A Systolic Array for Computing $B A^{-1}$

PIERRE COMON, MEMBER, IEEE, AND YVES ROBERT

Abstract—**This paper is devoted to the design of a new systolic array of $n(n+1)$ elementary processors which computes $B A^{-1}$ within $4n + p - 2$ time steps, where A is a dense (nonsingular) n by n matrix, B is a dense p by n matrix. The array can be directly extended to compute the vector $Y = B A^{-1}R$, where R is a vector with n components. Such computations arise frequently in signal processing applications.**

I. Introduction

As pointed out in [14] and [20], many signal processing applications impose a severe demand on the I/O bandwidth and computation power of general-purpose computers. The systolic concept offers guidelines in building cost-effective systems that balance I/O with computations. A systolic architecture is a network or array of elementary processors which are locally interconnected [15], [18]. The regularity and simplicity of such systems leads to modular designs highly suitable for VLSI implementation [1], [19], [21]. The great interest in systolic architectures for signal processing has been demonstrated by the large number of systolic special-purpose arrays as well as systolic digital signal processors, which have been recently introduced in the literature (see [3]–[5], [9], [11], [14]–[16], [20], [21], [23], [25], [28], [30], and [31] among others).

In this paper, we introduce a new systolic array of $n(n+1)$ elementary processors which can compute $B \cdot A^{-1}$ within $4n + p - 2$ time steps, where A is a dense (nonsingular) n by n matrix, and B is a dense p by n matrix. The array can be directly extended to compute the vector $Y = B A^{-1}R$, where R is a vector with n components. Such computations arise frequently in signal processing applications, as we briefly elaborate in the next section.

We point out that our array receives the matrix A followed by the matrix B from the host, and directly outputs the solution matrix $B A^{-1}$ after $4n + p - 2$ time steps. This one-array computation is much more attractive for direct hardware implementation than a solution based on the successive utilization of several simple special-purpose arrays.

II. Computing $B \cdot A^{-1}$ in Signal Processing Applications

Detection and estimation are essential operations in signal processing [29]. Focusing on multivariate discrete-time stationary processes observed on a finite duration, all the linear operators entering the problem of detection and estimation are of finite rank and are often expressed with the help of projections.

The projection $\phi_S(\boldsymbol{R})(k)$ of an $n \times 1$ observation vector onto the linear subspace spanned by a $p \times 1$ random signal $S(k)$ is expressed as:

$$\phi_S(\boldsymbol{R})(k) = \Gamma_{SR}(k)\, \Gamma_{RR}(k)^{-1}\, \boldsymbol{R}(k)$$

where k stands for a time, frequency, space, or wavenumber variable. $\Gamma_{SR}(k) = E\{S(k)\, \boldsymbol{R}(k)^+\}$ and $\Gamma_{RR}(k) = E\{\boldsymbol{R}(k)\, \boldsymbol{R}(k)^+\}$ ($^+$ denoting complex conjugated and transposed) are covariance matrices of respective dimension $p \times n$ and $n \times n$, and can eventually not depend on k, according to the investigated problem.

The projection $\phi_S(\boldsymbol{R})(k)$ is used either in the frame of Bayesian approach, especially under Gaussian hypotheses, or in Wiener filtering. In these cases, some *a priori* knowledge of the signal and observation statistics is required [12], [29].

Similar expressions must be computed in linear optimal solutions in the least-squares sense, where quantities such as $\Psi_S(\boldsymbol{R})(k)$ are needed:

$$\Psi_S(\boldsymbol{R})(k) = \Gamma_{SR}(k)\, \Gamma_{RR}(k)^{-1}\, \boldsymbol{R}(k).$$

Here, $\Gamma_{SR}(k) = \Sigma_{\mu=1}^{N} S_\mu(k)\, \boldsymbol{R}_\mu(k)^+$ and $\Gamma_{RR}(k) = \Sigma_{\mu=1}^{N} \boldsymbol{R}_\mu(k)\, \boldsymbol{R}_\mu(k)^+$ are sample covariance matrices that are computed from the observations.

These kinds of expressions also arise in maximum *a posteriori* or maximum likelihood approaches if the covariance matrices are unknown [7].

Hence, usual optimal solutions in digital signal processing involve Toeplitz symmetric, circulant, or Hermitian covariance matrices. In fact, with regard to usual practice, observations of stationary processes can be roughly regrouped into three classes. First, the real-time series, considered as finite observations of infinite duration realizations, lead to real symmetric block-Toeplitz covariance matrices. Second, the real-time series assumed to be one period length realizations of periodic processes lead to real block-circulant covariance matrices [2]; this situation occurs in many processings using the discrete Fourier transform, that is, the circular assumption [6]. Third, complex-valued data defined in frequency domain lead to block-Hermitian covariance matrices.

