ON THE EFFICIENT DESIGN AND IMPLEMENTATION OF SYSTOLIC STRUCTURES

By

LEONIDAS ALEXANDROPOULOS

Bachelor of Science

University of Patras

Patras, Greece

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1988

ON THE EFFICIENT DESIGN AND IMPLEMENTATION OF SYSTOLIC STRUCTURES

Thesis Approved:

Donald D. Fisher
Thesis Adviser

J. Chandler

[illegible]

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express sincere gratitude to my thesis adviser Dr. D. D. Fisher, for his very helpful comments and suggestions throughout the writing of this thesis. I also thank him (even more) for the encouragement, sincere support and friendliness he showed me during these "hard" years.

I wish to express my deep appreciation to the other thesis committee members as well: Dr. G. E. Hedrick and Dr. J. P. Chandler for their overall help during my stay at Oklahoma State University.

Finally, I have to say this: Sofia, this degree and this thesis are as much yours as they are mine.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

ALU	arithmetic and logic unit
IU	interface unit
MDFL	matrix data-flow language
MFLOPS	million floating operations per second
MPP	massively parallel processor
PE	processing element
SA	systolic algorithm
SAP	systolic array processor
SIMD	single-instruction stream multiple-data stream
WAP	wavefront array processor
VLSI	very large scale integration

CHAPTER I

INTRODUCTION

Systolic Computing

The scientific community, from the time computers were invented, or even before that, was in need of great computing power. It seems that this need is never going to get saturated. Problems that were previously considered practically unsolvable are now efficiently computed while at the same time new areas are being explored and discovered.

Concurrency of operations is the main principle on which that increase in speed is based. To be more specific: inherent parallelism, that is, architectures which are designed to exploit parallelism, are giving computer science that boost we notice in the recent years. Large and especially Very Large Scale Integration (LSI and VLSI) made it possible to have architectures with tens or hundreds of processors, cooperating to solve a single problem.

Inexpensive hardware, faster circuit technologies, smaller feature sizes and using old disciplines (eg. pipelining, concurrency) which were succesfully used on conventional Von Neumann computers, all contributed in the development of the new parallel architectures.

A highly parallel architecture has three main characteristics.

1) It is composed of a large number of possibly heterogeneous computing elements.

2) The number of these elements is conceptually expandable, at a hardware cost not much greater than linear, achieving a speed-up that is not much lower than linear.

3) It is used to solve one single problem at a time; (unlike networks such as the Xerox PARC for example). We will focus our attention on systolic computers. From now on we will refer to a systolic array processor as SAP, a systolic algorithm will be called a SA, and the processing elements PEs or cells.

General Description

These systems are mostly special-purpose computers, used for applications that are computation intensive, such as matrix computations, signal processing, image processing, etc. Sometimes, parallel systems exhibit an I/O and computation imbalance; that is, I/O interfaces can not keep up with the very fast device speed, thus deteriorating the overall performance, drastically. The systolic architectures however, permit multiple computations for each memory access, thus speeding execution without increasing I/O requirements. In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through as many processing cells as possible, before it is returned to memory. So, in essence, we have a large number of PEs (processing elements), connected together in a local fashion, with some boundary processing cells performing the I/O. (Systolic arrays are usually attached to a host computer). The systolic model of computation implies communication via I/O queues; i.e. the output of a PE becomes the input to its neighbor(s). Local memory may be attached to each PE in some cases, but it is small and the accesses are limited and are in general undesirable. A more detailed look at the structure of systolic systems reveals the following 5 characteristics.

1. The computer consists of identical PEs, named cells, which are simple, performing maybe a few operations on incoming data and pumping the results out to nearby cell(s). Some of the cells are assigned as I/O cells, i.e. some "boundary cells" communicate with the outside world.

2. The interconnections between the cells are local and regular. It is very important to have local interconnections considering the fact that the number of cells may reach a few thousand. Systolic systems can be linear arrays. It is amazing to see how many algorithms can be executed on such a simple structure. Rectangular, hexagonal or triangular configurations are also common, and increase the parallelism even more. Because of its regular

and modular structure, such a system can be easily implemented, reconfigured, and expanded.

3. Even though a SAP is often compared to a pipeline, it is not always true that the data flow is unidirectional. We may have bidirectional flow of data and in the case of other than linear configurations, data may flow toward any direction (but always to nearest neighbor PEs).

4. Systolic devices are synchronous. That is, the cells operate under a common global clock and may (see 5) be computing at each tick of the clock. There is a very notable and interesting exception to that rule. It is called wavefront array processor (WAP) and it operates using the exactly opposite principle as far as timing is concerned. The WAP is completely data driven; it is a systolic data flow machine. It will be presented in a subsequent chapter.

5. It is desired that all of the cells operate at all cycles, for maximum utilization of the array. This, however, is not always possible due to the characteristics of each algorithm. Usually a symmetry exists in the operation of cells, such as: all of the odd numbered cells are working in one cycle and all of the even ones during the next. The "net effect" of this operation of the PEs is that the programmer can create data streams with different speeds travelling through the SAP.

It is usefull to see how SAPs differ from other multiprocessor schemes; we will list a few of these differences as compared to three other architectures.

(1) MPP (a) in general SAPs do not include programmable connectivity of the array edges so that we get different configurations such as a cylinder, torus, or leave them open as in the MPP. (b) SAPs do not operate in bit - slice like the MPP does.

(2) SIMD arrays (a) require global buses for broadcasting data and instruction codes, which is one of the features we try to avoid in SAPs, especially if a large number of PEs is involved. (b) In addition, they store a relatively large amount of data local to each processor. SAPs on the other hand, have small local memories and data flow regularly through the network with limited access to memory.

(3) Hypercube architecture (a) communicate using "packets". These packets contain headers as well as pure data. Systolic configurations on the other hand pass pure data to their neighbor PEs; furthermore, the communication in the former is not synchronous. (b) The hypercube can expand in such a way, that each of its nodes can be connected to a vast number of other nodes. In SAPs, the number of interconnections is small, due to the limited configurations of the arrays. Six connections exist at the most, in the case of a hexagonal array.

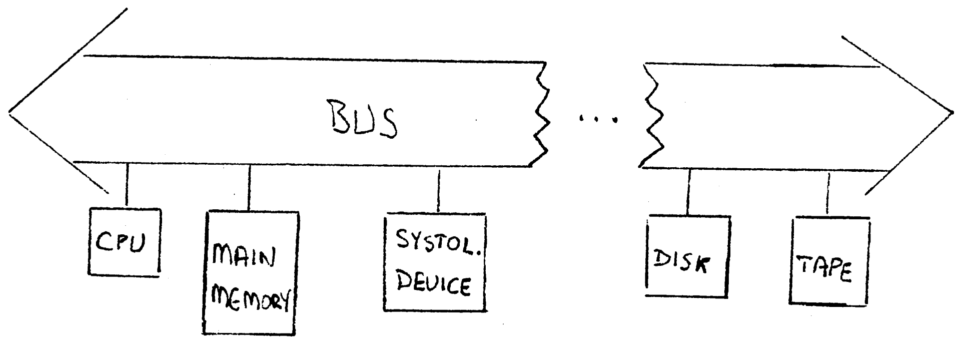

Figure 1. A systolic device connected to the bus of a computer system.

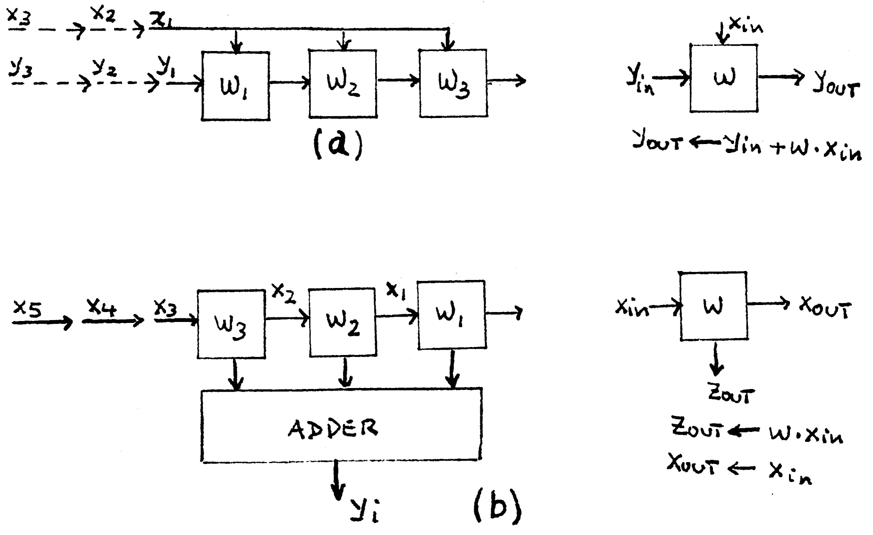

Figure 2. Two systolic arrays for convolution using (a) broadcasting; (b) fan-in.

Eliminating I/O Bottleneck

It has been stated earlier that "I/O" could become a bottleneck and thus deteriorate the overall performance of the systolic device. (In this section we will refer to "I/O" as accesses to main memory or to secondary storage devices). The problem occurs of course in I/O-bound computations, but in compute-bound problems as well, if the architecture used is a conventional one. This occurs because for every operation, at least one or two operands have to be fetched from, or stored to, memory. So, the total amount of "I/O" is proportional to the number of operations rather than the number of inputs and outputs. This means that even a compute-bound problem may become "I/O"-bound during its execution! Systolic architectures tend to overcome this problem since one access (to main memory or disk) usually ensures multiple computations. However, the problem may still exist.

Three techniques are used to maximize the throughput of a SAP, without increasing memory bandwidth.

1) The most obvious technique is to have all of the PEs perform a computation on each input data. (No idle cells exist at any time).

2) Broadcasting: a data item is fetched from memory and then transmitted to all cells simultaneously.

3) Unbounded Fan-In: data items from all cells are collected, either to be further processed or to be stored as results into memory.

Broadcasting and fan-in imply, of course, global data communications; this, in turn, means that a bus or a tree-like network must be used. This becomes a problem as the number of cells increases; wires become too long and we may have to slow down the system clock. So the first technique is preferable, if we want to retain the modular expandability of the system. In fact, the following can be said at this point, that also concerns the elimination of global communication in SAPs. If the sizes of the input and the output of a problem are larger than the size of the SAP, then all the inputs and intermediate results have to move during the computation. In this case, to achieve the greatest possible number of interactions among data we should let the data flow in both directions simultaneously. Furthermore, two-way pipelining is a powerful construct in the sense that it can eliminate the need for using undesirable <u>feedback</u> <u>loops</u>. This non-local communication would be needed in computing reccurences, etc.

In the following example we use the same problem with two different designs of a systolic array, to illustrate methods 2) and 3).

The Convolution problem: consider a vector $X = \{x_i\}$, i=1,...,n and a vector of weighting coefficients, W = $\{w_j\}$, j=1,...,k. In general n>>k. The convolution of X by W giving Y is defined as $y_s = \sum_{i=0}^{k-1} w_{i+1} * x_{i+s}$, s=1,...,n-k+1; (assume k=3).

- Design 2).

* Weights are preloaded to the cells one at each cell and are static.

* Partial results y_i are initialized to 0 and move systolically from left to right, one cell during each cycle, accumulating the correct result.

* The input sequence of x_i's is broadcast to all cells during each cycle. A few cycles are shown below.

* 1st cycle: $y_1 = w_1 x_1$ and $y_2 = y_3 = 0$.

  2nd cycle: $y_1 = w_1 x_1 + w_2 x_2$; $y_2 = w_1 x_2$; and $y_3 = 0$.

  3rd cycle: $y_1 = w_1 x_1 + w_2 x_2 + w_3 x_3$ (output); $y_2 = w_1 x_2 + w_2 x_3$; and $y_3 = w_1 x_3$ and so on.

- Design 3).

* As previously, weights are preloaded to the cells and do not move.

* The x_i's move systolically, from left to right.

* After the third cycle, an adder receives (Fan-In) as inputs $w_1 * x_1$, $w_2 * x_2$, $w_3 * x_3$ and its output is the sum of these terms i.e. y_1; then, during the next cycle the inputs to the adder will be $w_1 * x_2$, $w_2 * x_3$, $w_3 * x_4$ and the result is y_2, etc. (Obviously, the cells in 2) are of a different

structure than those of 3); the latter ones, perform only a multiplication).

Literature Review

In recent years, a lot of attention has been given to systolic architectures and algorithms; many of the aspects of systolic computing have been examined, although in our opinion, there are areas still to be examined, or to be examined more deeply.

SAs is the area with the most research done. Apostolico (1984) proposed algorithms that detect repetitions and statistics in strings. Chazelle (1984) deals with algorithms for geometrical problems that can be implemented on 1-dimensional SAPs; the interesting situation arose where the flow of data was irregular and not predetermined. Savage (1981, 1984) examines the design of a systolic chip for graph or spanning tree connectivity problems. Others that have examined data structures problems in terms of systolic computing include Shih (1987); he proposes four algorithms for examining 'All Pairs of Elements'. Leiserson in his book (1983), presents systolic priority queues. Kung H. T. (1980) and Lehman (1981) construct SAPs and SAs for efficient implementation of relational database operations and in 1986 , Kung H. T. proposed SAs for image processing operations. Kung S. Y. (1987) proposed hexagonal and orthogonal arrays for execution of the Warshall algorithm (for the transitive closure problem) and for the Floyd algorithm (shortest path problem). This paper also proposes a mapping procedure of these SAs to SAPs. Signal processing related algorithms and systolic computing can be found in Fisher (1981), (running order statistics problem); Cappello (1981) and in Kung S. Y. (1984). Proposed SAPs can be found in Annaratone (1986) and Kung H.T. (1986) where the WARP array is examined; Kung S. Y. (1982, 1987) discusses the design of a data-flow SAP called wavefront array, mainly suitable for signal processing applications. One of the few truly general - purpose SAPs can be found in Foulser (1987); the Saxpy Matrix-1 is a very flexible, matrix-oriented systolic architecture, with a very good performance. More material on integration of systolic devices into a system, or pipelining of the arithmetic units in a systolic array, etc. can be found in Bromley (1981), Drake (1987), Fortes (1987), Hockney (1981), Kung H. T. (1981, 1983) and Mead (1980). A variety of systolic designs and problems can be found in Leiserson (1983) and Kung H. T. (1982). We can say that the work done by Kung H. T. (and Leiserson) is the bible of systolic processing, as the basic ideas were proposed by them, around 1978, at CMU. Fundamental to the efficient design of SAPs and to the execution of SAs is mapping and partitioning. Moldovan (1982, 1983, 1986) proposes mapping schemes based on the mathematical transformations of index sets and data dependence vectors.

Li (1985) presents a mapping procedure, based on parameters characterizing systolic processing, which leads to an optimization problem. Another method based on graphs and data dependencies, can be found in Miranker (1984). O'Keefe (1986) examines briefly two of the mapping methods, while Guerra (1986) is concerned with a mapping procedure for non-uniform data flow algorithms. A very good article on parallel algorithms is presented in Kung, H. T. (1980). A corresponding (to the mapping) procedure, for partitioning, can be found in Moldovan (1986). Other partitioning methods are discussed by Navarro (1986, 1987) and they concern matrix related algorithms.

CHAPTER II

SYSTOLIC ARCHITECTURES AND MODELS

Systolic Architectures

It is true that a lot of attention has been given in the recent years to systolic processing. It is also true however, that very few systolic devices have actually been built, tested, and successfully operated. This should not come as a surprise, because systematic research in the area of systolic computing is only a few years old. Most of the existing systolic devices are used as the main processing unit of real-time systems, where very fast responses are required; some SAPs can even be considered as nothing more than hardware implementations of given algorithms.

Design optimality criteria

In order to find an optimal design of a SAP, the optimality criteria must include many factors. The final choice of the optimality criteria is application dependent. Some typical factors are listed below.
- Pipelining period: the time interval between two successive computations for a processor, it is denoted by 'a'. This means that the processor is busy for one out of

every 'a' time intervals.

- Computation time: the time interval between the start of the first computation and the end of the last computation of a problem instance by the SAP.
- Block pipelining period: the time interval between the initiations of two successive problem instances by the SAP.
- Array size: the number of processors in the array. The array size determines the basic hardware cost.
- I/O channels: the number of input/output lines between the processor array and the outside world (the host computer).

For the construction of a general-purpose systolic system, techniques are needed, so that the array has the capability of efficiently executing algorithms as diverse as possible. This can be accomplished in two ways.

(a) Adding hardware mechanisms so as to reconfigure the topology and interconnection pattern of the SAP and to emulate the requirements of a specialized design. An example of this approach is CHiP (Configurable Highly Parallel computer), which has a programmable lattice of switches for reconfiguration purposes.

(b) Use of software to map different algorithms to a fixed architecture. The cost to this (more flexible) approach, is that it usually requires the use of programming languages capable of expressing parallelism, development of compilers, operating systems, etc. The above apply to Warp, a systolic array, developed at Carnegie Mellon University.

Next, we will present some of the most successful systolic architectures now in use. The two machines that follow can be considered general-purpose systolic architectures.

Warp

Warp is a 10 (or more) cell linear systolic array, mainly used for computation in the areas of signal, image and low-level vision processing. The systolic array is integrated into a UNIX system.

GENERAL DESCRIPTION and FEATURES. The machine consists of three major components:

1) the Warp processor array (Warp array);

2) the interface unit (IU);

3) the host.

