This paper presents a very simple conceptual model for systolic arrays. Based on this simple model, we illustrate how systolic arrays can be derived from problem specification in a systematic manner. At the heart of the synthesis method is an optimization procedure which finds linear transforms that yield optimal systolic arrays. Various design criterions can be formulated in terms of different objective functions for optimization.

Synthesizing Systolic Arrays from Recursion Equations

Marina C. Chen

Research Report YALEU/DCS/RR-527
March 1987

Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0296 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

Synthesizing Systolic Arrays from Recursion Equations

Marina C. Chen

1 Introduction

Systolic arrays as a class of high-performance computational structures have attracted a great deal of attention since their conception [15]. The class arouses fascination probably as much by its intricate pattern of data flow as by its efficiency and cost performance ratio. You may at once admire the ingenuity that went into a design but be totally frustrated at the attempt to modify it for solving a similar problem. The mystery associated with the working of systolic arrays does not just challenge a novice; even experienced designers find it extremely time consuming to set up both the array and inputs correctly. The design of systolic arrays is indeed a process that requires not only ingenuity, but lots of time to get the details right. We might ask: Is there a design methodology which provides some guidelines as to how one might come up with a new design? If we do have some idea about a new systolic design, can the tedious part of the design process be made easier? Can the design of systolic arrays be automated?

The question about how well automatic program synthesis in general might work has been a controversial one. Often it seems that correct and efficient solutions to problems can only be obtained by either relying on certain insights from the programmer or employing some kind of a search through a large library which can effectively provide derivation paths from problem specifications to solutions. Unfortunately, the latter approach is the difficult one of commanding a large body of relevant knowledge and applying it appropriately in an efficient manner. In attempting to synthesize systolic arrays, we encounter a similar problem. Thanks to the developments in the systolic design methodology, however, the synthesis approach can work extremely well — sometimes even better than a human designer — for a class of problems that can be described in a special stylized form. Hence the problem of synthesizing systolic arrays can be divided into two subproblems — a difficult one and a solvable one. The general synthesis issue: does there exist for a given problem a description of the aforementioned special form? And if so, what are the guidelines to derive it? The special synthesis issue: given a problem description in the special form, how do we derive systolic solutions, and automate the derivation?

Since Johnsson and Cohen's [6] z-transform method, Cappello and Steiglit's work on unifying systolic arrays by linear transforms, Moldovan's work on linear transformation of dependency vectors [23], and Leiserson and Saxe's work in retiming [17], — all attempts to systematically design systolic arrays — the mystery associated with the intricate data flow of systolic arrays has started to unveil. A sizable body of literature now exists on the systolic design methods addressing the special synthesis issue. These methods can be grouped into roughly two approaches, the transformational approach

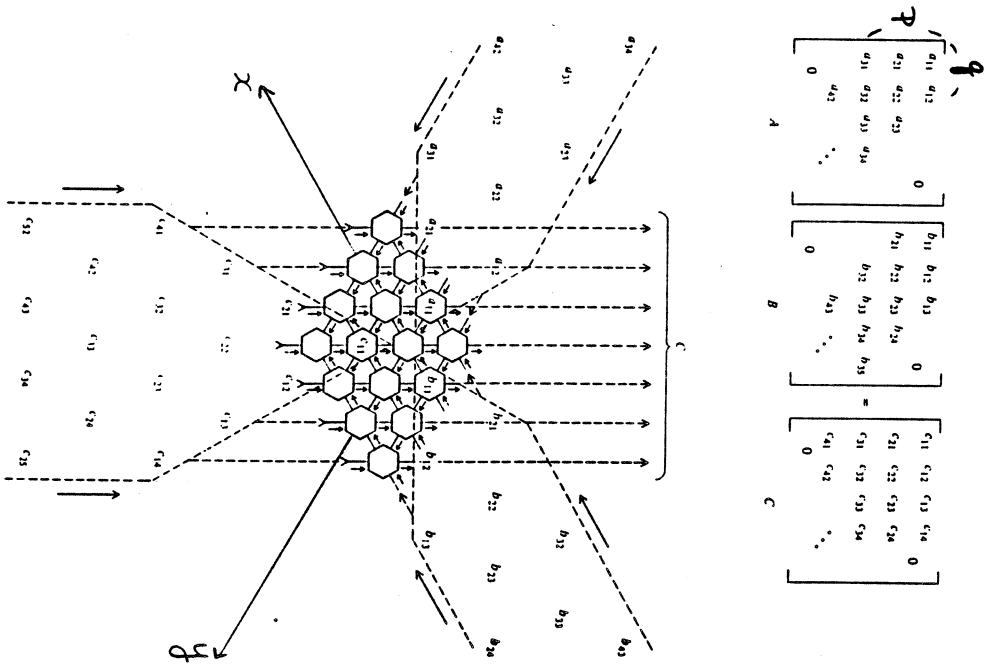

Figure 1: Kung and Leiserson's systolic array for band matrix multiplication for $p = 3, q = 2, w = 4$.

[6,5,14,23,22,21,26,18,4,12,8,9,10] and the graph theoretical approach [17,16,11,13].

This article describes a design method from the standpoint of the transformational approach and presents a survey of the state-of-the-art solutions to the special synthesis issue. It also presents some recent work in addressing the general synthesis issue and indicates some future work in that direction. The organization of this article is as follows: Section 2 introduces a systolic array by characterizing it in terms of five design parameters. Section 3 establishes a conceptual model for systolic computations. Section 4 and Section 5 address the special synthesis issue. The former describes the concept of a space-time map and the general methods for finding space-time maps. The latter describes an expedient procedure for finding linear space-time maps. Section 6 addresses the general synthesis issue of transforming a problem specification to a special class of recursion equations. Section 7 contains bibliography notes and a discussion of future work.

2 Understanding Systolic Arrays

Figure 1 illustrates the original systolic array of Kung and Leiserson for band matrix multiplication, where the band-width of the input matrix is $w = p + q - 1$ as shown in the figure. In this particular case, both matrices have the same band-width.

$$c_{ij} = \sum_{k=i-p+1}^{j+q-1} a_{ik} \times b_{kj} \text{ for } 0 < i, j \leq n, \text{ and } -w < i - j < w. \quad (1)$$

The systolic design can be characterized by the following parameters:

1. Interconnection: each processor in the array has three connections to neighboring processors; in the south to north, northwest to southeast, and northeast to southwest directions, respectively.

2. Functionality of each processor: the top output equals the product of the two "side" inputs plus the bottom input. Each of the side outputs is identical to the input from the opposite side.

3. Total number of processors and the locations of the processors: the array is of size $w_1 w_2$, where w_1 and w_2 are the bandwidth of the input matrices, and for this example, $w_1 = w_2 = w = 4$. To be more specific, if a coordinate system is given as shown, then the x-coordinates of the processors are ranging from $-q < x < p$, and similarly, $-q < y < p$.

4. Total execution time of the algorithm: the total elapsed time in between the moment when one of the first input data c_{11}^0 enters the array until the moment when the last output data c_{nn} leaves the array is $3(n-1) + min(w_1, w_2) = 3(n-1) + w$.

5. The locations of the input streams: given the coordinate system as shown, input matrix elements a_{ij} should appear in location $(i-j, -i-2j+2)$, b_{ij} in location $(-2i-j+2, -i+j)$, and c_{ij} in location $(2i+j-2, i+2j-2)$.

Readers are encouraged to verify the fact that the formula above indeed gives the locations of the matrix elements in the figure. With the above parameters, we know what each processor is supposed to do; we know how they are connected together; we know how inputs should be fed into the array; and we know how long to wait before fetching the answer. That is, these parameters specify completely a synchronous systolic array. Several other characteristics of the design can in turn be derived. For instance, input matrix elements are fed into the array diagonal-wise. Any two matrix elements lying on the same diagonal are separated by two zero's (or anything else) filling the gap, and the total length of each input stream is $3n$ where n is the order of the matrices. Each processor is only active at one out of every three clock cycles.

In their 1981 paper [28], Weiser and Davis presented an alternative systolic array as shown in Figure 2. It differs from Kung and Leiserson's array in several aspects: the direction of one of the connections is reversed; the input streams are placed differently; there are no filler elements in the input streams and therefore the total length of the input stream is n; processors are busy at every clock cycle; and most importantly, given a fixed length of time T, the number of matrix multiplications it can compute (or its throughput) is 3 times that of the other design.

