FOLDING AND UNROLLING SYSTOLIC ARRAYS*

K. Culik II and J. Pachl
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-82-11
April, 1982

* This research was supported by the Natural Sciences and Engineering Research Council Canada under grant No. A7403.

Abstract

This paper is about two constructions for transforming planar systolic arrays. By a systolic array we mean a uniformly structured processor configuration that allows only local communication (between neighbour processors) and supports high throughput by pipelining. Mead and Conway argue that such configurations are suitable for VLSI implementation ([4], p. 271).

We show how new connections in a systolic array can be established by folding the array along a line; and how computations in one-dimensional arrays unroll to two-dimensional structures resembling trellis automata [2]. We construct systolic arrays for computing large powers of a matrix; for matching keywords; for recognizing square-free words; and for finding the transitive closure of a relation.

1. Folding Systolic Arrays

When constructing complex systolic arrays from simple ones, one often wishes to connect the nodes in one set to the corresponding nodes in another. E.g. in the array in Figure 1, the node a_1 is to be connected to b_1, a_2 to b_2, etc.

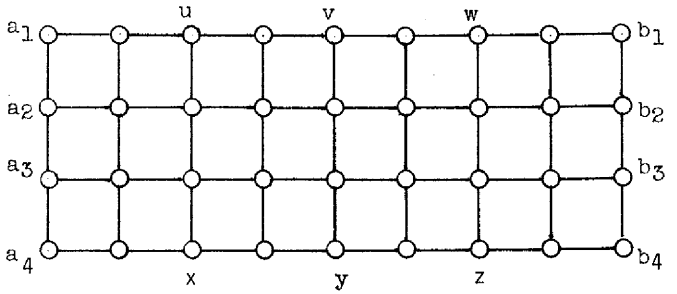

Figure 1

Long connections, like those in Figure 2, violate the systolic design principles; we would prefer to keep the lengths of all connections in the array approximately equal.

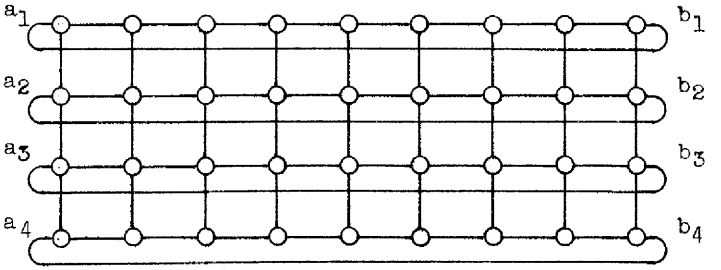

Figure 2

By folding the array in Figure 1 along the line vy, we preserve the original grid pattern and establish the desired connections (Figure 3).

1. Folding Systolic Arrays

When constructing complex systolic arrays from simple ones, one often wishes to connect the nodes in one set to the corresponding nodes in another. E.g. in the array in Figure 1, the node a_1 is to be connected to b_1, a_2 to b_2, etc.

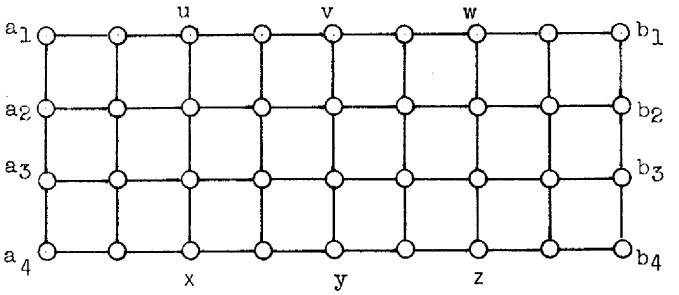

Figure 1

Long connections, like those in Figure 2, violate the systolic design principles; we would prefer to keep the lengths of all connections in the array approximately equal.

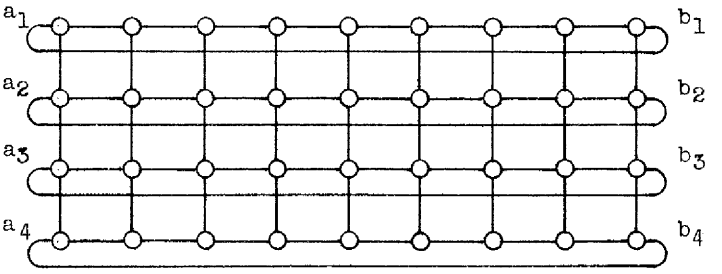

Figure 2

By folding the array in Figure 1 along the line vy, we preserve the original grid pattern and establish the desired connections (Figure 3).