The Warp array consists of a linear systolic array of 10 cells. These cells are identical and programmable. Each cell has its own program memory of 4K words and two functional units; these handle 32-bit floating point multiplication and other general operations. The floating point processors can deliver up to 5 MFLOPS (this is for one cell alone). All of these components along with some buffers are interconnected using a crossbar switch. The cell microinstruction is 112-bits wide. Each cell also has its own microsequencer, which generates the next address for the microprogram of each cell.

Data flow through the array on two data paths named X and Y, while addresses and systolic control signals travel on the Adr path. One of the features of Warp that makes it very efficient is its high I/O bandwidth. Each Warp cell can transfer up to 80 Mbytes to and from its neighboring cells per second; hence, it avoids that bottleneck).

The interface unit handles the input/output between the array and the host and generates addresses and control signals for the Warp array. For address generation, the IU has an integer ALU capable of generating two addresses every 200 ns. During data transfers, the IU can convert 8-bit or 16-bit integers from the host into 32-bit floating point numbers for the Warp array and vice versa. One of the primary reasons that the Warp array is so powerful, is that address generation - for the cells, is basically done in the IU; the same holds for the loop controls. In this way, the cells perform mostly actual computations, rather than have their functional units busy, generating addresses.

The IU is controlled by a 96-bit wide programmable microengine, which is similar to the Warp cell controller in programmability.

The host system is UNIX based. It executes those parts

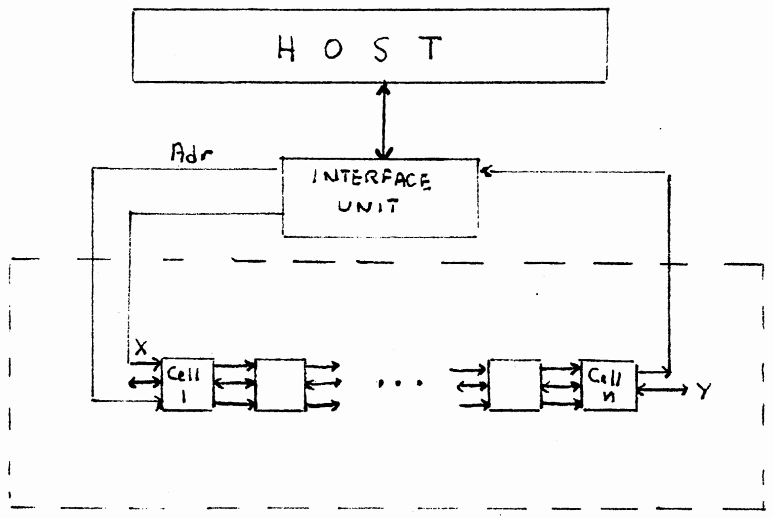

Figure 3. Warp machine overview.

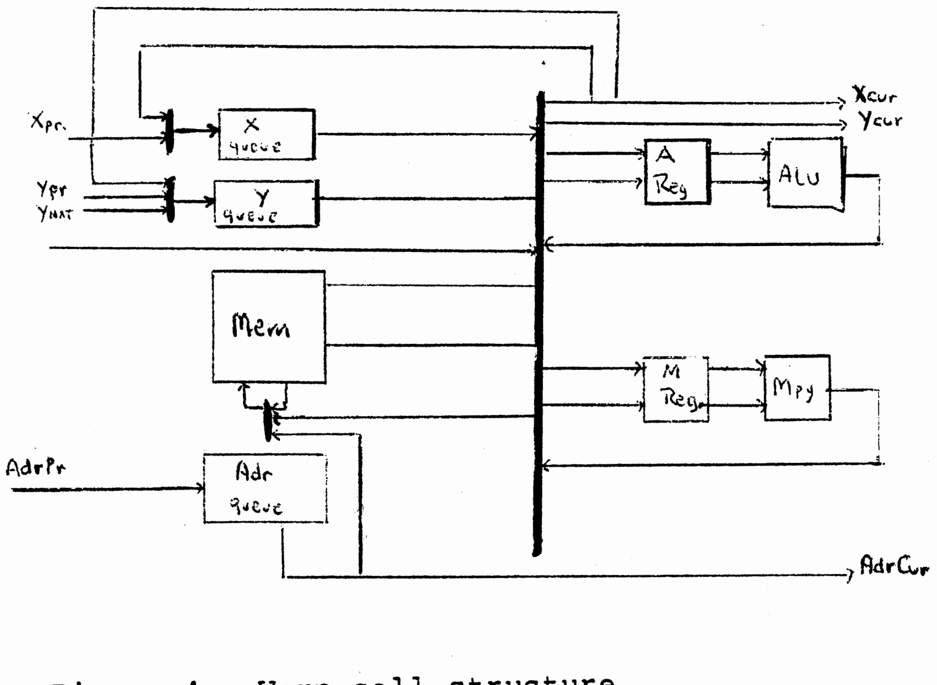

Figure 4. Warp cell structure.

of an application that do not map well onto the Warp array. It has total control of the clock generator and coordinates all the peripherals. The host itself consists of a workstation, supporting UNIX (master) and an "external host", built around a bus. The "external host" consists of three microprocessors. Two of these work in parallel during computation, each handling a uni-directional flow of data to/from the Warp processor through the IU. The support processor controls peripheral I/O devices and handles floating point exception and other interrupt signals from the Warp array.

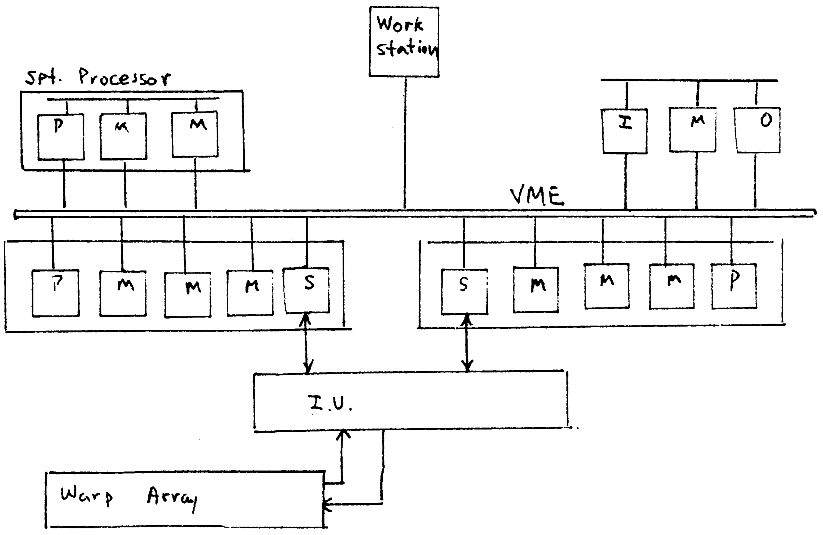

Figure 5. Host of the Warp machine.

Concluding: Warp is a powerful and usable machine. It it relatively easy to program. To the programmer, Warp is an array of simple sequential processors, communicating asynchronously! Its cells are very flexible and the intra-cell bandwidth is high, so that a communication bottleneck is avoided. The implementation of Warp was done with rather old and conventional existing parts, so that an improvement is easily feasible and expected in the near future.

Saxpy Matrix-1

Matrix-1 uses an SIMD control hierarchy, a large global memory and a small number of fast processing elements. In the systolic array, the data paths can be either purely systolic or global. Other features include:
the use of FORTRAN as the programming language; the emphasis on block algorithms; the provision in the hardware of double-buffered, software-managed local memory, for the systolic array to support block algorithms.

The system consists of five principal components.
1) The system controller, a general-purpose computer that executes the application program and allocates Matrix-1 resources.
2) The matrix processor, a linear array of up to 32 pipelined, floating-point processors that have systolic and global interconnections.
3) The system memory, which stores all data arrays for use

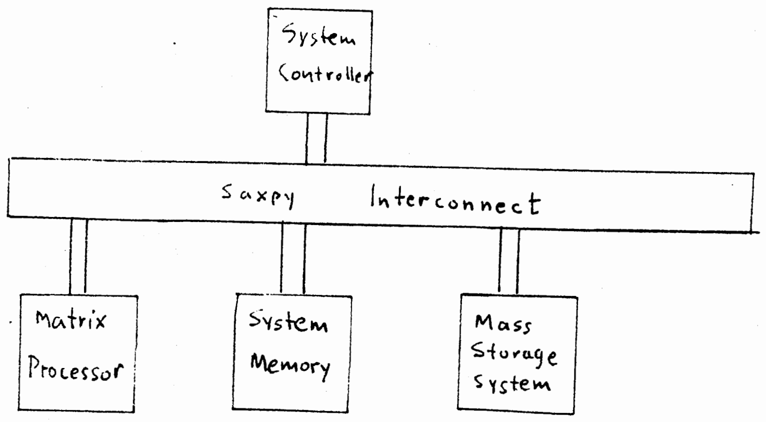

Figure 6. Block diagram of the Matrix-1 system.

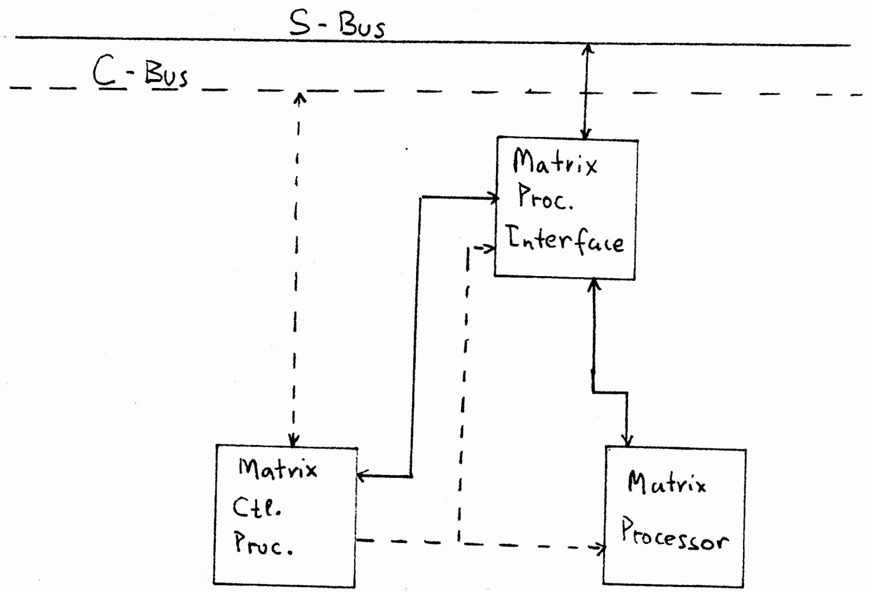

Figure 7. Matrix processor unit subsystem.

by the matrix processor.

4) The mass storage system, an I/O interface that provides access to high-speed data-storage peripherals.

5) The Saxpy interconnect, a combined control and data bus that links the other four units of the Matrix-1.

The system controller is the host, a VAX that runs VMS. Its functions are to compile, link and execute the application program, send control information across the system and to coordinate resources; in addition to the above, practically all floating-point computation is performed in this unit.

The matrix processor subsystem has 3 components: the matrix processor (the programmable array), the matrix processor interface (the data pathway between the processor array and the Saxpy interconnect) and the matrix control processor, (a processor that decodes commands and controls the interface and processor array).

An important feature of Matrix-1 is the architecture of the matrix processor. It is an array of 8, 16, 24 or 32 vector processors that are called <u>computational zones</u>. Each of the computational zones consists of an arithmetic and logic unit, a multiplier and a local memory of 4K. All of the units operate on 32-bit floating point data. The peak computing rate of all 32 zones working is close to 1000 MFLOPS. The architecture of the array is such

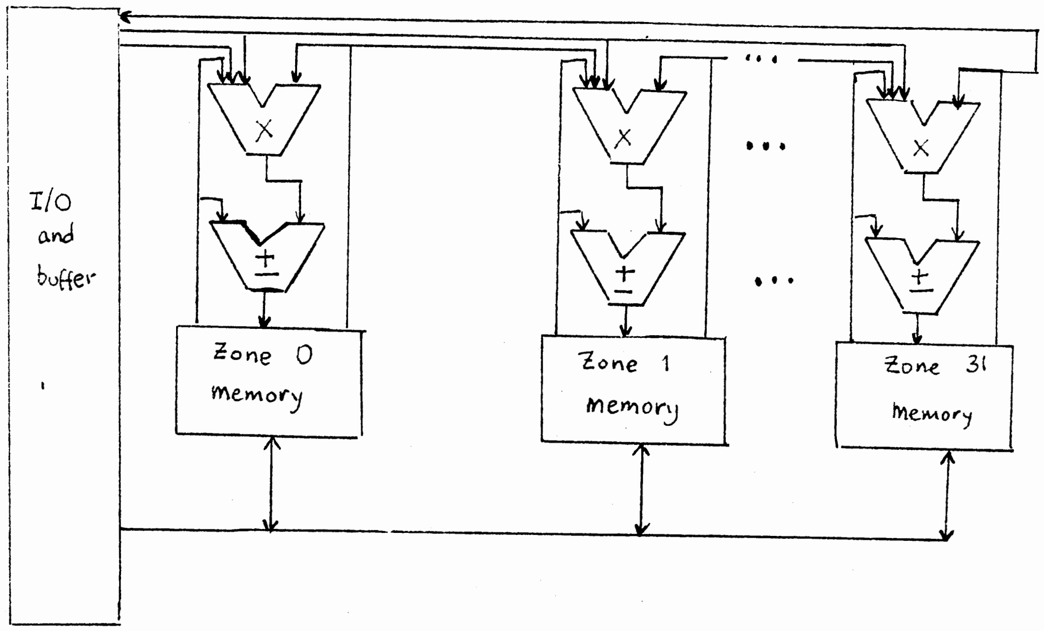

Figure 8. The Matrix processor zone architecture.

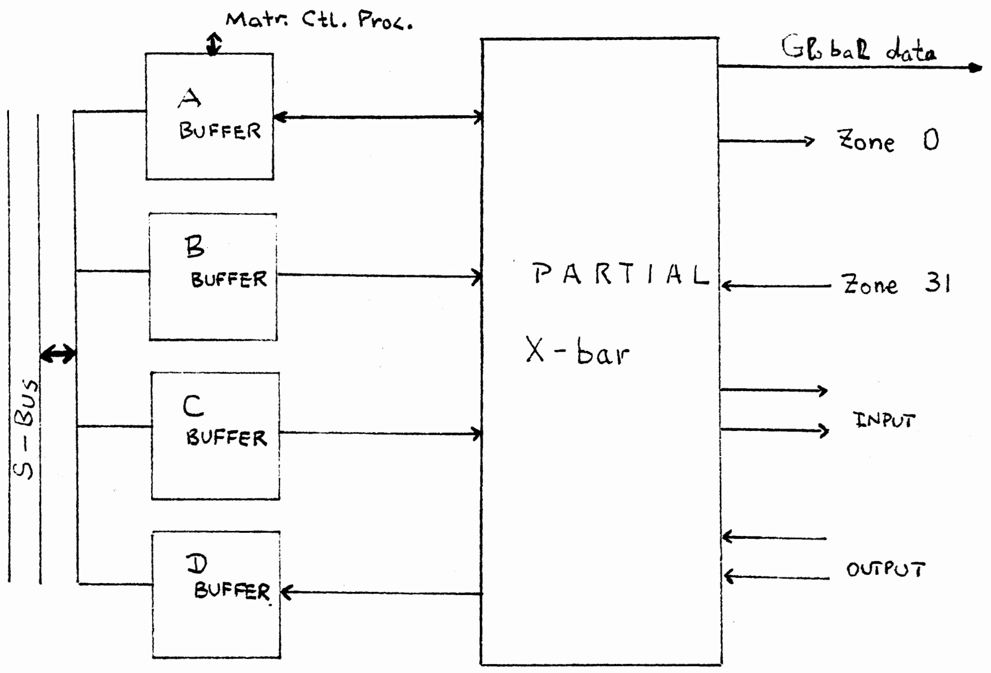

Figure 9. The Matrix processor interface.

that makes it versatile.

The computational zones can function in systolic mode, or in block mode. (In the first case, data are transferred linearly across the zones; in the second case all zones operate independently and use local data). Any subset of the zones may be disabled by masking. Finally, the zone memories allow indirect addressing, in which elements of one vector are used as pointers into another vector.

The matrix processor interface, mediates between the system data bus and the internal buses of the matrix processor. Four, two-ported buffers allow for fast concurrent transfers with system memory and transfers with the matrix processor zone memories.

The matrix control processor executes computational subroutines to control the flow of data between system memory and the matrix processor and to issue the computational instructions to the zones. Also, large arrays are decomposed here if needed, and blocks of data are executed. The details are hidden from the application level.

In the system memory reside the data that are processed by Matrix-1. Its size ranges from 16 M - 128 M words. Each job is guaranteed to have all the available memory; no virtual addressing is used. Performance is therefore predictable and is not affected by swapping or other schemes. If however, a specific application exceeds the memory available, then off-line memory management is needed. Memory cycle time is 100 ns and a wideword of 8 adjacent 32-bit words is read/written in each cycle. Some further comments follow.

- How does the relatively slow system controller (VAX), direct the very fast matrix processor?
- This can be done because the system features

1) asynchronous execution of the system controller and matrix processor;
2) hierarchical control;
3) storage of data in system memory.