Next, let's look at the problem of LU-decomposition. Figure 3 shows Kung and Leiserson's design, which is basically the design for matrix multiplication, inverted, with one of the processors performing a division rather than multiplication and some of the processors

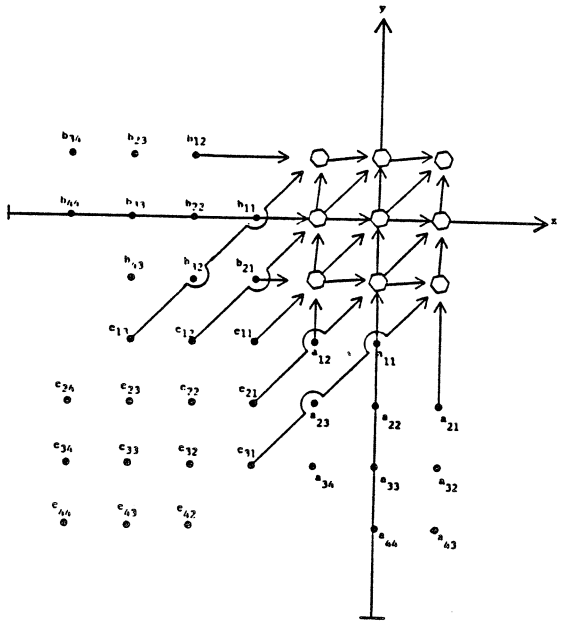

Figure 2: Weiser and Davis's systolic array for band matrix multiplication for $p = q = 2, w = 3$.

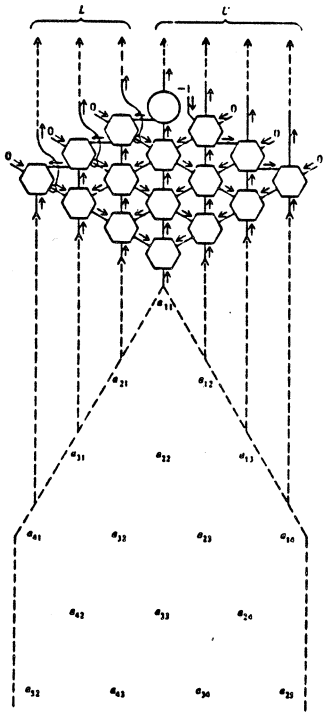

Figure 3: Kung and Leiserson's systolic array for LU-decomposition of band matrix for $p3, q = 2, w = 4$.

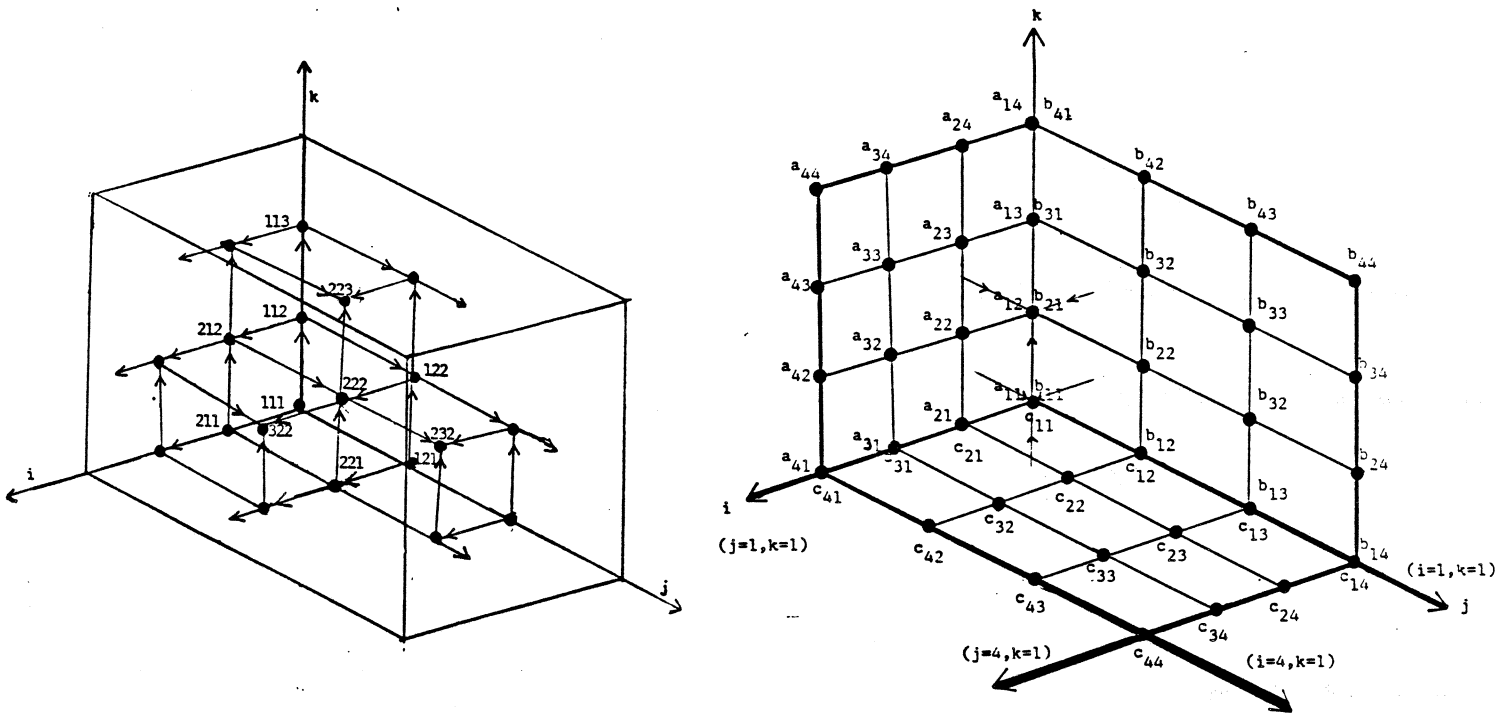

Figure 4: Matrix multiplication on a 3-dimensional array of processors for $p = q = n = 4$.

rotated 120 degrees. Now the question is: can Weiser and Davis's array be modified in a similar way to perform LU decomposition? And if so, how? A good understanding of systolic arrays should provide a simple answer to the above question. In the following section, we will construct a conceptual model for systolic computations so as to arrive at an answer that is simple enough to understand.

3 Conceptual Model of Systolic Computation

3.1 A Straightforward Design

We will first look at a simple and straightforward but non-optimal solution to the problem of multiplying two matrices. Figure 4 illustrates a three-dimensional array of processors in the coordinate system as shown. Processors are located at the integral grid point (i, j, k) within the cube where $0 < i, j, k \leq n$. Let input matrices be placed behind the two back faces of the cube: matrix element a_{ik} goes to grid point $(i, 0, k)$, and b_{kj} goes to grid point $(0, j, k)$. We also initialize below the bottom face of the cube at grid point $(i, j, 0)$ with c_{ij}^0. Now let each matrix element a_{ik} be copied to a row of processors along the j-axis. Similarly, let b_{kj} be copied to a row of processors along the i-axis. At this point, by accumulating partial products from each processor of the bottom face up, the column of processors along the k-axis, the resulting product c_{ij} now appears at the top face. The computation takes n time steps to do the copying for all rows at the same time, and n time steps to do the accumulation for all columns at the same time.

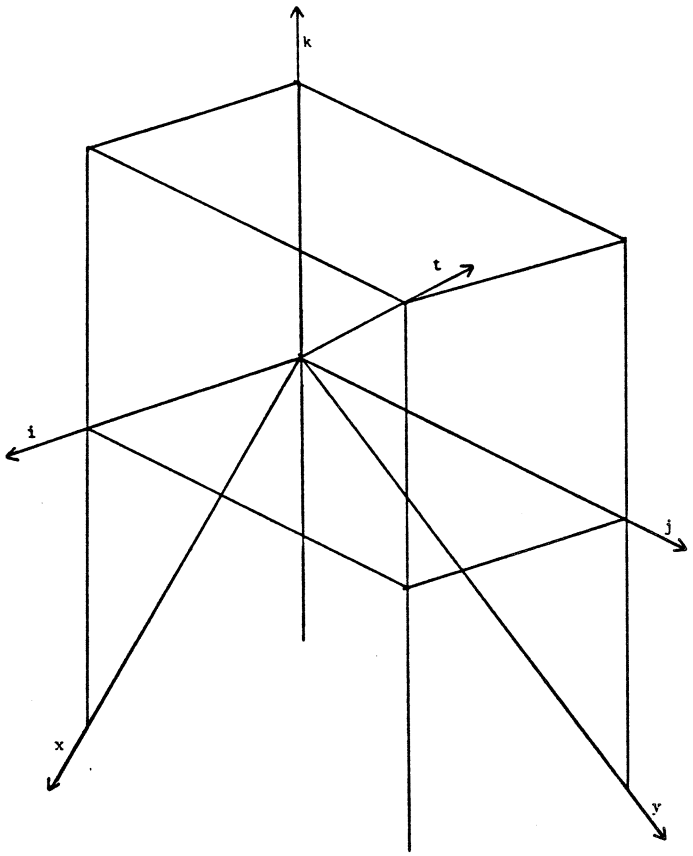

Figure 5: A new coordinate system for the same collection of processors.