In another connection, as emphasized in [10], adaptive processing techniques can be divided into two main families: "block" processings and "recursive" processings. In the first, incoming data are divided into independent

Manuscript received February 3, 1986; revised October 29, 1986.

P. Comon is with Stanford Electronics Laboratories, Stanford University, Stanford, CA 94305 on leave from the Random Phenomena and Geophysics Study Center (CEPHAG), Grenoble, France.

Y. Robert is with the IBM European Research Center ECSEC, Roma, Italy, on leave from the Centre National de la Recherche Scientifique, TIM3, Grenoble, France.

IEEE Log Number 8613860.

0096-3518/87/0600-0717$01.00

blocks, and from each block assumed as a whole, an estimate is computed. On the other hand, in the second, the estimates are assumed to be *a priori* dependent to the preceding estimates. Generally, this recursive procedure does not use a block division. If the estimates are updated as soon as new sample data income, the covariance matrices at time k can be computed from their value at time $k-1$ by a rank 1 modification [10]. As for us, we accord special attention to the first kind of processing.

Hence, regarding adaptive processings of "block" type, we have to deal with the general problem to compute $B\,A^{-1}R$, where B and A are dense complex matrices of respective dimension $p \times n$ and $n \times n$, and R is a vector with n components. Here, we do not take into account further assumptions about the particular form of matrices A and B (Toeplitz symmetric, circulant, Hermitian, . . .), so that our approach is available in any of the three classes aforementioned. The only restriction from the numerical point of view is that the matrix A is positive definite or diagonal dominant.

Such a computation requires $O(n^3)$ floating-point operations on a sequential computer, and this motivates the use of a dedicated systolic processor whenever real-time solutions are needed.

III. The Systolic Array

A. The Gauss–Jordan Diagonalization Algorithm

We first concentrate on the design of a systolic array to compute the product $B \cdot A^{-1}$, and we differ the problem of postmultiplication by the vector R until Section III-C.

Let C denote the $n+p$ by n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix}.$$

We reduce C to the matrix

$$\begin{bmatrix} I \\ B\,A^{-1} \end{bmatrix}$$

by the Gauss–Jordan diagonalization algorithm. But since we want to compute $B\,A^{-1}$ rather than $A^{-1}\,B$, we have to use combinations of columns rather than combinations of rows. The way to proceed is to postmultiply C by n elementary $n \times n$ matrices $J_1, J_2, \cdots, J_n$ in order to obtain after n steps

$$CJ_1J_2 \cdots J_n = \begin{bmatrix} I \\ B\,A^{-1} \end{bmatrix}.$$

Therefore, define $C_k = C_{k-1} \cdot J_k$, starting with $C_0 = C$, and let C_k° be the $(n+p-k) \times n$ matrix built up with the last $n+p-k$ rows of C_k. We choose J_k so that the first k rows of C_k are those of I_n, the identity matrix of order n. Thus, C_k has the following structure:

$$C_k = \begin{bmatrix} \text{the } k \text{ first rows of } I_n \\ C_k^\circ \end{bmatrix}.$$

In particular, C_n° is the desired matrix $B\,A^{-1}$. The matrix J_k only defers from I_n by its kth row, which we denote as

$$[c_{k1}^{(k)} \cdots c_{kn}^{(k)}].$$

The coefficients of C_k° are denoted $(c_{ij}^{(k)})$, $k+1 \le i \le n+p$, $1 \le j \le n$. The values of the $c_{ij}^{(k)}$, $k \le i \le n+p$, $1 \le j \le n$, $1 \le k \le n$, are computed recursively by the following algorithm, starting from $c_{ij}^{(0)} = c_{ij}$:

for $k := 1$ **to** n
begin
  {compute J_k}
  $c_{kk}^{(k)} := 1 / c_{kk}^{(k-1)}$ (A)
  for $j := 1$ **to** $n, j \neq k$
    $c_{kj}^{(k)} := -c_{kk}^{(k)} * c_{kj}^{(k-1)}$ (B)
  {compute C_k°}
  for $i := k+1$ **to** $n+p$
  begin
    for $j := 1$ **to** $n, j \neq k$
      $c_{ij}^{(k)} := c_{ij}^{(k-1)} + c_{ik}^{(k-1)} * c_{kj}^{(k)}$; (C)
    $c_{ik}^{(k)} := c_{ik}^{(k-1)} * c_{kk}^{(k)}$; (D)
  end;
end;

Recall that $c_{kj}^{(k)}$ refers to the kth row of J_k, while $c_{ij}^{(k)}$ with $k < i$ refers to the matrix C_k°.

