network.

The difference introduced by the permutation network is major due to the linear dependence on the size of the matrix. The price paid is a hardware structure with size in $O(p \times log\ p)$, compared to the one without the permutation network which has size in $O(p)$.

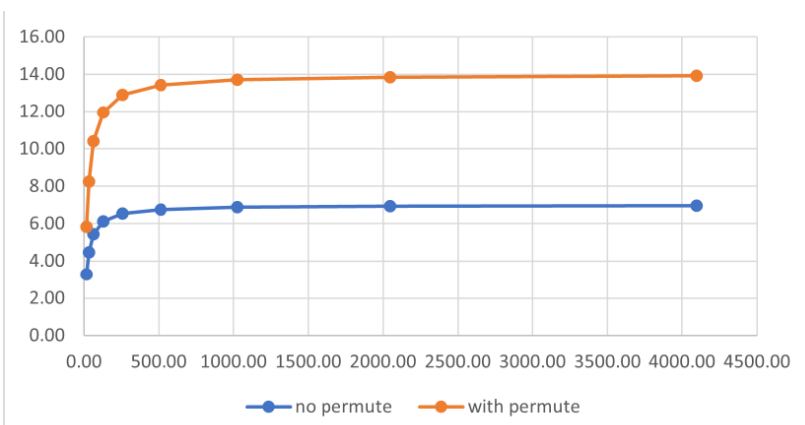

Figure 5: Acceleration for $p \times p$ matrices on p-cell accelerator considering the data transfer time.

If we take into account the effect introduced by the transfer time for the maximum size matrices, then the accelerations illustrated in Figure 5 result. The limitation of the performance is due to the acceleration of the computation beyond the limit at which the time allocated to it becomes insignificant compared to the transfer time. The twice higher performance when using the permutation network is obtained by increasing the size of the system from $O(p)$ to $O(p \times log\ p)$.

7 Conclusion

The conducted work aligns with the principles of "experimental algorithmics" as outlined in McGeoch's work [12], incorporating an additional criterion that considers the size and complexity of the hardware employed. The proposed implementations of the algorithm for transposition assume specific hardware structures, necessitating consideration when selecting a solution in a broader context.

While the purely theoretical assessment, employing orders of magnitude such as $O(f(m))$, deems the loading of certain matrices into the accelerator solely for transposition as inefficient, the evaluation based on precise cycle-level measurements indicates that transposing some matrices using the accelerator is justified. This is attributed to the substantial number of cycles, specifically 14, in the mono-core sequential solution.

The following observations are required regarding the application of the matrix transposition function:

- it is preferable that the function be applied to the matrices originating in **M** as a consequence of a calculation and not of a transfer from the external memory of the accelerator
- the version that assumes the permutation function must be used only when, in the running application, only performance matters, and efficiency (price, energy) is not taken into account
- when efficiency (price, energy) is taken into account, the solution without the exchange network is recommended
- when the accelerator offers a function to the host from a heterogeneous computing system to perform the transposition of large matrices (beyond what the accelerator can offer), it is preferable to use the version without the permutation network, especially when the efficiency – not only performance – is important.

References

[1] S. Amberger. (2018). A Parallel, In-Place, Rectangular Matrix Transpose Algorithm, *Master Thesis, Johannes Kepler University Linz*.

[2] M. Antonescu, G. M. Ștefan. (2020). Multi-Function Scan Circuit, *Proceedings of the 43nd International Semiconductor Conference CAS*, Sinaia, Romania, pp. 123-126.

[3] Václav E. Beneš. (1968). *Mathematical Theory of Connecting Networks and Telephone Traffic*. New York: Academic.

[4] J. C. Bowman, M. Roberts. (2015). Adaptive Matrix Transpose Algorithms for Distributed Multicore Processors. In: Cojocaru, M., Kotsireas, I., Makarov, R., Melnik, R., Shodiev, H. (eds) *Interdisciplinary Topics in Applied Mathematics, Modeling and Computational Science*. Springer Proceedings in Mathematics & Statistics, vol 117. Springer, Cham.

[5] C. Calvin, D. Trystram. (1996). Matrix Transpose for Block Allocations on Torus and de Bruijn Networks, *Journal of Parallel and Distributed Computing*, **34**(1):36-49.

[6] J. Choi, J. Dongarra, D. W. Walker. (1995). Parallel matrix transpose algorithms on distributed memory concurrent computers, *Parallel Computing*, **21**(9):1387-1405.

[7] K. Ding, C. Hob, J. Tsaya. (1998). Matrix transpose on meshes with wormhole and XY routing, *Discrete Applied Mathematics*, **83**:41-59.

[8] F. G. Gustavson, D. W. Walker. (2019). Algorithms for In-Place Matrix Transpose, *Concurrency and Computation: Practice and Experience*, **31**(13).

[9] B. Hanounik X. Hu. (2001). Linear-time Matrix Transpose Algorithms Using Vector Register File with Diagonal Registers, *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*.