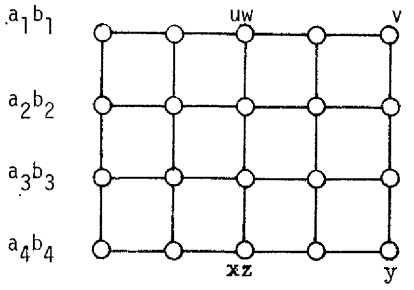

Figure 3

The small size of the array in Figure 3 may be misleading; the nodes and connections have to do more work than those in Figure 2. The new layout can be more or less expensive, depending on the cost function used. Nevertheless, Figure 3 is more regular than Figure 2, and it conforms to the systolic philosophy. It can also be implemented in two layers; or, alternatively, the two overlapping computations can be multiplexed in time.

For the square grid in the plane, the (repeated) folding construction can be used to identify any two sequences that are congruent in the sense of the following definition.

Definition. A (planar) layout is a collection of curves in the plane; the endpoints of the curves are the nodes of the layout. Two (finite or infinite) sequences (a_i) and (b_i) of nodes are congruent (relative to the layout) if there is a distance-preserving 1-1 transformation of the plane that maps the layout onto itself and each a_i to b_i. □

The square grid in the plane has the following property, which is also satisfied by a number of other planar patterns. If A and B are two congruent sequences of nodes in the infinite planar square grid,

then A can be transformed to B by applying at most four reflections. (In this paper, a reflection is always that of the plane about a line. We only consider the reflections that preserve the layout in question.) It follows that A and B can be identified (connected) by applying at most four foldings. Repeating the construction $k-1$ times, we get this result:

Theorem 1. If $A_1, A_2, \ldots, A_k$ are pairwise congruent sequences in the infinite planar square grid, then $A_1, A_2, \ldots, A_k$ can be identified by applying at most $4(k-1)$ foldings.

Proof: The general case follows from the case $k = 2$. Let $A = (a_1\ a_2\ \ldots\ a_n)$ and $B = (b_1\ b_2\ \ldots\ b_m)$ be two congruent sequences in the infinite planar square grid. Assume, without loss of generality, that the sequence A is not collinear and $a_1 \neq a_2$ (hence also B is not collinear and $b_1 \neq b_2$).

A plane translation, which is a composition of two layout-preserving reflections, maps a_1 to b_1; let $C = (c_1, c_2, \ldots, c_n)$ be the image of A. Thus $c_1 = b_1$, and B and C are congruent. If B and C have the same orientation then a rotation about b_1 maps C to B; since B and C are congruent relative to the grid, the rotation is by a multiple of 90°, and therefore it is a composition of two layout-preserving reflections. If B and C have opposite orientations then a single layout-preserving reflection maps c_2 to b_2 and hence also c_i to b_i for all $i > 2$.

We conclude that there are four or fewer reflections whose composition maps A to B. The composition of the corresponding

foldings identifies A and B .

□

In many cases the connections can be established with fewer than $4(k-1)$ foldings. It is obviously desirable to use as few foldings as possible, to keep the combined nodes small.

The same results hold for the triangular grid, which is used to lay out the hex-connected arrays ([4], ch. 8).

The theorem can also be applied to "approximately congruent" sequences. It is not necessary to exactly identify (a_i) and (b_i) in order to connect them; if (a_i) is congruent to (c_i) and if b_i and c_i are endpoints of the same curve of the layout for each i, then (a_i) can be identified with (c_i) and hence connected to (b_i) by one curve of the layout. This point arises in Example 1 below, where such an approximate identification requires fewer foldings than exact identification would.

Example 1. Using the folding technique we construct a systolic array on triangular grid, with no communication except between adjacent nodes, to compute A^m for an input matrix A of fixed dimension n and $m = 2, 4, 8, \ldots, 2^k, \ldots$.

It is based on the Kung and Leiserson matrix-multiplication array ([4], pp. 276-280), used here for dense matrices. A straightforward modification of the array from [4] is shown in Figure 4, for $n = 3$. First we feed the input matrix A from both left and right, to compute $A^2 = A \cdot A$. Then in each iterative step we compute $A^{2^{k+1}} = A^{2^k} \cdot A^{2^k}$ by feeding the output from the k-th step as both "left"