The application program in the system controller issues control packets and not single control instructions. These control packets are buffered in queues and maintained by the system management interface so the system controller proceeds independently of I/O and the matrix processor.

Application programs are written in a high-level language as FORTRAN or C and run on the system controller. In order to direct the matrix processor to operate on the large data arrays located in system memory, the application program makes calls to matrix processor subroutines. Each such subroutine performs a substantial amount of computation on a data array in system memory. So the matrix processor is fast but very busy too and this is why the VAX can keep up with it.

In conclusion, Matrix-1 is probably the most successful general-purpose systolic processor. It is flexible, since its matrix processor can be reconfigured, and the global data path can be used. The memory management contributes a lot to the high performance of the system. (No virtual memory, no cache, but a small and very fast local memory is used - the block store. A cache would be ineffective for each of the 32 processors). Extensive buffering is used as is a global buffer containing common data, that are shared by all the zones. The system is user-friendly too, as the applications programmer uses high-level operators and is not burdened with the low-level algorithmic and hardware details.

The Wavefront Array Processor

We will briefly describe the architectural model and the basic ideas behind the wavefront array processor (WAP). This computing structure differs from the architectures presented thus far in that its operation is asynchronous; that is, the computation is purely data-driven. The most promising configuration for a WAP seems to be the orthogonal one, because matrix operations are easily implemented on it. (A lot of problems can be transformed into matrix operations, including many signal processing problems, for which the WAP was invented). Conceptually, the requirement for correct <u>timing</u> in the systolic array is now replaced by a requirement for correct sequencing in the wavefront array.

The main reason that this architecture arose is that for a synchronous system it is necessary to distribute a clock signal over the entire array. For very large systems, the clock skew incurred in global clock distribution is a nontrivial factor, causing unnecessary slowdown in the clock rate.

Computational wavefront. It is a term describing effectively the computation in a WAP. The computation activities resemble a wave propagation phenomenon. More precisely, the recursive nature of the algorithm, in conjuction with the localized data dependency, points to a continuously advancing wave of data and computational activity.

Example. Consider the case of an orthogonal N x N array; the computation could start at the processor in the upper left corner, then move to processors (1, 2) and (2, 1), etc. Immediately after the first wave propagates we can execute/start a second wave, etc.

Suppose that we want to execute a matrix multiplication C = A x B where all matrices are N x N. A recursive formula to accomplish this, is:

$$C^{k} = C^{k-1} + A_{k} * B_{k}, \; k=1, 2, \ldots, N,$$

where A_i is column i of A, and B_i is row i of B. In this case, each wavefront would correspond to a recursion.

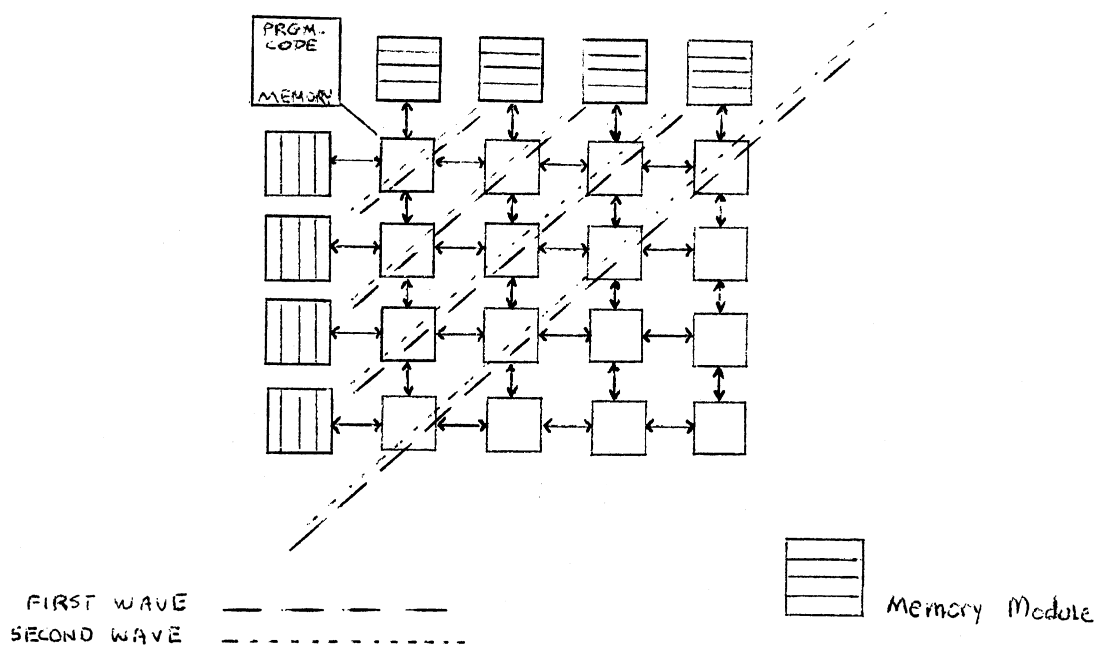

Figure 10. The WAP configuration.

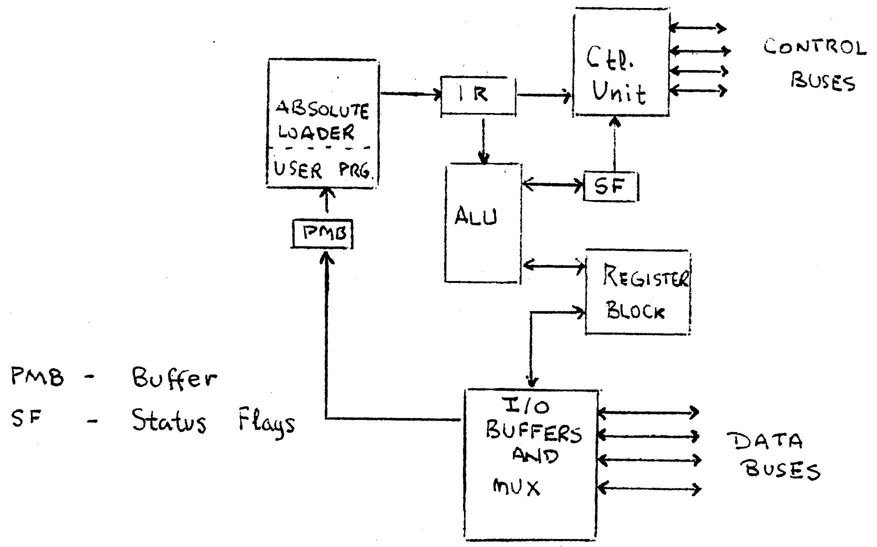

Figure 11. Architecture of an Interior PE.

The entries of A are stored to the left (in columns), while those of B are stored in the memory modules on top (in rows). In general the (i,j)th processor will execute the k-th recursion

$$c_{ij}^{k} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{ik} b_{kj} \ .$$

After N wavefronts, the PEs would each contain one element of the product matrix C.

Central to the development of any data-flow computer is its language. The WAP is no exception. MDFL or matrix data-flow language is the language developed for the WAP. MDFL has two levels of programming as shown below.

1) Global MDFL describes the algorithm from the viewpoint of a wavefront. The perspective of a global MDFL programmer is of one wavefront passing across all the processors.

2) Local MDFL describes the actions of each processing element and the perspective of a programmer at this level is that, of one processor encountering a series of wavefronts.

The instruction set is a reduced one, (RISC) and we can divide the operations into data transfer instructions, recursion oriented instructions, conditional instructions, and internal processor instructions. Of special interest are two constructs described below.

1) Space invariance: the tasks performed by a wavefront in a particular kind of processor must be identical at all (2n-1) fronts.

2) <u>Time invariance</u>: recursions are identical.

The global MDFL provides two repetitive constructs, the <u>space repetitive construct</u>

```
        WHILE WAVEFRONT IN ARRAY DO
          BEGIN <TASK T> END
```

(T is repeated at all fronts); and the <u>time repetitive construct</u>

```
    REPEAT <ONE RECURSION> UNTIL TERMINATED
```

(so that the same recursion is repeated).

Each processor is a hardware interpreter of local MDFL. The architecture of a PE is rather conventional, consisting of an internal program memory, a control unit, an arithmetic-logic unit, and a set of registers. The only exception is the communication with its neighbor PEs. All of the processors can be categorized into 4 classes according to their communication needs (eg. access of the memory modules, etc.). These types are Corner, FirstRow, FirstCol and Interior processors.

One feature that is necessary is a very fast and accurate ALU, required by signal processing. An effort has been made to implement all instructions as one-cycle instructions (which is a characteristic of RISC architectures anyway). Correct execution of the WAP is ensured by a two-way control scheme (handshaking). The other type of asynchronous communication scheme is the one-way control, in which data are sent without waiting for the acknowledgement signal of the receiver. The latter method is safe, as long as large buffers are available.

Two existing WAPs are STC-RSRE and MWAP.
1) The STC-RSRE WAP system, was developed in Britain. This system is reconfigurable for many applications, but is mainly used for adaptive beamforming. In this case, the STC-RSRE system consists of 33 identical PEs 21 of which are organized as a triangular wavefront array, performing the adaptive beamforming function, while the 12 remaining PEs do data correction and other secondary functions.
2) The Memory-linked WAP (MWAP) was developed at Johns Hopkins University. Its performance is very high because of its very advanced ICs. Many MWAPs are connected on a ring network to form a large system. The characteristics of this architecture are its memory addressing structure and the coupling of PEs and memory modules.

Hockney Description

Hockney devised a notation to describe computers in a few lines which will contain the architecture's primary characteristics. For example, C = I [E-M] denotes a simple von Neumann computer that defines the computer C to be a single instruction processing unit I, controlling the units in the brackets. These are a single execution unit E for performing arithmetic, connected by a single data path (-)

to an unbanked memory unit M. The notation is structural and based on a shorthand indicating the number of instruction units, execution units and memory units and the manner of their interconnection and control. An exact mathematical definition of the syntax of the notation exists in Backus normal form (BNF). The interested reader is referred to Hockney (1981) for more details.

Next we describe/summarize Warp and Matrix-1, in the Hockney notation. All of the information available at the moment about these computers is included here. In the case of alternatives in the design we chose the ones that have already been tried out. In a few cases, comments were used to express more clearly the specific point.

(i) Warp:

$C(\mathrm{Warp}) = C1\ [\{C2-,\ -2M_{\{2.5*10^5\}*32}\},\ \texttt{<-->}\ ,2\{C2-\ ,-3\overline{M_{\{2.5*10^5\}*32}},$

$-H\},\ \texttt{<--/-->}\ \{IO-E^{100}\}\ \texttt{<--/}_{\frac{100}{32}}\texttt{-->}\ 10\overline{P}(\mathrm{Warp\ array})]_{ls}\ ;$

$C1 = C(\mathrm{Sun}\ 2/160);$

$C2 = C(\mathrm{Motorola}\ 68020);$

$10\overline{P} = 10\{\ \{M1^{200}_{4K*32}\ ,\ 3M2_{128*32}\ \}\ X\ \{F_S\ (*),\ F_S\ (ALU)\}\ \}^{1-nn}\ ;$

At the end of line two of the description, the subscript "s" stands for the "skewed model of computation".

(ii) Saxpy Matrix-1:

$C(\mathrm{Matrix\text{-}1}) = C1[P-,\ -M^{100}_{128M*32}--,\ -U]_h;$

$P(\mathrm{Matr.\ processor\ subs.}) = I(\mathrm{Matr.\ ctl.\ proc.})[32\overline{E}\ X$

$IO1(\mathrm{Matr.\ \ proc.\ interface})]_{li}\ ;$

$32\bar{E}(\text{zones}) = 32\{M1^{32}_{4k*32}(\text{local mem.})-, -F^{64}_{\rho}(*)-, -F^{64}_{\rho}(\text{ALU})\}^{1-nn}$;

C1(Syst.controller) = C(VAX);

IO1(Matr. proc. interf.) = {4M2(buffers)--H(xbar)};

U(Mass storage syst.) = {IO2(interface) <--> M3(disk, tapes)};

At the end of the description of the matrix processor subsystem the subscript "i" stands for the "independent execution of the zones". At the description of the matrix processor interface the connection of the buffers to H is as follows:

buffer A to H is full duplex;

buffers B and C to H are simplex to H;

and, finally buffer D is simplex from H.

CHAPTER III

SYSTOLIC ALGORITHMS

The Space of Systolic Algorithms

This part of the thesis is concerned with various questions/aspects of systolic algorithms (SAs).
Can any particular algorithm be executed efficiently on a systolic device? Are there any characteristics or properties that a SA should necessarily have? These questions will be investigated.

Parallel architectures, including systolic architectures, are definitely more demanding than serial computers as far as algorithm design is concerned. Careful design of the algorithm is required if we want to exploit the systolic (or any other parallel) architecture as much as possible. There are issues such as:
- synchronizing the processors for correct (and efficient) execution;
- distributing the computation among the available processors;
- rearranging of the data is as necessary;
- partitioning of the problem into subproblems, that can be

solved in the number of PEs available;

- determining the speed of data flow in the array;
- reconfiguring the available architecture; (somebody may ask, which comes first, the algorithm design or the reconfiguring of the hardware?).

We view a parallel algorithm as a collection of independent task modules that can be executed in parallel and that communicate with each other during the execution of the algorithm. Three main properties of parallel algorithms constitute the space of parallel (and systolic as well) algorithms. The dimensions of {Computation Unit (granularity), Communication Patterns and Patterns of Reference to Data} - or - {Concurrency Control, Module Granularity and Communication Geometry} have been proposed. We will consider the latter triple, as proposed by H. T. Kung.

1) Concurrency Control is needed because more than one task module can be executed at a time; we need to ensure/enforce the desired interactions among the modules so that the execution of the algorithm is correct. (Examples of different types of control are simplex/complex local control which can be synchronous/asynchronous, and also centralized/distributed).

2) Module Granularity refers to the maximal amount of the computation a typical module can do before having to

communicate with other modules. The module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive. An algorithm can have small constants, or small or large module granularities. A small granularity means that the modules communicate often with each other; by contrast, a large granularity implies that substantial computation is done within a module, without having to communicate with other modules.

3) Communication Geometry: suppose that the task modules of a parallel algorithm are connected to represent intermodule communication. Then a geometric layout of the resulting network is referred to as the communication geometry of the algorithm. Typical geometries are crossbar, square, linear array, shuffle, hexagonal array, etc.
Each of the dimensions 1), 2), 3) can be represented as a tree, with its leaves giving the possible choices for that particular property of the algorithm.

The SAs have distributed control, achieved by simple local control mechanisms. The control is synchronous with the exception of algorithms for the wavefront array, whose control is asynchronous (data-driven). Task modules of SAs communicate often with each other; thus the granularity is small; furthermore, the module granularity has to be constant. Finally it it desirable that communication geometries be simple and regular. Such structures lead to cheap implementations and high densities of systolic chips. In turn, high density implies both high performance and low overhead for support components. So the task modules of a SA should be simple; their execution should require a small constant amount of time (thus leading to faster arrays) and space.

Utilization rates

One of the most important characteristics of any multiprocessor system, including the systolic processors, is its utilization rate. In other words, do we keep the processors busy (working - not idle) for most of the computation time? A low utilization rate probably implies an inefficient algorithm or SAP design; (it is here where mapping can be very important). Some factors that affect the utilization rate are mentioned below.

1) The size of the SAP (i.e. the number of the PEs). If a problem will never saturate a SAP for instance, thus never reaching a full utilization, then this SAP is obviously too big for this problem.

2) The configuration of the SAP (linear, orthogonal, etc.); obviously an orthogonal array can be more "parallel" in its execution than a linear one. This often minimizes the completion time and thus <u>usually</u> increases the utilization rate. (We must say again, that some algorithms map better onto some configurations than onto others, so this must be

taken into consideration).

3) The size of the problem. The larger the problem size the higher the utilization rate. This occurs because the completion time does not increase too fast, due to the large amount of parallelism and pipelining in the array; on the other hand, we are able to keep the array working saturated for a larger period of time.

4) Timing. If only one data stream enters a SAP, do we have its elements separated by one clock cycle or more - in which case a number of idle cells exists. If multiple streams of data enter a SAP, do all enter the array at the same time or are there relative delays between each stream?

Utilization rates in the area of 80% - 85% can be considered high. It is obvious that the combinations of different layouts of SAPs, sizes, timing, etc., produce a very large number of scenarios. It is true however, that the majority of algorithms for SAPs uses a very small number of all the possible configurations. We will present the analysis and utilization formulas for most of these commonly encountered configurations.

We define the utilization rate as

$$U = \frac{S}{\#PE * T} \quad (1),$$ where

U : is the utilization rate of the SAP.

S : is a sum of active cells during each period.

#PE : is the total number of processing cells in the SAP.

T : is the completion time. This includes loading and draining time.

One can see from the formula above, that the optimum U would be U = 1 or 100%. In this case we would have <u>all</u> of the PEs working <u>all</u> of the time, i.e. U = T / T = 1. Obviously this is not feasible since during the loading and the draining phases of the array, the utilization rate is far below optimum. We can identify three phases in the execution of a SAP.

1) <u>Loading phase</u>: it is the beginning of the execution of the algorithm; during this phase the "filling" of the array takes place. "Filling" does not necessarily mean that all of the PEs are busy; there may be a number of idle cells in the SAP. However, a saturation point exists.