3.2 Space-time Interpretation

Recall that the coordinate system shown in Figure 4 is chosen arbitrarily for the notational conveniences in describing the solution. Now that we know the solution, nothing can be hurt if we use a different coordinate system and start referring to processors with a different tuple of coordinates. Suppose now that we chose three new axes x, y, and t as shown in Figure 5. They relate to the old system by the following relation: $x = i - k$, $y = j - k$, and $t = i + j + k$. Every processor now has a new name, for instance, processor (i, j, k) in the old system becomes processor $(i - k, j - k, i + j + k)$ in the new system.

3.2.1 Re-usable Processors

Observe that any two connected processors in the old system have exactly two of the three coordinates i, j and k being the same and the third coordinate differing by 1. The new coordinate system has the curious property that any two connected processors p and q must have distinct values for the t coordinates. Furthermore, if a processor has a new coordinate $t = t_0$, then all its three neighboring processors receiving its outputs have new coordinates $t = t_0 + 1$ while those three neighboring processors from which it receives inputs have $t = t_0 - 1$. We say that the new coordinate system is *t-directed* if the t coordinates of any two processors P_1 and P_2 (say P_1 receives inputs from P_2) are related by $t_1 = t_2 + a$ for some positive integer a.

Notice that the functionality of each processor is to receive three inputs from processors with $t = t_0 - 1$: compute the new partial product and do the copying, and then send the

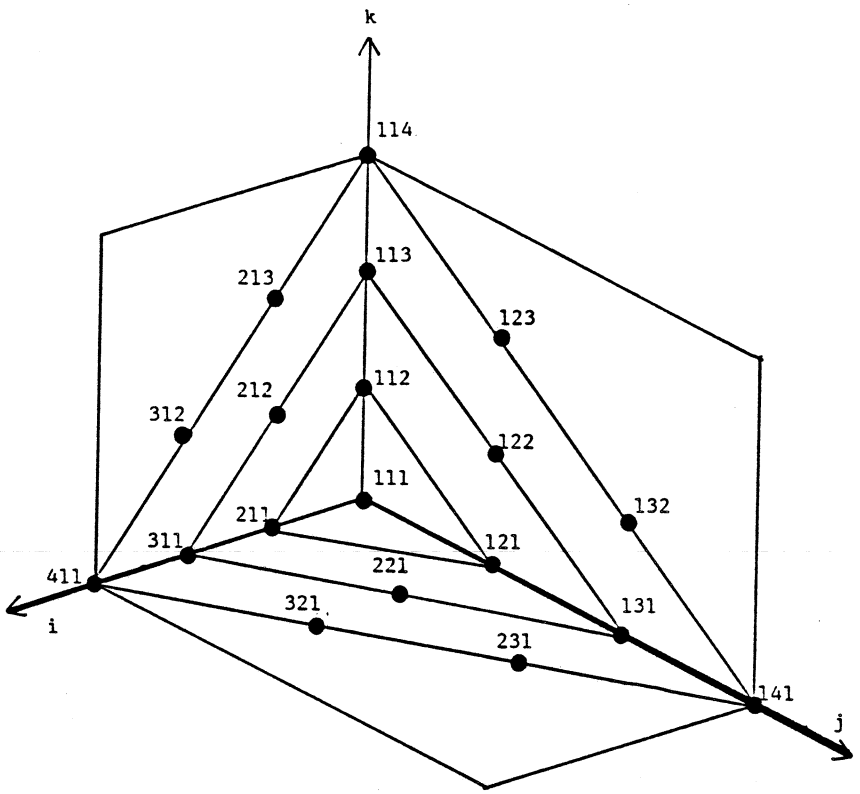

Figure 6: The time-slices.

outputs to processors with $t = t_0+1$. Hence due to the *t-directedness* property, a processor with $t = t_0$ needs neither to exist before $t = t_0$ nor after t_0+1, or in other words, a processor with $t = t_0$ can be reclaimed and reused for computations of processors with $t = t_0 + 1$, and this is true for all values of t. Note that all processors satisfying $i + j + k = t$ for any fixed t are all lying on the same plane, called a *t-slice* (or a time slice), as shown in Figure 6. Since any two t-slices do not interfere, a two-dimensional array of "reusable" processors should suffice to implement the design. Let t be interpreted as the time step and (x, y) be interpreted as the location of processors in space. Two distinct processors in the old system may now be at the same location in space but at different time steps under this *space-time interpretation.*

3.2.2 Synchronization of Inputs

Under the new interpretation, however, the initialization of the design must be modified for it to work correctly. Note that all of the three inputs for processor (i, j, k) must arrive at the processor at time step $t = i+j+k$. It takes a matrix element a_{ik} on the ik-backface $(j = 0)$ j steps to reach processor (i, j, k), and similarly, a matrix element b_{kj} on the kj-backface $(i = 0)$ i steps and c^0_{ij} on the bottom face $(k = 0)$ k steps to reach processor (i, j, k). Thus to synchronize all three inputs, the initialization of input matrix elements at $t = 0$ must be: a_{ik} placed at location $(i, -(i+k), k)$ so that it spends $(i+k)$ steps to reach $(i, 0, k)$ on the ik-backface and altogether $i + j + k$ time steps to reach (i, j, k). Similarly,

b_{kj} should be placed at location $(-(j+k), j, k)$, and c^0_{ij} at location $(i, j, -(i+j))$.

3.2.3 Derive a Systolic Array

To see that the space-time interpretation indeed results in a systolic array, let us derive the design parameters described in Section 2.

1. Interconnection: under the interpretation, the connection from processor (i, j, k) to processor $(i, j+1, k)$ results in a connection from processor $(x, y) = (i-k, j-k)$ at time $t = i+j+k$ to processor $(x, y+1)$ at $t+1$. In other words, processor (x, y) sends its output to processor $(x, y+1)$ via the northwest to southeast connection in Figure 1. Similarly, an output is sent to processors $(x+1, y)$ via the northeast to southwest connection and one to processor $(x-1, y-1)$ via the south to north connection.

2. Functionality: the partial product is obtained by multiplying two inputs from $(i, j-1, k)$ and $(i-1, j, k)$ and adding to the third input from $(i, j, k-1)$ and sending it to $(i, j, k+1)$. Under the interpretation, the first two inputs are indeed the two "side" inputs and the third one the bottom input. Similarly, each of the two side inputs is copied and sent to the opposite side as described in Section 2.

3. The location (x, y) of processor (i, j, k) will be $(i-k, j-k)$ from the space-time interpretation: since for band matrix multiplication a processor (i, j, k) appears in the design in Figure 4 only when $-q < i-k < p$ and $-q < j-k < p$, we have $-q < x < p$ and $-q < y < p$, the same as that of Kung and Leiserson's systolic array.

4. The total execution time: since $t = i+j+k$, the smallest time step is $t = 3$ when $(i, j, k) = (1, 1, 1)$ and the largest $t = 3n$ when $(i, j, k) = (n, n, n)$. Since node $(1, 1, 1)$ is first executed at processor $(x, y) = (i-k, j-k) = (0, 0)$, c^0_{11} must travel p steps from the bottom of the array to processor $(0, 0)$, and similarly q time steps must be added for the answer c_{nn} to travel from processor $(0, 0)$ to the top boundary of the array. So the total elapsed time is $3(n-1) + p + q - 1 = 3(n-1) + w$.

5. The location of the input streams: from Section 3.2.2, a_{ik} is placed at $(i, j, k) = (i, -(i+k), k)$. In the new coordinate system, a_{ik} is placed at $(x, y, t) = (i-k, -(i+k) - i, i - (i+k) + k)$, i.e., processor $(i-k, -2i-k)$ at time 0. What is shown in Figure 1 are inputs at $t = 2$. In this case, a_{ik} is at processor $(i-k, -2i-k+2)$, the same as described in Section 2. Similarly, the location of inputs b_{kj} and c_{ij} can be calculated.

To summarize, what we have done here is first to construct a straightforward design. We then find a function which maps the original coordinate system of the design to

another that has the t-directedness property. Next, we synchronize the inputs of the original design according to the t-coordinates. Finally, we interpret the original design under the new coordinate system where t becomes the time steps and (x, y) the processor-id. Note that once the relation between the old and new coordinate systems becomes known, the derivation of the resulting design is mechanical. Such a mechanical procedure alleviates a great deal of the tedious process of ensuring the correctness of a systolic array. The question is then how one finds a new coordinate system that has the t-directedness property.