[10] M. Kim, Y. J. Jang, W. W. Ro. (2011). Parallel transpose of matrix multiplication based on the tiling algorithms, *2011 IEEE 54th Int. Midwest Symp. on Circuits and Systems (MWSCAS)*.
[11] M. Maliţa, G. M. Ştefan, D. Thiébaut. (2007). Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation, *ACM SIGARCH Computer Architecture News*, **35**(5)32:38. Special issue: ALPS'07 – Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing, June 17, 2007, Seattle, WA, USA.
[12] C. C. McGeoch. (2007). Experimental Algorithmics, *Communications of the ACM*, **50**(11):27-31.
[13] G. M. Ştefan. (2006). The CA1024: A Massively Parallel Processor for Cost-Effective HDTV, *Spring Processor Forum Japan*, June 8-9, 2006, Tokyo.
[14] G. M. Ştefan, A. Sheel, B. Mĭţu, T. Thomson, D. Tomescu. (2006). The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing, *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford Univ., Aug. 20-22, 2006.
[15] G. M. Ştefan, M. Maliţa. (2014). Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation, *18th Int. Conf. on Circuits, Systems, Communications and Computers*, Santorini Island, Greece, pp. 582-597.

Appendices

A. Program for Transpose on Mono-Core Architecture

```
// Report2324/GeorgeAssembly/matrixTranspose.c
// >gss -S matrixTranspose.c // creates assembly
// in Windows run: m.exe
// for report do this (only in Linux)
// >gcc -o m.exe  matrixTranspose.c  -pg
// >./m.exe
// > gprof  m.exe > report5.txt
#include <stdio.h>
#define n 16
/*
void display(int X[n][n]) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            printf("%d ", X[i][j]);
        }       printf("\n");
    }
}//display
*/
int main() {
    int A[n][n] = {0};
    int R[n][n] =  {0};
   //display(A);
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
                R[i][j] = A[j][i];
        }
     }
    //display(R);
    return 0;
}
```

B. Assembly Code on Mono-Core Architecture

The code compiled from the program listed in Appendix A is:

```
    .file   "matrixTranspose.c"
    .text
    .def    __main; .scl    2;  .type   32; .endef
    .globl  main
    .def    main;   .scl    2;  .type   32; .endef
    .seh_proc   main
main:
    pushq   %rbp
    .seh_pushreg    %rbp
    pushq   %rdi
    .seh_pushreg    %rdi
    subq    $2104, %rsp
    .seh_stackalloc 2104
    leaq    128(%rsp), %rbp
    .seh_setframe   %rbp, 128
    .seh_endprologue
    call    __main
    leaq    928(%rbp), %rdx
    movl    $0, %eax
    movl    $128, %ecx
    movq    %rdx, %rdi
    rep stosq
    leaq    -96(%rbp), %rdx
    movl    $0, %eax
    movl    $128, %ecx
    movq    %rdx, %rdi
    rep stosq
    movl    $0, 1964(%rbp)
    jmp .L2
.L5:
    movl    $0, 1960(%rbp)
    jmp .L3
.L4:
    movl    1964(%rbp), %eax
    movslq  %eax, %rdx
    movl    1960(%rbp), %eax
    cltq
    salq    $4, %rax
    addq    %rdx, %rax
    movl    928(%rbp,%rax,4), %eax
    movl    1960(%rbp), %edx
    movslq  %edx, %rcx
    movl    1964(%rbp), %edx
    movslq  %edx, %rdx
    salq    $4, %rdx
    addq    %rcx, %rdx
    movl    %eax, -96(%rbp,%rdx,4)
    addl    $1, 1960(%rbp)
.L3:
    cmpl    $15, 1960(%rbp)
    jle .L4
    addl    $1, 1964(%rbp)
.L2:
    cmpl    $15, 1964(%rbp)
    jle .L5
    movl    $0, %eax
    addq    $2104, %rsp
    popq    %rdi
    popq    %rbp
    ret
    .seh_endproc
    .ident  "GCC: (tdm64-1) 10.3.0"
```

The main loop is labeled .L4. It is repeated m^2 times. It has 17 instructions and is executed in ~ 14 clock cycles taking into account that the number of instructions per cycle is around 1.2 for the x86 architecture.

* **MIHAELA MALITA**, Ph.D. is Associate Professor of Computer Science in the Department of Mathematics and Computer Science at Rivier University. She conducted her doctoral research focusing on learning models at the University of Bucharest, Romania. Formerly a faculty member at Saint Anselm College, she holds the distinguished title of Professor Emerita. Dr. Malita has taught as a visiting professor at Smith College and Amherst College in Massachusetts. She played a pivotal role as a leading member of the technical team that spearheaded the establishment of BrightScale, a Silicon Company. Additionally, she co-founded CSTA-NH (Computer Science Teacher Association for the New Hampshire Chapter) in 2010 and hosted High School Programming Contests from 2006 to 2022. More at https://ypologist.com/mmalita17.

** **GHEORGHE M. ŞTEFAN** teaches digital design and computer architecture at the National University of Science and Technology – University POLITEHNICA of Bucharest (Bucharest, Romania). Its scientific interests are focused on digital circuits, computer architecture, and parallel computation. In the 1980s, he led a team which designed and implemented the Lisp machine DIALISP. In 2003-2009 he worked as Chief Scientist and co-founder in BrightScale, a Silicon Valley startup which developed the BA1024, a many-core chip for the HDTV market. More at http://users.dcae.pub.ro/~gstefan/.