2) "<u>Computing</u>" <u>phase</u>: it is in this phase that the array is utilized in its maximum. The objective of a designer is to keep the SAP working "saturated" as much as possible. (The quotes around the word Computing are there because computing also occurs in 1) and 3), but it is in 2) where the bulk of computation takes place.

3) <u>Draining phase</u>: after the data streams have been exhausted, utilization drops. This is the final phase; no new data enters the array, hence it can not be kept in the saturated state. Due to symmetry of the SAPs, the loading and the draining usually take about the same time.

The assumption made throughout is that there are enough data elements to fill the array completely (phase 2)) for at least one clock cycle.

<u>Linear Array</u>

1) Assume that k is the total number of PEs and n is the number of data elements entering the array, one at each cycle. The completion time is $T = k+(n-1)$. k cycles for the first data element to reach the "output end" of the array and n-1 cycles are needed to output the remaining n-1 elements.

The working PEs for phase 1) are:

$1 + 2 + 3 + \ldots + n-1$.

The working PEs for phase 3) are (note the symmetry):

$n-1 + n-2 + \ldots + 2 + 1$.

Full utilization occurs for n-k+1 cycles.

Therefore - refering to phases 1) and 3) -

$S = 2\sum_{j=1}^{k-1} j = k(k-1)$; (this of course will be divided by the #PE).

We can easily conclude that $U = n / (k+n-1)$. To get an idea for the values of U in this case, for k = 100 we have:

$U(n=200) = 0.668$; $U(n=500) = 0.834$; $U(n=1000) = 0.909$.

2) Another common scenario for linear arrays is that a stream of data of size n enters the array (with a delay of one cycle between its elements), East - bound and k streams of n elements each enter the array South - bound, with each stream having one clock cycle delay <u>relative to its</u>

neighboring streams. The utilization for this configuration is the same as the one we obtained above. (A use of 2) is matrix - vector multiplication Ax = y; dim(A) = nxk; dim(x) = kx1; dim(y) = nx1. y is East - bound, x is permanently stored in the array and A is entered South - bound by columns. Another use of this configuration is for integer addition. To implement this, an additional set of North - bound streams (mirror images of the South - bound streams) is needed. These streams are the numbers to be added while the carry is moving East (to higher order bits).

Orthogonal Array

1) Assume that we have an orthogonal array of size kxk. Furthermore, kxn data elements enter the SAP in the North - South direction and similarly, kxn data elements enter from the East - West direction. Each data stream has n elements and all streams enter the array at the same time (see Fig. 12).

The completion time is T = k+(n-1) as in the case of linear arrays. (When one of the pipelines has finished, all have finished). In the loading phase, elements in the first row and column are working; in the second cycle, we add the first row and column of the square internal to the previous one, etc. Let us list the number of the working cells as the computation progresses.

$$\begin{aligned}&2k-1\\ &+ 2k-1 + 2(k-1)-1\\ &+ 2k-1 + 2(k-1)-1 + 2(k-2)-1\\ &+ \ldots + \qquad\qquad\qquad\qquad\qquad\qquad (2)\\ &+ 2k-1 + \ldots + 3.\end{aligned}$$

The cycle after the (k-1)st (last line), fills the array (phase 2)). Full utilization takes place for n-k+1 cycles. The draining phase, due to symmetry, results in a sum which is the same as that of (2). Hence

$S' = 2((k-1)(2k-1) + (k-2)(2k-3) + (k-3)(2k-5) +\ldots+ 3)$

So the formula becomes

$$U = \frac{S'}{\#PE * T} + \frac{n-k+1}{T} .$$

2) Assume an orthogonal array with kxk PEs. As previously, kxn data elements enter the array in N-S direction and kxn enter in E-W direction. The difference between 1) and 2) is that the streams in the N-S direction enter the array with a relative delay of one clock cycle relative to each other. Similarly, for the ones in the E-W direction.

The completion time is $T = k + (n-1) + (k-1) = 2k+n-2$. The pipeline that has the biggest delay, of k-1 clock cycles, determines the completion time.

The array is filled by its diagonals, hence :

$1 + (1+2) + (1+2+3) + \ldots + (1+2+3+\ldots+k)$,

fills the array up to its main diagonal. The sum

(1+2+3+...+k+(k-1)) + ... + (1+2+3+...+k+...+2)
fills the array beyond the main diagonal. The next cycle (i.e. the one after the last addend in the sum above), utilizes the array fully. This phase (2) lasts for n-2k+2 cycles. We mention the four points that led to this.

1. The number of diagonals is = 2k-1.

2. Full utilization occurs as long as there are elements in the upper right - corner cell which started the earliest.

3. The array needs 2k-1 cycles to fill up (see 1), hence the <u>remaining</u> data elements of the first row (or column) will be n - (2k-1) + 1 = n-2k+2.

4. Points 1, 2, 3, give the number of cycles of full utilization. The draining phase, due to symmetry gives rise to a sum that is the same as in the loading phase.

Further work gives us:

$$S' = \sum_{j=1}^{k}(j+1)j + (k-2)(k+1)k + 2\sum_{j=2}^{k-1}(k-j)(k-j+1)$$

and finally

$$U = \frac{S'}{\#PE * T} + \frac{n-2k+2}{T} .$$

Here too, the assumption has been made that the array operates saturated for at least one clock cycle, i.e. n-2k+2 >= 1 or n >= 2k-1. (The number of elements in each of the data streams is greater than or equal to the number of the diagonals in the array).

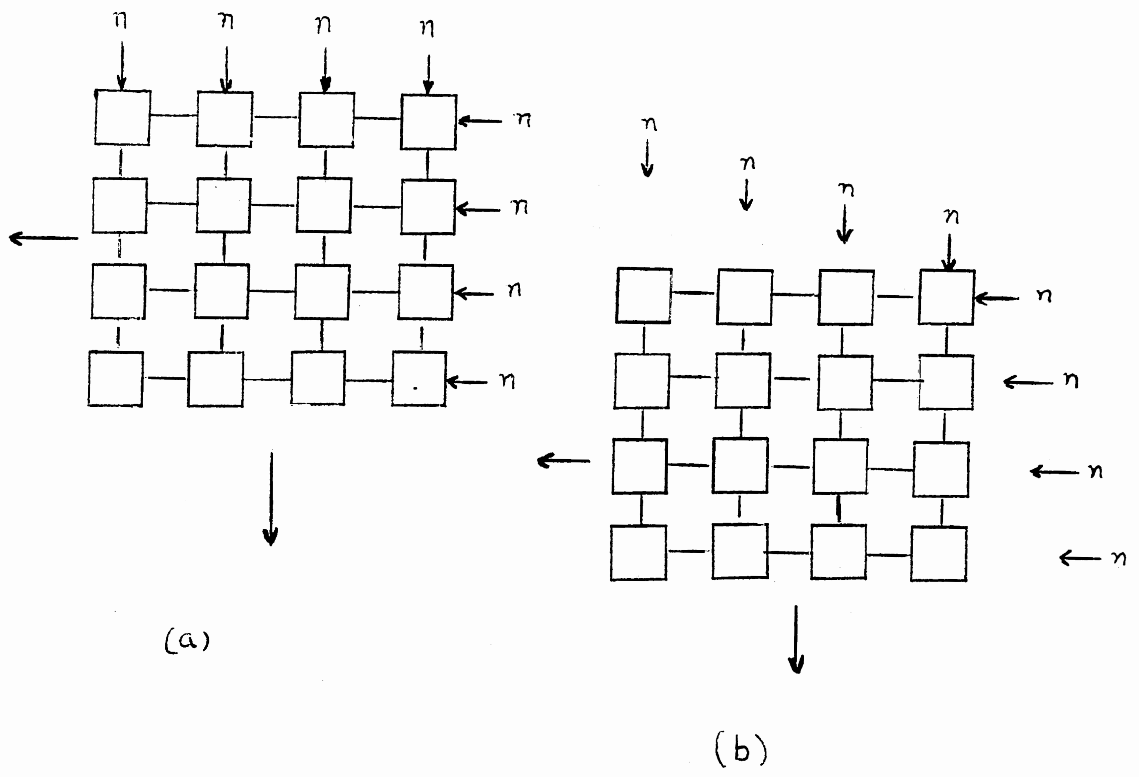

Figure 12. Orthogonal arrays with two types of input streams.

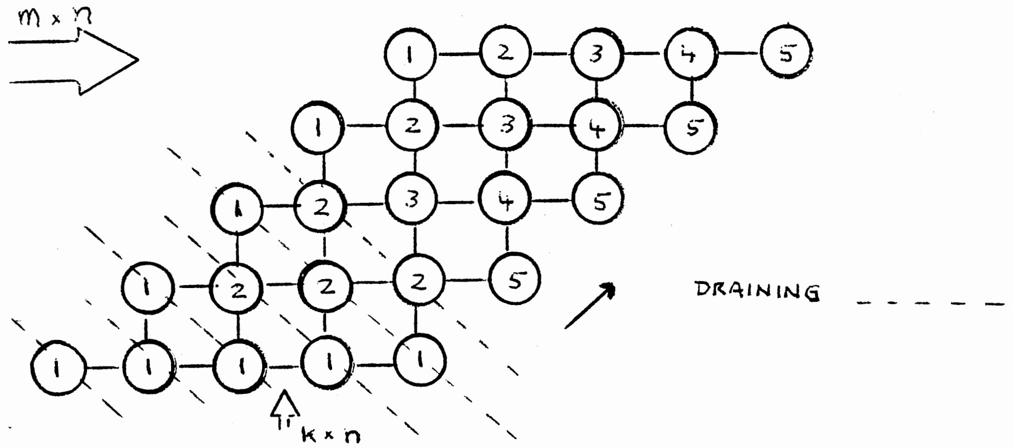

Figure 13. Band systolic array with k odd.

Band Array

We call this the band array, because its structure resembles the elements of a banded matrix. This type of array is used often in manipulating matrices, especially in reordering its elements, or transposing elements, etc.
We assume that we have m rows of PEs and each row has k PEs, with m >= k. m data streams of n data elements each flow in the West - East direction; k streams of n elements each move in the South - North direction. No relative delays exist in either case.
The structure of the cell is described as follows: each cell receives two inputs (from the South and the West directions); it performs a computation which may involve either or both of the inputs and routes the result to the North and to the East directions. Usually these types of arrays associate one of the directions of data flow with control; that is, streams of control bits are pumped through the array, thus controlling the actions of each cell. So only if input from the West exists, will we have out - routing to two directions; else North - bound control bits just pass through the cells. Two cases are considered.

1) _k is even_. First we examine the number of cells that are working during each cycle.
- 1st cycle: bottom row of of PEs and leftmost PEs of each row, all work. So m+k-1 cells work.
- 2nd cycle: the cells of cycle 1, plus the second cell in

each row (m-1) plus the cells of the second row from bottom, excluding some cells already counted and one that is not working. This gives us (k-3) cells. So, a total of (m-1)+(k-3) additional working cells are busy this cycle. Let t_i denote the new (additional) working cells at cycle i, during the loading phase. Careful work reveals the following terms.

$t_1 = m+(k-1)$

$t_2 = (m-1)+(k-3)$

$\vdots$

$t_{k/2} = m-(k/2 -1) + k-(k-1)$

$t_{k/2+1} = m- k/2$

$t_{k/2+2} = m- (k/2 +1) + 2$

$t_{k/2+3} = m- (k/2 +2) + 4$

$\vdots$

$t_{k-1} = m- (k-2) + 2((k-2) - k/2)$

The array is filled during the k-th cycle. The last addend, depicted above is the one that takes place at the (k-1)-th cycle.

From the above we conclude that the sum of working cells during the loading phase Sl, is given by

$Sl = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) +...+ (t_1 +...+ t_{k-1})$.

The full utilization lasts for Sc = n-k+1 cycles.

The pattern of the draining of the array is quite <u>different</u>

from that of the loading phase. It is easy to see that this SAP is drained by its diagonals! Thus, at each <u>cycle of the draining phase</u>, we simply "cross - out" the PEs along a diagonal, starting of course at the bottom leftmost cell. It turns out that the diagonal(s) containing the largest number of PEs, for k even, contain k/2 elements. The number of the diagonals in any array of dimensions pxq is p+q-1; in our case we have m+(m+k-1)-1 = 2m+k-2 diagonals. Furthermore, there exists a pattern in the number of elements (PEs) each diagonal contains; it is
1, 1, 2, 2,..., k/2 -1, k/2 -1, k/2 (main diagonal; this occurs 2m-k+2 times), k/2 -1, k/2 -1,..., 1, 1.
The above holds for k >= 4, k even. In the case where k = 2 - a trivial situation -, the draining is done one cell at a time. Hence, the following sum of working cells is obvious now.

```
Sd =  mk-1 +
    + mk-(1+1) +
    + mk-(1+1+2) +...
 ...+ mk-(1+1+2+...+ k/2 -1 + k/2 -1) +
    + mk-(1+...+ k/2) +...
 ...+ mk-(1+...+ k/2 (2m-k+2) ) +...
    + mk-(1+...+ k/2 (2m-k+2) +...+1 ).
```

2) <u>k is odd</u>. The thinking is similar to the one presented in case 1. For the loading phase, let t_i denote the additional (new) working cells at cycle i. Careful work

reveals the following terms.

$t_1 = m+(k-1)$

$t_2 = (m-1)+(k-3)$

$\vdots$

$t_{\lfloor k/2 \rfloor+1} = (m-\lfloor k/2 \rfloor)+(k-k)$

$t_{\lfloor k/2 \rfloor+2} = (m-(\lfloor k/2 \rfloor+1))+1$

$\vdots$

$t_{k-1} = (m-(k-2) + ((k-3)-\lfloor k/2 \rfloor)2 +1.$

From the above we conclude that the sum of active cells during the loading phase is

$Sl = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) +\ldots+ (t_1 +\ldots+ t_{k-1})$.

The formula for the sum above, holds for $k > 3$, k odd. In the case where $k = 3$, the sum becomes for the loading phase

$Sl = m+(k-1) + m+(k-1) + m-1 = 3m+2k-3$.

The full utilization lasts for $Sc = n-k+1$ cycles.

The calculation for the draining phase is similar to that of 1. The difference is that now, we have two main diagonals; the one has length (PEs that is) $\lceil k/2 \rceil$ and the other $\lfloor k/2 \rfloor$. They occur alternately, starting and ending with the largest diagonal. The largest of the main diagonals occurs $\lceil(\#\text{diag.} - 4\lfloor k/2 \rfloor)/2\rceil$ times while the smaller one occurs $\lfloor(\#\text{diag.} - 4\lfloor k/2 \rfloor)/2\rfloor$ times. (Actually this one occurs four times more, but for symmetry reasons we do not count it as a diagonal then). Of course #diagonals = $2m+k-2$ as we have previously shown. Thus the sum of the working cells becomes

$$\begin{aligned} Sd = {} & mk-1 + \\ & + mk-(1+1) + \dots \\ \dots & + mk-(1+1+\dots+\lfloor k/2 \rfloor + \lfloor k/2 \rfloor) + \\ & + mk-(1+1+\dots+\lfloor k/2 \rfloor + \lceil k/2 \rceil) + \dots \\ \dots & + mk-(1+1+\dots+\lfloor k/2 \rfloor + \lceil k/2 \rceil + \dots + 2+1). \end{aligned}$$

The above holds for m >= k, k >= 3. As far as completion time goes, it can be divided in three parts.

Time for loading Tl = k-1 cycles.

Full utilization time Tc = n-k+1 cycles.

Time for draining Td = 2m+k-3 (= #diag. -1).

Hence completion time T = Tl + Tc + Td.

From the above we can easily see, for both cases, that

$$U = \frac{Sl + Sd}{T * mk} + \frac{Sc}{T} .$$

Algorithms for systolic architectures have been devised for almost all scientific fields. Indicative, and by no means complete, is the following list of applications/algorithms implemented as systolic: signal processing (FIR filter, Fourier transforms, convolutions, etc.); matrix operations (multiplication, LU decomposition, QR factorization etc.); relational database operations, data structure problems (sorting, queues, graph algorithms, etc.), pattern matching, recurrence evaluation, implementation of arithmetic units, computational geometry.

