A VERSATILE SYSTOLIC ARRAY FOR MATRIX COMPUTATIONS

Henry Y. H. Chuang and Guo He†

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

Two critical factors affecting the utility of a VLSI processor array are 1) the versatility of the array, and 2) the size problem due to I/O constraints. In this paper we present a feedback systolic array system for matrix computations which, in addition to being able to produce high throughput, has improved utility. The array system can solve many matrix problems containing arbitrarily large matrices. It can also process sparse matrices efficiently by skipping blocks of zeros.

1. Introduction

VLSI processor arrays such as the systolic arrays can produce very high throughput due to the high degree of concurrent processing possible in them. However, the utility of such arrays suffers because 1) different array is generally required for different algorithm, and 2) only problems of fixed size can be solved in an array. Several recent research efforts have aimed at removing these shortcomings. Kung's programmable systolic chip[4] and Snyder's reconfigurable processor array [1] represent two major efforts to remove the first shortcoming, while the problem-size independent systolic array approach of Chuang and He [2] and the partitioned matrix algorithms for VLSI arithmetic systems of Hwang and Cheng [6] represent the efforts to remove the second shortcoming. A less drastic approach to remove the first shortcoming is to find algorithms and their array implementations which are general-purpose within a class of problems. Several signal processing arrays and matrix computation arrays [7,8] are representatives of this approach. As can be expected, this approach generally results in simpler processor and/or simpler interconnection, and thus more array cells can be put into a single chip.

For general-purpose matrix computations, Nash et. al[8]. have proposed a VLSI processor array based on Faddeev's algorithm[3]. This array, however, can only perform computations on fixed size matrices (usually small) and the submatrices cannot pipeline through the array due to the need of preloading. In this paper we present a more versatile VLSI array for matrix computations which is also independent of problem size. We obtain this array (or array system to be exact) by implementing Faddeev's algorithm with a systolic array which can be organized into a feedback array system capable of matrix computations involving arbitrarily large matrices. Our feedback array can also process sparse matrices efficiently by skipping blocks of zeros. In section 2 of this paper we describe Faddeev's algorithm and its functional implementation as a systolic array. In section 3 we discuss the I/O bandwidth and the size problems, and our approach to solve them. Section 4 describes our feedback array system. In section 5 we discuss how to efficiently deal with sparsity in matrices.

† Visiting scholar from Wuhan Digital Engineering Institute, Wuhan, China.

2. Faddeev's Algorithm and Systolic Array Implementation

Consider four matrices A, B, C, D of order n arranged in the following form:

$$\left[\begin{array}{c|c} A & B \\ \hline\hline -C & D \end{array}\right] = \begin{bmatrix} a_{11} & a_{1n} & b_{11} & b_{1n} \\ . & . & . & . \\ a_{n1} & a_{nn} & b_{n1} & b_{nn} \\ -c_{11} & -c_{1n} & d_{11} & d_{1n} \\ . & . & . & . \\ -c_{n1} & -c_{nn} & d_{n1} & d_{nn} \end{bmatrix} \tag{1}$$

It can be shown that if suitable linear combinations of the rows above the double line are found and added to the rows below the double line so that the lower left part becomes a zero matrix, then the lower right part will be equal to $E = CA^{-1}B + D$. When $D=0$, $C=I$ (the identity matrix), and $B=b$ (a column vector), E becomes the solution $A^{-1}b$ of the linear system $Ax=b$. When $D=0$ and $A=I$, E is the product of C and B. When $D=0$ and $B=C=I$, E is the inversion of A. Finding linear combinations of upper rows to eliminate the lower left part can be carried out by Gaussian elimination.

For the implementation of Faddeev's algorithm, Nash et. al[8]. have suggested a hexagonally connected rectangular array. For solving $Ax=b$, it requires $(n+1)^2$ array cells, while for the other operations $2n^2$ cells are required. The array can only process matrices of fixed size, and the matrices cannot pipeline through it due to the necessary preloading. Furthermore this array can only process Faddeev's algorithm.

