FAST AND ACCURATE MATRIX TRIANGULARIZATION USING AN ITERATIVE STRUCTURE

L. Ciminiera - A. Serra - A. Valenzano

CENS - Istituto di Elettrotecnica Generale Politecnico di Torino
Corso Duca degli Abruzzi, 24 - 10129 Torino - Italy

Abstract

The paper presents a new iterative array, which performs the triangularization of a dense matrix, using the Givens rotation algorithm. Two slightly different arrays are presented: the first one performs the factorization of a single matrix; the second one performs the recursive triangularization. The implementation of the cell in the array is based on the on-line arithmetic, which allows us to obtain high performances. Furthermore, the cell implementation requires only three types of arithmetic units (multiplication/addition, square root, division) and shift registers for data buffering and for generating the timing signals.

1. Introduction

Several numerical methods have been developed during the last few years regarding matrix triangularizations and the solution of linear systems of algebraic equations. In particular, many efforts have been made to obtain fast, stable and accurate results with low-cost implementations. In order to reach a satisfactory speedup, parallel processing is necessary. On the other hand, stable and accurate solutions of linear systems may be achieved by triangularizing the matrix A of coefficients in $Ax = b$, by means of the orthogonal factorization. Sameh and Kuck [1,2] have proposed some basic algorithms which are well suited to these purposes. The advent of VLSI technology has shown that a hardware implementation of complex sequences of computations will soon be possible. From this point of view, Kung and Leiserson have proposed [3] their systolic arrays which carry out the L - U decomposition and the triangular system solution; a highest efficiency is also reached in this case by carefully pipelining data in the array.

In this paper we present a cellular structure which realizes the reduction of a N x N matrix A to the upper triangular form by means of Givens' trasformations [4]. Givens' rotations, in fact, may be carried out in parallel and lead to a higher stability in the results. Our array may perform a single triangularization or recursive triangularizations of A. In real time environments, such as adaptive controls, a succession of reductions of A is often required before other computations may start. When a new data row is added to the previous factorized matrix, a new triangularization is necessary. This may be obtained by simply feeding our structure with data rows as they are sampled by the physical system. Modularity and regularity of communication paths permit an easy implementation of the array even if N is very large. Since the set of arithmetic operations is heterogeneous, the on-line arithmetic is particularly suitable for implementing such an array [5]. In fact, it enables us to reach a high computational speed, overlapping the operations at the digit level. In section 2, the algorithm implementation is described, then the structure of the array is introduced and the operation mechanism is presented. In section 3, the internal organization of the cells in the array is shown, assuming that on-line arithmetic is used for implementing them. The formulas and graphics in section 4 show the advantages obtained using on-line arithmetic rather than the conventional one.

2. Array structure and operation

We are interested in developing an efficient scheme for implementing Givens' reduction method of a matrix A of N^2 elements. Generally, in fact, stable solutions of dense systems of linear equations $Ax = b$, where $A \in R^{nxn}$, are obtained by triangularization of the matrix A, by means of orthogonal factorization tecniques. An efficient method for the reduction of A to its upper triangular form is based on Givens' transformations. Let us consider two general rows of A, with index i and i + 1 respectively, for the annihilation of $a_{i+1,1}$ we have

$$\begin{bmatrix} \tilde{a}_{i,1} & \tilde{a}_{i,2} \cdots \tilde{a}_{i,N} \\ 0 & \tilde{a}_{i+1,2} \quad \tilde{a}_{i+1,N} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{i1} & a_{i2} \cdots a_{iN} \\ a_{i+1,1} & a_{i+1,2} \cdots a_{i+1,N} \end{bmatrix} \tag{2.1}$$

where

$$\rho = \sqrt{a_{il}^2 + a_{i+1,1}^2} \qquad (2.2a)$$

$$c = a_{il}/\rho \qquad (2.2b)$$

$$s = a_{i+1,1}/\rho \qquad (2.2c)$$

ρ, c and s are the basic parameters of the rotation. With a suitable scheme of reductions it is possible to annihilate every a_{ij} with $i > j$. Givens' transformations, as is well known, lead to more accurate solutions of Ax=b than Gaussian elimination-based methods. Furthermore, rotations may easily be carried out in parallel and are suitable for on-line operations.

