On the Design of Algorithms for VLSI Systolic Arrays

DAN I. MOLDOVAN, MEMBER, IEEE

Abstract—This paper is concerned with the mapping of cyclic loop algorithms into special-purpose VLSI arrays. The mapping procedure is based on the mathematical transformations of index sets and data dependence vectors. Necessary and sufficient conditions for the existence of valid transformations are given for algorithms with constant data dependences. Two examples of different algorithms are given to illustrate the proposed mapping procedure; first is the LU decomposition of a matrix which leads to constant data dependence vectors, and secondly is the dynamic programming which leads to dependences which are functions on the index set and are more difficult to be mapped into VLSI arrays.

I. INTRODUCTION

THE TECHNOLOGY available to produce central processor units (CPU) and computer memories has always influenced the architecture of computers. Improvements in technological processes resulted in higher computer performances. The present semiconductor technology has reached already the level of maturity beyond which no significant breakthroughs are expected for switching speeds. The level of integration, however, continues to grow and in the next ten years it will be possible to incorporate one million logic gates on a die of a chip. This very large scale integration (VLSI) is a new technological environment which requires new ideas in computer organization, theory of computing, and other related fields.

In this paper we are concerned with the development of algorithms for special-purpose VLSI arrays. As will be seen below, while the VLSI technology offers remarkable advantages to the system designer, it also imposes restrictions on the design of algorithms. The most important of these restrictions is the necessity for reduced communication complexity.

The paper is organized as follows: in Section I we discuss the implications of the VLSI technology on computer architectures and algorithm design. Sections II and III contain the main results of the paper; first, a technique is proposed to transform algorithms with loops into highly parallel forms suitable for VLSI devices; then, a procedure is proposed to map these transformed algorithms into VLSI systolic arrays. In Section IV, algorithms for the LU decomposition of a matrix and dynamic programming are used as examples to show how previously proposed architectures can be formally derived using appropriate algorithm transformations.

A. VLSI Algorithms and Architectures

The main advantages offered by the VLSI technology are: large amount of hardware available at very low cost, reduced power consumption and physical size, and increased reliability at the circuit level. Additionally, the high level of integration can conceivably eliminate the need to physically separate processors from memory, thus eliminating the bottleneck between them. Parallelism and pipelining are two classical concepts without which the efficient utilization of the large hardware resources offered by VLSI is not possible. Parallelism implies the operation of many units at the same time. Pipelining also requires a multitude of resources, but in contrast with parallelism, the resources work in a chain allowing data to flow only from one unit to the next one. Both, parallelism and pipelining, can be seen at different logic levels. The first level of parallelism is offered by partitioning a computational task into smaller computational modules. The second level of parallelism is found within each computational module. The last level of parallelism is offered by the simultaneous processing of all the bits in a word; and this level is present in almost all computers. The focus of this paper is the parallelism at the second level.

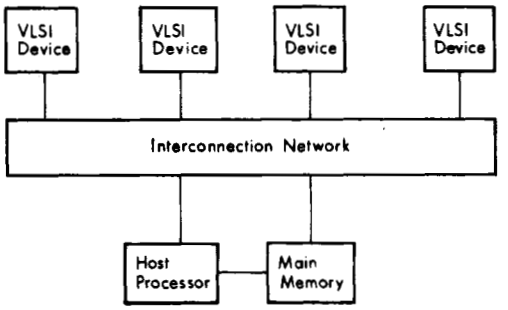


Fig. 1. Organization of a computer system containing several special-purpose VLSI processor arrays, interconnection network, host processor, and main memory.

The exploitation of parallelism at the first level is often necessary because computational problems are larger than a single VLSI device can process at a time. If a parallel algorithm is structured as a network of smaller computational modules, then these modules can be assigned to different VLSI devices. The communications between these modules and their operation control dictates the structure of the VLSI system and its performances. In Fig. 1, a simplistic organization of a computer system consisting of several VLSI devices, main memory, and an interconnection network are shown. Each VLSI device has a number of processors working in parallel.

The I/O bottleneck problem in VLSI systems presents a serious restriction imposed on the algorithm design. The challenge is to design parallel algorithms which can be partitioned such that the amount of communication between modules is as small as possible. Moreover, data entering the VLSI

Manuscript received April 22, 1982; revised October 20, 1982. This research was partially supported by the National Science Foundation under Grant ECS 8119509.

The author is with the Electrical Engineering Department—Systems, University of Southern California, Los Angeles, CA 90089.

0018-9219/83/0100-0113$01.00

device should be utilized exhaustively before passing again through the I/O ports.

Another potential problem which can deteriorate the performance is the data communication within the VLSI device. The interconnections between logic gates are as expensive as the logic itself and the signal propagation is comparable with the logic switching time. An efficient utilization of silicon area, time, and energy is achieved only if the hardware contains local interconnections. The solution to this problem is to design algorithms which, when mapped into VLSI hardware, require only local data transfers. In the next two sections it is shown that some algorithms can be transformed to meet these requirements.

Systolic array architectures have been proposed by Kung [1], [2] and others as a possible solution to these VLSI problems. In the systolic concept, VLSI devices consist of arrays of interconnected processing cells with a high degree of modularity. Each processor operates on a string of data that flow regularly through the network. If the I/O problem is ignored, the throughput of such computational structure is proportional to the number of cells.

In order to match better the characteristics of algorithms with the characteristics of computer architectures, and consequently to increase the efficiency of computation, a careful mapping of the computational problem to the machine is necessary. The mapping of algorithms into systolic arrays is different than the mapping of algorithms into architectures with fixed number of processors and interconnections. In the case of systolic arrays, one has to deal with issues ranging from the organization of the network of cells to the detailed operation of the cells. In fact, the mapping is nothing but the design of the VLSI array according to the properties of the algorithm and a set of design goals.