B. Description of the Array

We use a bidimensional array of orthogonally connected processors (see Fig. 1). The array is composed of n rows, each row k including $n+1$ processors numbered from left to right $P_{k,1}, \cdots, P_{k,n+1}$.

The matrix A, followed by the matrix B, is fed into the array column by column. More specifically, column k of the matrix C is input to processor P_{1k}, one new element each time step, beginning at time $t = k$. This input format is depicted in Fig. 1.

Before describing in detail the operation of the array, let us give an informal explanation of how it works. There are n phases in the Gauss–Jordan diagonalization algorithm. We map each of them onto a row of the array. More precisely, the kth instance of the external loop is performed by the row k of the array, which receives the matrix C_{k-1}° from the row $k-1$, and delivers the matrix C_k° to the row $k+1$. There are two different things to do during each phase k of the algorithm: first compute the elementary matrix J_k, then perform the multiplication $C_k^\circ = C_{k-1}^\circ J_k$. The action of the processors in row k is defined to match these two different computations: the processors in row k first compute and store the coefficients of matrix J_k, then they perform inner-product step computations.

As soon as all the elementary matrices J_k are computed and stored in the array, we say that the array is initialized. Once initialized, the array can be viewed as a linear operator that transforms any matrix B into the matrix $B\,A^{-1}$. This is only a conceptual vision of the array, since the initialization and multiplication phases are pipelined. Of course, if we let a third matrix B' follow B, the array will

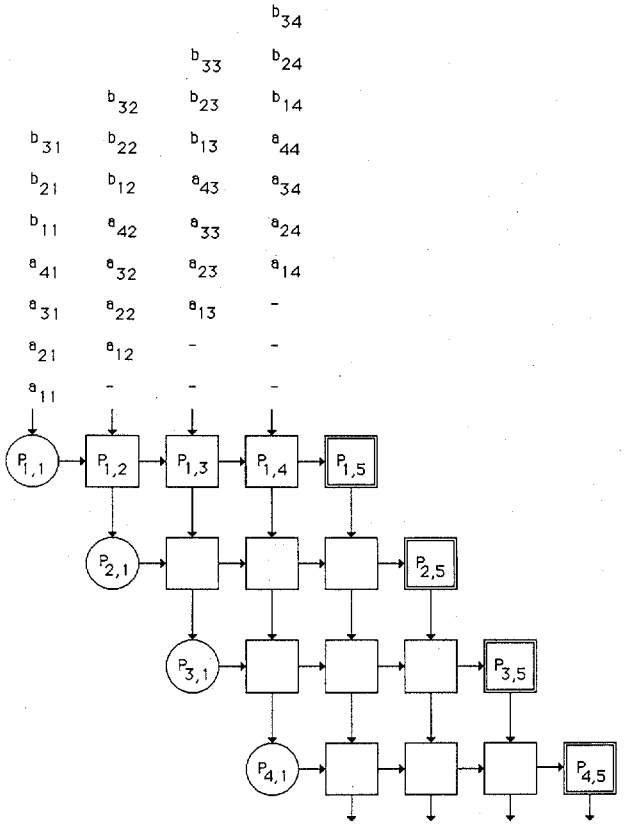


Fig. 1. Input of the systolic array ($n = 4, p = 3$).