Our implementation of Faddeev's algorithm is based on the matrix triangularization systolic array of Gentleman and Kung[5]. We extend their triangular array into a trapezoidal array and slightly modify the array cells. Fig. 1a

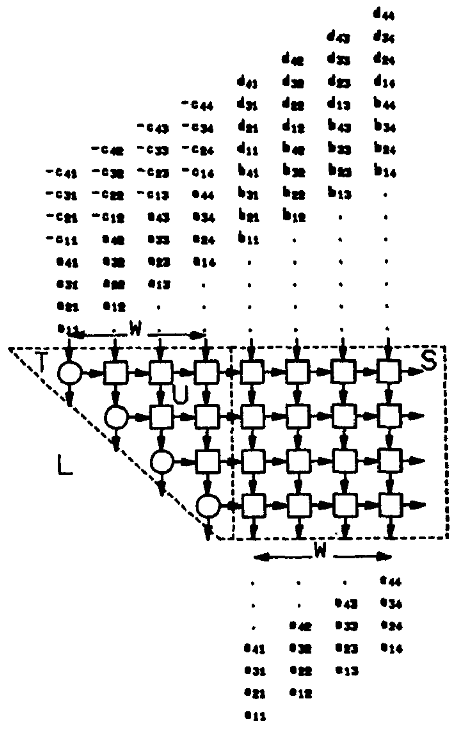
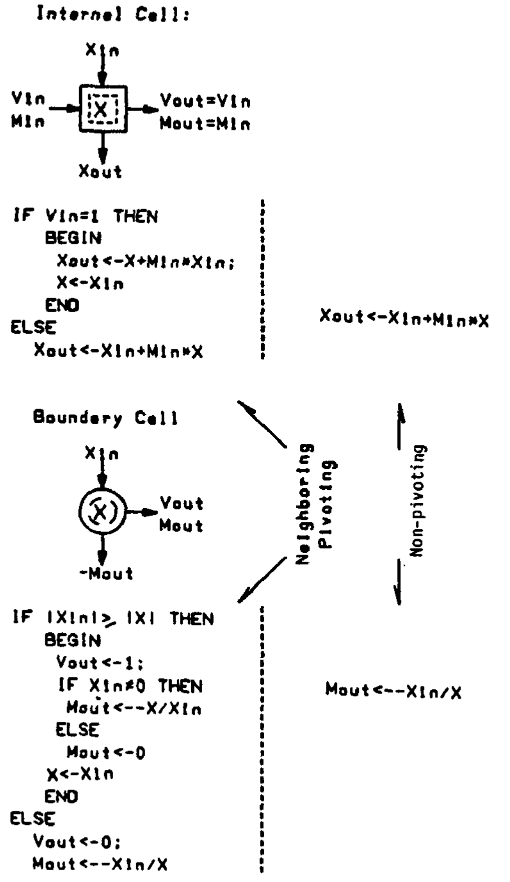


Figure 1.a



```
IF Vin=1 THEN
    BEGIN
     Xout<-X+Min*Xin;
     X<-Xin
    END
ELSE
  Xout<-Xin+Min*X
```

```
Xout<-Xin+Min*X
```

```
IF |Xin|≥ |X| THEN
    BEGIN
     Vout<-1;
     IF Xin≠0 THEN
      Mout<--X/Xin
     ELSE
      Mout<-0
    X<-Xin
    END
ELSE
  Vout<-0;
  Mout<--Xin/X
```

```
Mout<--Xin/X
```

Figure 1.b

shows the array and the input and output data flows which are skewed. Fig. 1b gives the cell functions for both pivoting and non-pivoting. The pivoting function is used to process the first half of the data flow (i.e., the part consisting of matrices A and B), and the non-pivoting function is used to process the second half (i.e., the part consisting of matrices C and D). The X value in each cell is preset to 0 before the operation begins. This array is decomposed into a triangular array T and a square array S. It can be shown that the T array can perform LU decomposition and triangularization, and the S array can compute matrix multiplication, inner product, and convolution. The combination of the T and the S arrays implements the Faddeev's algorithm by carrying out Gaussian elimination. Faddeev's algorithm by itself does not compute LU decomposition, convolution, or inner product. But, since the array can perform Gaussian elimination and the component arrays can perform other operations, more problems can be solved with it.

Each boundary cell stores an input data element and sends a modification factor rightwards to modify input data entering internal cells on the same row. Each internal cell also stores a data value arriving from the top and passes downwards all the following data after modification. Thus as a column of $2n$ input data flows downwards through the array, its length is shortened by n to become a column of n output data. It is easy to see that, when the array matches the I/O bandwidth, $5n$ steps are required to obtain $\cdot CA^{-1}B + D$, $4n+1$ steps are needed to solve a linear system of n equations, and $3n$ steps to do LU decomposition.