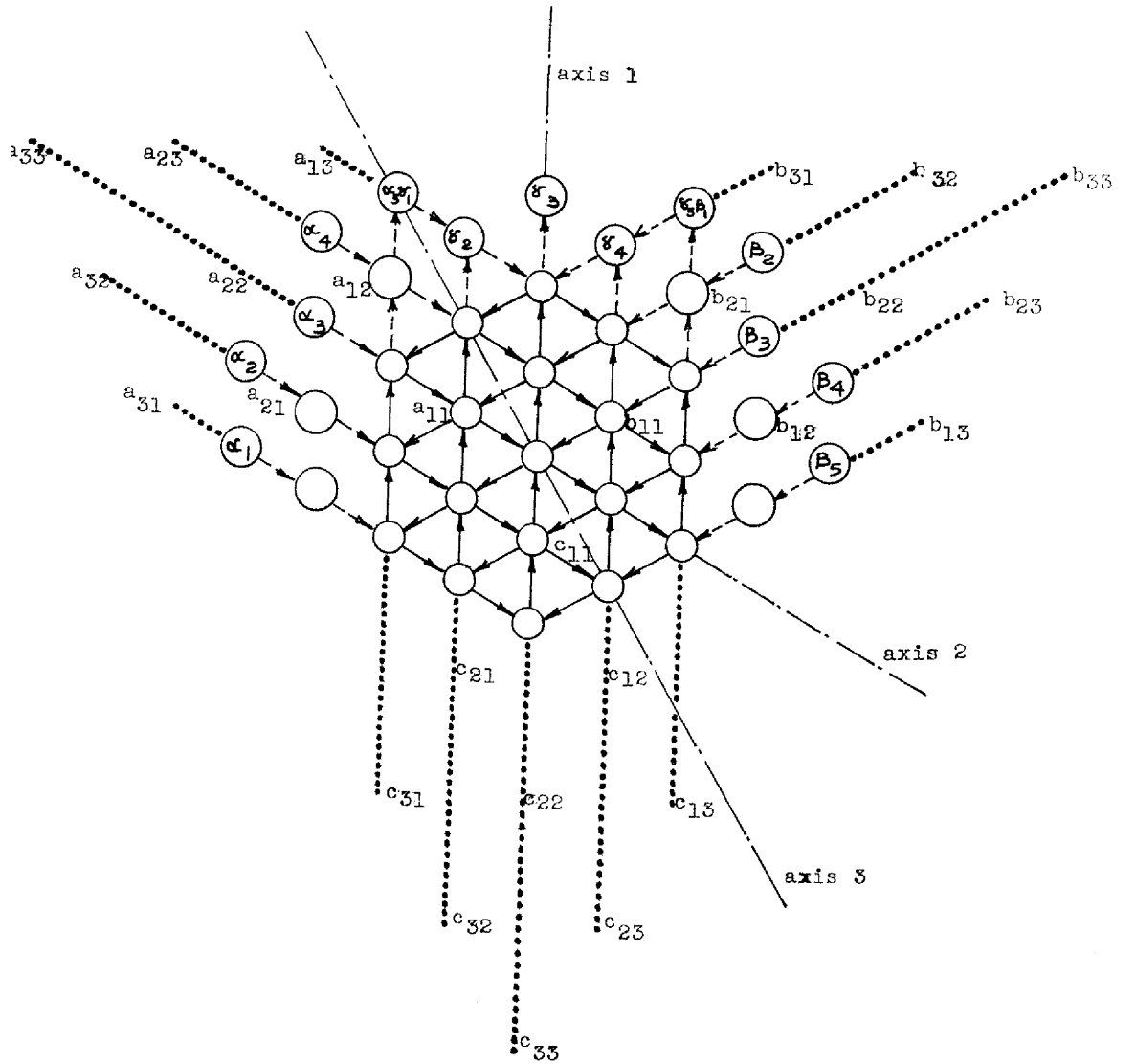
axis 1
axis 2
axis 3
a_{33}
a_{23}
a_{13}
a_{32}
a_{22}
a_{12}
a_{31}
a_{21}
a_{11}
b_{31}
b_{32}
b_{33}
b_{21}
b_{22}
b_{23}
b_{11}
b_{12}
b_{13}
c_{11}
c_{21}
c_{12}
c_{31}
c_{22}
c_{13}
c_{32}
c_{23}
c_{33}
α_1
α_2
α_3
α_4
β_2
β_3
β_4
β_5
γ_1
γ_2
γ_3
γ_4
γ_5

Figure 4

Now we apply the folding technique explained after Theorem 1 in order to achieve "local" connections between α_i and γ_i and between β_i and γ_i for $i = 1, 2, \ldots, n$. This is done by folding the layout along the axes numbered 1, 2, and 3, in that order. and "right" input of the (k+1)-th step. This is done by non-local connections from γ_i to α_i and β_i, $i = 1, \ldots, n$, and suitable delays between the original array and point γ_i and between $\alpha_i(\beta_i)$ and the original array.