A number of special-purpose VLSI architectures have been proposed in the last few years. Kung's early work in parallel algorithms for VLSI has stimulated a considerable interest. He proposed systolic arrays for matrix–vector and matrix–matrix multiplications, LU decompositions, recurrence evaluations, etc., [1], [2]. The VLSI implementation of some combinational algorithms has been investigated by Guibas *et al.* [3]. Algorithms for solving systems of equations have been proposed by Kung [4], Hwang and Cheng [5], and Preparata [6]. The special-purpose VLSI computing structures have found immediate application in signal processing where many algorithms have regular structures [7], [8].

B. A VLSI Model of Computation

A model of the VLSI computing structure is needed in order to relate the features of an algorithm to the realities of the hardware. Tradeoffs are possible between various parameters of the VLSI device in order to improve one performance or another. The approach taken here is to distinguish between the operation of the systolic system at the array level and the activities taking place inside the processing cells. The array level is called the global level, and the processor level is called the local level. At both levels, the operation should be examined in time and space. Fig. 2 shows the main steps involved in the design of a special-purpose VLSI chip.

In this paper, we will focus only on the step from the parallel algorithm to the global model. A model of the processing cell and the transition from the global model to the local model can be found in [9]. The organization and the operation of the VLSI array can be described by the network geometry G, the functions F performed by the processing cells, and the network timing T.

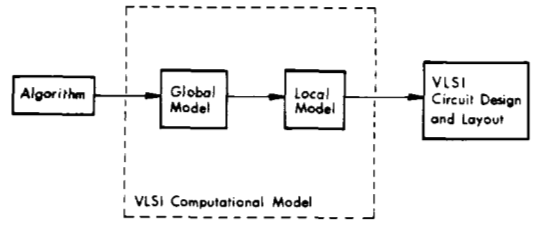


Fig. 2. Main steps in the design of a special-purpose VLSI device.

The assumptions about the VLSI systolic network are as follows:

a) The network consists of a planar mesh connected network of processing cells.

b) The cells can be of different types and perform different functions.

c) The interconnections between cells are buses which transfer parallel words.

d) The operation of the network is synchronous.

The *network geometry* G refers to the geometrical layout of the network. The position of each processing cell in the plane is described by its Cartesian coordinates. By choosing the grid arbitrarily small it is possible to represent these coordinates by integers. Then, the interconnections between cells can easily be described by the position of the terminal cells. These interconnections support the flow of data through the network; a link can be dedicated only to one data stream of variables or it can be used for the transport of several data streams at different time instances. A simple and regular geometry is desired.

The *functions* F associated to each processing cell represent the totality of arithmetic and logic expressions that a cell is capable to perform. We assume that each cell consists of a small number of registers, ALU, and control logic. Several different types of processing cells may coexist in the same network; however, one design goal should be to reduce the number of cell types.

The *network timing* T specifies for each cell the time when the processing of functions F occurs and when the data communications take place. A correct timing assures that the right data reach their destination at the right time. The speed of the data streams through the network is given by the ratio between the distance of communication link over the communication time. Networks with constant data speeds are preferable because they require a simpler control logic.

In summary, the global model of the VLSI array can be formally described by a set of 3-tuples (G, F, T). The more regular the network is the simpler these functions become. This model is quite general and it is sufficient for the purpose of this paper of developing a methodology for designing VLSI algorithms.

In the next section, a technique is developed to study and to modify computational algorithms for the purpose of mapping them into VLSI processing arrays. While any algorithm can be analyzed using this technique, only some algorithms can be mapped directly into simple systolic arrays.

II. Transformations of Algorithms for VLSI Systems

A. Data Dependences

The intention of mapping computational algorithms into VLSI circuits implies first a transformation of algorithms into equivalent but more appropriate forms for VLSI. The basic structural features of an algorithm are dictated by the data and control dependences. These dependences refer to precedence

relations of computations which need to be satisfied in order to compute the problem correctly. The absence of dependences indicates the possibility of simultaneous computations. These dependences can be studied at several distinct levels: blocks of computations level, statement (or expression) level, variable level, and even bit level. In this paper, since we concentrate on algorithms for VLSI systolic arrays, we will focus only on data dependences at the variable level which is the lowest possible level before the bit level.

The analysis of data dependences in high-level language (HLL) programs for the purpose of detecting concurrency of operations has received considerable attention in the last decade. Muraoka [10] and Kuck *et al.* [11] have studied the parallelism of simple loops and have introduced the notion of dependence relations between assignment statements. Towle [12], Banergee [13], and Banerjee *et al.* [14] extended the methodology of transforming ordinary programs into highly parallel forms. Based upon dependences between statements, they have provided algorithms for exploiting parallelism in loops. Techniques such as loop freezing, wave from method, the splitting-lemma, and loop interchanging have been introduced. Recently, Kuhn [15] has proposed a methodology to analyze data dependences using transformations on convex index sets. Results on program transformations were also reported by Lamport [16].

All these results were aimed mostly towards program speed-up and compiler design. Although the program transformations proposed before contain many basic results, they are not adequate enough for VLSI implementations. In addition to a high degree of parallelism, VLSI arrays suggest pipelining and reduced communication distance and time.

Previous work in algorithm transformations has focused on deciding whether a pair of occurrences is dependent or not. The present approach is based not only on the detection of dependences but also on their modification. The very structure of algorithm interconnections has to be modified in order to increase the "locality" of communications and to meet other VLSI requirements.

In what follows, algorithms written in HLL are considered. Other forms to express algorithms are possible, but the results would be similar. Consider a Fortran loop structure of the form

```
DO 10  I^1 = l^1, u^1
DO 10  I^2 = l^2, u^2
...
DO 10  I^n = l^n, u^n
S_1(Ī)                                   (1)
S_2(Ī)
...
S_N(Ī)
10 CONTINUE
```

where l^j and u^j are integers value linear expressions involving $I^1, \cdots, I^{j-1}$ and $\bar{I} = (I^1, I^2, \cdots, I^n)$. $S_1, S_2, \cdots, S_N$ are assignment statements of the form $X = E$ where X is a variable and E is an expression of some input variables.