3. The I/O Bandwidth and The Size Problems

Since a VLSI processor array is typically attached to a host (CPU, memory, and interface), the throughput attainable in it is limited by the I/O bandwidth between the array and the host. A faster and larger array would not produce higher throughput. In the case of systolic arrays, a larger array might not be able to solve a larger problem either because of the strict timing a systolic array demands of its input data. The size and the processing speed of the array should, therefore, be chosen to match the I/O bandwidth.

When the problem to be solved is larger than the array, it is generally decomposed into subproblems to be solved in the array one at a time. The result of each subcomputation is then stored in the host for further processing in the array or within the host. The decomposition, if possible, and the postprocessing my be complex or time-consuming. Passing intermediate results between the host and the array reduces the throughput the I/O bandwidth can support. The interrupted data flow due to problem decomposition further reduces the throughput because of the resulting pipeline flush.

In order to more fully utilize the high throughput potential of VLSI arrays and not to burden users with the task of problem decomposition and postprocessing, we solve the problems of I/O bandwidth and problem size by structuring the array as a feedback array system. The feedback array system simulates the operation of an arbitrarily large array by using the small arrays over and over, with the output of the small arrays fed back to be processed with other input data at proper time.

The I/O path between the host and the array determines the width of input data which can enter the array and since this is generally smaller than the width of the input data flow required by a large array, the input data flow has to be decomposed. Suppose the width of the I/O path is w. We cut the data flow into strips of width w parallel to the direction of data flow, or bands of width w vertical to data flow. The strips and the bands are further cut into blocks of length w. Depending on the order in which the blocks are fed into the array, we have parallel, vertical, or hybrid decomposition as shown in Fig. 2.

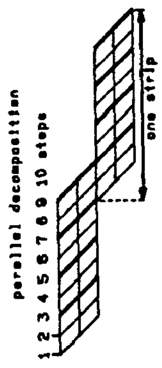


Figure 2.a

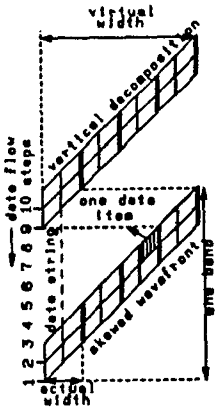


Figure 2.b

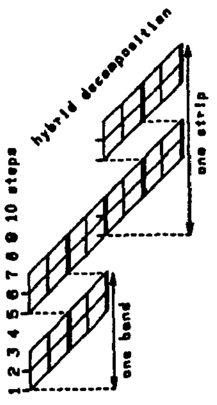


Figure 2.c

4. Feedback Systolic Array Systems

Different decompositions of the input data flow result in different feedback array systems, due mainly to the different buffer memory requirements for the feedback data. In this connection, we also want to point out that there are two types of internal data flow, one that varies as it moves across a cell and one that does not vary. Since the invariant type once saved can be used over and over, the write access to the buffer storing such internal data is drastically reduced. This means lower buffer access rate and possibly less memory ports. On the other hand, the variant type requires higher buffer memory access rate. It is, therefore, preferable to feed input data into the array in such a way that the requirement to save the variant type data in external buffer memory is lower. In the following, we shall discuss the feedback array systems for the three types of input data decomposition.

4.1 Feedback array for parallel data flow decomposition

To compute $CA^{-1}B + D$ for matrices of size n, the input data flow (the matrix in (1)) is $2n$ long and $2n$ wide. Suppose the host's I/O bus is w wide. We decompose the input data flow along its direction into $2m$ w wide strips $V_1, \ldots, V_{2m}$, where $n{=}mw$. Fig. 3 shows a full-size array (for $m{=}4$) composed of T and S component arrays of size w. Since only one input data strip can be fed into the array at a time, this full-size array will not work under the given I/O constraint. But the feedback array system of Fig. 4, which receives the input strips one at a time continuously without interruption, can process correctly under the same input constraint. The feedback array system simulates the large array by using the four S arrays and the single T array over and over with the output of the T array (the horizontal data) fed to the S arrays to meet the vertical data at the proper time. In order to use the component arrays over and over, a special value is appended to the end of each data strip to reset the arrays so that $X{=}0$ in each array cell before the next strip comes in. In this feedback array, feedback corresponds to repetition of S array in the horizontal direction, one feedback loop for each row in the array. The buffer memories in the feedback loops store the output data of the T array which are broadcast to the S arrays.