Sameh and Kuck [1] have presented a carefully chosen scheme for implementing parallel transformations in the most efficient way. We shall introduce a pipelined triangular array which achieves the triangularization (or alternatively recursive triangularizations) of A, by means of the Sameh and Kuck algorithm. For this purpose, let us consider the succession of computations involved in every plane rotation. A precedence graph for the operations required by an elementary Givens' transformation is shown in Fig. 1. It is obvious that products and sums in (2.1) may be carried out in parallel, once the basic parameters of the rotation ρ, c and s have been computed, using (2.2 a, b, c). The whole Givens' reduction may be obtained by employing a linearly connected array of N cells, like that shown in Fig. 2. The first cell (or "square root cell") can compute ρ, c and s in a time period T_{sq}. Cells with numbers 2,... N or "multiplication cells" are able to carry out the product of a 2x2 matrix U for a column vector of two elements. In particular, the cell number j will calculate $\tilde{a}_{i+1,j}$ in (2.1) by executing

$$U \begin{bmatrix} a_{ij} \\ a_{i+1,j} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{ij} \\ a_{i+1,j} \end{bmatrix} = \begin{bmatrix} \tilde{a}_{ij} \\ \tilde{a}_{i+1,j} \end{bmatrix} \qquad (2.3)$$

in a time period T_m. On the whole, a plane rotation will consist of the following logical steps:

a) Initialization: $a_{i+1,j}$'s for j=1,... N are loaded in the array on figure 2 one for each cell ($a_{i+1,K}$ is loaded in the cell number K) from the line X_K;

b) a_{i1} enters the square root cell; after a time period T_{sq}, c and s are available on lines X_i and X_{i+1} of each multiplication cell. Furthermore, ρ has also been computed and has replaced $a_{i+1,1}$ in the first cell.

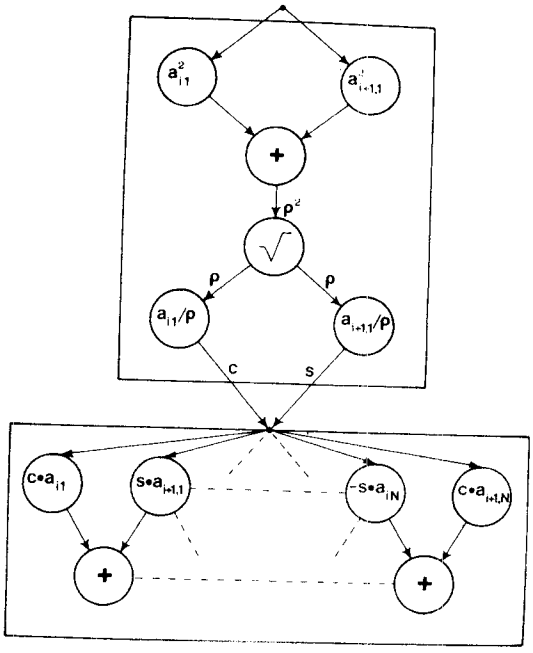


Fig. 1. Precedence graph of a basic Givens' rotation.

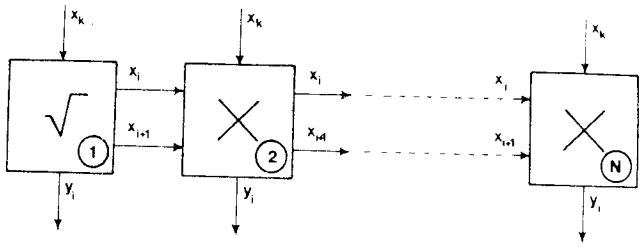


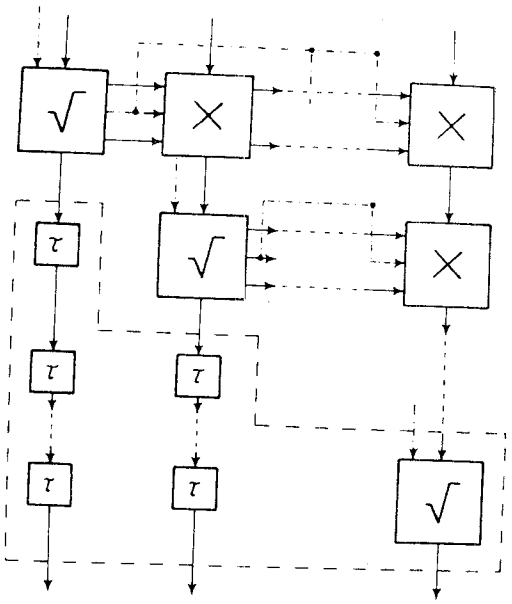
Fig. 2. A linearly-connected [illegible] executing a Givens' rotation.

c) a_{ij}'s, j=2,... n, enter the multiplication cells; after a time period T_m, $\tilde{a}_{ij}$ is available in the cell number i, while $\tilde{a}_{i+1,j}$ appears on the line y_i.