TABLE I

UTILIZATION RATES FOR LINEAR SYSTOLIC ARRAYS

Total #Cells	Data Elements/Stream	Utilization (%)
10	10	52.63
10	25	73.53
10	35	79.55
10	50	84.75
10	60	86.96
10	70	88.61
10	80	89.89
10	90	90.91
10	100	91.74
10	200	95.69
10	300	97.09
10	400	97.80
10	500	98.23
10	800	98.89
10	1000	99.11
100	100	50.25
100	200	66.89
100	300	75.19
100	400	80.16
100	500	83.47
100	600	85.84
100	700	87.61
100	800	88.99
100	900	90.09
100	1000	90.99
100	1500	93.81
100	2000	95.28
100	2500	96.19
100	3000	96.81
100	4000	97.58
100	5000	98.06
250	250	50.10
250	300	54.64
250	400	61.63
250	500	66.76
250	600	70.67
250	700	73.76
250	1000	80.06
250	2000	88.93

TABLE II

UTILIZATION RATES FOR ORTHOGONAL SYSTOLIC ARRAYS

Total #Cells	Data Elements per Stream	Utilization rate (%) Case (1)	Case (2)
25	10	82.86	55.56
25	25	91.72	75.76
25	50	95.56	86.21
25	75	96.96	90.36
25	100	97.69	92.59
100	20	80.34	52.63
100	30	85.38	62.50
100	40	88.37	68.97
100	50	90.34	73.53
100	60	91.74	76.92
100	70	92.78	79.55
100	80	93.60	81.63
100	90	94.24	83.33
100	100	94.77	84.75
100	120	95.58	86.96
100	140	96.17	88.61
100	160	96.63	89.89
100	180	96.98	90.91
100	200	97.27	91.74
100	220	97.51	92.44
100	240	97.71	93.02
100	260	97.88	93.53
100	280	98.03	93.96
100	300	98.16	94.34
100	400	98.61	95.69
100	500	98.88	96.53
100	600	99.06	97.09
100	700	99.20	97.49
100	800	99.30	97.80
100	900	99.37	98.04
100	1000	99.44	98.23
100	3000	99.81	99.40
400	40	79.07	51.28
400	80	87.53	67.80
400	100	89.62	72.46
400	500	97.62	92.94
400	1000	98.79	96.34

CHAPTER IV

FORMAL APPROACHES TO OBTAINING SYSTOLIC ARRAYS

Mapping and Partitioning

Even though there are systolic systems now in operation, little work has been done in devising methodologies to design systolic arrays that are optimal for a large class of problems. This is referred to as the mapping problem. In order to match best the characteristics of algorithms with those 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 examine issues ranging from the organization of the network of cells to the detailed operation of the cells. In fact, the mapping is nothing less than the design of the VLSI array, according to the properties of the SA and a set of design goals.

The Mapping Problem

Given a class of algorithms with certain characteristics, obtain a set of constraints which reduces the possible systolic architectures to a set, from which, optimal design(s) can be found.

The other problem associated with SAPs and SAs is the partitioning problem. Most of the SAs that are designed assume the existence of a SAP with the required number of cells available. Unfortunately, the situation in many practical cases is that the interconnection topology and the number of PEs are fixed. This implies that some transformations of the original data structures are needed.

The Partitioning Problem

Consider an algorithm and a fixed size SAP. If the size of the problem is larger than the SAP can handle, then partition the original problem so that the transformed algorithm can be executed on the available SAP. (Size of a problem can mean the number of nested loops, or, the number of rows of a matrix, etc. The size of a SAP, on the other hand, is its number of PEs).

Below are the basic issues and points for the MAPPING and the PARTITIONING methods to be efficient and correct.

1. The mapping procedure should involve classes of algorithms that are as broad as possible.

2. The parameters/constraints on which the mapping is based should be complete; by complete, we mean that it must examine most of the aspects of a systolic execution, such as intervention from the host, I/O, control and data dependencies, etc.

3. From the mapping procedure, we should obtain most of the basic features of the proposed array, such as type of interconnection of PEs, type of operation for each PE, timing of the whole array, size of the array for a problem of given size, etc.

4. The partitioning techniques should apply to a large class of problems.

5. The partitioning must have data transformations with low generation difficulties, which do not require any increase in the complexity of the PEs.

6. The constraints imposed by the partitioning on the size of the problem and on the size of the SAP must be minimal (i.e. we must have a flexible/adaptable partitioning, so that for a variety of given sizes of SAPs, we can obtain transformations that allow the execution of a SA).

7. The transformed problem should be equivalent to the original one, i.e. the set of solutions is correct and complete.

8. The computation time of a partitioned algorithm is proportional only to the product of the number of partitions

and the time to process one partition. In other words, no additional delays caused by the partitioning process are allowed.

9. The amount of overhead in external hardware and external communication caused by partitioning is as small as possible.

We will now present the two most interesting and methodical procedures that can be used for mapping onto systolic arrays. They are the parameter method and the method using linear transformations (dependency method).

The Parameter Method

This method is based on the work done by G. J. Li and B. W. Wah. The systolic arrays are characterized by three classes of parameters: the velocities of data flows, the spatial distributions of data and the periods of computation. By relating these parameters, in constraint equations that govern the correctness of the design, the design is formulated into an optimization problem. The size of the search space is a polynomial of the problem size, and a methodology to search and reduce this space systematically and to obtain the optimal design is proposed.

Thus, a systematic methodology for the design of optimal pure planar systolic arrays is proposed.
A systolic array that does not have broadcast (global) buses and implements the algorithm in pipelines extending in

different directions is called pure. (By contrast, a semi-systolic array uses global communications which can be faster, but introduces problems as the number of cells increases).

Planar systolic arrays are those in which the interconnections can be laid out in a plane without crossing each other.

Further restrictions are the following: the method works for linear recurrence processes; the inputs must be one or two dimensional and inputs with a larger number of dimensions have to be partitioned first. Finally, for a two dimensional array X used as input or output of a SAP, the elements along a row or column are arranged in a straight line and are equally spaced as they pass through the systolic array; their relative positions are iteration independent. No other forms of data distributions are considered.

Linear recurrences for the computation of a two dimensional result Z from two two dimensional inputs X and Y can be expressed as

$z_{i,j}^{k} = f(z_{i,j}^{k-\delta}, x(i,k), y(k,j))$, δ=1 or -1, where f is a function to be executed by a PE and k is a positive integer bounded by a linear function of i, j and the problem size. We will use only backward recurrences. That is, z^{k} is defined in terms of z^{k-1}. (The opposite can be true too, and this is called forward recurrence). In designing systolic algorithms, both types have the same result; what matters, is only the order of evaluation of the variables involved in the computation; this may affect the complexity of the resulting design.

The parameters for the mapping.

Some assumptions are needed for the model of the systolic array onto which we expect to execute the algorithm. Furthermore, the theory is built on a basis which includes assumptions regarding the distance of the PEs and the time unit. The SAP consists of a mesh of interconnected PEs operating in synchrony. As far as timing goes, a clock cycle is a unit of time during which one iterative operation is computed in a PE, and data advance into neighboring PEs or buffers. We assume that we may have buffers, equally spaced between PEs. Each PE or buffer delays the data flow by one clock cycle. Furthermore, the distance between two directly connected PEs is defined to be unity. The three parameters are defined below.

1) Velocity of data flow. The velocity of a datum x is defined as the directional distance passed by x during a clock cycle and is denoted by $\vec{x}_d$. The magnitude of $\vec{x}_d$ is a rational number i/j, where i, j are integers i<=j. This means that in j clock cycles, x has propagated through i PEs and j-i buffers.

2) Data distribution. Suppose, the row and column indexes of an input or output two dimensional array X are i and j, respectively. The row displacement of X is defined as the directional distance between $x_{i,j}$ and $x_{i+1,j}$ as X passes through the systolic array and is denoted by $\vec{x}_{is}$. Similarly a column displacement is defined and it is denoted by $\vec{x}_{js}$. If X is a one dimensional array, the index in accessing X is implied and we simply have the item displacement of X ($\vec{x}_s$), which is the distance between x_i and x_{i+1}. (Remember, that one of the assumptions made, was that the elements along a row or a column are equally spaced, so the row and column displacements are independent of the values of i, j).

3) Period. This parameter is a scalar; two time functions are needed. $\underline{T_c}$ is the time at which a computation is performed, and $\underline{T_a}$ is the time at which an input is accessed for a particular computation. The following periods, concerning systolic execution can now be defined. The periods of i and j for two dimensional outputs are:

$t_i = T_c\,(z^{k}_{i+1,j}) - T_c\,(z^{k}_{i,j})$ and

$t_j = T_c\,(z^{k}_{i,j+1}) - T\,(z^{k}_{i,j})$.

The period of iterative computation for two dimensional outputs is

$t_k = T_c\,(z^{k+1}_{i,j}) - T_c\,(z^{k}_{i,j})$. Note that t_k is always positive because the recurrence is expressed in backward form. In computing $z_{i,j}$ items, $x_{i,k}$ and $x_{i,k+1}$ are accessed sequentially and so are $y_{k,j}$ and $y_{k+1,j}$. (Because of the general formula). Define the periods of X and Y with respect to k, in the computation of $z_{i,j}$, as the time between accessing successive elements of X and Y.

Thus, $t_{kx} = T_\alpha(x_{i,k+1}) - T_\alpha(x_{i,k})$ and

$t_{ky} = T_\alpha(y_{k+1,j}) - T_\alpha(y_{k,j})$.

t_{kx} and t_{ky} may be negative depending on the order of access defined in the <u>subscript</u> <u>access</u> <u>functions</u> $\underline{x}(\underline{i},\underline{k})$ <u>and</u> $\underline{y}(\underline{k},\underline{j})$. (Note: do not confuse x(i,k) with $x_{i,k}$. The first one mentioned is a function giving a subscript, while the latter is an element of matrix X).

Note that the computations in a SAP are periodic and hence all the periods are independent of i, j, k. There is a total of 13 parameters for two dimensional linear recurrences, of which 3 are for the velocities of data flow, $\vec{x_d}$, $\vec{y_d}$, $\vec{z_d}$, 6 are for data distributions $\vec{x_{is}}$, $\vec{x_{js}}$, $\vec{y_{is}}$, $\vec{y_{js}}$, $\vec{z_{is}}$, $\vec{z_{js}}$ and 4 are for the periods t_{kx} , t_{ky} , t_i , t_j . For one dimensional problems, only 9 parameters exist: $\vec{x_d}$, $\vec{y_d}$, $\vec{z_d}$, $\vec{x_s}$, $\vec{y_s}$, $\vec{z_s}$, t_{kx} , t_{ky} , t_i . The following theorem states the relationships among these parameters; (it actually describes the fundamental space - time relationships in systolic processing). The relationships that follow, form the basis on which the mapping is done; they derive the speed and direction of data flow, the data distribution, etc. In addition to these, another set of "core" equations (7-14) exists which basically optimizes the objective function(s).

Theorem of Systolic Processing. Suppose a two dimensional recurrence computation

$$z_{i,j}^{k} = f(z_{i,j}^{k-1}, x(i,k), y(k,j))$$

is implemented in a SAP; then the velocities, data distributions and periods must satisfy the following vector equations:

(data movement for X, Z between computing $z_{i,j}^{k-1}$, $z_{i,j}^{k}$);

$$t_{kx}\vec{x}_d + \vec{x}_{ks} = t_{kx}\vec{z}_d \qquad (1)$$

(data movement for Y, Z between computing $z_{i,j}^{k-1}$, $z_{i,j}^{k}$);

$$t_{ky}\vec{y}_d + \vec{y}_{ks} = t_{ky}\vec{z}_d \qquad (2)$$

(data movement for X, Y between computing $z_{i,j}^{k}$, $z_{i+1,j}^{k}$);

$$t_i\vec{x}_d + \vec{x}_{is} = t_i\vec{y}_d \qquad (3)$$

(data movement for Y, Z between computing $z_{i,j}^{k}$, $z_{i+1,j}^{k}$);

$$t_i\vec{z}_d + \vec{z}_{is} = t_i\vec{y}_d \qquad (4)$$

(data movement for X, Y between computing $z_{i,j}^{k}$, $z_{i,j+1}^{k}$);

$$t_j\vec{y}_d + \vec{y}_{js} = t_j\vec{x}_d \qquad (5)$$

(data movement for X, Z between computing $z_{i,j}^{k}$, $z_{i,j+1}^{k}$);

$$t_j\vec{z}_d + \vec{z}_{js} = t_j\vec{x}_d \qquad (6).$$

For one dimensional problems, only (1)-(4) are necessary. Proof: see Appendix A.

Before we can look at how to minimize an objective function we need to define and discuss a few terms. (Note: it has been stated earlier, that a variety of objective functions exist. The choice of one, depends on the design requirements, the problem size, etc.). The following are needed to further enhance the mapping procedure with constraints and ways to determine the number of processing cells. First we will define the number of streams of data flow of an input/output matrix, (1). This is then used in (2) to determine the number of PEs in the SAP. Finally, more constraints are introduced for the design in (3).

- (1) Consider a matrix X; the <u>number</u> <u>of</u> <u>streams</u> <u>of</u> <u>data</u> <u>flow</u> of X, in the direction of data flow, is defined as the number of distinct lines that must be drawn in parallel to the direction of data flow, so that each element of the matrix, lies in exactly one line. For a one dimensional matrix X with n elements, the number of streams can be one (serial input), or n (parallel input). For a two dimensional n-by-n matrix, this number depends on the directions of $\vec{x}_{is}$, $\vec{x}_{js}$; if they are in the same or opposite directions, then the number of streams can be one (serial input), or n x n (parallel input - this is the extreme case where each element lies in a different stream).

In general, if $\vec{x}_{is}$, $\vec{x}_{js}$ are in different directions, then the number of streams is given by
n + (n - 1)j, where 0 <= j <= n.
(eg. when j=0 then, each row or column of a matrix lies in one stream and so, the number of streams is n).

- (2) <u>#PE</u> (<u>the</u> <u>number</u> <u>of</u> <u>cells</u>) depends on the directions in which the inputs are moving. There are four possible cases.

First, one of the input or output matrices remains in the systolic array (is static) and the others move. Then #PE is given by the size of the stationary matrix (when, of course, all of its elements are used).

Second, both input and output matrices are moving in the same or opposite directions. Then, #PE = (min. number of streams of data flow) x (the distance traveled between the time that the first elements of the input matrices meet and the time that the last elements of the input matrices meet).

Third, is the case where there are two independent directions of data flow (involving input or output matrices). If the two input matrices are flowing in the same or opposite directions and the output matrix is flowing in a different direction, #PE is given by the number of streams of data flow of the output matrix. If the two input matrices are flowing in different directions and the output matrix is flowing in a direction of one of the inputs and if each stream of data flow in an input matrix has to interact with every other stream of data flow in the other input matrix, #PE = (number of streams of data flow of input matrix-1) x (number of streams of data flow of input matrix-2). If the interaction of the input streams is not complete, #PE is as above, reduced by a term, determined from the recurrence.

Fourth, if there are three independent directions of data flow, #PE for the two input matrices can be computed as above; however, this number can be further reduced by the flow of the output matrix.

- (3) In addition to the equations of Theorem 1, the minimization of the objective function(s) is subject to the following:

$$1/t_{jmax} <= |\vec{x}_d| <= 1 \quad \text{or} \quad |\vec{x}_d| = 0 \qquad (7)$$

$$1/t_{imax} <= |\vec{y}_d| <= 1 \quad \text{or} \quad |\vec{y}_d| = 0 \qquad (8)$$

$$1/t_{kmax} <= |\vec{z}_d| <= 1 \quad \text{or} \quad |\vec{z}_d| = 0 \qquad (9)$$

$$1 <= |t_i| <= t_{imax}; \quad 1 <= |t_j| <= t_{jmax};$$

$$1 <= |t_k| <= t_{kmax} \qquad (10)$$

$$|t_k|\,|\vec{z}_d| = k_1 <= t_{kmax}; \quad |t_i|\,|\vec{y}_d| = k_2 <= t_{imax};$$

$$|t_j|\,|\vec{x}_d| = k_3 <= t_{jmax} \qquad (11)$$

$$|\vec{x}_{is}| \neq 0; \quad |\vec{x}_{ks}| \neq 0; \quad |\vec{y}_{ks}| \neq 0 \qquad (12)$$

$$|\vec{y}_{js}| \neq 0; \quad |\vec{z}_{is}| \neq 0; \quad |\vec{z}_{js}| \neq 0 \qquad (13)$$

$$t_k = |t_{kx}| = |t_{ky}| \qquad (14)$$

Recurrence determines the relative signs of (14). The signs depend on the order of access of the elements of X and Y. k_1 , k_2, k_3, t_{kmax}, t_{imax}, t_{jmax}, are integers. All other parameters are rational numbers. Moreover, t_{kmax}, t_{imax}, and t_{jmax}, are functions of the problem size (i.e. depend on k) and T_{serial}. T_{serial} , is the number of times the function f in

the recurrence has to be executed in order to compute all the required results. k_1, k_2, k_3, in (11), represent the distances traversed between computations. Since a computation must be performed in a PE, the distance traversed must coincide with the location of PEs. Their upper bounds are the maximum values of t_k, t_i, t_j, because the maximum values of speeds are 1 (see (7), (8), (9)). This is true because the maximum value of speed is obviously, travelling from a PE to its neighbor PE - distance of 1 -, during one cycle. Thus, no intermediate buffers exist. The lower bounds of (7), (8), (9) are obtained as it is described below. The fraction has its minimum value when the denominator is maximum, that is t_{jmax} or t_{imax} or t_{kmax} and the nominator indicates the unity distance. That is, the slowest possible speed is travelling from PE to PE in the maximum amount of time. When speed = 0 then this means that the data is static.
The total computation time is a function of t_k, $|t_i|$, $|t_j|$. In order for systolic processing to be more efficient than serial computation, $T <= T_{serial}$ must be true. If in this inequality we use the minimum values for two of the periods ($t_k = 1$, $|t_i| = 1$, $|t_j| = 1$), the upper bound for the other period (t_{kmax}, t_{imax}, t_{jmax}) is found. The constraints in (10) follow from their definitions.

Based on the above, we can obtain a systolic design, which minimizes the objective functions #PE x T^2 or T. At least one complete/detailed example will be presented, illustrating all of the definitions as well as minimization of both objective functions.