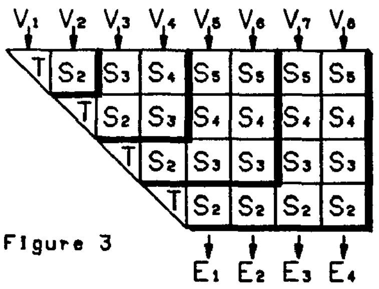
The total number of steps required to compute $CA^{-1}B + D$ in this feedback array system is

$$(2m \times 2mw + w) + mw = (4m + 1)n + w = O(mn)^{\dagger}$$

The first part on the left-hand side of this equation is the time required to input all the $2m$ input strips, and the second part is the additional time for V_{2m} to pass through the array system. This system is not quite independent of problem size because it requires m S arrays and m buffers. But, it uses much smaller number of array cells to produce maximum throughput under the given I/O constraint.

Because all the S arrays in this system are identical, we can replace the column of S arrays with a single S array with feedback. This corresponds to the repetition in the vertical direction. One more buffer memory B_v is needed to hold the internal data which flow down from one S array to the next in the array of Fig. 4. The resulting system shown in Fig. 5 consists of one T array and one S array only, and is completely independent of problem size except for the buffers which, however, can be implemented in an external memory. An input data strip no longer followed by another one immediately. Feedback data from the output of the S array have to be inserted between adjacent input data strips.

† $O(k)$ denotes order of k.



V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8
T S_2 S_3 S_4 S_5 S_5 S_5 S_5
T S_2 S_3 S_4 S_4 S_4 S_4
T S_2 S_3 S_3 S_3 S_3
T S_2 S_2 S_2 S_2
E_1 E_2 E_3 E_4

Figure 3

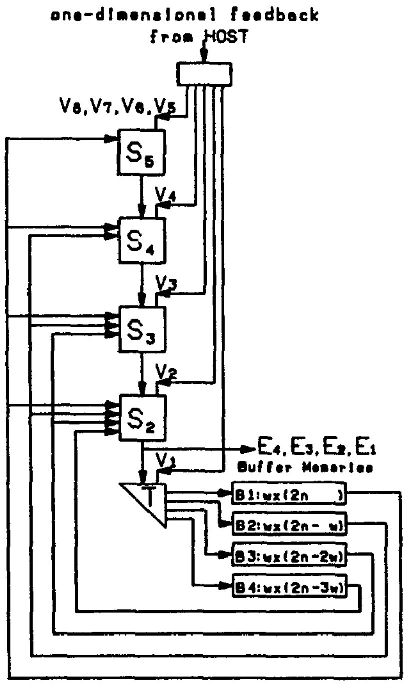


Figure 4

The throughput of this one-S one-T feedback array system is of course lower. The processing of the first strip needs $2mw$ steps. From the second to the m^{th} strip, processing one strip, e.g. the r^{th} strip, needs $\sum_{k=0}^{r-2}(2m-k)w$ steps. From the $m+1^{st}$ strip to the $2m^{th}$ strip, processing one strip needs $\sum_{k=0}^{m-1}(2m-k)w$ steps. The last strip needs w more steps to enter the array and w more steps to pass through the array, because its last block can not overlay with other strips. Therefore, the total processing time is

$$2mw + \sum_{k=1}^{m}(2m-k)(2m-k+1)w + 2w = \frac{7}{3}m^2n + \frac{5}{3}n + 2w = O(m^2n)$$

Using more than one S array in the feedback loop will reduce the processing time proportionally.