For a more efficient use of the array, pipelining must be employed. In this way, a further set of parameters ρ, c and s may be determined while the $\tilde{a}_{ij}$'s and the $\tilde{a}_{i+1,j}$'s are computed. A more detailed analysis of the operation of each kind of cell will be given in section 3. Using several linearly connected arrays we can build the orthogonally-connected structure shown in Fig. 3. Note that outputs y_i of each row of cells are tied to the inputs X_k of the strictly subsequent group. In this situation it is possible to triangularize a NxN matrix using N (N-1)/2 - 1 cells (the square root cell in the bottom row of Fig. 3 must be omitted). A recursive triangularization of A, using N (N-1)/2 cells, may be obtained also.

Let us examine the computing sequence for both cases. Each data row enters the upper line of the array and is transmitted downward, after it has been modified. First of all, the a_{Nj}'s are introduced in the upper cells, then the $a_{N-1,j}$'s, and so on, until the whole matrix A is exhausted. Cells belonging to the first row of the array annihilate the a_{i1}'s; the second row annihilates the a_{i2}'s while the cells in the last row are responsible for the reduction of the a_{iN}'s when recursive triangularizations of A are performed. Every group of cells on the same line implements a Givens' rotation, in a time period T which is the sum of T_{sq} and T_m. The sequence of the annihilation followed by the triangular array, when N=4 and a non recursive triangularization is performed, is shown Fig. 4. Note that a_{ij} denotes the original value of the corresponding element in the matrix A, while $\tilde{a}_{ij}$ represents the result of the successive modifications of a_{ij}, performed by the cells in the array.

Each zero introduced during the time period T_i is marked with the integer i. While the 0's appearing in Fig. 4 are those entered to empty the pipe. On the whole, the entire reduction of A to the upper triangular form may be completed in 2 (N-1) ti



-.-.- control signals

Fig.3. A triangular array for non-recursive and/or recursive factorizations.

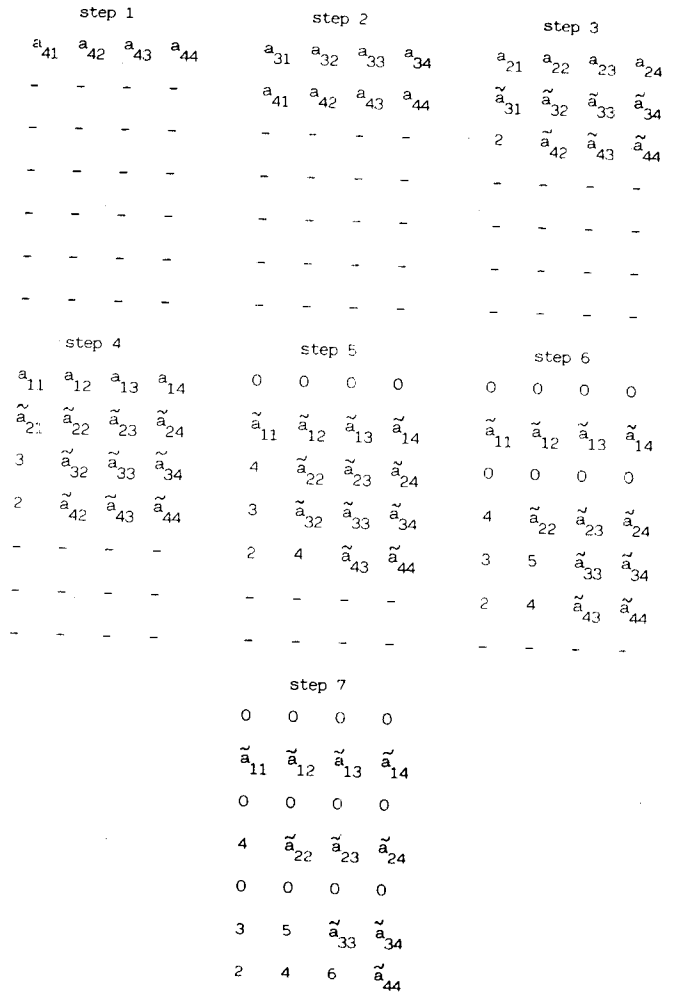
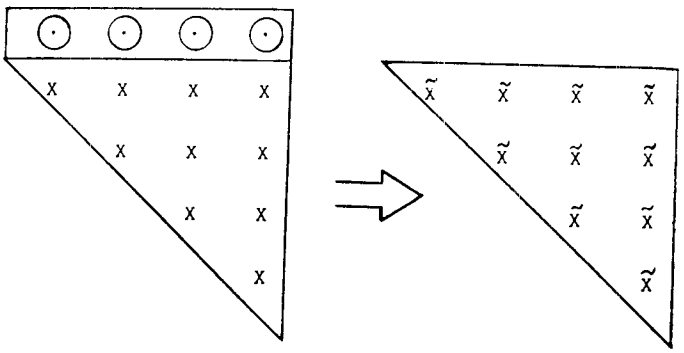