Let $\mathcal{I}$ denote the set of all integers and $\mathcal{I}^n$ denote the set of n-tuples of integers. The index set of loop (1) is a subset of $\mathcal{I}^n$ and is defined as

$$\mathcal{L}^n(\bar{I}) = \{(I^1, \cdots, I^n):\ l^1 \leqslant I^1 \leqslant u^1, \cdots, l^n \leqslant I^n \leqslant u^n\}.$$

When loop (1) is executed, the elements of $\mathcal{L}^n$ are ordered in a lexicographical ordering. This is an induced ordering which is not essential and can be modified. Let f and g be two integer functions defined on the set $\mathcal{L}^n$. Denote X and Y two variables whose indices are f and g; we write $X(f(I))$ and $Y(g(\bar{I}))$. Variables X and Y are generated in statements $S(\bar{I}_1)$ and $S(\bar{I}_2)$, respectively. Variable $Y(g(\bar{I}))$ is said to be dependent on variable $X(f(\bar{I}))$ and write $X(f(\bar{I})) \rightarrow Y(g(\bar{I}))$ if

a) $I_1 < \bar{I}_2$ (throughout the paper "$<$" means "less than" in lexicographical sense)
b) $f(\bar{I}_1) = g(\bar{I}_2)$
c) $X(f(\bar{I}))$ is an input variable in statement $S(\bar{I}_2)$.

The vector $\bar{d} = \bar{I}_2 - \bar{I}_1$ is called the data dependence vector. An algorithm has a number of such dependence vectors. In general, the dependence vectors are functions of the elements of set $\mathcal{L}^n$, i.e., $\bar{d} = \bar{d}(\bar{I})$, as will be seen in Section III-B. There is, however, a large class of algorithms with fixed, or constant data dependence vectors.

B. Transformation of Index Set and Data Dependences

Denote the ordering imposed by the data dependences on set $\mathcal{L}^n$ with R. The elements of $\mathcal{L}^n$ and ordering R form together a well-defined algebraic structure $\langle \mathcal{L}^n, R \rangle$. We seek now a transformation T such that

$$T:\ \langle \mathcal{L}^n, R \rangle \rightarrow \langle \mathcal{L}^n_T, R_T \rangle \tag{2}$$

with the following properties:

a) T is a bijection and a monotonic function (2a)
b) The data dependences of the new structure $\langle \mathcal{L}^n_T, R_T \rangle$ can be selected by us. (2b)

Since T is a bijection, the two structures are said to be isomorphic, and since T is monotonic with respect to R and R_T

$$\bar{d} > 0 \rightarrow \bar{\delta} = T(\bar{d}) > 0.$$

It simply means that the transformation T preserves the sense of the data dependences. The meaning of the second condition will soon become clear. The transformation T is partitioned in two functions as follows:

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}. \tag{3}$$

The mapping Π is defined as

$$\Pi:\ \mathcal{L}^n \rightarrow \mathcal{L}^k_T, \qquad n > k$$

$$\Pi(I^1, I^2, \cdots, I^n) = (J^1, J^2, \cdots, J^k), \qquad \text{with } \bar{J} \in \mathcal{L}^n_T.$$

The mapping S is defined as

$$S:\ \mathcal{L}^n \rightarrow \mathcal{L}^{n-k}_T$$

$$S(I^1, I^2, \cdots, I^n) = (J^{k+1}, J^{k+2}, \cdots, J^n).$$

The dimensionality of functions Π and S is marked by k; and k is selected such that Π alone establishes the ordering R_T. The first k coordinates of elements $\bar{J} \in \mathcal{L}^n_T$ can now be related to time and the last $n - k$ coordinates be related to the geometrical properties of the algorithm. In other words, the time is associated to the new lexicographical ordering imposed on the elements $\bar{J}$ and this is given only by their first k coordinates. The last $n - k$ coordinates can be chosen by us to satisfy our expectations about the geometrical properties of the algorithm. For all elements $\bar{I} \in \mathcal{L}^n$ for which $\Pi(\bar{I}) =$ constant, the first k coordinates of $\bar{J}(\bar{I})$ are also constant. It follows that all such $\bar{I} \in \mathcal{L}^n$ can be processed concurrently. $\Pi(\bar{I}) =$ constant represents hypersurfaces with property to contain elements which are not data dependent.

The freedom of selecting $(J^{k+1}, \cdots, J^n)$ can be used advantageously to satisfy property (2b); that is, to localize the data communications in the VLSI system.

Consider the case when an algorithm of form (1) with n loops provides m constant data dependence vectors. These are grouped in a matrix $D = [\bar{d}_1, \bar{d}_2, \cdots, \bar{d}_m]$, $D \in R^{n \times m}$. A linear transformation T is sought, i.e., $\bar{J} = T \cdot \bar{I}$. Since T is linear $T(\bar{I} + \bar{d}_j) - T(\bar{I}) = T\bar{d}_j = \bar{\delta}_j$ for $1 \leqslant j \leqslant m$. These equations can be written as

$$TD = \Delta \tag{4}$$

where $\Delta = [\bar{\delta}_1, \bar{\delta}_2, \cdots, \bar{\delta}_m]$. The matrix Δ represents the modified data dependences in the new index space $\mathcal{L}^n$, and according to the requirements (2b) they are assumed known.

The interesting question now is under what conditions can such T exist? System (4) represents $n \times m$ diophantine equations with n^2 unknowns. T exists if system (4) has solution and the solution consists of integers. The following theorem indicates the necessary and sufficient conditions for valid linear transformations and it can be used as a tool to preselect Δ.