3.3 DAG as an Abstract Model

We now introduce Directed Acyclic Graph (DAG) as an abstract model for describing systolic computations. The purpose is to use a uniform representation from problem specification to a straightforward design, and to computation on the final systolic array.

3.4 Representing Design by DAG

A processor may perform a sequence of actions through time. Each action would require some input data and perhaps the previous state of the processor and would produce some output data and the new state of the processor. Each such action, characterized by a state transition function from inputs and current state to outputs and next state, will be represented by a node $\mathbf{v} \in V$. A directed edge $e \in E$ goes from node $\mathbf{u}$ to node $\mathbf{v}$ and represents an output (or next state) of action $\mathbf{u}$, or conversely, an input (or current state) to action $\mathbf{v}$. As we have interpreted each node as a state transition function, the directed graph (V, E) must be acyclic to make sense. The set of edges, in fact, represents a binary relation on the set of nodes. We say that node $\mathbf{u}$ *precedes* node $\mathbf{v}$ if there is a directed edge going from node $\mathbf{u}$ to node $\mathbf{v}$, denoted by $\mathbf{u} \prec \mathbf{v}$.

In the above straightforward design for matrix multiplication, each processor (i, j, k) is represented by a node because it has a single action that takes three inputs and produces three outputs. Hence the DAG contains exactly $n \times n \times n$ nodes connected as a cube. Edges are directed, from every node (i, j, k) to $(i + 1, j, k)$, $(i, j + 1, k)$, and $(i, j, k + 1)$, respectively.

3.5 Generating a DAG from Specifications

A problem specification, on the other hand, is just a symbolic representation of a DAG. An action of a processor is described by a tuple of transition functions, each responsible for producing an output (or next state). Each transition function can be symbolically described as an equation where the output (or next state) appears on its left-hand side while inputs and current state needed to produce that output are on its right-hand side. A problem specification consists of a system of such equations, one for each output (or next state).

For instance, the DAG of Figure 4 can be described by the following equations:

$$A(i,j,k) = \begin{cases} j = 0 \rightarrow a_{ik} \\ 0 < j \leq n \rightarrow A(i,j-1,k) \end{cases}$$

$$B(i,j,k) = \begin{cases} i = 0 \rightarrow b_{kj} \\ 0 < i \leq n \rightarrow B(i-1,j,k) \end{cases} \quad (2)$$

$$C(i,j,k) = \begin{cases} k = 0 \rightarrow c0_{ij} \\ 0 < k \leq n \rightarrow C(i,j,k-1) + A(i,j,k) \times B(i,j,k) \end{cases}$$

where $A(i,j,k)$, $B(i,j,k)$, and $C(i,j,k)$ defined on the left-hand side of the equations are outputs of node (i,j,k), and $A(i,j-1,k)$, $B(i-1,j,k)$, and $C(i,j,k-1)$ on the right-hand side of the equations are the inputs.

A DAG can also be generated for such a system of equations. Every tuple of indices (e.g. (i,j,k) in the above equation) is a node, and there is a directed edge from node **u** node **v** is if **u** appears on the right-hand side of an equation while **v** appears on the left. For instance, an edge from $(i,j-1,k)$ to (i,j,k) is generated for all nodes such that $j > 0$ for the above system of equations.

3.6 Systolic DAG

If a DAG has the following properties, it is called a *systolic DAG*. (1) **Bounded-degrees:** each node only connects to a bounded number (independent of problem size) of other nodes, (2) **Local communications:** the shortest path between any two connected nodes has a bounded length. (3) **Proper time-slices:** each time-slice of the DAG has $O(N^p)$ number of nodes, where $p \geq 1$ and N is the problem size. From now on, we will call a DAG that has these properties a *systolic* DAG.

For example, in the matrix multiplication DAG, each node precedes and depends on, respectively, at most three other nodes. The shortest path between any two connected nodes is 1. Finally, the number of nodes per time-slice starts from 1 and grows larger to $3, 6, \ldots$, and then shrinks back down to $6, 3, 1$ again. Symbolically, it is the number of elements in the set $\{(i,j,k) \mid 0 \leq i,j,k \leq n : i+j+k = t\}$, which is of $O(n)$.

4 Synthesis by Space-time Mapping

In this section, we examine the special synthesis issue, that is, to find, for a systolic DAG, a new coordinate system that has the t-directedness property. We will seek a pair of functions (f,g), called a space-time map, where f is a timing function and g a space-mapping function.

4.1 Find the Timing Function

We will find for each node **v** of a DAG a timing function f where $t = f(\mathbf{v})$. For any given DAG, obviously any node **v** cannot start execution until all other nodes **u** preceding it

have finished execution. Hence the earliest time step for each node $\mathbf{v}$ to execute is simply the maximum path length over all incoming paths from source nodes (those that have no incoming edges). One can easily justify that the time step for each node defined this way always exists and has a unique value. We call the function f_{op} that maps each node to its time step value the *optimal timing function.* Obviously, by a breadth-first search procedure on the DAG, the optimal timing function f_{op} can be constructed node by node. Since it does not matter how we count the intial time step t_0 of a system, there can in fact be a family of timing functions $f(\mathbf{v}) = f_{op}(\mathbf{v}) + t_0$ for some constant t_0, which are essentially equivalent. We call the timing function f found by calculating the maximum path length the *optimal timing function with initial timing* t_0.

Taking the DAG of Figure 4 as an example, node $(1, 1, 1)$ in Figure 4 has three incoming paths originating from $(0, 1, 1)$, $(1, 0, 1)$, and $(1, 1, 0)$, respectively. Since all these nodes are sources, all three paths have length 1, which is the maximum length. Hence $f(1, 1, 1) = 1$. Node $(1, 1, 2)$ has three incoming paths from $(0, 1, 2)$, $(1, 0, 2)$, and $(1, 1, 1)$, respectively. Since node $(1, 1, 1)$ itself has maximum path length 1, the maximum path length for node $(1, 1, 2)$ is 2. In general, node (i, j, k) has three incoming paths from three neighboring nodes, and it is not hard to see that its maximum path length is $g(i, j, k) = i + j + k - 2$. We let $f = i + j + k$ where constant t_0 is set to 2 be the (optimal) timing function. Note that in this case, f is a linear function, even though in general a timing function may not be.

4.2 Find the Space-mapping Functions

Once the timing function has been found, the DAG can be sliced according to the time steps as shown by the series of parallel planes in Figure 6. To gain maximum possible parallelism, we would use a one-to-one function that maps every node in a given slice to some processor so that they can all be executed in parallel. Obviously, when each slice only contains a small number of nodes, then not much parallelism can be gained. Hence we require a systolic DAG to have the proper time-slice property in order to prevent trivial mappings that does not increase parallelism in any real sense.

The choices for such a space-mapping function for each slice are numerous. However, when we consider the entire DAG as a whole, some choices are obviously better than others Examples of optimization principles for space-mapping are: (1) Locality: two connected nodes in the DAG, now in separate slices, should be mapped to processors as close in space as possible to minimize the extra time delay that might be incurred due to the geometry of the processor network. (2) Minimizing array size: the locations of processors for different slices should overlap as much as possible so as to minimize the total number of processors in the array. (3) Regularity of network: the flow of data should be as regular as possible. We can choose space-mapping functions with respect to a particular optimization principle, or a combination of them. Intuitively, we can view the process of