The resulting array has only local connections on the triangular grid. At most eight processors of the original array are mapped to one. The same holds for the communication lines. The resulting array has $O(n^2)$ processors and computes the matrix A^m in $(3n-2)\log m$ clock cycles.

2. Linear Systolic Arrays with One Parallel Input

A linear systolic array is a semi-infinite one-dimensional configuration of processors in which only neighbours communicate in each clock cycle (Figure 5).

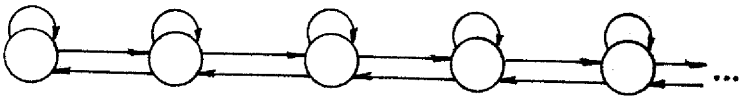

Figure 5

The circles are memoryless processors. Each arrow represents a communication that takes place during one time unit (typically the arrows will be implemented by clocked registers).

We now consider an array to which external input (a sequence of symbols) of length n is fed in parallel, one symbol to each of the n leftmost cells. Assume that the output is emitted from the leftmost processor; all the other processors are identical (Figure 6).

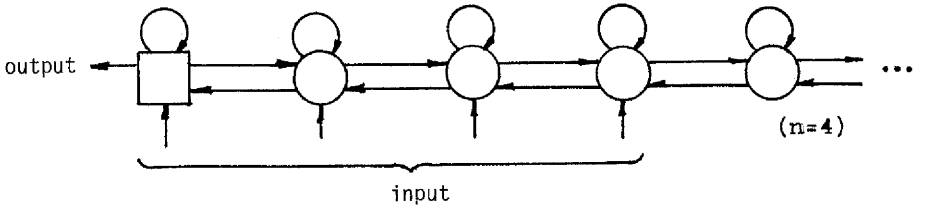

Figure 6

If the input length is n, we are particularly interested in the output symbol produced n time units after the input has been read in. This is the earliest time when the output depends on all input symbols; we say that the output is produced in pseudo-realtime.

The pseudo-realtime computation in the linear systolic array can be graphically arranged as a two-dimensional map, in which one dimension represents time (Figure 7).

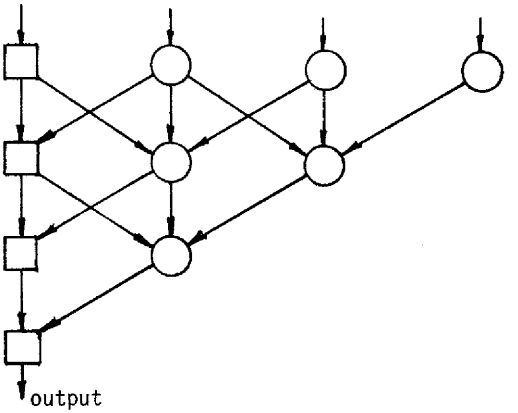

Figure 7

This picture may be regarded either as the logical description of the computation in the linear array or as the physical layout of a two-dimensional array which computes the same output. By creating additional nodes and changing transmission paths, we transform the structure into one whose connections are as in the trellis automata of [2]; see Figure 8.

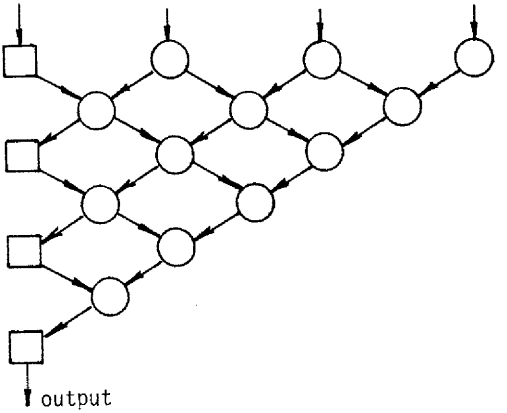

Figure 8

This is one half of a trellis in the sense of [2] . The construction shows how trellises, which are well understood, can be used to design linear systolic arrays. The pseudo-realtime computation of the array on an input sequence w is simulated by a trellis computation on the sequence $\$^{n-1}w$ (where $\$$ is a new symbol and n is the length of w).

Theorem 2. To every linear systolic array corresponds a homogeneous trellis such that the pseudo-realtime output of the array on the parallel input w is the same as the output of the trellis on the input $\$^{|w|-1}w$. Conversely, every homogeneous trellis whose input alphabet contains $\$$ corresponds to a linear systolic array in this way.