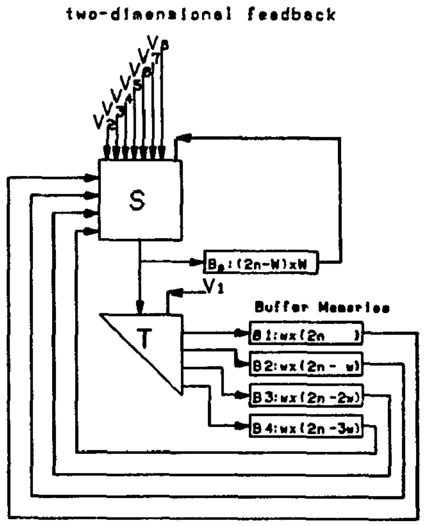


Figure 5

4.2 Feedback array for vertical data flow decomposition

An array wider than the I/O can solve a matching large problem only if the input data flow is skewed and it is decomposed vertically as shown in Fig. 2b. To see why this is so, again suppose the I/O is w wide and the array is $2n=2mw$ wide. We partition the array into subarrays of size w and interface the array with the host through a $2m$-way demultiplexor in the input side and an m-way multiplexor in the output side as shown in Fig. 6. The $2n$ by $2n$ input data flow is decomposed into $2m$ bands, vertical to its direction. Each band is further divided into $2m$ blocks, and the blocks in a band enter the array through the demultiplexor one by one. Because the input data flow is skewed, each block overlays with its left and right neighbors and the whole band enters the array continuously as if it were not decomposed. The last block of a band also overlays with the first block of the following band. So, the $2m \times 2m$ blocks enter the array continuously, and the total number of steps to complete the process is

$$(2m \times 2mw + w) + mw = (4m+1)n + w$$

which equals the processing time of the one-dimensional feedback array system of Fig. 4. Although the array of Fig. 6 has many more subarrays, the processing speed is not higher. This is because the array has many more cells than that the I/O bandwidth can support causing many array cells to idle.

We can remove the excess subarrays and at the same time achieve problem-size independence by using feedbacks. The feedback array system shown in Fig. 7 simulates the large array of Fig. 6, and performs the computation with the same number of steps. As in the array of Fig. 6, data enter the array system block by block without interruption.

Since the blocks in a band are processed continuously, the data moving rightwards (the invariant type) need not be saved, and no buffer is needed in the feedback loop. On

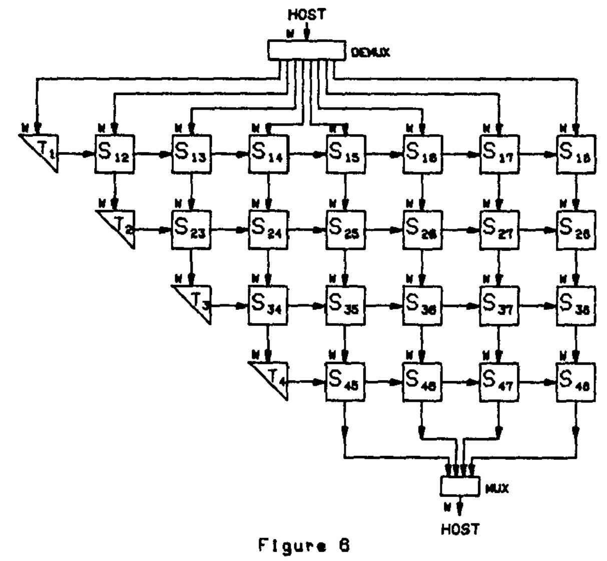


Figure 6

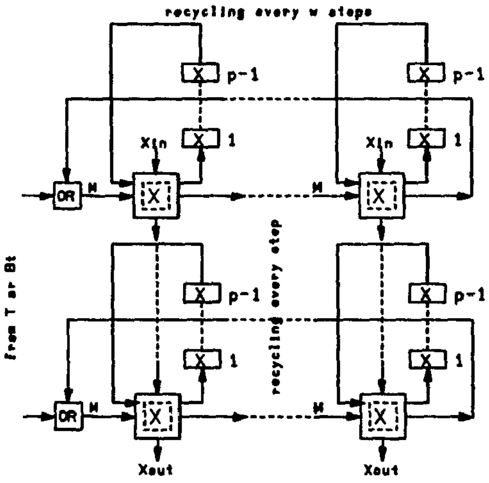


Figure 8

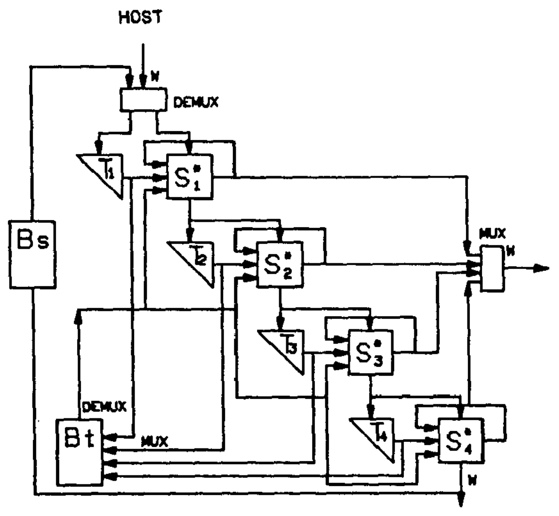


Figure 7

the other hand, the X values in the cells of the S array need to be saved as shifting into neighboring block begins, because these X values will be used in processing the next band of data. These X values are stored in a recycling shift register in each cell of the S array. The S^* array in the figure is the S array augmented with recycling shift registers for storing the X values and feedback loops for propagating the M values, as shown in Fig. 8. The B_s is for storing the X_{out} values when the problem is too big ($2n>4w$ for the array system of Fig. 7). In this case, the X_{out} values stored in B_s are processed after the whole data flow (or the whole strip) has passed the array system. Storing the X values in the recycling shift registers rather than in an external buffer simplifies control and reduces memory access and, through time sharing of memory, the number of memory ports.