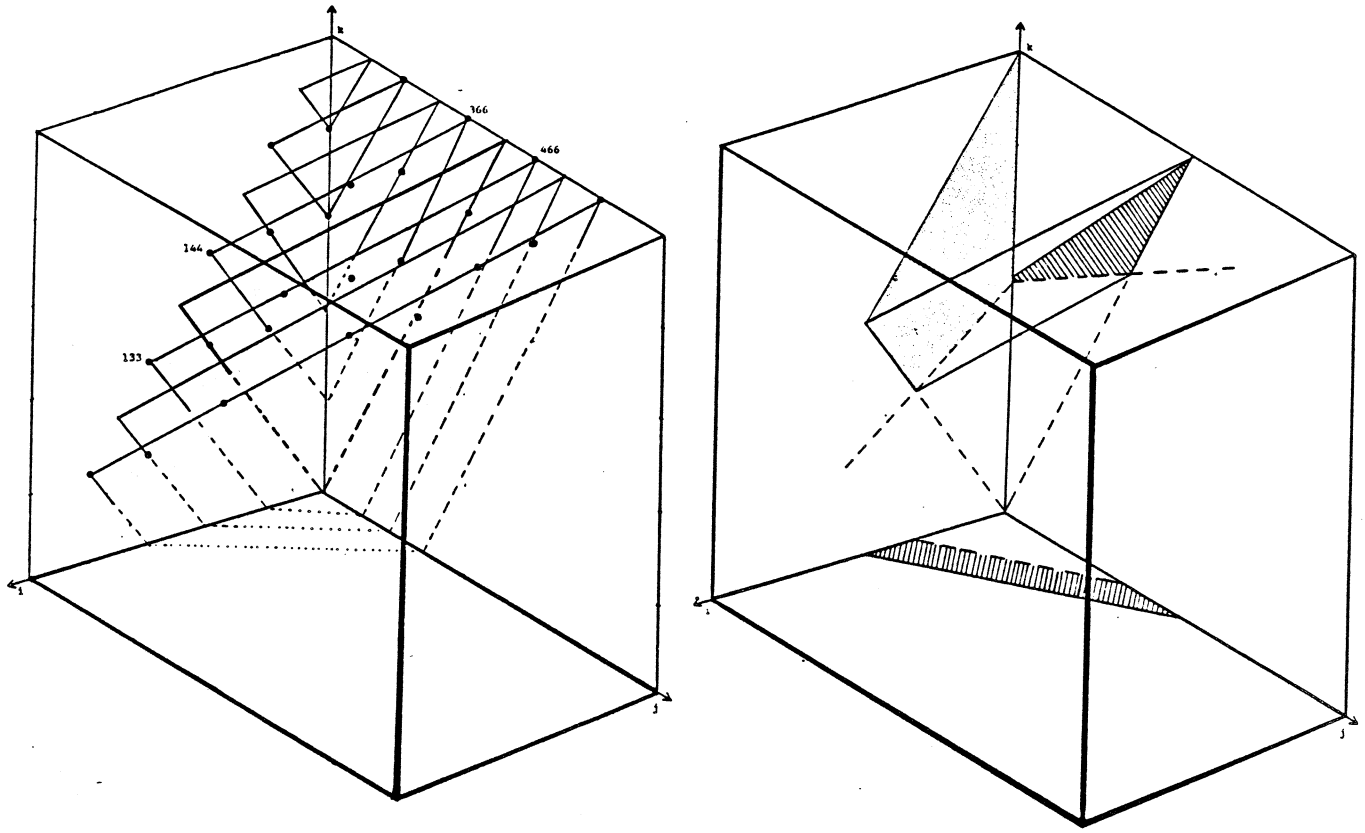

Figure 7: (a) Time slices in the dynamic programming DAG. (b) Projections on each of the three planes.

finding the space-mapping function g as follows: try projecting the time-slices to various (hyper-)planes, say, those normal to an axis for instance, as long as the projection for each slice is a one-to-one function, and then choose the projection, say, with minimal array size.

In Figure 7, on the left is the systolic DAG and its time-slices for the dynamic programming problem. On the right are the projections (shaded areas) of one slice (enclosed in solid line segments) onto plane ij, plane ik, and plane jk, respectively. Figure 8 shows the resulting arrays for the projection onto plane ij and plane ik, whose size is roughly only half of the former.

4.3 Further Refinement of the DAG Model

Since regularity and uniformity of data flow are of major concern for systolic arrays, it makes sense to consider *linear* space-time maps. For a subclass of problems, indeed, it is possible to find linear space-time maps which always yield regular systolic arrays with minimal time and a minimum number of processors. To do so, first some notations and definitions:

Consider the nodes of a DAG as elements of the vector space over the rationals. Given this interpretation of the nodes, then a natural interpretation of an edge is the difference of the two elements, called a *dependency vector*. For instance, in Figure 4, the edge from node (i,j,k) to node $(i+1,j,k)$ represents the dependency vector $(1,0,0)$, and there are

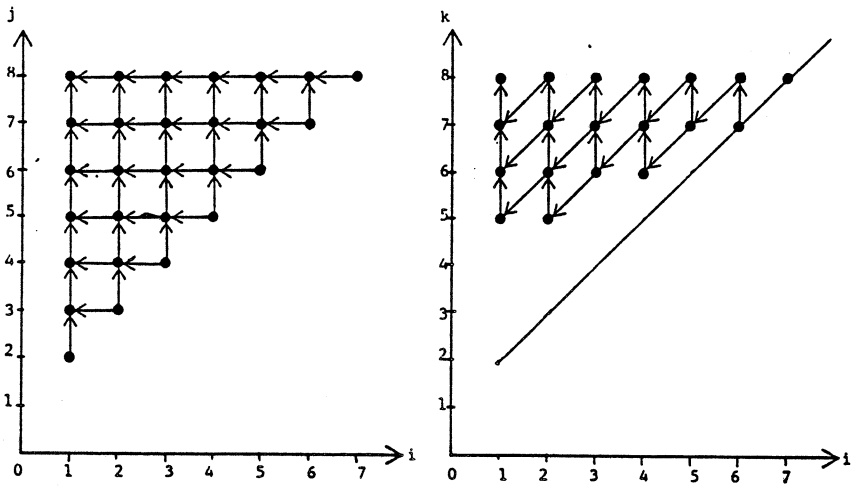

Figure 8: Systolic arrays resulting from two different projections.

altogether three dependency vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ in the DAG. We say that a DAG is of dimension m if the matrix formed by all the dependency vectors, called the dependency matrix, is of rank m.

4.4 A General Linear-map Procedure

The problem of seeking a linear space-time map can be formulated in terms of the dependency matrix, a mapping matrix T, and a time direction $\mathbf{t}$, which is the last row of matrix T. We can prove that for a given DAG with dependency matrix D, there exists a linear time map T with the t-directedness property if and only if there exists a time direction $\mathbf{t}$ such that for all dependency vectors $\mathbf{d}_i$, $\mathbf{d}_i \cdot \mathbf{t} > 0$. The goal is then to find T such that the resulting systolic array (characterized by a matrix $E = TD$), has minimized time steps and/or a minimized number of processors. Since time steps and processors must be counted by integers, in general, an integer programming procedure is performed.

5 An Expedient Linear-map Procedure

We now discuss an expedient procedure which finds the optimal linear space-time map without integer programming. Such a procedure is possible for a special class of DAG's. The expedient procedure is interesting for two reasons: (1) No costly integer programming procedure is needed, and its computational time complexity is proportional only to the

dimensionality of the DAG, not the size of the DAG; (2) all existent systolic arrays and their variants fall into the special class for which the procedure is applicable. The second reasons is more important, as better understanding of the special class may provide insight into the understanding of the nature of systolic computations.

5.1 Communication Matrix of Networks

In the above discussions, we view a DAG from the standpoint of a problem specification or a straightforward design. On the other hand, a DAG can be viewed from the perspective of a processor network. Each network can be viewed as defining a set of linearly independent ***basis communication vectors*** describing the data flow of a computation over the network. For instance, in an $(m-1)$-dimensional hypercube, a processor has $m-1$ connections to its nearest neighboring processors. Each of the $m-1$ communication vectors (one for each connection), will have m components. The first $m-1$ components indicate the movement in space and the m'th component, called the t-component, indicates the unit communication time, which is always the positive integer 1. These $m-1$ communication vectors, together with the communication vector $[0,0,\ldots,0,1]$ representing the processors's communication of its current state to its next state, form the basis communication vectors C, called, when written in the form of a matrix, the ***communication matrix.*** For an $(m-1)$-dimensional network, many possible sets of basis communication vectors are possible. Taking a two-dimensional hexagonal network as an example, where $m=3$, a diagonal connection has a communication vector $(1,1,1)$. The communication matrix C_1 below serves as the basis communication vectors as well as matrix C_2. Similarly, matrix C_3 gives another set of basis communication vectors, as shown in Figure 9.

$$C_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \; C_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \; C_3 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

For any $(m-1)$-dimensional network which has nearest neighbor connections and is regularly connected and indefinitely extensible, all possible basis communication matrices can be obtained by the enumeration of its symmetry groups (Lin and Mead [20]).

5.2 Uniform DAG

For an m-dimensional DAG, we can choose any m linearly independent dependency vectors as the ***basis dependency vectors***, or to form a basis dependency matrix B. We say that a DAG of dimension m with dependency matrix D is uniform if there exists a basis dependency matrix B, such that every dependency vector of the DAG can be expressed as a linear combination of the basis dependency vectors with non-negative integral coefficients, i.e., there exists a matrix A with only non-negative integral components such that $D = BA$.