Proof: The half-trellis in Figure 8 is simulated by the homogeneous trellis in Figure 9 as follows: Use two new symbols, $\cent$ and $\#$. The input symbol $\$$ is transformed to two $\cent$'s. If a cell receives at least one $\cent$, it sends $\#$ to both outputs; if it receives $\#$ on both inputs,

it sends ¢ to both outputs; and if it receives # from the left and another symbol from the right, then it sends ¢ to the left and on the right it simulates the leftmost cell of the linear array.

Conversely, every homogeneous trellis can be folded in the middle to yield a half-trellis such that the half-trellis on any input w simulates the trellis on the input $\$^{|w|-1}w$. The half-trellis, such as that in Figure 8, then corresponds to an unrolled computation of a linear systolic array, such as that in Figure 7.

□

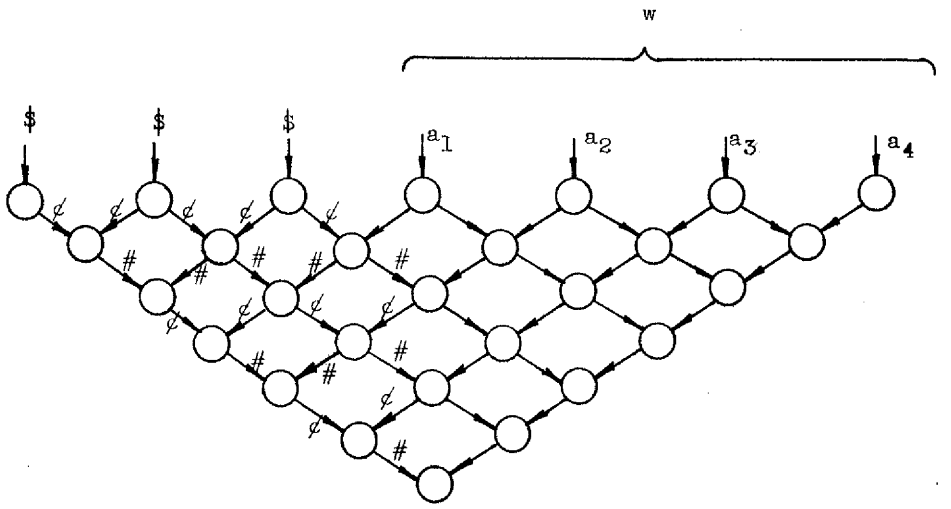

Figure 9

Corollary. Every linear context-free language is accepted in pseudo-realtime by a linear systolic array.

Proof: It is proved in [2] that every finite intersection of linear languages is accepted by a homogeneous trellis. If L is a linear language, then $L_{\$} = \{\$^{n-1} w \mid w \in L , |w| = n\}$ is the intersection of two linear languages. It follows that $L_{\$}$ is accepted by a homogeneous trellis and, by the theorem, L is accepted by a linear systolic array in pseudo-realtime.

□

Example 2. There is a linear systolic array that decides whether at least one of finitely many keywords occurs in a given string. The array is universal; the string and the keywords are read as input. The construction is based on the fact that the language

$$\left\{ s\#k_1^R\#k_2^R\#\ldots\#k_\ell^R \mid \exists i , \ 1 \le i \le \ell , \ k_i \text{ is a substring of } s \right\}$$

is linear. (# is a new symbol, and k_i^R is the keyword k_i in reverse order.) Theorem 2 shows how to design a parallel linear systolic array that accepts or rejects every string $s\#k_1^R\#k_2^R\#\ldots\#k_\ell^R$ (in pseudo-realtime) depending on whether s does or does not contain a substring k_i.

3. General Linear Systolic Arrays

In the previous section we concentrated, for the sake of simplicity, on the linear systolic array that reads its input at a single moment in time and is then left on its own to compute the output. A more general linear systolic array reads new inputs throughout its computation.

As before, we restrict our attention to the part of the array computation that leads to the output produced by one cell at one time. (The whole computation is an interleaved aggregate of such simple computations.)

We again unroll the computation in time, and arrange it as a trellis-like structure. The difference is that the resulting half-trellis reads input symbols in various interior nodes, while the half-trellis in the previous section has all its input nodes within one row. In the array in Figure 10,

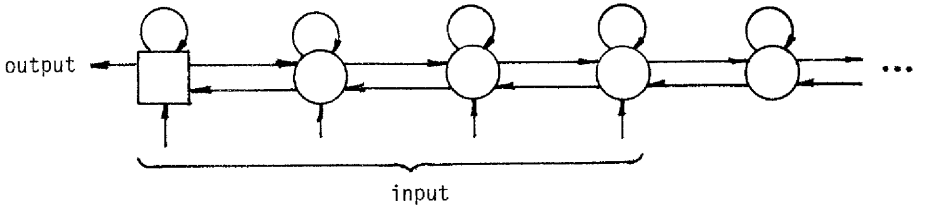

Figure 10

the computation that leads to the output produced after three time units unrolls as in Figure 11.