Fig.4. Annihilation sequence for the non-recursive triangularization (N=4).

me periods T.

Let us now consider the process of recursive triangularization of the matrix A (this is required, for instance, in adaptive controls based on LSQ methods). After a transient condition, during which A is triangularized, using a scheme very similar to the previous one, a steady state is reached, where it is necessary to annihilate only the subdiagonal elements (a_{ij}'s with i=j+1) at each data acquisition. This is summarized in Fig. 5.

When a new data row is obtained, N subdiagonal items must be annihilated; the whole procedure becomes recursive. It may be realized simply by feeding the array continuously with data rows sampled at each time period. Moreover, as we said, the lowest cell must also be employed to complete the structure as shown in Fig. 3: in fact, this cell is responsible for the annihilation of elements belonging to the n-th column of the matrix. Note that the delay elements τ inside the dotter line of Fig. 3 are used for obtaining all the rows of each triangularized matrix in parallel. Fig. 6 contains the steady state operation for the recursive triangularization when N=4; one may straightway verify that the steady condition is reached after 2N time periods, then a factorization is completed every T: in particular, during the period T_i, the computations started at T_{i-2N+1} are terminated. We have noted that the reduction of $A = [\underline{N} \times \underline{N}]$ may be obtained employing N (N-1)/2-1 cells, while recursive factorizations require N (N-1)/2 cells; if we consider the solution of $Ax=b_k$, K=1,2...,r b_k must also be updated, together with A. In this situation



⊙ entering element
X matrix element before the transformation
$\tilde{X}$ matrix element after the transformation
Fig. 5. Recursive triangularization.

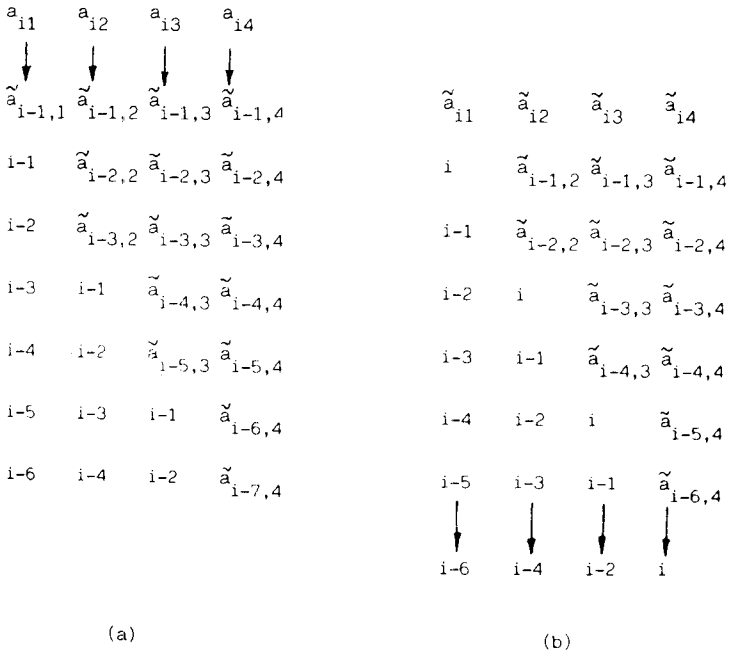


Fig. 6. a) initial condition of the i-th step of the recursive triangularization.
b) final condition of the i-th step of the recursive triangularization.