A complete example - the Minimization procedure

The (ever-present) two-dimensional matrix multiplication algorithm, will illustrate derivation and definitions of the data distribution vectors, the periods, etc. and the minimization of two types of objective functions. The multiplication C = A x B can be expressed by the recurrence:

$$c^{0}_{i,j} = 0 \qquad 1 <= i,j <= n$$
$$c^{k}_{i,j} = c^{k-1}_{i,j} + a_{i,k}\ b_{k,j} \qquad 1 <= i,j,k <= n$$

This is a backward recurrence since the k-th term is computed in terms of the (k-1)-th term. The data distribution vectors of A are defined by $\vec{a}_{is}$ and $\vec{a}_{ks}$. This is so because A is referenced by indexes i, k. The data distribution vectors of B, C are analogous. The periods of A and B with respect to k (t_{ka}, t_{kb}) are 1, because $a_{i,k}$ is accessed one cycle before $a_{i,k+1}$ and so are $b_{k,j}$,$b_{k+1,j}$.

The elements of A, B are accessed in this order, because we have assummed that the recurrence is in backward form. This not only means that the (k+1)-th output element $c_{i,j}$ is computed by the k-th $c_{i,j}$, but also that the subscript functions for A, B are of the form a(i, k+1), b(k+1, j)

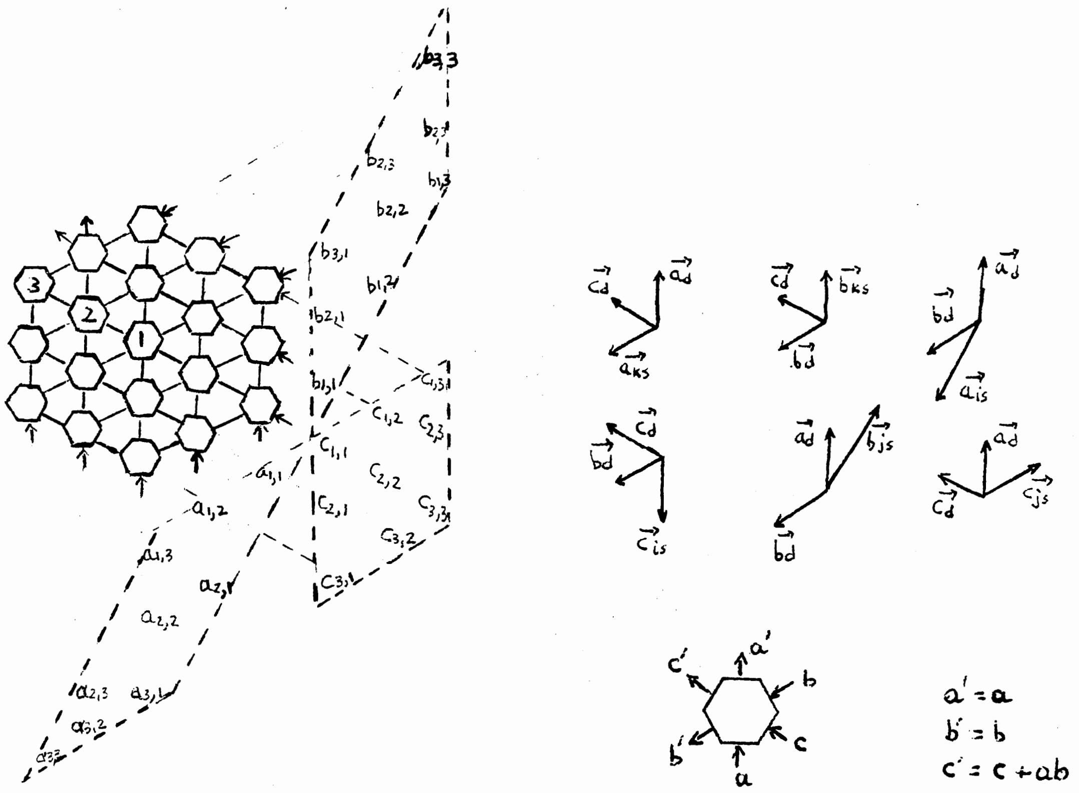

Figure 14. SAP for 2-dimensional matrix multiplication that minimizes T.

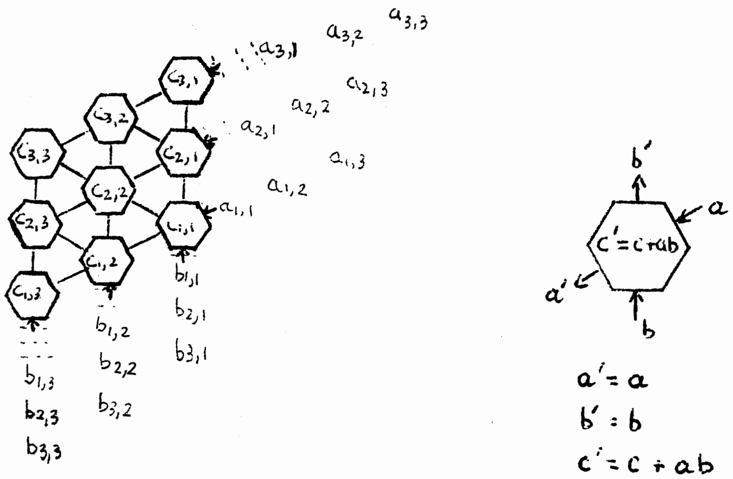

Figure 15. SAP for 2-dimensional matrix multiplication that minimizes PE x T^2.

which in this case simply give i, k+1 and k+1, j as subscripts. Thus we obtain the order of access as described. The periods of i, j for the output C are $t_i = t_j = 1$. For example: $c_{2,1}$ is fully computed at cycle 4, while $c_{3,1}$ is at cycle 5, thus $t_i = 5-4 = 1$. It can also be seen that the number of streams of data flow, for each of the matrices A, B, C, is 5. (Thus, in the formula, $n + (n-1)j$, $n = 3$, and $j = 1$).

Streams of data flow. There are 3 independent streams of data flow; this means that #PE will be the product of the number of streams of data flow for the input matrices A, B i.e. 25. The output matrix C, flows in a different direction, so we know that the #PE can be further reduced. Truly, cutting off two corners of 3 cells each give us the hexagonal 19-cell SAP; we did this by examination - these cells simply did not perform any useful computation.

Completion time. Let us now examine the completion time of this algorithm's implementation; furthermore, using the completion time we will obtain upper bounds for the periods.

The T_{serial} is the time needed for the serial execution. C has 3 x 3 elements; 3 recurrences are needed for each element in C, so $T_{serial} = 27$. Execution on the SAP, however, requires

$T = nt_K + (n-1)|t_i| + (n-1)|t_j|$. nt_K steps are needed to compute $c_{1,1}$; $(n-1)|t_i|$ steps are required for computing $c_{1,1}$ to $c_{n,1}$, and $(n-1)|t_j|$ for computing $c_{n,1}$ to $c_{n,n}$. For the example under consideration, n = 3 and assuming that all the periods are 1 we obtain T = 7. This is easily verifiable; the last element to be computed is $c_{3,3}$; this element is separated by one clock cycle from other elements in its data stream. Eventually it reaches cell 1 where it computes its first recurrence at time 5. At time 7, it has been fully computed at cell number 3. As we said earlier, if in T we substitute the minimum for two of the periods we can obtain the maximum for the other period. (The minimum for any of the periods is 1). Proceeding in this manner we get $t_{imax} = t_{jmax} = 11$ and $t_{kmax} = 8$.

Search space complexity. Next we will obtain the complexity of the search space for the method. The number of buffers b among the PEs is what regulates the values of speeds and periods of the SAP. Specifically, for t_k, $|\vec{z}_d|$, k_1 and t_{kmax} , k_1 represents the number of PEs ("distance" = time x velocity, see (11)) traversed by a datum between two successive iterative computations; its maximum is t_{kmax}. Let p be the maximum number of iterations required for computing a result. For a given k_1 , the maximum number of PEs in the pipeline is $(p-1)k_1+1$ PEs, (i.e. remaining iterations x speed). Then it is obvious that the number of buffers in this pipeline, satisfies:

$0 <= b <= ((p-1)t_{kmax} + 1) - ((p-1)k_1 + 1) = p((t_{kmax} - k_1)$.

(Let this be inequality (15)). For the last part of (15) we have used that $k_1 <= t_{kmax}$, see (11).

Once <u>b</u> <u>is</u> <u>chosen</u>, $|\vec{z}_d|$ and $|t_k|$ can be determined. From (10), (11), (15) we obtain:

$|\vec{z}_d| = ((p-1)k_1) / ((p-1)k_1 + b)$.

From the definition of k and the above relation we obtain:

$|t_k| = k_1 / |\vec{z}_d| = k_1 + b / (p-1)$.

As a result, there are $O(pt^2_{kmax})$ combinations of values of t_k and $|\vec{z}_d|$; this is an immediate result of (15) and (11). A reasoning that is absolutely similar to the above, concerning t_i and $|\vec{y}_d|$, and t_j and $|\vec{x}_d|$ gives us $O(pt^2_{imax})$ and $O(pt^2_{jmax})$ combinations of values, respectively.

<u>Reduction</u> <u>of</u> <u>search</u> <u>space</u> <u>complexity</u>. The optimization of design of a systolic array for a given recurrence has a finite search space of complexity $O(p^3 t^2_{imax} t^2_{jmax} t^2_{kmax})$. This complexity is quite large, therefore we need to reduce it. There are two ways to do this.

- (1) Instead of requiring that $T <= T_{serial}$, use relation $T <= O(T_{serial} / \#PE)$, which is a reasonable assumption for a SAP. This reduces the search complexity.

- (2) The equations of Theorem 1 indicate that correctness of design is independent of problem size. So to reduce the search complexity, an optimal design for a smaller problem can be found. This in turn, is used to extend the systolic design for a larger version of the same problem. Note that this method does not necessarily lead to an optimal design; this is true because the objective function is monotonically increasing with the problem size, i.e., if a design is best for a problem of a given size, it does not mean that it is going to be optimal for a problem of another size.

We will elaborate some more on the minimization procedure for two objective functions. First in 1 and 2 we describe the overall procedure for the optimization of each of the objective functions. Following that, the specific actions for our example are described.

1. Minimize the completion time T. We can identify seven steps.

(a) Determine the different directions of data flows (one out of possible five).

(b) Find the maximum values of t_k, t_i, t_j.

(c) Select from the set of possible values, a subset of t_k, t_i, and t_j that minimizes the completion time.

(d) The speeds of data flow are evaluated from (11) by using ($k_1 = k_2 = k_3 = 1$ initially) the values assigned to k_1, k_2 and k_3 - see step (g) below.

(e) If no feasible solution is found, repeat the procedure by finding another set of periods t_k, t_i, t_j so that the completion time T is increased by the least amount. (Thus we go back to step (c)).

(f) Repeat steps (b) - (e) for all 5 data flow directions!

(g) If still no feasible solution is found then, increase by 1 one of k_1, k_2, k_3 and repeat the whole procedure.
The first feasible solution found, is the optimal solution that minimizes the completion time.

2. Minimize #PE x T^2. First we need to know the lower bound on #PE. For linear recurrences with two-dimensional (n-by-n) inputs, the lower bound on #PE can be 1 (both inputs serial), or n (one input is serial and the other has n streams of data flow), or n^2 (both inputs have a degree of parallelism - n streams of data flow). Serial inputs usually do not lead to feasible solutions - so the lower bound is n^2.
Then we repeat the procedure 1 as described above. This leads to a feasible solution. Assume that we have found a design which requires T_1 clock cycles to complete and P_1 PEs. Then we can easily see that any design with #PE = n^2 (minimum) and a completion time T_2 , such that $n^2 T_2^2 < P_1 T_1^2$ is better. Thus, an upper bound on the completion time is obtained:
$T_2 >= \sqrt{P}\ T_1/n$; T_2 will NOT lead to a better solution. So the search is continued to find better solutions with completion time between T_1 and T_2.

Methods 1. and 2. are illustrated using the example used earlier. The computation time needed for the 3-by-3 matrix multiplication is

$T = 3t_k + (3-1)|t_i| + (3-1)|t_j|$. Obviously, it is minimized when t_k, $|t_i|$, $|t_j|$ are as small as possible. We start the search with $t_k = t_i = t_j = 1$, on all combinations of directions of data flows. If no feasible solution is found, the signs of t_i or t_j are negated and we repeat the search. In this specific example, when data are flowing in three different directions $t_k = t_i = t_j = 1$ results in a solution that satisfies constraints (1) - (14) and minimizes the completion time. From Theorem 1, we can easily get the vectors depicted in Fig. 14 from equations (3) - (6). Using these vectors we can obtain the other vectors and a basic cell design. Concluding: we connect the cells into a mesh, eliminate the cells that do not perform any computation and we obtain the SAP as depicted. This is the fastest matrix - multiplication scheme, with completion time 7 units of time and 19 cells.

To minimize the other objective function, #PE x T^2, the search has to be continued to find out all the feasible designs with completion time less than $\sqrt{19} \times 7/3 = 10.2$. By assuming that the output matrix is stationary so $\vec{c_d} = 0$, we find that a feasible design with $t_k = t_i = t_j = 1$ needs 7 units of computation time and 3 units of drain time, for a total of 10 time units of completion time. In fact, this is the optimal solution that minimizes #PE x T^2, assuming we retain the #PE and the structure of the cells as in the previous example. The configuration with the output matrix

stationary, is depicted in Fig. 15.
Again the fundamental relations of systolic processing led to this design. For example (the interpretation is not strict but, nevertheless sufficient...)
$\vec{c}_{js} = \vec{a}_d$ from (6) : means that the column displacement of C, has the same direction as the data flow of A.
$\vec{c}_{is} = \vec{b}_d$ from (4) : the row displacement of C has the same direction as the data flow of B. We can further assume that $t_{ka} = t_{kb} = 1$, which gives us:

$\vec{a}_d + \vec{a}_{ks} = 0$, (1) and

$\vec{b}_d + \vec{b}_{ks} = 0$, (2).

The first equation means that column displacement of A and the flow of direction of A are opposite. The second relation means that the row displacement of B and the direction of data flow of B are opposite. This follows directly from simple vector arithmetic, and the definitions of velocity and displacement of A and B. One can see that the discussion above leads to unique ways of distributing the output matrix in the SAP; its relation to the flow of inputs is also determined; we can also conclude that matrix A is inputted by columns, while matrix B is inputted by rows. Furthermore corrective delays for the data streams ensure correct execution of the algorithm.

The procedure described, can be summarized in 5 steps.
Step 1. Write the recurrence formula for the problem to be solved. Choice of the formula is important, as it affects

the design.

Step 2. Write the corresponding systolic processing equations (theorem of systolic processing for two - dimensional or one - dimensional case) and the constraints on the values of parameters. Note that additional constraint equations may be imposed on the design, depending on the particular problem.

Step 3. Select and write the objective function, based on the design requirements in terms of the systolic parameters and the problem size.

Step 4. Find the parameter values that minimize the objective function by enumerating over the limited search space.

Step 5. Design a basic cell for the systolic array and find a possible interconnection of cells from the parameters obtained. Eliminate cells that do not perform any useful computation.

Mapping Using Linear Transformations

This method views an algorithm as a set of nested loops. (This class of algorithms includes matrix computations and many signal processing algorithms as we have mentioned earlier). Furthermore, the computations performed within each loop should be simple and if possible identical. This is needed, so that the processing cells can be made identical. If the mathematical expressions inside a loop involve too many computations, the loop can be split into several simpler loops; in any case we can assume that the computations are almost identical over the entire <u>index space</u> (= the set of all loop indices).

A computational problem of size N, is measured by the number of elements in the index set, that is
$N = I_1 \times I_2 \times \ldots \times I_n$, where I_i indicates the number of elements along the i-th coordinate of the index set.
The mapping is done as follows: first, computational models are introduced for VLSI systolic arrays and algorithms; second, the transformation of the algorithm into a "suitable" form takes place; third, the actual construction (mapping) of the array is done. The two models are related by using a transformation function. This method seeks to minimize the processing time and the interconnection time of the SAPs.

The notation for this section is given next.
Z: refers to the set of all integers.
I: refers to the set of all nonnegative integers.
Cartesian powers are superscripts of the sets, eg. Z^n.
The points in the index space (an n-tuple in general) are denoted by $\bar{j}^1$, $\bar{j}^2$, etc.; the coordinates of a point in the index space are denoted by j_1, j_2, etc. The first of the models and definition is concerned with the type of the systolic array, its interconnections and its size.

The VLSI array model.

It is assumed that the computational resource consists of a mesh connected network of processing cells.

Definition: a mesh connected array processor is a tuple

($\underline{J}^{n-1}$, $\underline{P}$), where $J^{n-1} \subset Z^{n-1}$ is the index set of the array. That is, each processor can be identified by its set of coordinates. $P \in Z^{(n-1) x r}$ is a matrix of interconnection primitives.

Although we consider for the sake of generality that SAPs are (n-1)-dimensional, practical arrays have a planar layout. The interconnections between the cells are described by the difference vectors between the coordinates of adjacent cells. The matrix of interconnection primitives is

$P = (\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_s)$, where $\bar{p}_j$ is a column vector indicating a unique direction of a communication link. Thus P establishes all possible interconnections between the cells; however, which of these connections is used, and how, is established later.

Examples. Consider the array in Fig. 16; its model is described by (J^2,P), where

$$J^2 = \{(j_1, j_2): 0 <= j_1 <= 2 , \quad 0 <= j_2 <= 2\}.$$

and $$P = \begin{pmatrix} 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \end{pmatrix} \qquad (0)$$

This array has 8-neighbor bidirectional connections and also a connection within the cell. The connection within the cell gives us $\overline{p}_1 = (0,0)^t$, while the bidirectional connections marked by a, b, give us:

$\overline{p}_8 = (1,0)^t$, $p_9 = (-1,0)^t$ for a, and
$p_6 = (0,1)^t$, $p_7 = (0,-1)^t$ for b.