Theorem 1: For an algorithm with a constant set of data dependences D, the necessary and sufficient conditions that a valid transformation T exists are as follows:

i) The new data dependence vectors $\bar{\delta}_j$ are congruent to the dependences $\bar{d}_j$ modulo c_j, where c_j is the greatest common divisor (gcd) of the elements of $\bar{d}_j$

$$\bar{\delta} \equiv \bar{d}_j (\text{mod } c_j). \tag{5}$$

ii) System (4) can be solved for T.

iii) The first nonzero element of vector $\bar{\delta}_j$ is positive.

Proof: Sufficient: Condition i) indicates that the elements of $\bar{\delta}_j$ are multiples of the gcd of the elements of respective $\bar{d}_j$. This is a necessary and sufficient condition that each of the $n \times m$ diophantine equations can be solved for integers [17]. According to ii) system (4) has solution. Since the first nonzero elements of $\bar{\delta}_j$ are positive it follows that $\Pi \bar{d}_j > 0$, thus T is a valid transformation. *Necessary:* Transformation T is a bijection and consists of integers, hence i) and ii) are required conditions. Because T preserves the ordering R_T, $\bar{\delta}_j > 0$ and this implies that the first nonzero element is positive. QED

In the selection of Δ one should choose the smallest possible integers for its elements. In this way, the processing time and the communication requirements of the transformed algorithm are being optimized.

C. Mapping Algorithms into Hardware

The transformation of the index sets described above is the key towards an efficient mapping of an algorithm into a special-purpose VLSI array. It is shown in what follows how the global model of the VLSI device introduced in Section I can be derived directly from the transformed algorithm.

The functions F performed by the cells are derived directly from the mathematical expressions indicated in the algorithm. An algorithm of form (1) contains assignment statements in one loop body which is executed repeatedly for all iteration points in set $\mathcal{L}^n$. This implies that all the processing cells can be made identical. The peripheral cells performing input/output operations are, of course, different than the rest. If the mathematical expressions inside the loop involve too many computations, the loop can be split into several simpler loops. Algorithms with several distinct loop bodies normally require different processing cells.

The network geometry G refers to the physical underlying of the network, and it is derived from the mapping $S: \mathcal{L}^n \to \mathcal{L}_T^{n-k}$. A processing cell is assigned to each distinct element of $\mathcal{L}_T^{n-k}$. Assuming that in algorithm (1), $u_i - l_i = O(N)$ for all $i = 1, 2, \cdots, n$ where N is the size of the problem, it follows that the total number of processing cells is $O(N^{n-k})$. The position, or the identification number of each cell is given by $S(\bar{I}) = (J^{k+1}, \cdots, J^n)$. The interconnections between cells necessary for the data communication are derived directly from the last $n - k$ components of the modified data dependence vectors $\bar{\delta}_j^S = S(\bar{I} + \bar{d}_j) - S(\bar{I})$, which becomes $S\bar{d}_j$ for linear transformations. For each cell, the vectors $\bar{\delta}_j^S$ indicate the relative destination of the variable associated to that dependence vector. These interconnections are then replicated to the entire network. Although three-dimensional and multilayer VLSI networks may be attempted, the most practical is the planar arrangement. If $n - k > 2$, an additional one-to-one mapping S' is necessary $S': \mathcal{L}_T^{n-k} \to \mathcal{L}_T^2$.

The network timing T is derived from the mapping $\Pi: \mathcal{L}^n \to \mathcal{L}_T^k$. The exact time when the processing related to an element $\bar{I} \in \mathcal{L}^n$ occurs is simply $\Pi(\bar{I})$. The communication time for a data stream associated with a dependence vector $\bar{d}$ is given by $\Pi(\bar{I} + \bar{d}) - \Pi(\bar{I})$, which in the case of a linear transformation reduces to $\Pi\bar{d}$. The total running time for VLSI algorithm is $[\max \Pi(\bar{I}) - \min \Pi(\bar{I})]$. It can be seen that linear transformations yield a running time $O(N^k)$, while higher order mappings Π will normally lead to higher order processing times. Notice that the running time includes only the computation time and the communication time and not the input/output time. Another observation is that keeping k as small as the transformation permits should be one goal in designing VLSI algorithms. This will increase the concurrency of operations at the expense of the number of processors.

It remains to demonstrate that indeed the VLSI model executes the algorithm correctly. Since we consider here only the global model, it will be sufficient to show that the data flow through the VLSI network is correct. We say that the data flow through the network is correct if all the variables necessary to compute the mathematical expressions of the algorithm are available at the proper time at the proper cell. The following theorem refers to linear transformations.

Theorem 2: A transformation

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$$

of an algorithm which satisfies Theorem 1 maps that algorithm into a systolic array in which the data flow is correct.

Proof: Consider a typical assignment statement $x = E(v_1, v_2, \cdots, v_r)$ executed at $\bar{I} \in \mathcal{L}^n$. From the definition of data dependence vectors we have

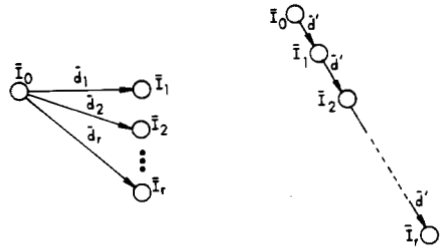
$$\bar{I} = \bar{I}_1 + \bar{d}_1 = \bar{I}_2 + \bar{d}_2 = \cdots = \bar{I}_r + \bar{d}_r \tag{6}$$

where $\bar{I}_i \in \mathcal{L}^n$ and $\bar{d}_i$ correspond to the generation of variable v_i. Apply the linear operators Π and S to (6)

$$\Pi\bar{I} = \Pi\bar{I}_1 + \Pi\bar{d}_1 = \Pi\bar{I}_2 + \Pi\bar{d}_2 = \cdots = \Pi\bar{I}_r + \Pi\bar{d}_r \tag{7}$$

$$S\bar{I} = S\bar{I}_1 + S\bar{d}_1 = S\bar{I}_2 + S\bar{d}_2 = \cdots = S\bar{I}_r + S\bar{d}_r. \tag{8}$$