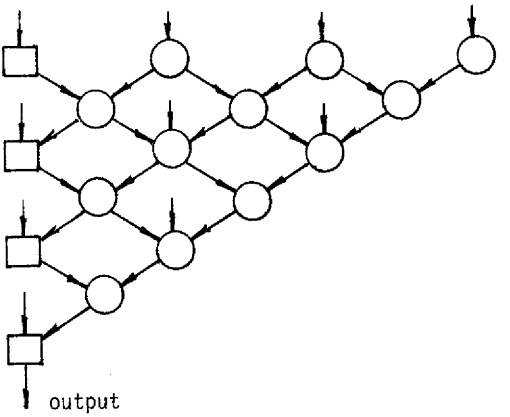

Figure 11

The vertical arraows in Figure 11 stand for external input and output.

The one-dimensional iterative array of Cole [1] is a special case of a linear systolic array; the input is read sequentially by the leftmost cell. The corresponding unrolled trellis-like structure (Figure 12) is a useful design tool.

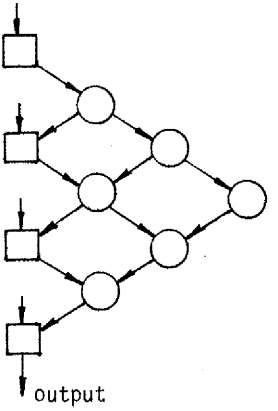

Figure 12

For instance, the now standard problem of constructing a one-dimensional iterative array to accept palindromes in realtime [1, 3] can be easily solved, and, what is perhaps more improtant, the solution can be intuitively understood, on this diagram.

Similarly, the one-dimensional iterative array to recognize the language $\{ww \mid w \in \{a, b\}^*\}$ (see [1]) unrolls as in the following example.

Example 3. We want to design a linear array (with sequential input) to recognize the language $\{ww \mid w \in \{a, b\}^*\}$ in real time. We start with the diamond-shaped trellis structure in Figure 13. The solution combines three concurrent computations. The first sends every input symbol diagonally right; as a result, the second half of the input word of length n is available at the bottom right edge of the diamond

corresponding to the input of size n . (In Figure 13, the word abaab is available at the bottom right edge when the input word abaababaab has been read in.)