The triangular array depicted in Fig. 17 is modeled by (J^2,P), where

$$J^2 = \{(j_1, j_2),\ j_1 <= 3\ ,\ 0 <= j_2 <= j_1\},\quad \text{and}$$

$$P = (\overline{p}_1, \overline{p}_2, \overline{p}_3) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

This array does not have bidirectional communication. Triangular arrays have been proposed for algorithms such as matrix inversion, Cholesky decomposition, etc.

<u>The algorithm model</u>.

The class of algorithms with nested loops is considered. In this model we want to include the following information about the algorithm:

- The algorithm index set; (since our main concern is nested loops).
- The computations performed at each index point.
- The data dependencies which ultimately dictate the algorithm communication requirements.
- The algorithm input and output variables.

The first definition that follows, deals with the information about the algorithm that we want to include in the model. It refers to the static properties of the algorithm. The next two, examine the execution and the equivalence of algorithms. (The dynamic aspects of an

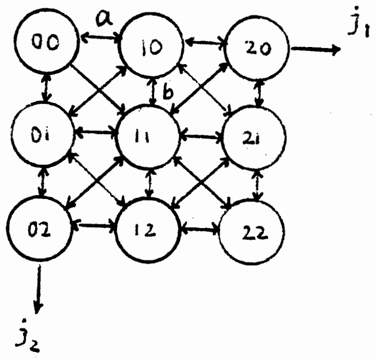

Figure 16. A square array with 8-neighbor connections.

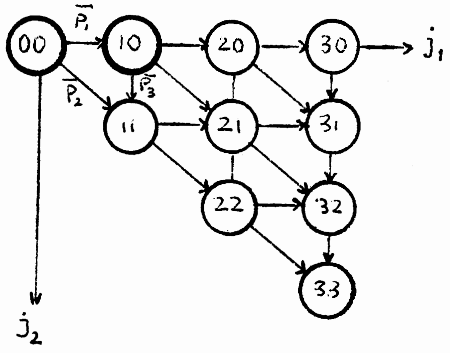

Figure 17. A triangular array with 3-neighbor connections.

algorithm).

(The following refer to the next definition; an algebraic structure S is a set of elements, with some operations defined on them. In what follows, when we refer to an algebraic structure and its carrier (set) we will use the same symbol).

Definition. An algorithm A over an algebraic structure S is a 5-tuple A = (J^n, C, D, X, Y) where:

J is a finite index set of A; $J^n \subset I^n$; that is, a finite subset of the cartesian product of all positive integers.

C is the set of all computations. It is a set of triples $(\bar{j}, v, t)$, where $\bar{j} \in J^n$ is a point in the index set, v is a variable and t is a term built from operations of S and variables ranging over S (the carrier). We call v the variable generated at $\bar{j}$; v is on the left hand side of an assignment operator - an output variable. Any variable appearing in the term t is a used variable; this variable appears on the right hand side of an assignment operator.

X is the set of input variables of A.

Y is the set of output variables of A.

D denotes the data dependencies in A; it is a set of triples $(\bar{j}, v, \bar{d})$, where $\bar{j} \in J^n$, v is variable and $\bar{d}$ is an element of Z^n. (In fact, d's are column vectors of a matrix D, as we will see later). There are three types of dependencies in D.

1) Input dependence : $(\bar{j}, v, \bar{d})$ is an input dependence if

$v \in X$ (v is an input variable) and v is an operand of t in computation $(\bar{j}, v, t)$;

2) <u>Self</u> <u>dependence</u> : as 1), only v is not an input variable. For both types of dependencies, by definition $\bar{d} = 0$.

3) <u>Internal</u> <u>dependence</u> : $(\bar{j}, v, \bar{d})$ is an internal dependence if v is an operand of t in computation $(\bar{j}, v, t)$, generated at $(\bar{j}^*, v, t)$; by definition $\bar{d} = \bar{j} - \bar{j}^*$. So v is "defined" - generated in $\bar{j}^*$, and used in $\bar{j}$; so j^* must precede $\bar{j}$.

So, if $\bar{j}$ depends on $\bar{j}^*$,this can be depicted by a vector, from $\bar{j}^*$ to $\bar{j}$ in the n-dimensional index space. (Hence our definition of $\bar{d}$). The basic structural features of an algorithm are dictated by the data dependencies. These dependencies refer to <u>precedence</u> <u>relations</u> <u>of</u> <u>computations</u>, which need to be satisfied in order to compute the problem correctly. The absence of dependencies indicates the possibility of simultaneous operations.

The <u>levels</u> at which one can examine dependencies are, blocks of computations level, statement (or expression) level, variable level, and even bit level. Our attention will focus on dependencies at the variable level. The data dependencies determine the algorithm's communication requirements. Systolic algorithms are in need of <u>local</u> and <u>regular</u> communications. Hence, the method proposed, transforms (among other things), the data dependencies of the algorithm, in order to increase the locality of communications.

Representation of dependencies. It is practical to represent all (internal) dependencies as a matrix. Every column of D is the last element of the triple $(\bar{j},v,\bar{d})$, and is labeled $\bar{d}_{v\bar{j}}$. The subscripts imply that the dependency refers to variable v, at index point $\bar{j}$. Usually, the point $\bar{j}$ is omitted, if the dependencies are valid for every index point.

Example. The following algorithm, will be used throughout our discussion to exemplify the various aspects of the method.

```
    for $j_0$ = 1 to N
      for $j_1$ = 1 to N
        for $j_2$ = 1 to N
S1:     $a(j_0,j_1,j_2) = a(j_0-1,j_1+1,j_2) * b(j_0-1,j_1,j_2+1)$
S2:     $b(j_0,j_1,j_2) = b(j_0-1,j_1-1,j_2+2) + b(j_0,j_1-3,j_2+2)$
        end $j_2$                                              (1)
      end $j_1$
    end $j_0$.
```

The model for the algorithm (1) is as follows: the index set is $\{J^3 = (j_0,j_1,j_2),\ 1 <= j_0 <= N,\ 1 <= j_1 <= N$ and $1 <= j_2 <= N\}$. The set of computations C is

$\{ (\bar{j},\ a,\ a(j_0-1,j_1+1,j_2) * b(j_0-1,j_1,j_2+1))$ and

$(\bar{j}, b, b(j_0-1, j_1-1, j_2+2) + b(j_0, j_1-3, j_2+2))$ }.

In this example, at every point in the index space, a multiplication and an addition are performed. The data dependencies can be described (as we have explained previously) by the <u>difference vectors</u> of the index points, where a variable is used and where that variable was generated. The four dependence vectors are:

$\bar{d}_1 = (1,-1,0)$ for pair $\{ a(j_0, j_1, j_2), a(j_0-1, j_1+1, j_2) \}$
$\bar{d}_2 = (1,0,-1)$ for pair $\{ b(j_0, j_1, j_2), b(j_0-1, j_1, j_2+1) \}$
$\bar{d}_3 = (1,1,-2)$ for pair $\{ b(j_0, j_1, j_2), b(j_0-1, j_1-1, j_2+2) \}$
$\bar{d}_4 = (0,3,-2)$ for pair $\{ b(j_0, j_1, j_2), b(j_0, j_1-3, j_2+2) \}$

These dependencies form the matrix D, (the order of columns is not important).

$$D = (\bar{d}_1\ \bar{d}_2\ \bar{d}_3\ \bar{d}_4) = \begin{pmatrix} 1 & 1 & 1 & 0 \\ -1 & 0 & 1 & 3 \\ 0 & -1 & -2 & -2 \end{pmatrix}$$

The first column refers to variable a, while the following three columns refer to b. Finally, the set of input variables X and the set of output variables Y are easily identified, using the indexes; (eg. during the first iteration, in statement S1, we see that a(0,2,1) and b(0,1,2) must be input variables).

Next we examine the execution of the algorithm; as our example algorithm executes, its index points are ordered in lexicographical order. This is an artificial ordering, i.e.

it can be modified, so that parallelism extraction is possible, without altering the results of the algorithm.

Definition : the execution of an algorithm $A = (J^n, C, D, X, Y)$ is described by
1) the specification of a partial ordering on J^n (called execution ordering); we will use the symbol >. This ordering will be such, so that for all $(\bar{j}, v, \bar{d}) \in D$ we will have $\bar{d} > 0$.
2) The execution rule. Until all computations in C have been performed, execute $(\bar{j}^*, v, t)$, for all $\bar{j}^* > \bar{j}$ for which $(\bar{j}, v, t)$ have terminated.
The ordering (larger than zero) > is used in lexicographical sense, i.e. if $\bar{d} = \bar{j} - \bar{j}^* > 0$, it means that the computations indexed by $\bar{j}^*$must be performed before those indexed by $\bar{j}$.

Definition : two algorithms $A = (J^n, C, D, X, Y)$ and $\hat{A} = (\hat{J}^n, C, \hat{D}, X, Y)$ are said to be T equivalent if and only if:
1) Algorithm $\hat{A}$ is input - output equivalent to A; this is denoted by $A = \hat{A}$. This means that these algorithms map any set of input variables to the same set of output variables. The following establish a stronger equivalence between algorithms than the usual input - output equivalence.
2) Index set of $\hat{A}$ is the transformed index set of A; $\hat{J}^n = T(J^n)$, where T is a bijection function; (T is a transformation).
3) To any operation of A there corresponds an identical

operation in $\hat{A}$ and vice versa.

4) Dependencies of $\hat{A}$ are the transformed dependencies of A; $\hat{D} = T(D)$.

This equivalence allows us to obtain a transformed algorithm that is equivalent to the original one. The index set and the dependencies of the new algorithm are obtained via a simple linear transform. This algorithm is now suitable for VLSI implementation.

On the transformation

The mapping is going to be done, by linearly transforming the algorithm's index set and dependencies. A linear transformation can be expressed in general as $y = Ax$, where A is a matrix containing the transformation (mapping) of x to y. More formally we have the following.

The data dependencies impose an ordering R on the index set J^n. The elements of the index set along with the ordering form an algebraic structure $<J^n,R>$. The transformation T we seek is

$$T : <J^n,R> \dashrightarrow <\hat{J}^n,R_T>,$$

where $\hat{J}^n$ is the transformed index set and R_T is the ordering imposed in the transformed index set (by the transformed data dependencies). T should have the following properties:

1) T is a bijection and a monotonic function and

2) the data dependencies of the new structure can be

selected by us! (see Theorem 1, in this section).

The transformation T is partitioned into two functions

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

Mapping Π is defined as Π: $J^n \dashrightarrow \hat{J}^1$ and S: $J^n \dashrightarrow \hat{J}^{n-1}$. Mapping Π results in an execution ordering. That is, the first coordinate of the transformed index set preserves the correctness of computation by maintaining an execution ordering. We need to have relation $\Pi d_i > 0$, for all column vectors of the dependence matrix. This constraint arises from the requirement that a variable must be generated before it is used in a computation. We elaborate on this later. The rest of the coordinates can be selected , by the algorithm designer to meet some communication requirements. Hence, roughly speaking Π deals with time, while S with the geometry and communication.

Consider an algorithm with n nested loops (index space of size n) with m constant data dependence vectors. So $D = (\bar{d}_1, \ldots, \bar{d}_m) \in R^{n \times m}$. A linear transformation T is sought such that $\hat{J}^n = TJ^n$ (transforms the index set). Now since T is linear

$$T(\bar{i}+\bar{d}_j) - T(\bar{i}) = Td_j = \hat{d}_j, \text{ for } 1 <= j <= m; \ \bar{i} \in J^n.$$

The expression holds for all index points and so we can collect all of the equations in $TD = \hat{D}$. These questions arise: under what conditions does T exist? Furthermore, is the mapping correct? (Given a computation at a cell, do all the necessary variables arrive at that cell at the correct time?). The following two theorems answer these questions, and clarify some more the method. $TD = \hat{D}$ represents a system of nxm equations with n^2 unknowns (the elements of T). This is so, because it has been stated earlier (property (2) of T), that we _assign_ the data dependencies in the new structure, hence $\hat{D}$ is assumed to be known. The following theorem indicates the necessary and sufficient conditions for the existence of linear transformation T; furthermore, it can be used as a guide to _preselect_ $\hat{D}$.

Theorem 1. For an algorithm with a _constant_ set of data dependencies D, there are three necessary and sufficient conditions so that a valid transformation T exists.

(1) The new data dependence vectors $\hat{d}_j$, satisfy the equation $\hat{d}_j = \bar{d}_j \pmod{c_j}$, for all $1 <= j <= m$, where c_j is the greatest common divisor of the elements of $\bar{d}_j$.

(2) System $TD = \hat{D}$ can be solved for T.

(3) The first nonzero element of vector $\hat{d}_j$ is positive.

Proof. Sufficient. Condition (1) indicates that the elements of $\hat{d}_j$ are multiples of the greatest common divisor of the elements of the respective $\bar{d}_j$. (This is how we can _preselect_ $\hat{D}$). This is a necessary and sufficient condition that each of the nxm diophantine equations can be solved for

integers. According to (2) the system has a solution. Since the first nonzero element of $\hat{d}_j$ is positive, it follows that $\Pi \bar{d}_j > 0$, thus T is a valid transformation. (It is necessary for the correct execution of the transformed algorithm, that the timing is correct; in other words an execution ordering is maintained. Thus, it is necessary that the transformed data dependencies $\hat{D}$ have for each column their first nonzero element positive).
Necessary. Transformation T is a bijection and consists of integers, hence (1) and (2) conditions are required. Finally, T preserves the ordering (R_T), that is $\hat{d}_j > 0$. This implies that the first nonzero element is positive.

In the selection of $\hat{D}$ 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 optimized.
The second theorem, ensures the correctness of the global level mapping. (We distinguish between the operation of the systolic system at the array level and the activities taking place inside the PEs. The array level is called the global level and the processor level is called the local level. Most methods, including this one, focus their attention into the mapping from the algorithm to the global model as this is the most critical one).

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 $x = E(v_1, v_2, \ldots, v_r)$ executed at the index point $\bar{j} \in J^n$. From the definition of data dependence vectors we have

$$\bar{j} = \bar{j}^1 + \bar{d}_1 = \bar{j}^2 + \bar{d}_2 = \ldots = \bar{j}^r + \bar{d}_r$$

where $\bar{j}^i \in J^n$ and $\bar{d}_i$ correspond to the generation of variable v_i. If we apply the linear operators Π and S to the above relation we get:

$$\Pi\bar{j} = \Pi\bar{j}^1 + \Pi\bar{d}_1 = \Pi\bar{j}^2 + \Pi\bar{d}_2 = \ldots = \Pi\bar{j}^r + \Pi\bar{d}_r \qquad (2)$$

$$S\bar{j} = S\bar{j}^1 + S\bar{d}_1 = S\bar{j}^2 + S\bar{d}_2 = \ldots = S\bar{j}^r + S\bar{d}_r \qquad (3)$$

If the computations at $\bar{j}^i \in J^n$ produce correctly v_i, then it follows from (2) and (3) that all the input variables are available for $\bar{j} \in J^n$ at the _same time_ (obtained from (2)) and at the _same processing cell_ (obtained from (3)). For each v_i there corresponds a $\bar{d}_i$ and $\hat{D}$ can be selected as desired. It follows that there is no overlap in the flow of the data streams and no cell is required to perform more than one operation at any one time. We can now say the following about transformation T and the specific meaning of its parts.

Time. A computation indexed by $\bar{j} \in J^n$ in the original algorithm is processed at the time $\Pi \bar{j} = \hat{j}_0$. That is, we can regard correctly the first coordinate of the transformed algorithm $\hat{j}_0$ as the time coordinate. This is true, because Π is selected so that $\Pi \bar{d}_i > 0$ for any $\bar{d}_i \in D$. The total running time of the new algorithm is usually

$t = \max \Pi(\bar{j}^1 - \bar{j}^2) + 1 = \max \hat{j}_0 - \min \hat{j}_0$; this assumes a unitary time increment. In general however, the time increment may not be unitary; in this case it is given by the smallest transformed dependence, i.e. $\min \Pi \bar{d}_i$. Thus the execution time of the parallel algorithm is the number of hyperplanes Π sweeping the index space J^n and it is given by the ratio

$$t = \left\lceil \frac{\max \Pi(\bar{j}^1 - \bar{j}^2) + 1}{\min \Pi \bar{d}_i} \right\rceil$$

for any $\bar{j}^1$, $\bar{j}^2 \in J^n$ and $\bar{d}_i \in D$ (4).

(Notice that the running time includes only the computation time and the communication time and not the input/output time). The communication time for a data stream associated with a dependence vector $\bar{d}$ is given by $\Pi(\bar{j} + \bar{d}) - \Pi(\bar{j}) = \Pi \bar{d}$, since Π is a linear transformation.

Network geometry. A processing cell is assigned to each distinct element of $\hat{J}^{n-1}$(remember $S : J^n \text{-->} \hat{J}^{n-1}$). The position or the identification number of each cell is given by $S(\bar{j}) = (\hat{j}_1, \hat{j}_2, \ldots, \hat{j}_n)$. The interconections (interprocessor communications) result from the transformed data dependencies; in fact they are derived from the last n-1 rows of $\hat{D}$. (The first row is associated with timing). Two observations can be made concerning the above: First, if the mapping of S results in the dimension of its range being greater than 2, then an additional one-to-one mapping is needed. This occurs, because multilayer VLSI networks may be attempted but planar arrangements are preferable. Second, the mapping T can be generalized so that the dimensionality of Π and S is marked by a positive integer k; see appendix C.