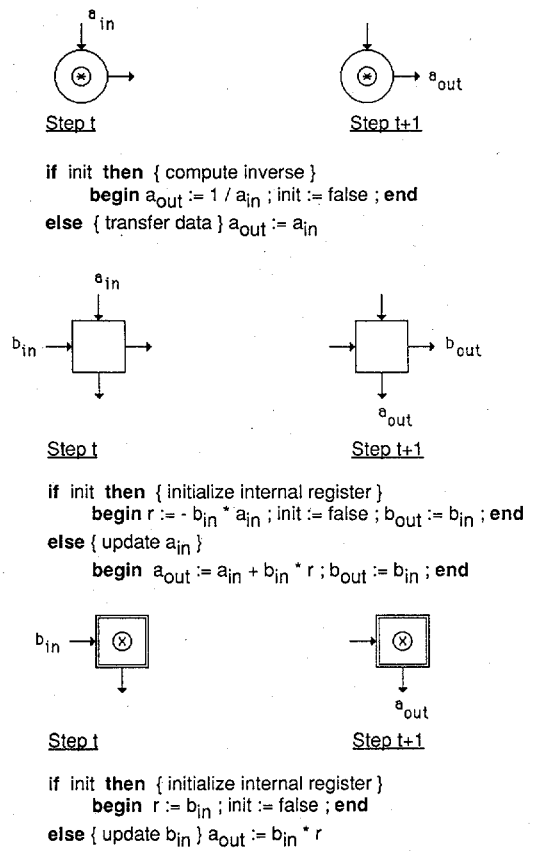


Fig. 2. Operation of the processors (at the beginning, all variables *init* are set to "true").

output $B'\,A^{-1}$ after $B\,A^{-1}$. The inverse of A has been stored in the array as n elementary matrices $A^{-1} = J_1 J_2 \cdots J_n$.

The operation of each processor is detailed in Fig. 2. There are three types of processors as follows.

- *Type 1:* Circle processors compute the inverse of their first input data, according to the instruction (A) in the algorithm. Afterwards they simply act as delay cells.
- *Type 2:* Square processors first initialize their internal register r by storing after modification their first input data [instruction (B)]; then they act as classical multiply-and-add cells ([15], [18]) to perform instruction (C).
- *Type 3:* Double-square processors actually operate as square processors, with the exception that their internal register r is not initialized in the same way (see Fig. 2). They perform instruction (D).

In the kth row of the array, the leftmost processor $P_{k,1}$ is of type 1 and the rightmost processor $P_{k,n+1}$ is of type 3. All the other processors $P_{k,2}, \cdots, P_{k,n}$ are of type 2.

As we already mentioned, the action of a given processor depends on whether it is the first data item it receives. As shown in Fig. 2, each processor has a control Boolean *init* initialized to true. This Boolean specifies the operation to be performed and the line along which the data are to be sent out. In an actual implementation, the instruction to start would be input to the top-left corner of the array and systolically propagated to all the processors. Since each processor starts operation when the processor above it sends data downwards (through a_{out}) for the first time, it can be checked from Fig. 1 that processor P_{kj} operates for the first time at step $3k + j - 3$, so that the start signal would move at full speed to the right and at half-speed downwards.

Let us describe now the operation of the first row of the array. At step 1, processor P_{11} computes $c_{11}^{(1)} = 1/c_{11}^{(0)}$ and sends it rightwards. From step 2 until the end of the computation, P_{11} operates as a one-step delay cell. At step 2, P_{12} initializes its internal register with $c_{12}^{(1)} = -c_{11}^{(1)} * c_{12}^{(0)}$ and then operates as a multiply-and-add cell: indeed at step 3, it updates $c_{22}^{(0)}$ into

$$c_{22}^{(1)} := c_{22}^{(0)} + c_{21}^{(0)} * c_{12}^{(1)},$$

and so on with $c_{32}, c_{42}, \cdots$.

$P_{13}, P_{14}, \cdots, P_{1n}$ operate like P_{12}, starting, respectively, at step 3, 4, n. At step $n + 1$, $c_{11}^{(1)}$ reaches processor $P_{1,n+1}$ and is stored in its internal register. From step $n + 2$ to the end, processor $P_{1,n+1}$ updates the first column of C_1^9. Indeed, at step $n + 2$, it computes $c_{21}^{(1)} = c_{21}^{(0)} * c_{11}^{(1)}$, and so on, with $c_{31}, c_{41}, \cdots$, according to instruction (D).

The following table shows the operation of the first row of processors during phase 1 (we choose $n = 4$ in this example):

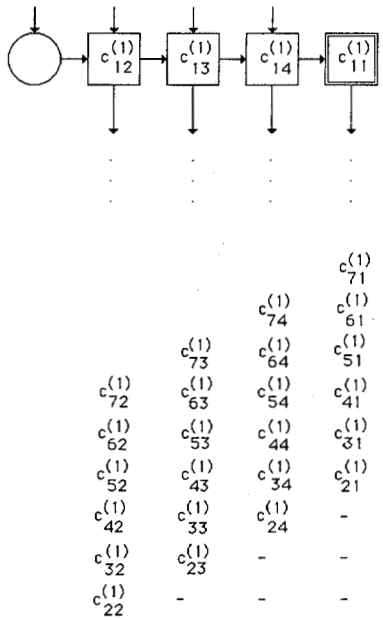


Fig. 3. Operation of the first row of the array.