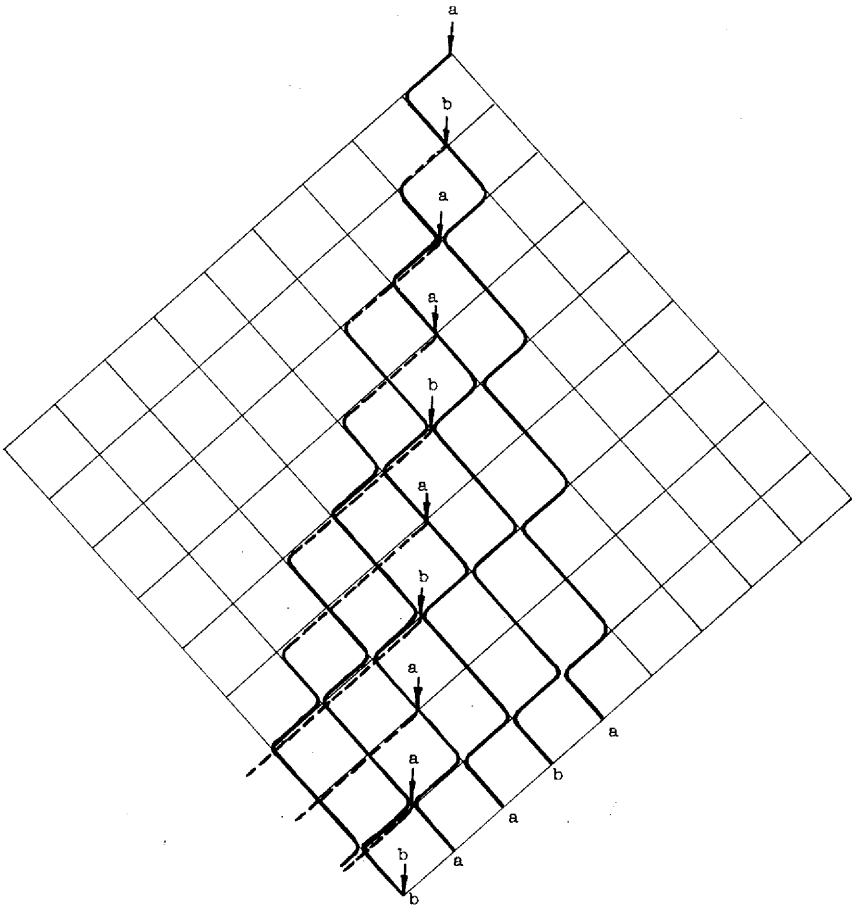

Figure 13

The second computation transports the first half of the input string to the same position at the bottom right edge. The data paths are traced in Figure 13. Every input symbol moves diagonally left (along a dashed line in the figure) until it hits a free trajectory to the right. Then it moves right as indicated by the full line: two moves right and one left. Note that an edge in the trellis is shared by at most two data paths in this second computation; only finite amount of information is therefore stored at each node.

The third computation collects the results of comparisons along the bottom right edge of the diamond. If all comparisons are successful, the output (at the bottom corner of the diamond) indicates success.

To get the structure of Figure 12 (the unrolled computation of a one-dimensional iterative array), we fold Figure 13 along its vertical axis of symmetry.

Example 4. The systolic array in Example 3 is a basic component in the array that we now construct, which recognizes the language $\{ww^kv \mid w \in \Sigma^+, u,v \in \Sigma^*\}$ for a fixed integer $k \geq 2$. We construct two versions of the array; one reads its input sequentially, the other in parallel.

We do not know whether the language can be recognized by a realtime systolic array with sequential input; our array needs additional time to broadcast input symbols and to collect results.

First we modify the construction in Example 3 to get an array that recognizes the language $\{w^kv \mid w \in \Sigma^+, v \in \Sigma^*\}$. Then we use the

unrolled form of the modified array (like that in Figure 12) and apply it to every suffix of input. There are two ways of doing this, depending on whether the input is read sequentially or in parallel.

In the case of sequential input, the n-th input symbol is broadcast to the top n input points in Figure 12. When the input string has been read, the results for all its suffixes are available along the left edge of the array: The answer whether the whole string belongs to the language $\{w^k v \mid w \in \Sigma^+, v \in \Sigma^*\}$ is at the bottom square box in Figure 12, the answer for the next shorter suffix at the box above it, etc. It then remains to collect the answers and OR them together. The broadcast and collection operations can be implemented via external connections (e.g. by a binary tree, whose time cost is logarithmic in some models); alternatively, either broadcast or collection can be implemented for free by means of the systolic conversion theorem [3].

The array with parallel input operates in pseudo-realtime. Again we start with the unrolled array of Figure 12 that recognizes $\{w^k v \mid w \in \Sigma^+, v \in \Sigma^*\}$, but now input is fed in parallel to the square boxes along the left edge. Every square box remembers its input symbol and retransmits it with each clock pulse. The answers produced sequentially by the array are ORed together at the bottom box. If the length of input is n then after n time units the bottom box has the answer.

We conclude with an example in which unrolling and folding are combined to construct a systolic array for computing the transitive closure of a finite relation.

Example 5. Our goal is to construct a systolic array that computes the transitive closure of a relation. Both the input and the output relation are represented by matrices of zeros and ones. We start with the linear systolic array ("linearly connected network" in the terminology of Kung and Leiserson) for matrix-vector multiplication described in ([4], pp. 274-276). By unrolling the computation in time, we get the trellis structure (direction from top-left to bottom-right) in Figure 14 (for $n = 3$), which computes