The condition of non-negative components is motivated by the fact that, when it is satisfied, we can use as the space-time map the linear transform $T = CB^{-1}$ from the

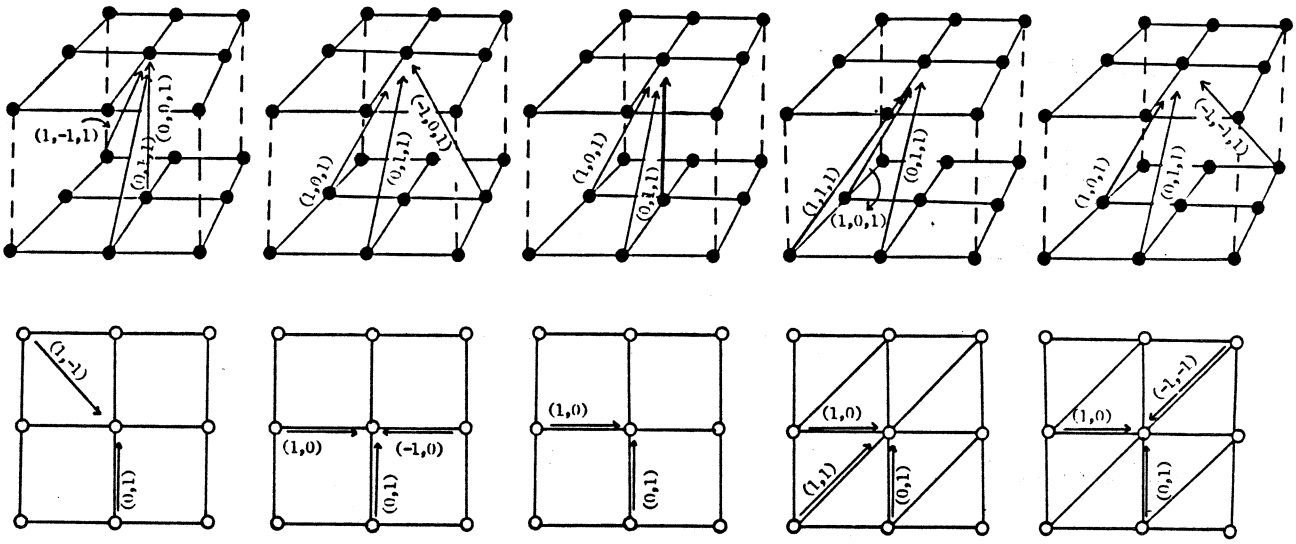

Figure 9: Networks of processors and corresponding DAG's in space and time.

basis data dependency vectors B of a DAG to the basis communication vectors C of a network, where B^{-1} is the inverse of B. We can check that the space-time map T results in a t-directed coordinate system: any dependency vector of a DAG $\mathbf{d}_i = B\mathbf{a}_i$ becomes $\mathbf{e}_i = C\mathbf{a}_i$ under the new system. Matrix E consists of such vectors $\mathbf{e}_i$ that describe the directions of data flow in the resulting systolic array. Since the t component of very basis communication vector is 1, and $\mathbf{a}_i$ has non-negative components, and therefore the t component of $\mathbf{e}_i$ equals the sum of those non-zero components of $\mathbf{a}_i$ which is positive. Hence we have found a new coordinate system for the DAG such that all nodes on which a given node depends will have smaller t coordinates than its own t coordinates, i.e., the t-directedness property.

The constraints of integral components of matrix A may rule out many DAG's which could otherwise be considered uniform. But these constraints make it possible to expedite the procedure of finding optimal space-time map for those DAG's that are indeed uniform. In this case, the main calculation is to test the uniformity condition and choose some optimal matrix B such that A has the smallest non-negative integral components. Since only those dependency vectors having components that divide those of the others need to be tested as basis, no costly integer programming needs to be performed.

Matrix A with smallest components, or the corresponding basis dependency matrix B may not be unique. Each choice of optimal B will generate several different systolic designs each of which corresponds to a communication matrix C.

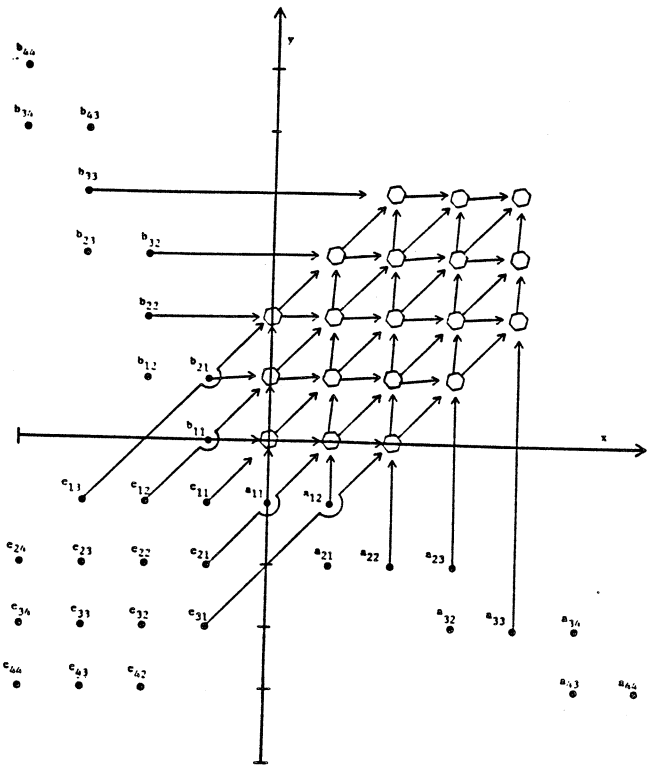

Figure 10: Another systolic array for band matrix multiplication for $p = q = 2, w = 3$.

5.3 Example

For example, in matrix multiplication, there are three dependency vectors, each written as a column vector of the matrix D They are linearly independent, and therefore are basis dependency vectors themselves, i.e., $B = D$. Now let's choose communication matrix C_1 below. Then the space time map $T = B^{-1}C_1 = C_1$ since B is an identity matrix. The direction of data flow is described $E = TD = TB = C_1$. Note that matrix T maps any node of a DAG, now written as a column vector $\begin{pmatrix} i \\ j \\ k \end{pmatrix}$ to its new coordinates $\begin{pmatrix} x \\ y \\ t \end{pmatrix}$, where $(x, y) = g(i, j, k) = (i - k, j - k)$ and $t = f(i, j, k) = i + j + k$, which is exactly the new coordinate system given in Section 3.2.1 that yields Kung and Leiserson's systolic design.

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; C_1 = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}, \; C_2 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Other designs are possible, for instance, we can choose communication matrix C_2 above. Following the same derivation as shown in Section 3.2.3, we obtain a different systolic design as shown in Figure 10. We can see that this systolic array has three streams of data flowing in the same directions as Weiser and Davis's array. It also has the same throughput rate as theirs, which is three times that of Kung and Leiserson's. However, it has a serious defect: the number of processors needed grows with the order of the input matrices, as can be seen from the figure, as opposed to growth proportional only to the band-width w_1 and w_2 of the input matrices. What choice of communication matrix yields Weiser and Davis's systolic array then? The reader will be kept in suspense until later.

5.4 Optimality of the Expedient Procedure

The above procedure takes advantage of the special case of a uniform DAG and finds a space-time map without resorting to solving an otherwise standard integer programming problem. But we need to show that the resulting systolic arrays are indeed optimal, just as if we had used the more expensive procedure.

The proof hinges on the fact that the basis dependency matrix B is chosen so that components of A are minimized and that each basis dependency vector belongs to D and is mapped to a "nearest neighbor" communication in a unit time step. Hence each column vector of matrix E has minimized t component, and minimized total number of time steps over all linear space-time maps. One may then choose from the various communication matrices C for a space-time map T that yields the minimum number of processors.

Examples of using such a simple procedure to find linear mappings of processes to parallel architectures for matrix products, LU decomposition, array multipliers, dynamic programming, etc., can be found in [3,4,2]. Most of the systolic algorithms reported in the literature can be obtained this way. New systolic algorithms are in fact discovered, due to the ability of being able to generate systematically all optimal basis dependency matrices and the communication matrices.

6 Program Transformation

If we managed to transform a problem specification to a systolic DAG, then the issue of seeking a systolic design is solved in the sense that if the systolic DAG is either uniform or has a time direction t, then linear time maps can be found by either the expedient or the general linear-map procedure; if the DAG fails the tests for both of these procedures, a non-linear space-time map can be found by a breadth-first search on the DAG for the timing function and then by projecting time-slices for the space-mapping functions. The remaining issue is on how to transform an initial problem specification to a systolic DAG.

Since a DAG can be generated from the symbolic representation of recursion equations and vice versa, the systolic properties of a DAG can be extracted from the equations as well. Hence the problem of constructing a systolic DAG for a given problem now becomes a problem of program transformation: from the initial problem specification to a system of recursion equations that have the systolic properties.