	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}		
Time Step	1	$c_{11}^{(1)}$	—	—	—	—	
	2	—	$c_{12}^{(1)}$	—	—	—	
	3	—	$c_{22}^{(1)}$	$c_{13}^{(1)}$	—	—	
	4	—	$c_{32}^{(1)}$	$c_{23}^{(1)}$	$c_{14}^{(1)}$	—	
	5	—	$c_{42}^{(1)}$	$c_{33}^{(1)}$	$c_{24}^{(1)}$	$c_{11}^{(1)}$	J_1
	6	—	$c_{52}^{(1)}$	$c_{43}^{(1)}$	$c_{34}^{(1)}$	$c_{21}^{(1)}$	C_1°

Fig. 3 shows the contents of the registers of the first row of the array, as well as its output format.

We briefly describe at the data stream level the operation of the kth row of the array, which is devoted to the computation

$$C_k^\circ = C_{k-1}^\circ J_k.$$

The leftmost processor $P_{k,1}$ transmits the input data arriving from the top to the right. Type 2 processors in the row modify and store the first data value arriving from the top and pass downwards all the following data after modification. Similarly, the rightmost processor $P_{k,n+1}$ stores the first input data arriving from the left and passes downwards all the following data after modification. Thus, after a column of $n + p$ input data flows through the whole array, its length is shortened by n to become a column of p output data. $C_0 = C_0^\circ = C$ is input to row 1 of the array, C_1° is input to row 2, $\cdots$, and C_{n-1}° is input to row n; finally, the array outputs the matrix $C_n^\circ = B A^{-1}$.

It is important to note that the columns are "reordered" in some sense when moving through the array: the input of the processors $P_{k1}, \cdots, P_{k,n}$ in row k are, respectively, the columns $k, k+1, \cdots, n, 1, \cdots, k-1$ of C_{k-1}°. The element $c_{kk}^{(k)}$ is computed according to $c_{kk}^{(k)} := 1/c_{kk}^{(k-1)}$ in processor $P_{k,1}$, and moves rightwards without modification until it is stored in the processor $P_{k,n+1}$. The nondiagonal coefficients $c_{ki}^{(k)}$, $i \neq k$, of the matrix J_k are computed and stored in the processors $P_{k,2}, \cdots, P_{k,n+1}$ (more precisely, $c_{ki}^{(k)}$ is stored in processor $P_{k,1+(i-k) \bmod n}$). After the processors are initialized, they perform the multiplication $C_k^\circ = C_{k-1}^\circ J_k$. The matrix C_k° is output from the row k of the array in column order, the leftmost column being now column $k+1$.

The reason for such a reordering is the following: in phase k of the algorithm, column k of the matrix must be combined with all the other columns (and not only with columns $j, j > k$, as would be the case for Gaussian triangularization). Moreover, $c_{kk}^{(k-1)}$ must be the first coefficient of the matrix C_{k-1}° to be input to the kth row of the array. The circular permutation of columns that the array implements allows us to fulfill these requirements, letting column k play exactly the same role in phase k of the algorithm as column 1 in the first phase.

Theorem: Given a dense nonsingular $n \times n$ matrix A and a dense $p \times n$ matrix B, the orthogonal systolic array of $n(n+1)$ processors can compute $B A^{-1}$ within $4n + p - 2$ time steps.

In the above presentation of the array, we have not ad-

dressed the issue of pivoting. As already emphasized in the Introduction, we are concerned with positive definite or diagonal dominant matrices A, for which the Gauss-Jordan algorithm is numerically stable.

Sure enough, it is well known that partial pivoting cannot be implemented on systolic arrays, since the scanning for a pivot would break down the regularity of the design [18]. We would like to mention the possibility of dealing with general matrices by programming some processors of the array to perform Givens rotations instead of Gaussian combination of columns, so that the strictly lower triangular part of the matrix A can be annihilated in a stable manner. Such a mixed strategy of elimination is detailed in [8].

C. Postmultiplication by a Vector R

It is easy to compute on-the-fly the vector $Y = B\,A^{-1}R$. For convenience, let us denote $M = B\,A^{-1}$ (instead of C_n°) the matrix which is output from the preceding systolic array.

To compute $Y = MR$, we add an $(n + 1)$th row to the array, as depicted in Fig. 4. This new row is composed of n multiply-and-add processors, denoted as triple square processors. The ith processor starting from the left, numbered $P_{n+1,i}$, has its internal register preloaded with the value R_i. A processor is to be activated when it receives its first input data from the top, that is, $P_{n+1,i}$ is to be activated at step $3n + i$. Once activated, it operates as detailed in Fig. 5.