If the computations at $\bar{I}_i \in \mathcal{L}^n$ produce correctly v_i, then it follows from (7) and (8) that all the input variables will be available for $\bar{I} \in \mathcal{L}^n$ at the same time and at the same processing cell. For each v_i it corresponds a $\bar{d}_i$ and Δ can be selected as desired. It follows that there is no overlap in the flow of



(a) (b)

Fig. 3. (a) Broadcasted variables. (b) Pipelined variables.

the data streams and no cell is required to perform more than one operation at any one time. QED

D. Procedure for Mapping Algorithms into VLSI Systolic Arrays

In this subsection, a procedure is proposed which summarizes the technique introduced above. This procedure will then be used in the next section to discuss some examples.

Step 1) Pipeline all variables in the algorithm.
Step 2) Find the set of data dependence vectors.
Step 3) Identify a valid transformation for the data dependence vectors and the index set.
Step 4) Map the algorithm into hardware.
Step 5) Prove correctness and analyze performances.

Explanation: The role of the first step is to eliminate all possible data broadcasts which may exist in the original algorithm. Consider, for example, that a variable v is generated at some element $\bar{I}_0 \in \mathcal{L}^n$ and used at several other elements $\bar{I}_i \in \mathcal{L}^n$, $i = 1, 2, \cdots, r$. There are r data dependence vectors for variable v, as shown symbolically in Fig. 3.

The highest parallelism for variable v is achieved when all the use statements $S(\bar{I}_i)$, $i = 1, 2, \cdots, r$, are performed in parallel, provided that there are no other restrictions; thus the generated variable v needs to be broadcasted at once. However, it is likely that in the VLSI implementation, this algorithm will be communication saturated. The goal is then to reduce the number of the original data dependences. The solution to this problem is to pipeline the propagation of variable v for the r usage statements, as shown in Fig. 3(b). New data dependences have been created by arbitrarily ordering the usage elements. If the new arrangement offers fewer dependences, then this will eventually translate into fewer communication requirements.

Typically, broadcasts are signaled by missing indices of variables in the loop. In order to avoid broadcasts and to increase pipelining we first complete all the missing indices and introduce new artificial variables such that for each generated variable there is only one destination.

Example: Consider the following loop which implements a matrix multiplication $C = AB$, where $A, B \in R^{n \times n}$.

```
DO 10  k = 1 to n
DO 10  i = 1 to n
DO 10  j = 1 to n                                   (9)
c(i, j) = c(i, j) + a(i, k) · b(k, j)
10 CONTINUE.
```

This loop can be written in an equivalent form in which variables a, b, and c are pipelined

```
      DO 10  k = 1 to n
      DO 10  i = 1 to n
      DO 10  j = 1 to n
S1:   a^{j+1}(i, k) = a^j(i, k)                     (10)
S2:   b^{i+1}(k, j) = b^i(k, j)
S3:   c^{k+1}(i, j) = c^k(i, j) + a^j(i, k) · b^i(k, j)
   10 CONTINUE.
```

The data dependence vectors of the algorithm can now be found using their definition. All possible pairs of generated (output) and used (input) variables are formed and their indices are equated. This is equivalent to writing $\bar{I}_1 + \bar{d} = \bar{I}_2$, from which the data dependence vector $\bar{d}$ can be found directly. It is possible that two different pairs of variables lead to the same data dependence vector. Caution should be exercised to identify only valid generated–used pairs of variables.

As an example, consider loop (10). For variable c, S_3 is both the generate and the use statement. The pair $\langle c^{k'+1}(i, j), c^k(i, j)\rangle$ is formed (in this example we use $'$ for the indices of the generated variable). This yields a dependence vector $\bar{d}_1 = (k - k', i - i', j - j')^t = (1, 0, 0)^t$.

The generated variable $a^{j'+1}(i', k')$ in S_1 is used in S_1 and S_3. However, only one distinct pair can be formed for variable a, i.e., $\langle a^{j'+1}(i', k'), a^j(i, k)\rangle$. It follows that $\bar{d}_2 = (k - k', i - i', j - j')^t = (0, 0, 1)^t$. Similarly, the data dependence vector is found for variable b, $\bar{d}_3 = (0, 1, 0)^t$. For the matrix multiplication algorithm pipelined as in loop (10), there are only three data dependence vectors $\bar{d}_1, \bar{d}_2$, and $\bar{d}_3$. Note that these vectors are independent of the index set.

Steps 3) and 4) have been discussed only for the case of linear transformations. As will be seen in the next section, the dynamic programming algorithm requires a more complex transformation. Step 5) is necessary in order to validate the mapping process and to ensure that the performances obtained are satisfactory.

III. Examples

A. LU Decomposition of a Matrix A

Consider a matrix A which can be decomposed into a lower and upper triangular matrices by Gaussian elimination without pivoting. VLSI computing structures for the LU decomposition problem have been proposed by Kung [1], Kung [4], Hwang and Cheng [5], and others. In this example it is shown that previously proposed architectures can be formally derived by using appropriate algorithm transformations.

The algorithm for the LU decomposition of a matrix $A = [a_{ij}]$ is expressed by the program written in Pidgin ALGOL

```
for k ← 0 until n - 1 do
    begin
    u_kk ← 1/a_kk
    for j ← k + 1 until n - 1 do
        u_kj ← a_kj
    for i ← k + 1 until n - 1 do                    (11)
        l_ik ← a_ik u_kk
    for i ← k + 1 until n - 1 do
        for j ← k + 1 until n - 1 do
            a_ij ← a_ij - l_ik · u_kj
    end.
```

This program can be rewritten into the following equivalent form in which all the variables have been pipelined and all the data broadcasts have been eliminated.