N (N-1)/2 + (N-1)r - 1 cells are required for the non-recursive operation, while the recursive method needs N (N-1)/2 + Nr elements.

3. Cell Implementation

The efficiency of the implementation of the array presented in section 2 is strongly affected by the arithmetic algorithms and by the number representation chosen. Using the classical number representation (2's complement) and arithmetic algorithms[6] and assuming a serial-parallel scheme, the implemented array shows some drawback - First, the classical algorithms for multiplication, square root and division, do not process the bits of the serial operand in the same order. This characteristic does not allow one to overlap the operations performed by the array; for example, if the square root follows a multiplication, the latter operation cannot start when only the first bit of the result produced by former operation are available, but it is necessary to wait until the whole result of the multiplication is produced before the square root operation may begin. Hence, the pipelined array implemented using the conventional arithmetic has a large latency time.

Moreover, when the system reaches its steady state, the throughput is limited by the time required for performing the slowest operation; in other words, if the operands and the results are expressed using n digits, the array presented in section 2 and implemented using the conventional serial - parallel

arithmetic cannot provide more than n digits every 2n clock cycles, since one multiplication requires 2n clock periods (including one period for the initialization). Therefore the throughput obtained is about one half of the maximum achievable.

Both drawbacks arising when the conventional arithmetic is used may be eliminated or lessened using the on line arithmetic. In fact, one of the main characteristics of the on-line algorithms is that the operation is performed in a digit serial manner from the most to the least significant digit; thus many different operations may be pipelined at the digit level, since each operation can begin as soon as a few digit of the previous result become available.

Furthermore, each on-line algorithm is able to provide the n most significant digits of the result using only $n+\delta+1$ clock periods, where δ in the on-line delay of the algorithm considered. The algorithm for multiplication addition [7], square root [8] and division [9], show that the value of δ is small.

In Fig. 7, the internal block diagram for the diagonal cells in the array is shown.

The computation of ρ^2 required by (2.2a) is performed recursively, in fact, the value of $a_{i+1,1}$ is equal to the value of ρ calculated in the previous step of the algorithm; hence, at each step, the value of ρ^2 is the sum of the square values of the number previously entered in the cell. The entering number is stored in the A and B registers, and the value of the previously calculated ρ^2 is accumulated in the C register. Then the result produced by the multiplication/addition block, indicated by the symbol x/+, is passed to a block, marked by a $\sqrt{}$, performing the square root. The output thus obtained is used for computing (2.2b) and (2.2c); furthermore it is stored in the D register, since the actual value of ρ is the value of $a_{i+1,1}$ in the next step. The outputs c and s are fed to the other cells in the same row.

The internal structure of the cells performing the calculations required by (2.3) is shown in Fig. 8. The P register holds the value of one element of the entering row, while the Q register holds one of the results of the previous step, that is the value $\tilde{a}_{i+1,j}$. The four arithmetic blocks in Fig.8 are multiplication/addition networks.

In Fig. 7 and in Fig. 8, the paths indicated with dashed lines are reserved for control signal which cause the parallel loading of the registers and the reset of the arithmetic units. Note that the loading of the P registers in the cells of Fig. 8 is delayed so that the operations begin when the first digits of s and c have been produced by the diagonal cell. In Fig. 9, the time diagram of the

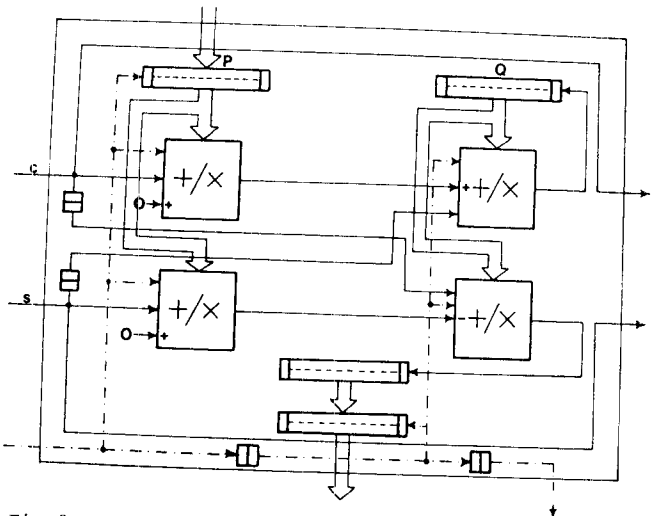


Fig.8. Internal structure of the multiplication cell.