Let us follow the computation of Y_1, initialized to 0. At step $3n + 1$, it is updated in processor $P_{n+1,1}$ according to $Y_1 := Y_1 + m_{11}\,R_1$. At step $3n + 2$, it is updated in processor $P_{n+1,2}$ according to $Y_1 := Y_1 + m_{12}\,R_2$. At step $4n$, Y_1 is output from processor $P_{n+1,n}$ with its definitive value. A new Y_i is then output every time step, until step $4n + p - 1$, where the last component of Y exits the array.

There are many possible variations. One can decide that the processors of the $(n + 1)$th row start their operation at step 1, and neglect the data output from $P_{n+1,n}$ until step $4n$ (in fact, only zeros are produced by the array until this step). It is also possible to include a mechanism to load the components of the vector R into the corresponding registers: one can simply input $R_1, R_2, \cdots, R_n$ to the left input port of $P_{n+1,1}$, each datum moving right until it finds an empty register to be stored in.

Proposition: Given a dense nonsingular $n \times n$ matrix A, a dense $p \times n$ matrix B, and a vector R with n components, the orthogonal systolic array of $n(n + 2)$ processors can compute $B\,A^{-1}R$ within $4n + p - 1$ time steps.

We point out that the array can be straightforwardly extended to the case where R is an $n \times q$ matrix instead of an $n \times 1$ vector. All one has to do is to replicate the bottom row of triple square processors, such that they pass their inputs from above downwards. It then takes $n(n + q + 1)$ processors and $4n + p + q - 2$ time steps to compute the matrix $B\,A^{-1}R$.

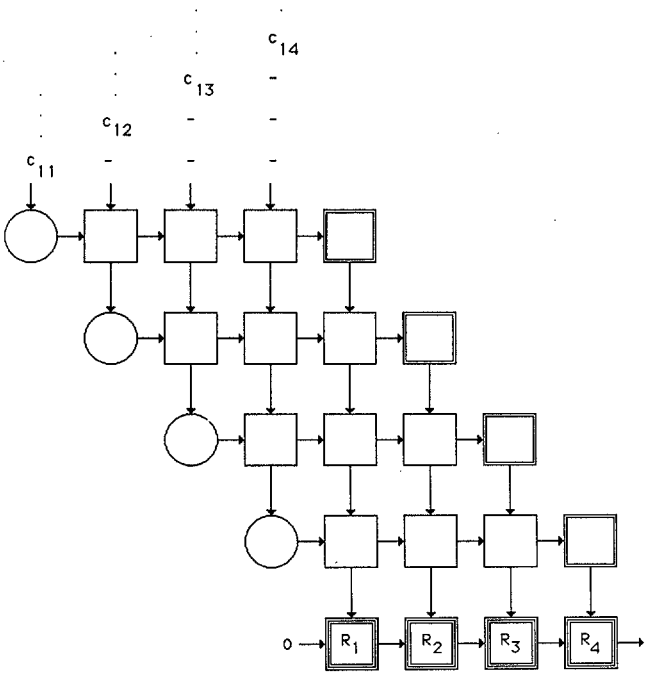


Fig. 4. Computing $y = B \cdot A^{-1} \cdot R\,(n = 4, p = 3)$.

if activated
then { update y_{in} } $y_{out} := y_{in} + R * b_{in}$

Fig. 5. Operation of triple square processors.

IV. Concluding Remarks

There are many systolic arrays that have been introduced in the literature to solve particular instances of linear algebra and digital signal processing problems. Given two dense matrices A and B of respective dimensions $n \times n$ and $p \times n$, the computation $B\,A^{-1}$ could be achieved by using several existing arrays. In particular, one could use the building blocks of [11] for i) computing the LU decomposition of A; ii) inverting the triangular matrices L and U; iii) computing the product $U^{-1}L^{-1} = A^{-1}$; and iv) computing the product $B\,A^{-1}$. Such an approach based on several simple (but more general) arrays is less efficient than our "one-array" scheme: it requires more processors, and the total computation time is larger, even if we simply add the processing time of each simple array. However, the important price to pay in time when using several subarrays is the extra delay needed to store partial results in the host, and most often to rearrange them in order to get the right input format for processing by the next building block (see the discussion of [24] for the problem of computing the product of three matrices).