TABLE I
DATA DEPENDENCES FOR LU DECOMPOSITION ALGORITHM

The pairs of generated-used variables	Date dependences k-k'	i-i'	j-j'
$\langle a^{k'}_{i'j'}, a^{k-1}_{ij} \rangle$	(1	0	$0)^T = \bar{d}_1$
$\langle u^{i'}_{k'j'}, u^{i-1}_{kj} \rangle$	(0	1	$0)^T = \bar{d}_2$
$\langle \ell^{j'}_{i'k'}, \ell^{j-1}_{ik} \rangle$	(0	0	$1)^T = \bar{d}_3$

```
for k ← 0 until n - 1 do
  begin
1:  i ← k;
    j ← k;
    u^i_kj ← 1/a^k_ij
    for j ← k + 1 until n - 1 do
2:    begin
        i ← k;
        u^i_kj ← a^k_ij
      end
    for i ← k + 1 until n - 1 do
3:    begin
        j ← k;                                   (12)
        u^i_kj ← u^(i-1)_kj ;
        l^j_ik ← a^k_ij · u^i_kj
      end
    for i ← k + 1 until n - 1 do
      for j ← k + 1 until n - 1 do
4:      begin
          l^j_ik ← l^(j-1)_ik ;
          u^i_kj ← u^(i-1)_kj ;
          a^k_ij ← a^(k-1)_ij - l^(j-1)_ik u^(i-1)_kj
        end
  end.
```

This algorithm is similar to the matrix multiplication (9). Indeed, both algorithms yield the same data dependences. The only three distinct pairs of generate and use variables and their respective data dependence vectors are summarized in Table I.

The data dependences for this algorithm have the nice property that $D = [\bar{d}_1 \bar{d}_2 \bar{d}_3] = I$. There are several other algorithms which lead to these simple data dependences, and they were among the first to be considered for the VLSI implementation.

Following the methodology of Section II, the next step (step 3) is to identify a linear transformation of form (3). This transformation must have the following properties: $\Pi \bar{d}_i > 0$, it offers the maximum concurrency, and T is a bijection. According to Theorem 1, T exists, and furthermore, $T = \Delta$. Denote

$$T = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ \hline t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}.$$

In this case, it is possible to have $k = 1$, thus $\Pi \bar{d}_i = t_{1i} > 0$. The smallest possible positive integers are $t_{11} = t_{12} = t_{13} = 1$. The first two conditions are satisfied; and Π is unique. In the selection of mapping S we are now restricted only by the fact that T must be a bijection and consists of integers. A large number of possibilities exist, each leading to different network geometries. We choose

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (13)$$

$$\begin{bmatrix} \hat{k} \\ \hat{i} \\ \hat{j} \end{bmatrix} = T \begin{bmatrix} k \\ i \\ j \end{bmatrix}.$$

The original indices k, i, j are transformed by T into $\hat{k}$, $\hat{i}$, $\hat{j}$. The organization of the VLSI array, for $n = 5$ generated by the transformation (13) is shown in Fig. 4.

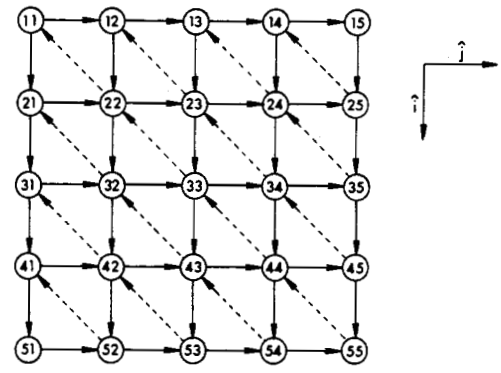


Fig. 4. VLSI array for the LU decomposition algorithm.

In this architecture variables a^k_{ij} do not travel in space, but are updated in time. Variables l^k_{ij} move along the direction $\hat{j}$ (east) with a speed of one grid per time unit, and variables u^k_{ij} move along the direction $\hat{i}$ (south) with the same speed. The network is loaded initially with the coefficients of A, and at the end the cells below the diagonal contain L and the cells above the diagonal contain U.

The processing time is $\Pi_{max} - \Pi_{min} = 3n - 5$. All the cells have the same architecture. However, their functions at one given moment may differ. It can be seen from the program (12) that some cells may execute loop 4, while others execute loops 2 or 3. If we wish to assign the same loops only to specific cells, then the mapping S must be changed accordingly. For example, the transformation

$$T = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

introduces a new data communication link between cells, toward north-west. These new links will support the movement of variables a^k_{ij}. According to this new transformation, the cells of the first row always compute loop 2, the cells of the first column compute loop 3, and the rest compute loop 4. The reader can now easily identify some other valid transformations which will lead to new organizations. By applying Theorem 2 to this example, one can prove the validity of an architecture.

In the next example, we will see data dependences which are no longer fixed, and this presents a challenge for finding a proper transformation.