6.1 Transformation Rules

A few quite general symbolic transformation rules are available for this purpose. Each rule is defined as a function which, when applied to various parts of the initial specification, generates a target specification that yields a systolic DAG. We will illustrate, in particular,

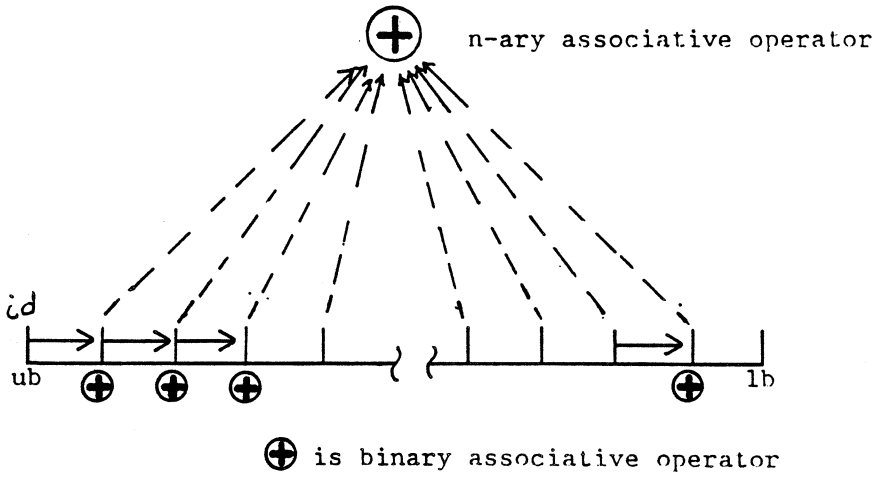

Figure 11: An n-nary associative operator has fan-in degree n is replaced by a series of binary oparators with fan-in degree 2.

the application of such transformation to the problem of matrix multiplication:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \text{ for } 1 \leq i, j \leq n. \tag{3}$$

can be transformed to the System of Equations (2).

- The input-pipeline rule:

$$inputPipe(S, \iota, lb, ub, I, op, id) = \begin{cases} \iota = lb \rightarrow id \\ lb < \iota < ub \rightarrow op(S(\iota - 1), I(\iota)) \end{cases} \tag{4}$$

  As shown in Figure 11, the *inputPipe* function will make a data stream S that moves along an axis indexed by ι starting after the lowerbound lb and ending before the upperbound ub. At each point in between the bounds, a binary associative operator op is applied to the partial value accumulated up to that point ($S(\iota - 1)$) and the input at that point ($I(\iota)$). In short, for any node in the DAG with large fan-in degree, supposing that the operator on the n-ary inputs are associative, then a stream S can be created to do the job by n binary operations performed serially.

- The $output - pipeline$ rule:

$$outputPipe(S, \iota, lb, ub, first, x) = \begin{cases} \iota = first \rightarrow x \\ lb < \iota < first \rightarrow S(\iota - 1) \\ first < \iota < ub \rightarrow S(\iota + 1) \end{cases} \tag{5}$$

  As shown in Figure 12, the *outputPipe* function will make a data stream S that moves along an axis indexed by ι starting at $\iota = first$ in two directions, one going from first down to the lowerbound lb and the other going up towards the upperbound ub. In the case that $ub = first$ or $lb = first$, only one direction of the stream exists. The stream simply copies the value at point $first$ to other points. Thus the large

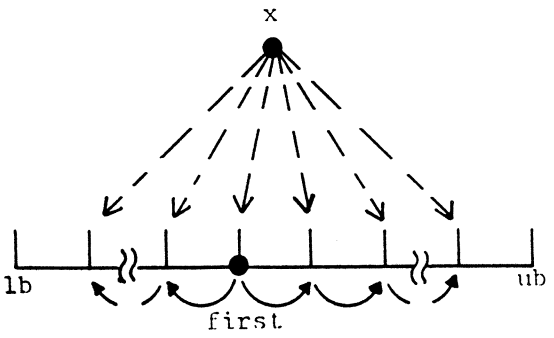

Figure 12: A value x goes to n places by broadcasting (fanout degree n) is replaced by a series of low fanout degree copying operations.

fan-out degree to all points between lb and ub is now reduced to 1 at each point. The choice of the point $first$ to assign the value x depends on two factors: (1) Locality: value x may be produced during the computation; $first$ should be close to that point (i.e., two points are separated by bounded path length). (2) Uniformity of data flow: when it is possible to be consistent with the locality constraint, $first$ should be set to either ub or lb to yield a stream that flows in only one direction.

The problem with Equation 3 is that the number of occurrences index pair ij for each fixed pair of indices ik or kj is proportional to n, and therefore the fan-out degree of a_{ik} or b_{kj} are not bounded. Conversely, the number of occurences of ik and kj for each fixed pair of indices ij is proportional to n, and therefore the fan-in degree of c_{ik} is not bounded. To obtain a systolic DAG of bounded fan-in and fan-out degrees, two function calls

$$outputPipe(A(i,k), j, 0, n+1, 0, a_{ik})$$
$$outputPipe(B(k,j), i, 0, n+1, 0, b_{kj})$$

are made to reduce the fan-out degrees of a_{ik} and b_{kj}. These two function calls yield the first two equations of the System 2. Next the function call

$$inputPipe(C(i,j), k, 0, n+1, (A(i,j) * B(i,j))(k), +, c^0_{ij}) \tag{6}$$

is made to reduce the fan-in degree and yields the third equation in the System 2. Note that the inputs $a_{ik} * b_{kj}$ in Equation 3 have been replaced by $A(i,j,k) * B(i,j,k)$, which are values from the two new data streams obtained by the two $outputPipe$ function calls. A slight abuse of notation is made in the above presentation when we use $A(i,j,k)$ interchangably with $A(i,j)(k)$ or $A(i,k)(j)$, or even distribute k outside an expression in $(A(i,j) * B(i,j))(k)$.

We have illustrated here that initial problem specifications can be transformed symbolically to a new specification that yields a systolic DAG. Both the procedure for determining the application of these transformation rules and the actual symbolic manipulation can be automated.

6.2 An Answer

Recall the question of whether there exists a systolic array à la Weiser and Davis for LU-decomposition. First, let's look at the space-time map T below which transforms the matrix multiplication systolic DAG of Figure 4 characterized by the dependency matrix D_1 to the Weiser and Davis array in Figure 2. This linear space-time map, however, cannot be obtained from either the expedient nor the general linear-map procedure because the last column has a t-component equal to -1, which makes no sense as a communication vector on any network, or stated in another way, the last row $(1, 1, -1)$ of T does not qualify as a t-direction for D_1 and the general linear mapping procedure only finds those linear mappings T that have a t-direction t for the corresponding DAG.

$$D_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ -1 & -1 & -1 \end{pmatrix}, \quad D_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}, E = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

If we had started with a systolic DAG characterized by D_2 above, then $E = TD_2$ above is exactly the matrix that characterizes Weiser and Davis's systolic array. This DAG can be obtained by applying the following function call instead of the one in (6).

$$inputPipe(C(i,j), k, 0, n+1, (A(i,j) * B(i,j))(k), +, c_{ij}^0) \tag{7}$$

This is saying that Weiser and Davis's array corresponds to the design of accumulating the partial products from the top face down to the bottom face as opposed to the one shown in Figure 4. Now it is clear that a Weiser and Davis type of array cannot exist for the problem of LU-decomposition because there is the inherent data dependency of iteration k on $k-1$ in its problem specification. However, there is another array, similar to the one shown in Figure 10 for which LU-decomposition is possible. It has the same data flow as Weiser's and Davis's array but is not suitable for band matrices due to the number of processors needed.

6.3 The General Synthesis Problem

The above transformation rules are of a quite general nature, and applicable to many problems [2]. However, it would be too naive to expect that a small set of rules would be adequate to transform any given problem specification to a systolic DAG. Unfortunately, the task of collecting a library of general purpose transformation rules for systolic DAG's resembles that for automatic program synthesis.

One good example that illustrates the problem of synthesis in general is the following: Transitive closure is a problem in that after the input-pipeline and output-pipeline rules are applied to the specification, the resulting systolic DAG has a non-linear function as an optimal timing function. A systolic array can be generated with a space-mapping function obtained by projecting the time-slices as described in Section 4.2. The resulting design

has a non-uniform data flow that needs more complicated control for timing (three types of delay elements). However, Rao, Citron and Kailath [27] have shown that, drawing upon the techniques developed for digital signal processing, the transitive closure problem can be solved by a systolic array with uniform data flow. Their graph extension method can be formulated in terms of program transformation to a systolic DAG with uniform flow only. It can then be mapped to a systolic array by a linear tranform obtained from the expedient procedure. Techniques such as graph extension have the property that they work extremely well for certain special classes of problems but are specialized and not applicable to problems outside the class. Similar to general purpose programming, knowledge and insight into a problem are required for developing elegant transformation from specifications into a systolic DAG. A completely automated synthesis system might only be achievable to a limited extent; however, when viewed as a parallel programming paradigm, the systolic synthesis methodology is extremely powerful. It takes care of the complexity of high-dimensional space-time relationships and the correctness problem in designing systolic arrays. The issues faced by the designer are elevated from extremely complex details of the operations on arrays to the high-level objectives of obtaining a systolic DAG, with or without uniform data flow.