4.3 Feedback array for hybrid data flow decomposition

The recycling shift registers of Fig. 8 have finite capacity of p. To cope with this finite capacity problem, a hybrid data flow decomposition as shown in Fig. 2c is used. The data flow is divided into parallel strips of pw wide. Blocks in each strip are processed as in vertical decomposition, and processing moves from strip to strip as in parallel decomposition.

During the processing of a block the X values stored in the output side of the shift registers (stored when processing the corresponding block in the previous band, i.e. the one p block ahead) will be consumed, and the new X values can be stored in the input side of the shift registers. When the processing shifts from one band to the next, however, the flow of the data moving rightward is interrupted. These data have to be saved until processing of the corresponding band in the next strip begins. The buffer B_t in Fig. 7 is for storing these data.

5. Sparsity in Matrix

Our feedback array system has another important merit in that we can skip blocks of zeros in the input data flow, and thus greatly reduce the processing time. Consider the linear system

$$AX = B \qquad (2)$$

where A is a lower blocked band matrix of order n, i.e.,

$$A = \begin{bmatrix} A_{11} & & & & & & \\ \cdot & A_{22} & & & & & \\ \cdot & \cdot & \cdot & & & & \\ A_{p1} & A_{p2} & \cdot & A_{pp} & & & \\ & A_{p+1,2} & \cdot & \cdot & A_{p+1,p+1} & & \\ & & \cdot & \cdot & \cdot & \cdot & \\ & & & A_{m,m-p+1} & & \cdot & \cdot \; A_{mm} \end{bmatrix}$$

and $B = \left[B_1, B_2, \ldots, B_m\right]^T$, $n = mw$, and each A_{ij} or B_i, $i=1,2,\ldots,m, 1 \leqslant j \leqslant i$, is a w by w submatrix.

The data flow is decomposed parallelly into w wide strips and w by w blocks as shown in Fig. 9a. The -1 blocks are the diagonal submatrices of the $-I$ matrix. Without loss of generality, we have assumed the B matrix to be n by w. As shown in Fig. 3, V_1 is processed by a T array which generates horizontal data flow (the M's) to modify data strips on the right, i.e. V_i, $i \geqslant 2$. V_2 is processed by one S array and then one T array which generates horizontal data flow to further modify data strips on the right. When the leading block of zeros in V_2 passes through the S array S_2, the X value in each cell of S_2 will be zero, and so $X_{out} = X_{in}$ in each cell. Namely, all the following blocks A_{i2}, $i=2,3,\ldots,m$ will pass through S_2 unchanged. This means the S array is not required to process V_2, and the first block of zeros can be skipped. Similarly, the first two blocks of zeros in V_3 result in $X=0$ in each cell of S_2 and S_3, and so we can skip these two blocks of zeros and use only a T array to process all other blocks in this strip. For the same reason every block of zeros above the main diagonal of matrix A can be skipped, and the whole input data flow except for the strip containing the B matrix can be processed with just one T array. In processing the input data strips with the T array, the blocks of zeros between the lower diagonal border of the band matrix and the main diagonal of the $-I$ matrix can also be skipped. This is because $M_{out}=0$ when $X_{in}=0$ in the boundary cell and $X_{out} = X_{in}$ when $M_{in} = M_{out} = 0$ in the internal cell. These blocks of zeros do not contribute to the modification of data strips on the right and, therefore, can be skipped. For the same reason, the blocks of zeros below the main diagonal of the $-I$ matrix can also be skipped.