B. Dynamic Programming

Many problems in computer science and engineering can be solved by the use of dynamic programming techniques. We consider here the VLSI implementation of an optimal paren-

TABLE II
DATA DEPENDENCES FOR THE DYNAMIC PROGRAMMING ALGORITHM

Pairs of generated-used variables	Data dependences $\ell-\ell'$	$i-i'$	$k-k'$
$\langle m^{\ell'}_{i'k'}, m^{\ell-1}_{ik} \rangle$	(1	0	0)$^T = \bar{d}_1$
$\langle m^{\ell'}_{k'+1,\, i'+\ell'}, m^{\ell-1}_{k+1,i+\ell} \rangle$	(1	-1	0)$^T = \bar{d}_2$
$\langle m^{\ell'}_{i',\, i'+\ell'}, m^{\ell-1}_{ik} \rangle$	(1	0	f)$^T = \bar{d}_3$
$\langle m^{\ell'}_{i',\, i'+\ell'}, m^{\ell-1}_{k+1,\, i+\ell} \rangle$	(1	-1	g)$^T = \bar{d}_4$

thesization algorithm based on dynamic programming. A string of n matrices are multiplied

$$M = M_1 \times M_2 \times \cdots \times M_n.$$

Let $r_0, r_1, \cdots, r_n$ be the dimensions of the n matrices with r_{i-1} and r_i dimensions of M_i. Denote by m_{ij} the minimum cost of computing the product $M_i \cdot M_{i+1} \cdots M_j$. The algorithm which finally produces m_{1n} is written as follows [18]:

```
for i ← 1 to n do m_ii ← 0
for l ← 1 to n - 1 do
    for i ← 1 to n - l do
      begin                                                    (14)
        j ← i + l
        m_ij ← MIN_{i≤k≤j} (m_ik + m_{k+1,j} + r_{i-1} r_k r_j)
      end.
```

Following the methodology of Section II, this program is transformed into the following equivalent form:

```
for i ← 1 to n do m_ii ← 0.
  for l ← 1 to n - 1 do
    for i ← 1 to n - l do
      for k ← i to i + l - 1 do
        begin                                                  (15)
          m^l_ik ← m^{l-1}_ik
          m^l_{k+1,i+l} ← m^{l-1}_{k+1,i+l}
          m^l_{i,i+l} ← MIN (m^{l-1}_ik + m^{l-1}_{k+1,i+l} + r_{i-1} r_k r_{i+l})
        end.
```

We will assume that the input data r are loaded on the network before the computations start, so we can neglect the dependences caused by the constant data r. This assumption is made only for the purpose of simplifying the explanation; in fact, the dependences caused by data r are similar to those generated by variable terms.

The data dependences derived from the above algorithm are shown in Table II. There are only four possible distinct pairs of used–generated variables.

The data dependence vectors for the first two pairs of generated–used variables are easily derived in the same manner as for the previous examples (see Table II). The last two dependences, however, require more attention. Consider, first, the pair $\langle m^{l'}_{i',i'+l'}, m^{l-1}_{ik} \rangle$; it yields that $l - l' = 1, i - i' = 0$, and $k = i' + l'$. Form program (15), k' takes values between i' and $i' + l' - 1$. It follows that $k - k' = l - 1, l - 2, \cdots, 1$. Similarly, the pair $\langle m^{l'}_{i',i'+l'}, m^{l-1}_{k+1,i+l} \rangle$ yields $l - l' = 1, i' + l' = i + l$, and $i' = k + 1$. From the first two equalities it results that $i - i' = -1$, and finally, since $k' = i', \cdots, i' + l' - 1$ it follows that $k - k' = -1, -2, \cdots, -l + 1$. Therefore, for both $\bar{d}_3$ and $\bar{d}_4$,

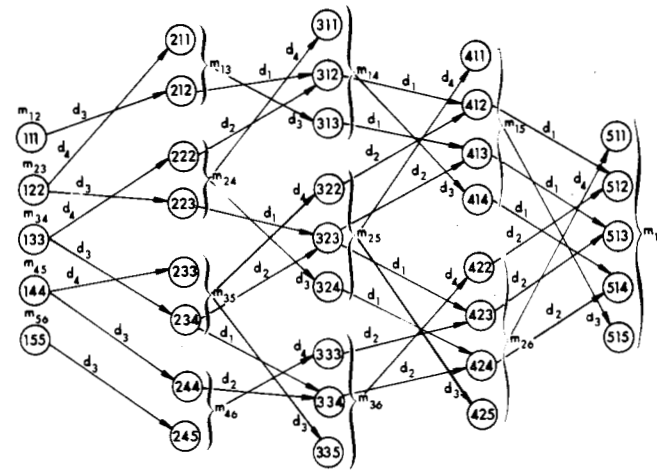


Fig. 5. Data dependence for the dynamic programming algorithm ($n = 6$). The encricled numbers correspond to elements of the index set l i k.

$k - k'$ can take many possible values

$$f = l - 1, l - 2, \cdots, 1$$

$$g = -1, -2, \cdots, -l + 1.$$

The difference $k - k'$ is not fixed because the order in which the minimization in loop k is performed is not specified. For instance, in program (15) if $m^l_{i,i+l}$ is generated when k takes the largest value, then $f = 1$ and $g = -l + 1$. Notice that, if the minimization procedure in loop k is performed sequentially, then either f or g will depend on the value of l. This fact constitutes an obstacle in finding a linear transformation for the dynamic programming problem. Fig. 5 shows the dependences between the iteration elements for $n = 6$. Each column corresponds to a different value of l and each group in the column corresponds to a different loop k, in which the order is not specified yet. For example, element 512 receives data from elements 412 and 422, but element 511 receives m_{26}, the result of elements 422, 423, 424, and 425.

The following mapping Π is proposed for the dynamic programming algorithm:

$$\Pi(l, i, k) = \max \begin{cases} \Pi_1(l, i, k) = 2l + i - k \\ \Pi_2(l, i, k) = l - i + k + 1. \end{cases}$$

The index set, which constitutes the domain of the mapping function Π, is separated into two disjoint sets, one for Π_1 and the other for Π_2. This mapping Π has the advantage that by exploiting possible concurrencies within loop k, it provides a processing time $O(n)$. The first half of any loop k uses Π_1 and the second half uses Π_2. Because of this new concurrency between the first and the last index elements of loop k, the dependences $\bar{d}_3$ and $\bar{d}_4$ are transformed respectively in $(1 \;\; 0 \;\; 1)^t$ and $(1 \;\; -1 \;\; -1)^t$. This is possible because Π_1 applies to $\bar{d}_4$'s while Π_2 applies to $\bar{d}_3$'s. The only inconvenience created by mapping Π is that the data flow in data streams does not have a constant speed. This is easily seen from the fact that $\Pi_1 \bar{d}_1 = 2 \neq \Pi_2 \bar{d}_1 = 1$ and $\Pi_1 \bar{d}_2 = 1 \neq \Pi_2 \bar{d}_2 = 2$.