Steps i)–iii) above can be reduced to a single one if we choose to use an array for matrix inversion, such as that of Kramer and Van Leeuven [13] or Rote [27] (matrix inversion is a particular instance of the algebraic path problem [27]). However, these two arrays require n^2 pro-

cessors and, respectively, $6n$ and $7n$ time steps to compute the inverse A^{-1} of an $n \times n$ matrix A. Again, there remains to store A^{-1} in the host before using an other matrix–matrix multiplication array to get the final result.

It is also possible to compute $B A^{-1}$ using the systolic array of [5] and [25], since this computation is a particular instance of the Faddeev algorithm (see [5] for details). However, this array requires $n(3n + 1)/2$ elementary processors and outputs the product $B A^{-1}$ after $4n + p - 2$ units of time. The array that we have introduced here delivers the solution in the same number of time steps, but it is composed of only $n(n + 1)$ elementary processors, hence—to our knowledge—providing with the best area-time performances.

Finally, it is worthwhile to point out that it is possible to compute the product $A^{-1} B$ without any change in the array. Given two dense matrices A and B of respective dimensions $n \times n$ and $n \times p$, just feed into the array the $(n + p) \times n$ matrix

$$C = \begin{bmatrix} A^t \\ B^t \end{bmatrix}.$$

The array will output the matrix $B^t A^{-t} = (A^{-1} B)^t$, which means that the matrix $A^{-1} B$ will be delivered column by column (in $4n + p - 2$ time steps). In the particular case $p = 1$, the matrix B is a vector b with n components, and the array computes the solution x of the system $Ax = b$ within $4n - 1$ time steps. This remark provides a connection with the array of Robert and Trystram [26], since computing $A^{-1} B$ is a possible extension of this array which was originally designed to solve the algebraic path problem as defined in [27].

The fit between the structure of the problem (computing $B A^{-1}$ or $A^{-1} B$) and the array presented here is worth pointing out. If only the computation of A^{-1} were needed (which is a special instance of the algebraic path problem), we should use the better array of Kung, Lewis, and Lo [22].

Acknowledgment

We thank the referees for their helpful comments and suggestions, and for pointing out an error in Section III-A.

References

[1] H. M. Ahmed, J. M. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Comput.*, vol. 15, pp. 65–82, 1982.

[2] D. R. Brillinger, *Time Series, Data Analysis, and Theory*. San Francisco, CA: Holden-Day, 1981.

[3] P. R. Cappello and K. Steiglitz, "Digital signal processing applications of systolic algorithms," in *Proc. VLSI Syst. Comput.*, H. T. Kung *et al.*, Eds. Rockville, MD: Computer Science Press, 1981, pp. 245–254.

[4] ——, "Completely pipelined architectures for digital signal processing," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-31, pp. 1016–1023, Aug. 1983.

[5] H. Y. H. Chuang and G. He, "A versatile systolic array for matrix computations," in *Proc. 12th Symp. Comput. Architect.*, June 17–19, 1985.

[6] R. Kumar, "A fast algorithm for solving a Toeplitz system of equations," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-33, pp. 254–267, Feb. 1985.

[7] P. Comon and J. L. Lacoume, "A robust adaptive filter for noise reduction problems," in *Proc. ICASSP*, Tokyo, Japan, Apr. 7–11, 1986, pp. 2599–2602.

[8] P. Comon, Y. Robert, and D. Trystram, "Systolic implementation of adaptive systems," (in French), Ecole Centrale Paris, Res. Rep. C.R.E.D.I. 4, 1986.

[9] R. A. Evans, D. Wood, J. V. McCanny, J. B. McWhirter, and A. P. H. McCabe, "A CMos implementation of a systolic multi-bit convolver chip," in *Proc. VLSI 83*, Trondheim, Norway, F. Anceau *et al.*, Eds. North-Holland, 1983.

[10] B. Friedlander, "Lattice filter for adaptive processing," *Proc. IEEE*, vol. 70, pp. 829–867, 1982.

[11] K. Hwang and Y. H. Cheng, "Partitioned matrix algorithm for VLSI arithmetic systems," *IEEE Trans. Comput.*, vol. C-31, pp. 1215–1224, Dec. 1982.

[12] T. Kailath, "A view of three decades of linear filtering," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 145–181, 1974.

[13] M. R. Kramer and J. van Leeuwen, "Systolic computation and VLSI," in *Foundations of Computer Science IV*, J. W. DeBakker and J. Van Leeuwen, Eds. Amsterdam, The Netherlands: Mathematisch Centrum, 1983, pp. 75–103.

[14] A. V. Kulkarni and D. W. L. Yen, "Systolic processing and an implementation for signal and image processing," *IEEE Trans. Comput.*, vol. C-31, pp. 1000–1009, Oct. 1982.