In Fig. 3 the data strip consisting of B, i.e. $V_{m+1} = B_1, B_2, \ldots, B_m$, passes through m S arrays $S_{m+1}, S_m, \ldots, S_2$ in that order. As the data strip passes S_{m+1}, B_1 is modified and stored in there. $B_2, B_3, \ldots, B_p$, where p is the number of non-zero blocks in a column of A, are modified and driven out. B_{p+1} to B_m pass it without modification because the T array on the same row, in processing the A matrix portion of V_1, produces only p non-zero blocks of M values to modify this data strip. The only non-zero block in the $-I$ matrix portion of V_1 produces a non-zero block of M values to turn the first zero block below B_m into a non-zero block $X_1(1)$. In a similar way the second S array S_m processes the sequence of data blocks entering it from the output of S_{m+1}.

Because we don't need S arrays to process V_1 to V_m, we can supply the M data required to modify V_{m+1} from the output of T arrays directly. If we use only one T array to process V_1 to V_m and do not store the M data in the buffers, we cannot use more than one S array to process V_{m+1} concurrently because the T array can only supply the

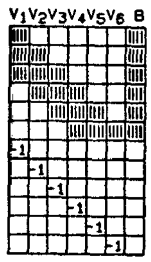


Figure 9.a

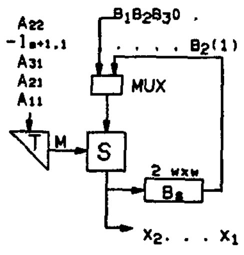


Figure 9.b

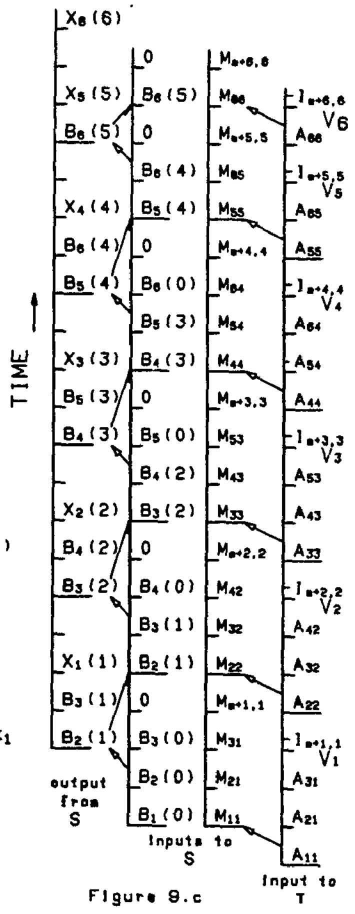


Figure 9.c

M data to one S array at a time. Therefore, we shall use a single S array with feedback to implement the sequence of S arrays S_{m+1} to S_2. Fig. 9.b shows a feedback array system for solving linear system with lower blocked band matrix. It is to be noted that this system, consisting of one T array and one S array, is a degenerated version of the one in Fig. 5 with output of the T array feeding the S array directly and with the input data strips bypassing the S array. Fig. 9.c shows the order in which the data blocks are processed in the T array and the S array. $B_j(k)$ denotes the j^{th} block of B after being processed by the S array k times. $X_j(j)$ is the j^{th} block of the solution generated during the j^{th} iteration. The empty arrows in the figure denote pipelining, while the solid arrows denote feedback.

Since in the array system of Fig. 9.b the processing of the data strip B and the processing of the rest of the data strips are concurrent, the total processing time is the time required to pipeline all the non-zero blocks in matrices A and $-I$ which is