The mapping S is selected such that the resulting VLSI architecture will be simple and regular.

$$S = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$S \cdot [\bar{d}_1 \bar{d}_2 \bar{d}_3 \bar{d}_4] = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

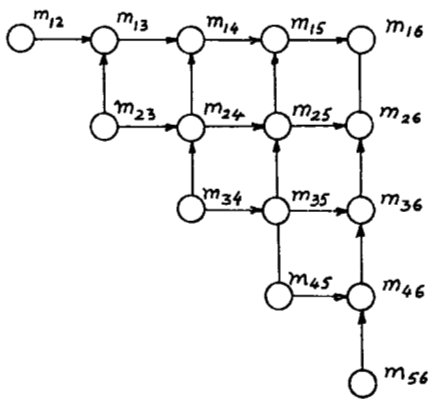


Fig. 6. VLSI array for the dynamic programming.

The mapping Π together with the mapping S form a transformation $T: (l, i, k) \rightarrow (\hat{l}, \hat{i}, \hat{k})$

$$T: \begin{bmatrix} \hat{l} \\ \hat{i} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} \max (2l + i - k, l - i + k + 1) \\ l + k \\ -k \end{bmatrix}.$$

This transformation leads to the VLSI architecture shown in Fig. 6. This architecture was first proposed by Guibas *et al.* [3]. All the processing cells perform the same functions, and no memories are required. There are $O(n^2)$ cells. The operation of this network and the proof of correctness become now particular cases of Theorem 2.

IV. Conclusions

The design of special-purpose VLSI devices is a multistep process (see Fig. 2). In this paper we have concentrated only on the step concerned with the mapping of linear cyclic algorithms into high-level VLSI models. The VLSI devices are assumed to be two-dimensional array processors with local communications. The model resulting from the mapping procedure specifies the complexity of processors, interprocessor connections, and the timing of the data flow. Although we have concentrated in this paper only on a class of algorithms, the methodology proposed here can constitute the foundation of a unifying approach to the design of VLSI algorithms.

Perhaps the most important information about an algorithm is contained in its data dependences because they determine the algorithms's communication requirements. The basic idea of this paper is to modify the data dependences vectors such that the new algorithm satisfies the VLSI requirements, while remaining input/output equivalent to the original algorithm. Transformations of other classes of algorithms into parallel forms constitute a further research topic.

An important feature of the technique proposed in this paper is that the idea of data dependence vectors can be extended to the next step of the VLSI design, that is, the actual design of the processors. This is achieved by studying the dependences at the register level and the bit level.

The design of algorithmically specialized VLSI devices is at its beginning. The development of specialized devices to replace mathematical software is feasible but is still costly. Several important technical issues remain unresolved, and deserve further investigation. Some of these are: I/O communication in VLSI technology, partitioning of algorithms to maintain their numerical stability, and minimization of the communication among computational blocks. Also, a better understanding of the design of parallel algorithms starting directly from the computational problem is necessary.

Finally, the concepts introduced in this paper are not restricted only to VLSI systems; they can also be used for mapping algorithms into some other fixed parallel computer architectures.

References

[1] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Caltech Conf. on VLSI*, pp. 65–90, Jan. 1979.
[2] —, "The structure of parallel algorithms," *Adv. Comput.*, vol. 19, pp. 65–111, 1980.
[3] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinational algorithms," in *Proc. Caltech Conf. on VLSI*, pp. 509–525, Jan. 1979.
[4] S. Y. Kung, "VLSI array processor for signal processing," presented at the Conf. on Advance Research in Integrated Circuits, MIT, Cambridge, MA, Jan. 28–30, 1980.
[5] K. Hwang and Y. H. Cheng, "VLSI computing structures for solving large scale linear system of equations," in *Proc. Parallel Processing Conf., 1980*, pp. 217–227.
[6] F. P. Preparata and J. Vuillemin, "Optimal integrated-circuit implementation of angular matrix inversion," in *Proc. Parallel Processing Conf., 1980*, pp. 211–216.
[7] J. M. Speiser, H. J. Whitehouse, and K. Bromley, "Signal processing applications for systolic arrays," in *Proc. 14th Asilomar Conf. on Circuits, Systems, and Computers*, Nov. 1980.
[8] J. G. Nash, S. Hansen, and G. R. Nudd, "VLSI processor array for matrix manipulation," presented at the Conf. VLSI Systems, Carnegie-Mellon Univ., Pittsburgh, PA, Oct. 1981.
[9] D. I. Moldovan, "Computational models for VLSI systems," Electrical Engineering Dep., Univ. of Southern California, Los Angeles, CA, Rep. DIM-82-3, 1982.
[10] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Feb. 1971.
[11] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Trans. Comput.*, vol. C-21, pp. 1293–1310, Dec. 1972.
[12] R. Towle, "Control and data dependence for program transformations," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Mar. 1976.
[13] U. Banergee, "Data dependence in ordinary programs," M.S. thesis, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Nov. 1976.
[14] U. Banerjee *et al.*, "Time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Comput.*, vol. C-28, pp. 660–670, Sept. 1979.
[15] R. H. Kuhn, "Optimization and interconnection complexity for: parallel processors, single stage networks and decision trees," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, 1980.
[16] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, pp. 83–93, Feb. 1974.
[17] L. J. Mordell, *Diophantine Equations*. New York: Academic Press, 1969, p. 30.
[18] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1975.