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

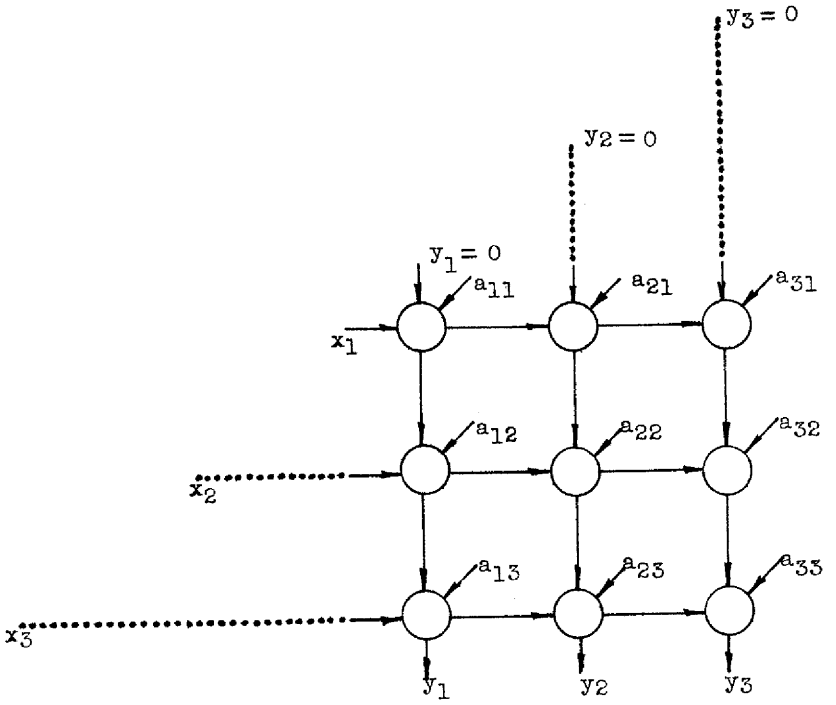

Figure 14

To find the transitive closure of the relation represented by an n x n matrix A , we want to compute $A + A^2 + \ldots + A^n$, with the + and · operations suitably interpreted.

We modify the array in Figure 14 by adding new non-local connections, shown in Figure 15, and then by folding to identify the sequences $(\alpha_{11}, \alpha_{21}, \ldots, \alpha_{n1})$, $(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1n})$ and

$(\alpha_{1n}, \alpha_{2n}, \ldots, \alpha_{nn})$. This can be done by three foldings. The result is a planar orthogonally connected array, which operates in two phases: In Phase I the matrix A is loaded to the array.

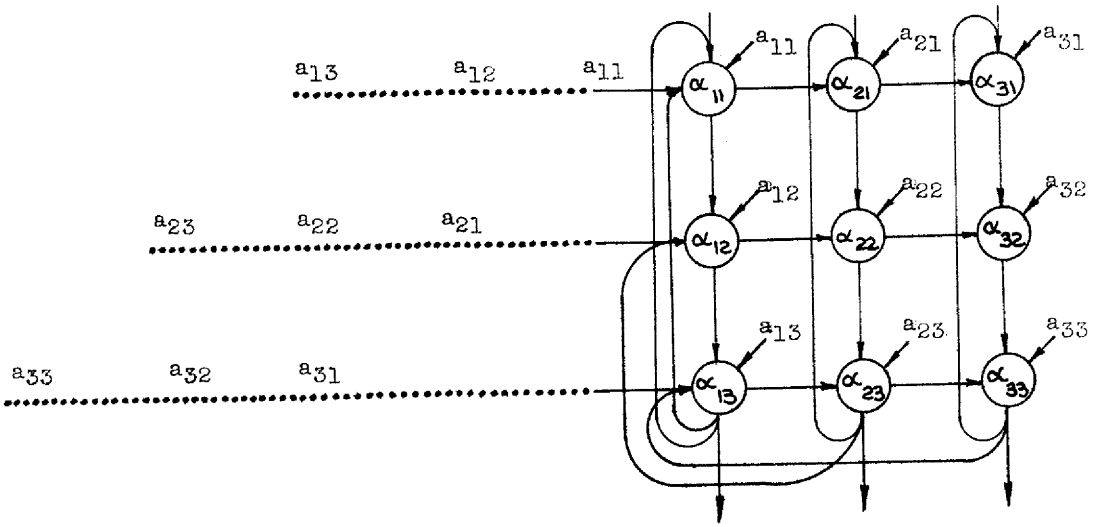

Figure 15

In Phase II the values a_{ij} remain stationary (as loaded in Phase I); initially the matrix A is again put on input x as shown in Figure 15 and zeros on input y , then the intermediate sums for $A + \ldots + A^n$ cycle back on both inputs x and y . The execution time is n clock cycles for Phase I and n^2 clock cycles for Phase II.

References

[1] S.N. Cole: Real-time computation by n-dimensional iterative arrays of finite-state machines, IEEE 7th Annual Symp. Switching Automata Theory (October 1966), 53-77.

[2] K. Culik II, J. Gruska and A. Salomaa: Systolic trellis automata (for VLSI), University of Waterloo Research Report CS-81-34 (December 1981).

[3] C.E. Leiserson and J.B. Saxe: Optimizing synchronous systems, 22nd Annual Symp. FOCS (October 1981), 23-36.

[4] C. Mead and L. Conway: Introduction to VLSI Systems, Addison-Wesley 1980.