$$\left(\sum_{k=m-p-1}^{m} k + m \right) w = (p+1)n - \frac{1}{2}p(p-1)$$

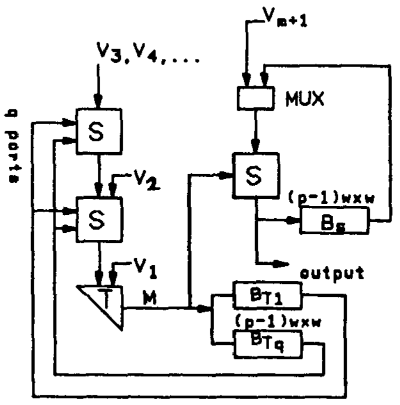


Figure 10

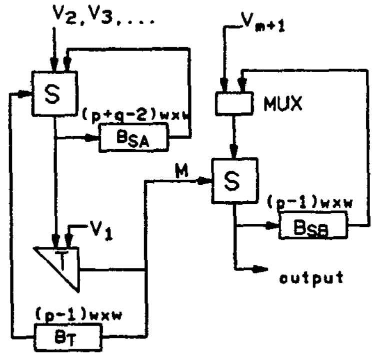


Figure 11

When matrix A has non-zero blocks above the main diagonal, the sparsity cannot be effectively utilized in Faddeev's algorithm directly to increase the processing speed. A single non-zero block above the main diagonal will produce, in the Gaussian elimination, a column of non-zero blocks in the $-I$ matrix above its main diagonal. Therefore, a large number of additional blocks have to be processed if the A matrix is not a lower blocked band matrix. A better way to solve a problem with such a matrix is to solve it in two phases. In the first phase the Gaussian elimination is applied to the part of input data flow consisting of only A and B resulting in an upper band matrix A_U and a modified B denoted as B_U. A_U is then transposed horizontally and vertically to obtain a lower band matrix A_L. B_U is turned upside down to become B_L. In the second phase, Faddeev's algorithm is applied to the whole input data flow consisting of A_L, B_L and the $-I$ matrix with the zero blocks skipped as mentioned above.

Suppose matrix A has an upper band of width q and a lower band of width p. The array system of Fig. 10 can process the first phase with maximum speed when the parallel decomposition of input data flow is used. V_1 to V_m are processed in the column of q S arrays ($q=2$ in the figure) and one T array on the left one by one. Concurrently V_{m+1} is processed in the S array with feedback loop on the right. By the same reason as explained previously, the blocks of zeros can also be skipped in this phase. In processing one data strip, the q non-zero blocks above the main diagonal are modified and stored in the q S arrays, the block on the main diagonal is modified and stored in the T array, and the remaining $p-1$ non-zero blocks in the lower band are modified and used to produce the M data to modify V_{m+1}. Therefore, the capacity of each of the buffers is $p-1$ blocks. Since the blocks of zeros can be skipped and the processing of B is concurrent with the processing of A, the processing time for the first phase equals the time to enter all the blocks between the upper and lower boundaries of the band in A:

$$(\sum_{i=0}^{p-1} m-i)\,w + (\sum_{j=1}^{q} m-j)\,w = (p+q)n - \frac{1}{2}(p^2 + q^2 - p + q)\,w$$

Because $p+q$ is the band width of matrix A, the processing time for the first phase is approximately the product of the order and the bandwidth of A. Comparing the array system in Fig. 9.b with the one in Fig. 10, we see that the system in Fig. 10 can also process the second phase. Since the processing time for phase 2 is approximately the product of the order and the lower bandwidth of matrix A, the total processing time is approximately $(2p+q)n$.

The array system of Fig. 10 is not problem-size independent because it requires q S arrays. To make it size independent, we can replace the column of q S arrays with a S array with feedback as shown in Fig. 11. The processing time for the first phase is now about q times as much because all data strips, except for the first one, have to be processed by the single S array q times. The processing time for the second phase is not affected because the column of q S arrays are not used in the phase. The total processing time in this array system is, therefore, approximately $q(p+q)n + (p+1)n$.

References

1. L. Snyder , "Introduction to the Configurable Highly Parallel Machine," *IEEE Computer*, pp. 47-64, January, 1982.
2. H. Y. H. Chuang and G. He, "Design of Problem-size independent Systolic Array Systems," *Proceedings of the International Conference on Computer Design: VLSI in Computer*, October, 1984 .
3. D. K. Faddeev and V. N. Faddeeva, in *Computational Methods of Linear Algebra*, pp. 150-158, W. H. Freeman and Company, 1963.
4. A. L. Fisher, H. T. Kung, L. M. Monier, H. Walker, and Y. Dohi, "Design of the PSC: A Programmable Systolic Chip," *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pp. 287-302, Computer Science Press, March, 1983.
5. W. W. Gentleman and H. T. Kung, "Matrix Triangularization by Systolic Arrays," *Proceedings of SPIE* --

The International Society of Optical Engineering, vol. 298, pp. 19-26, 1981.

6. K. Hwang and Y. H. Cheng, "Partitioned matrix Algorithm for VLSI Arithmetic Systems," *IEEE Trans. on Computer*, vol. C-31, no. 12, pp. 1215-1224, December,1982.
7. H. T. Kung, "Special-purpose Devices for Signal and Image Processing," *SPIE's 24-th Annual Technical Symposium*, San Diego, California, July 28 - August 1, 1980.
8. J. G. Nash, S. Hanson, and G. R. Nudd, "VLSI Processor Arrays for Matrix Manipulation," in *VLSI Systems and Computations*, pp. 367 - 378, Computer Science Press , October, 1981.