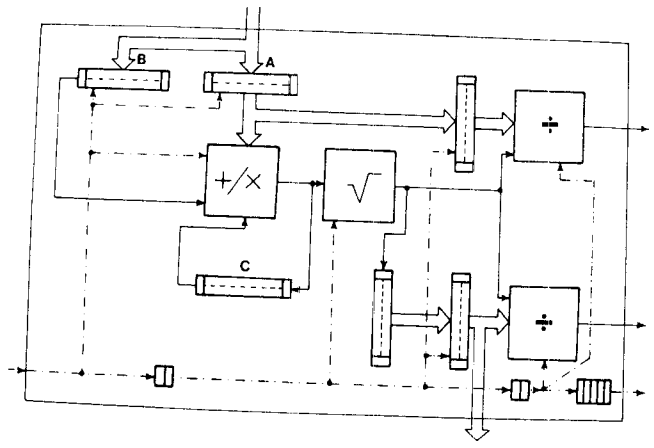


Fig.7. Internal structure of the square-root cell.

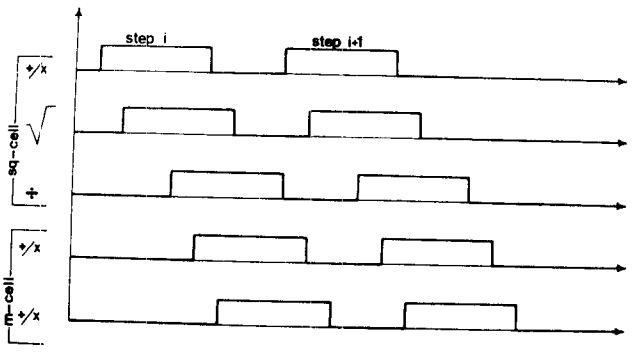


Fig.9. Timing diagram of the operations of the linearly-connected array.

operations performed by the array in Fig. 2 is shown. On each horizontal line, the output of one arithmetic unit is represented. It may be noted that there is a good overlap of the calculations, even if the set of the operations performed is heterogeneous.

4. Array Performance

In a pipelined system, the latency time and the throughput are the two most interesting performance indexes. In the array presentated here, the latency time is also the operation time for the non-recursive triangularization, while the throughput determines the maximum frequency at which the results are provided, in the recursive triangularization. Using a radix r=2 for the redundant representation needed by the on-line algorithms, the operation time for the non-recursive algorithm is given by the following formula:

$$T_{ol} = 2(N-1)\ (n+12)\ t_{ol} \qquad (4.1)$$

where n is the number of digits used for the number representation and t_{ol} is the clock period.

The steady state throughput of the array may easly be derived from Fig. 9, and it is given by:

$$f_{ol} = \frac{1}{t_{ol}} \frac{n}{n+6} \qquad (4.2)$$

Analogous equations may be derived, if the conventional arithmetic is used for implementing the array, using a serial-parallel scheme. In this latter case, the operation time is given by:

$$T_c = 2\ (N-1)\ (6n+6)\ t_c \qquad (4.3)$$

and the steady-state throughput for the recursive triangularization is:

$$f_c = \frac{1}{t_c} \frac{n}{2n+1} \qquad (4.4)$$

where t_c is the clock period.

A speed-up factor, due to the arithmetic implementation, for the non recursive triangularization may be defined as follows:

$$S_p = \frac{T_c}{T_{ol}} = \frac{6n+6}{n+12} \frac{t_c}{t_{ol}} \qquad (4.5)$$

In an analogous way, the speed up achieved for the recursive triangularization may be defined as follows:

$$S_p = \frac{f_{ol}}{f_c} = \frac{n+6}{2n+1} \frac{t_c}{t_{ol}} \qquad (4.6)$$

The ratio t_c/t_{ol} depends on the relative speed of the two implementations.

The curves giving the speedup as a function of the precision required, assuming $t_c = t_{ol}$, are shown in Fig. 10 for both recursive and non-recursive triangularization.