7 Bibiliography Notes and Future Work

After establishing DAG as a conceptual model for systolic computation, we can see that the graph-theoretical approach and the transformational approach are closely related. The graph-theoretical approach treats the two-stage synthesis procedure — from specification to systolic DAG to systolic array, in one step. Instead of using a symbolic representation (recursion equations) for the problem, a graph is used directly. Such a graph is essentially equivalent to a DAG. The timing function, instead of appearing as the t-coordinates of a DAG, now appears in the form of the weights of edges of the graph.

The transformational approach can be further classified by the transform functions (space-time maps) used and the method for seeking these functions. For linear space-time maps, the search is treated as an optimization problem by integer programming [23,22,21,26,18,12,8,19]. Variations on the optimization principles for different objectives are possible; O'Keefe and Fortes [8] discuss two different optimization objectives, one for optimal time, and one for optimal area and time product. Approaches to the optimization procedure also vary: Li and Wah [18] search the space of possible input configurations for determining the interconnection while others search for the space of possible mapping results in different interconnections. Quinton [26] suggested the method of first finding the linear optimal timing function and then an allocation function for space-mapping. This approach has been adopted by Huang and Lengauer [10] but with a different approach to implementation. Chen [4] gives the general breadth-first search method, called the inductive method, for non-linear timing function. There are other treatments of non-

linear mapping such as in the work by Guerra and Melhem[9]. The expedient linear-map procedure is proposed by Chen [3]. Program transformations from Fortran (initial specifications) to the equivalent of a systolic DAG were presented by Moldovan[23], and Miranker and Winkler [21]. Formal and mechanizable symbolic transformations and more transformation rules are dicussed by Chen [2]. Rao, Citron and Kailath [27] have shown that techniques developed for digital signal processing can be applied to designing systolic arrays.

One of the remaining problems in seeking non-linear space-time map is that the breadth-first search procedure on a DAG requires the instantiation of the DAG with a given input size, which implies that such a parameter must be given at compile time; otherwise the mapping procedure must be delayed to the run time and incur certain run-time overhead. In the case where a linear time function exists, the alternative linear transform method is used and therefore this parameter is not needed. In the general case where the linear timing function does not exist, however, an alternative technique to instantiating a DAG with size parameter is possible. The technique is based on the fact that the breadth-first search procedure for obtaining the timing function can be viewed as an inductive procedure on the DAG, which is a well-founded set. Given that a DAG is well-founded, by using Boyer and Moore's [1] heuristics for establishing an induction hypothesis, a program may now "guess" at a timing function by examining only the symbolic representation of the DAG, without knowing the size of the DAG.

With the special synthesis issue mostly resolved, the general synthesis issue is now becoming the focal point of research in this area. With the advent of large scale parallel machines, systolic design methods are likely to become a general purpose programming paradigm. We are likely to see, in specific application domains, methods being developed for transforming problem specifications to ones that are suitable for systolic implementation. Perhaps one of the most important implications of the development of systolic synthesis methods is to allow systolic methods to be used as programming tools as well as for designing special-purpose hardware.

Mapping larger systolic designs to a fixed sized array is an important pragmatic problem, and its discussions can be found in [24,25]. The systolic array methods can be extended to programming large scale parallel machines, and dealing with fixed interconnections becomes a new issue [19]. A stronger version of the existence theorem of linear maps discussed in Section 4.4, in which linear space-time maps are considered with respect to a network with fixed interconnections, can be found in [19]. Another is the issue of fault tolerance on systolic arrays. In the presence of faulty processing elements or links, routing becomes the main issue and the kind of stylized data cannot be s ustained anymore. The dividing line between special purpose hardware with fault tolerent capability and general purpose high performance parallel machines [7] starts to blur. We can see that issues arising in designing systolic arrays for a range of problems with a range of problem

sizes — mapping computation to fixed size arrays, arrays with fixed interconnections, and fault tolerance — are all reminiscent of those arising in programming large-scale, general-purpose parallel machines. Systolic design, as it seems, has developed and bifurcated, over the years, into two different areas of parallel processing. On the one hand, it has become a concept, a programming paradigm, that transcends its implementation. On the other, except for special purpose hardware that aims at specific problems targeted for specific problem size ranges, the design of a systolic machine becomes a practice very much similar to the design of a general purpose parallel machine. Indeed, as envisioned originally, we will continue to see more systolic designs for more applications. Interestingly enough, though, is that what sets out to be solely for *special* purpose *hardware* has now been transformed into a design concept that is applicable to *software* on *general* purpose parallel machines.

References

[1] R. Boyer and J. Moore. Academic Press, New York, 1979.

[2] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[3] M. C. Chen. The generation of a class of multipliers: a synthesis approach to the design of highly parallel algorithms in vlsi. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 116–121, October 1985.

[4] M. C. Chen. Synthesizing systolic designs. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 209–215, May 1985.

[5] Marina C. Chen. A semantics for general concurrent systems and an algebra for linear systems. In *The Proceedings of IEEE Workshop on Languages for Automation, Chicago, Illinois*, November 1983.

[6] Danny Cohen. Mathematical approach to computational networks. In *IEEE Int'l Conf. to Computer Design : VLSI in Computers*, 1983.

[7] W.J. Dally and C.L. Seitz. The torus routing chip. *Journal of Distributed Systems*, 1(3), 1986.

[8] J.A.B. Fortes and M.T. O'Keefe. A comparative study of two systematic design methodologies for systolic arrays. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing*, pages 672–675, IEEE and ACM, 1986.

[9] C. Guerra and R. Melham. Synthesizing non-uniform systolic designs. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing*, pages 765–772, IEEE and ACM, 1986.

[10] Chua-Huang Huang and Christian Lengauer. *The Derivation of Systolic Implementations of Programs.* Technical Report Austin, Univ. Texas, 1986.

[11] Karel Culik II and Ivan Fris. Toplogical transformations as a tool in the design of systolic networks. *Theoretical Computer Science*, (37):183–216, 1985.

[12] Delosme J-M and Ilse Ipsen. An illustration of a methodology for the construction of efficient systolic architecture in vlsi. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 268–273, May 1985.

[13] J. Jover, T. Kailath, H. Lev-Ari, and S. Rao. On the analysis of synchhronous computing arrays. In *1986 USC Workshop on VLSI and Signal Processing*, page , Stanford, 1986.

[14] H. Kung and W. Lin. *An Algebra for VLSI Algorithm Design.* Technical Report, Carnegie-Mellon University, April 1983.

[15] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI Processor Arrays*, chapter 8.3. Addison-Wesley, 1980.

[16] Sun-yuan Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72(7):867–884, July 1984.

[17] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference on VLSI*, pages 87–116, Caltech, March 1983.

[18] G.-J. Li and Wah B. W. The design of optimal systolic arrays. *IEEE Transactions on Computer*, C-34(1):66–77, January 1985.

[19] J. Li, M.C. Chen, and M.F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report 513, Yale University, 1986.

[20] T.Z. Lin and C.A. Mead. *The Application of Group Theory in Classifying Systolic Arrays.* Display File 5006, Caltech, March 1982.

[21] W. L. Miranker. Spacetime representations of computational structures. In *Computing*, pages 93–114, 1984.

[22] Dan. I. Moldovan. Advis: a software package for the design of systolic arrays. *Proceedings of ICCD*, 1984.

[23] Dan I. Moldovan. On the design of algorithms for vlsi systolic arrays. In *IEEE Transaction on Computer*, 1983.

[24] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. on Computers*, C-35(1), Jan. 1986.

[25] J.J. Navarro, J.M. Llaveria, and M. Valero. Solving matrix problems with no size restriction on a systolic array processor. In *Proceedings of the 1986 Int'l. Conf. on Parallel Processing*, pages 676–683, IEEE and ACM, 1986.

[26] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of 11th Annual Symposium on Computer Architecture*, pages 208–214, 1984.

[27] S. Rao, T. Citron, and T. Kailath. Mesh-connected processor arrays for the transitive closure problem. In *Proceedings of the 24th Conference on Decision and Control*, pages 1565–1570, Stanford, December 1985.

[28] U. Weiser and A. Davis. *A Wavefront Notation Tool for VLSI Array Design.* Computer Science Press, 1981.