[15] H. T. Kung, "Why systolic architectures," *IEEE Comput.*, vol. 15, pp. 37–46, 1982.

[16] ——, "Special-purpose devices for signal and image processing," in *Proc. SPIE's 24th Annu. Symp.*, San Diego, CA, 1980.

[17] H. T. Kung and M. S. Lam, "Wafer-scale integration and two-level pipelined implementations of systolic arrays," *J. Parallel and Distributed Comput.*, vol. 1, no. 1, pp. 32–63, 1984.

[18] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)," in *Proc. Symp. Sparse Matrices Comput.*, I. S. Duff *et al.*, Eds., Knoxville, TN, 1978, pp. 256–282.

[19] S. Y. Kung, "On supercomputing with systolic/wavefront array processors," *Proc. IEEE*, vol. 72, pp. 867–884, 1984.

[20] ——, "VLSI array processors," *IEEE ASSP Mag.*, vol. 2, pp. 4–22, 1985.

[21] S. Y. Kung and J. Annevelink, "VLSI design for massively parallel signal processors," *Microprocessors and Microsyst.*, Special Issue on Signal Processing Devices, vol. 7, no. 4, pp. 461–468, 1983.

[22] S. Y. Kung, P. S. Lewis, and S. C. Lo, "On optimally mapping algorithms to systolic arrays with application to transitive closure," in *Proc. IEEE Int. Symp. Circuits and Syst.*, San Jose, CA, 1986, pp. 1316–1322.

[23] S. Y. Kung, H. J. Whitehouse, and T. Kailath, Eds., *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[24] C. Moraga, "On a case of symbiosis between systolic arrays," *Integration, VLSI J.*, vol. 2, pp. 243–253, 1984.

[25] J. G. Nash, S. Hansen, and G. R. Nudd, "VLSI processor arrays for matrix manipulation," in *VLSI Systems & Computations*, H. T. Kung *et al.*, Eds. Rockville, MD: Computer Science Press, 1981, pp. 367–378.

[26] Y. Robert and D. Trystram, "Un réseau systolique orthogonal pour le problème du chemin algébrique," C. R. Acad. Sci. Paris, série I, vol. 302, p. 6, pp. 241–244, 1986.

[27] G. Rote, "A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion)," *Computing*, vol. 34, pp. 191–219, 1985.

[28] R. Schreiber and P. Kuekes, "Systolic linear algebra machines in digital signal processing," in *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1985, pp. 389–405.

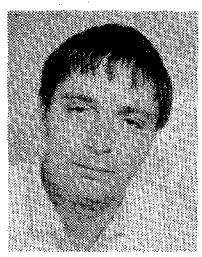
[29] H. L. Van Trees, *Detection, Estimation and Modulation theory*, 1st ed. New York: Wiley, 1968.

[30] T. Willey, R. Chapman, H. Yoho, T. S. Durrani, and D. Preis, "Systolic implementations for convolution, DFT and FFT," *Proc. IEE, F*, vol. 132, no. 6, pp. 466–472, 1985.

[31] H. C. Yung and C. R. Allen, "A programable VLSI array processor for IIR/FIR digital filtering," in *Digital Signal Processing*, V. Cappellini *et al.*, Eds. New York: Elsevier, North-Holland, 1984, pp. 197–201.

Pierre Comon (M'87) was born in Paris, France, in 1959. He received the Engineer degree in 1982, and the Thèse de Doctorat in 1985 both from the Institut National Polytechnique de Grenoble, France.

From 1982 to 1985 he was with Crouzet Valence, Department of Aerospace. Since 1981 he has been with the Random Phenomena and Geophysics Study Center (CEPHAG), Grenoble. In 1987 he is visiting the Information Systems Laboratory, Stanford University, Stanford, CA. His topics of interest include multivariate adaptive filtering, noise cancelling, high resolution analysis, and systolic arrays.



Yves Robert was born in Lyon, France, on September 5, 1958. He received the Thèse de Troisième Cycle degree in applied mathematics in 1982, and the Thèse d'Etat es Sciences in computer science in 1986, both from the Institut National Polytechnique de Grenoble, France.

Since 1983 he has been a Researcher at the Centre National de la Recherche Scientifique in the Laboratory TIM3, Grenoble. His fields of interest include design and analysis of parallel algorithms, systolic arrays, and VLSI architectures for digital signal processing. From November 1986 to November 1987, he will be with the IBM European Research Center ECSEC in Roma, Italy.