The speed-up is always greater than 1; more specifically, the better results are obtained for the non-recursive triangularization, since the on-line arithmetic produces a dramatic decrease of the latency time of the array. In this case, the curve in Fig. 10 shows that, in order to decrease the array complexity, it is possible to design the on-line arithmetic units so that t_{ol} is from about 3 to 5 times greater than t_c and the overall speed is about the same as for the conventional arithmetic implementation. Furthermore, an implementation using the conventional arithmetic would require complex units for data buffering. For example, in the "square root" cell, between the multiplication/addition and the square root unit a double LIFO buffer 2n-1 digit long must be placed. The LIFO buffer is required since the result of the multiplication /addition is produced from the least to the most significant digit, while the square root algorithm processes the input number from the most to the least significant digit. The double buffering is required in order to allow overlapping of the two operations. This fact increases the complexity of the conventional arithmetic-based implementations.

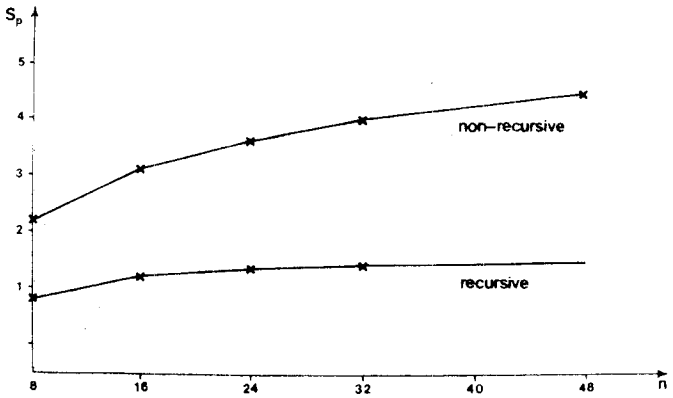


Fig. 10. Speedup factor obtained using on-line arithmetic.

On the other hand, the implementation presented here, based on the on-line arithmetic, uses only shift registers for data buffering and control signal transmission.

5. Conclusions

A new iterative array for matrix triangularization is presented in this paper. It performs matrix factorization using the Givens algorithm, which assures a good accuracy in the results obtained. Two situations are considered leading to two slightly different arrays. The first one deals with the classical problem of the triangularization of a single matrix. The second one is derived from the problem arising in on-line system identification for adaptive controls. In this case a recursive triangularization of dense matrix is required. The arrays presented are also able to operate on an arbitrary number of loading vectors. Since the set of the arithmetic operations performed by the array is heterogeneous, the cell implementation based on the on-line arithmetic is chosen. In fact, in this case, the on-line arithmetic allows us to achieve large performance improvements on an analogous implementation based on the classical arithmetic algorithms. Furthermore, the design of cells is simple, since only shift registers are required for data buffering and control signal distribution.

References

1. Sameh, A., Kuck, D. "A Parallel QR Algorithm for Symmetric Tridiagonal Matrices". IEEE Trans. Comptrs. C-26, (1977), pp. 147-153.

2. Sameh, A., Kuck, D. "On Stable Parallel Linear System Solvers" J. ACM 25, 1 (1978), pp.81-91.

3. Kung, H.T., Leiserson, C.E. "Systolic Arrays (for VLSI)" TR, April 1978, Computer Science Department, Carnagie-Mellon University.

4. Noble, B. "Applied Linear Algebra" Prentice-Hall, Inc, 1969.

5. Ercegovac, M.D., Grnarov, A.L. "On the performance of On-Line Arithmetic" Proc. IEEE International Conference on Parallel Processing, August 1980, pp. 55-61.

6. Chu, Y. "Digital Computer Design Fundamentals" Mc Graw-Hill, Inc, 1962.

7. Grnarov, A.L., Ercegovac, M.D. "VLSI Oriented Iterative Networks for Array Computations", Proc. ICCC, October 1980, pp. 60-64.

8. Ercegovac, M. D., "An On-Line Square Rooting Algorithm" Proc. Fourth IEEE Symposium on Computer Arithmetic, October 1978, pp. 183-189.

9. Trivedi, K.S., Ercegovac, M.D. "On line Algorithms for Division and Multiplication" IEEE Trans. Comptrs. C-26,7 (1977), pp. 681-687.