Implementation - example. We now proceed with elaborating on the mapping procedure and examples. To be more specific, two major parts can be identified in the mapping procedure. The first one, is the transforming of the algorithm in a suitable form for VLSI implementation. The second one is concerned with the derivation of the systolic array. For the first part we will see how S can be found and how it is related to other parameters. Matrix K is introduced which indicates the utilization of interconnection primitives in the array. What this method attempts to minimize is the completion time; it is also shown how space - time tradeoffs are possible in the design. Furthermore, it is explained how exactly the time - hyperplanes partition the index space, what they imply, and of course how they can be found.

I. Algorithm transformation.

Fundamental equations. We want to select the transformation S, so that the transformed dependencies are mapped into a VLSI array modeled as $(\hat{J}^{n-1}, P)$. (We assume a processor array model consisting of a grid which has the dimensionality of $\hat{J}^{n-1}$). This can be written as:

$$SD = PK \qquad (5).$$

P is the matrix of interconnection primitives, as we have previously said in (0). (The same P is used for this example as well). K is a matrix that indicates the utilization of primitive interconnections in matrix P. That is, of all the possible interconnection primitives in P, some may not be used, depending on K. These correspond to rows of matrix K with zero elements. Matrix K must satisfy the following:

$$k_{ji} >= 0 \qquad (6)$$

$$\sum_j k_{ji} <= \Pi \bar{d}_i \qquad (7).$$

Hence, all elements of the matrix must be nonnegative, due to (6), and the time between the generation and use of a variable must be greater than or equal to the number of interconnection primitives needed by the datum to travel from the PE in which it is generated to the PE in which it is used. Most often, many transformations S can be found, and each transformation leads to a different array. This flexibility apparently complicates matters, but in fact, it gives the designer the possibility to choose between a large

number of arrays with different characteristics . (Tradeoffs between space and time characteristics are possible). Let us use algorithm (1) as an example to illustrate the method.
(We must note that a program has been developed at the University of Southern California, called ADVIS - automatic design of VLSI systems -. This software package finds all valid transformations it is then up to the designer to choose an optimality criterion which will lead to the solution most suitable for his needs).

Determine Π. The first thing to determine is transformation $\Pi = (t_{11}\ t_{12}\ t_{13})$ so that it satisfies $\Pi \bar{d}_i > 0$; so if we multiply Π with matrix D of algorithm (1) we get a system of 4 inequalities in which unknowns are the 3 elements of Π. We can easily obtain that $t_{11} > t_{12} > t_{13}$. Additional constraints can limit the number of Π's. In this example the condition $\sum_{i=1}^{3} |t_{1i}| <= 3$ was used; the program found the following Πs

$$\Pi_1 = (2 \quad 1 \quad 0)$$
$$\Pi_2 = (1 \quad 0 \quad -1)$$
$$\Pi_3 = (1 \quad 0 \quad -2)$$
$$\Pi_4 = (0 \quad -1 \quad -2)$$
$$\Pi_5 = (2 \quad 0 \quad -1).$$

$\Pi_2 D = (1 \quad 2 \quad 3 \quad 2)$ (i.e. $\Pi \bar{d}_i > 0$, as required); (8),
and Π_2 minimizes the parallel execution time as given by

(4). Hence $\Pi = \Pi_2 = (1 \ \ 0 \ -1)$, and the time is $t = (N-1) - (1-N) + 1 = 2N - 1$. (The denominator is $\min \Pi \bar{d}_i = 1$, as given by (8). The index set for this example is a cube and function Π contains the coefficients of a family of parallel planes. The first index points visited by Π are (1,X,N) and the last (N,X,1), where X is "don't care". For instance (1,X,N) includes points {(1,1,N), (1,2,N),..., (1,N,N)}. So, each of the dotted lines in Fig. 18 contains a set of points that are going to be executed in times ranging from 1-N to N-1. Each of the points in these lines extends in three dimensions by including <u>all</u> the points in j_l. (The "don't cares"). Take for example points

A = (4,1,N), B = (3,1,N-1), C = (2,1,N-2) and D = (1,1,N-3). All of these points will be executed at time N-4; in fact, for each of these points, we can extend $j_1 = 1,2,\ldots,N$ and all of these index points can be executed at the same time. All index points $\bar{j}$ which are contained in one plane Π at a given moment (as in the above set), can be processed in parallel because there are no dependencies between them and they obey equation $\Pi \bar{j}$ = constant (=N-4 for the above example). In fact, the parallel processing time is nothing but the number of parallel planes Π necessary to cover all index points!

<u>Determine</u> <u>S</u>. The next step is to find transformation S. The program found twelve S matrices that satisfy

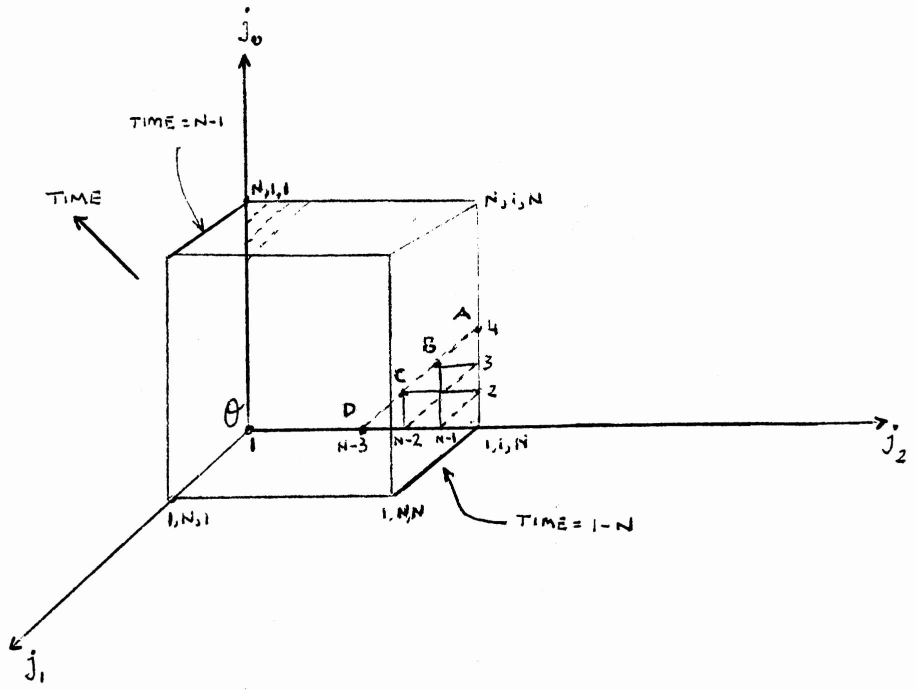

Figure 18. Index set and Hyperplanes.

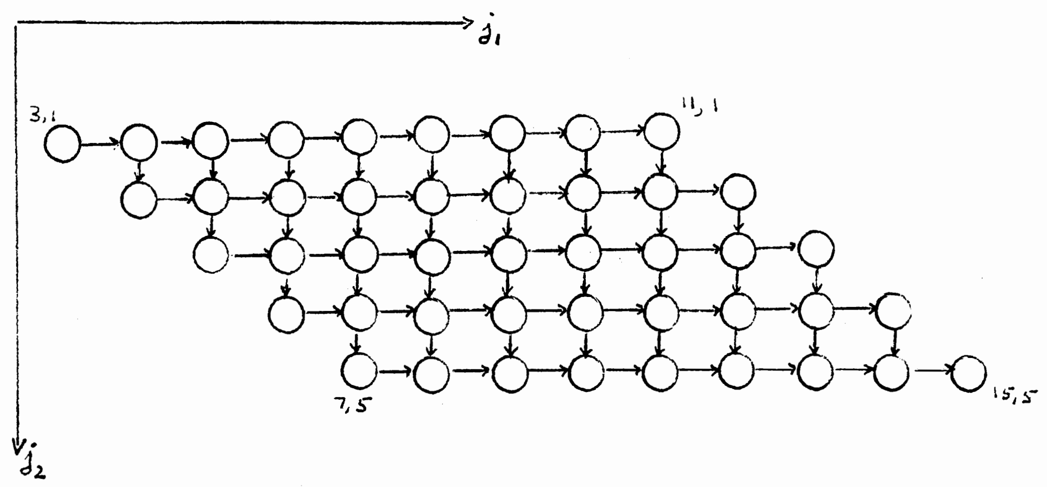

Figure 19. Systolic array of algorithm (1).

conditions (5)-(7). These S matrices, together with Π selected above, form twelve distinct valid transformations of the form

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

A utilization matrix K, which satisfies (6), (7) is

$$K = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This utilization matrix leads to transformation $T_7 = \begin{bmatrix} \Pi \\ S \end{bmatrix}$ because S_7 satisfies equation $S_7 D = PK$. (The system of diophantine equations is solved for S).

$$T = T_7 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(The program considers all possible utilization matrices K, which satisfy relations (6), (7)). Once the transformation is selected, then the new parallel algorithm results immediately from the definition of algorithm equivalence. The original index set J^n is transformed into new index set $\hat{J}^n$ so that to every point $\bar{j} \in J^n$ there corresponds a point $\hat{j} = (\hat{j}_0, \hat{j}_1, \ldots, \hat{j}_{n-1}) \in \hat{J}^n$, $\hat{j} = T_7 \bar{j}$; (in the specific example of course, n=3). Because of the way in which transformation T was selected, the first coordinate $\hat{j}_0$ indicates the time at which the computation indexed by the corresponding $\bar{j}$ is computed, and $(\hat{j}_1, \hat{j}_2)$ indicates the processor where that computation is performed. For instance, consider a computation indexed by (3,4,1) of algorithm (1); the transformed coordinates are $(\hat{j}_0, \hat{j}_1, \hat{j}_2) = T_7(3,4,1)^t = (2,8,3)^t$. This means that computation time is 2 and the processor cell at which the computation is performed is (8,3).

What has been our main concern so far was the transformation of the algorithm. Next, we want to construct the entire array in which T maps algorithm (1).

II. Array construction.

This constitutes the second major part of the mapping. The algorithm considered now is not the original one, but it is the transformed and equivalent one to algorithm (1). The interconnection primitives, indicate the direction of the communication between the cells. The transformed data dependencies, dictate the algorithm's communication requirements. Finally, we will see how we determine the direction of movement for each variable within the array, and the construction of each cell.

(1) Only two interconnection primitives are required; these are $(0\ \ 1)^t$ corresponding to North - South movement of data, and $(1\ \ 0)^t$ corresponding to West - East data movement. This happens because the utilization matrix K is very sparse. All but two rows of K are zero; the nonzero rows of

K correspond to the respective column vectors of P, which give us the interconnection primitives mentioned. In general, the simpler matrix K is, the simpler the systolic array we need. (It is of importance to notice that the array we started with was much more complex); we found however, that a much simpler one was needed).

(2) The transformed data dependencies

$$\hat{D} = T_7 D = \begin{array}{c} \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \\ \begin{array}{cccc} a & b & b & b \end{array} \end{array} \qquad (9)$$

The first row of the transformed dependencies is given by $\Pi D = (1 \quad 2 \quad 3 \quad 2)$. Each element in the first row indicates the number of time units allowed for its <u>respective</u> variable to travel from the processor where it is generated to the processor where it is used (<u>communication</u> <u>time</u>).

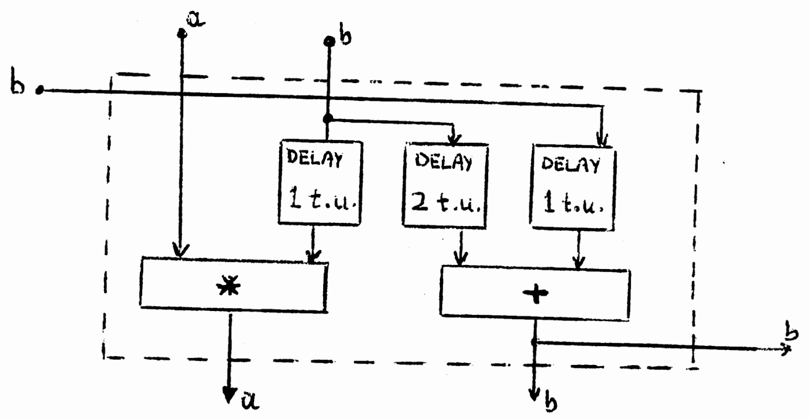

Figure 20. Cell structure.

(3) Furthermore, the direction of data flow, for each variable is represented by the last two rows in each column. All cells in the array are identical (see Fig. 19). The structure of a cell results from the computations required by the algorithm, as well as the timing and data communications, dictated by the new data dependencies. The cell consists of delay elements, an adder and a multiplier; it is assumed that the units that are doing the computations (+,*), also have one time unit delay. For example, variable a, which has dependence $\bar{d}_1$, moves from a cell to the next via a vertical North - South channel, $(0\ \ 1)^t$ and it has one unit time delay in each cell, introduced by the multiplier. For the second multiplication operand, which is variable b with dependence $\bar{d}_2$, there corresponds a vertical channel $(0\ \ 1)^t$ but an additional delay of one time unit is introduced, for a total of two time units delay.

Let us now summarize the steps of the procedure.

Step 1. Heuristically, find a transformation Π, such that $\Pi\bar{d}_i > 0$ and which minimizes

$$t = \left\lceil \frac{\max \Pi(\bar{j}^1 - \bar{j}^2) + 1}{\min \Pi\bar{d}_i} \right\rceil$$

for any $\bar{j}^1$, $\bar{j}^2 \in J^n$ and $\bar{d}_i \in D$. This step results the first row of the transformed dependencies ΠD.

Step 2. Generate all possible K matrices $K = (k_{ji})$, $K \in Z^{s \times m}$ which satisfy the following conditions:

(a) $0 <= k_{ji}$ and

(b) $\sum_j k_{ji} <= \Pi \bar{d}_i$.

Step 3. Find all possible transformations $S \in Z^{(n-1) \times n}$ which satisfy two conditions.

(a) Diophantine equation SD = PK can be solved for S and

(b) Matrix transformation T is nonsingular. As a result of this step, we may obtain some valid transformation T. If no S can be found to satisfy all the above conditions, then either we compromise the fast execution time by selecting another Π (step 1), or we compromise the locality of data communication by selecting another set of primitives P_i; (hence the array model changes).

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

Systolic computers seem to be a promising solution for obtaining a very high degree of parallelism with low cost. They consist of homogeneous processing cells that have a simple architecture. Hence, systolic arrays are easy to build and cheap in their implementation. The cells communicate locally with each other, via I/O queues; accesses to local memory are limited in most cases, thus resulting in a high bandwidth. SAPs are also easily expandable and a number of configurations is possible, thus providing flexibility to the designer. The speedup that can be expected from such a construct is in the area of O(N), where N is the number of processors. This is impressive, considering the very large number of processors usually involved in systolic computers. VLSI technology can give systolic architectures a major "thrust" forward in the near future.

It is important that an I/O bottleneck be avoided; careful analysis is required by the designer(s) to speculate on global communication and/or efficient algorithm design advantages and disadvantages. Not all algorithms are suitable for systolic implementation. The space of algorithms for systolic implementation has a module granularity which is small constants, and distributed control achieved by simple local control mechanisms; furthermore, the communication geometry of the systolic algorithms must be simple and regular.

SAPs are at the present time application dependent (special - purpose) devices. However, general - purpose systolic computers can be built in the next years. These computers will include flexible systolic arrays for computations applicable to systolic configurations; the processing cells increase in complexity and programmability and should be able to execute independently as well. Three general - purpose systolic schemes were examined. Warp, and Matrix-1, are already in operation are excellent examples of where systolic architectures are going to. The Hockney shorthand description was used to describe/summarize these computers and their primary characteristics. We also described WAP - a systolic dataflow computer; this somehow different approach to systolic computing, is equally promising and has the advantages (and disadvantages) that are present in any data - driven architecture. For example, it can be faster than a synchronized scheme; on the other hand there exists an overhead due to the additional information that has to be stored in the tokens.

An important metric of systolic computing is examined, the utilization rate of the processors. A variety of the most commonly encountered scenarios was analyzed and formulas were presented. A careful match between the size of the problem and that of the systolic array, will result in a high utilization rate. The configuration of the SAP for the specific type of problem also affects the utilization rate.

The key to efficient implementation of systolic arrays is mapping, i.e., obtaining a configuration of a SAP with most of the characteristics such as timing, function of the cells, etc. from a set of algorithms with common characteristics. Two methods with different approaches were presented, namely, the parameter method and the dependency method. Both obtain SAPs (in fact, they are equivalent!) for a rather limited class of problems. Luckily however, these classes of algorithms include a vast number of signal, matrix related problems and others which are suitable for systolic implementation. The parameter method easily expands its set of relations on which the mapping is based; i.e. additional constraints are obtained, depending on the specific problem. The dependency method on the other hand, has a background which is solid and has already been tried out (dependency of variables). Both are limited in that they preassume a "specific" model of SAP on which they seek to map the algorithm.

Future research. The existing systolic computers involve today a rather small number of processors which may also communicate with global buses (thus providing flexibility). This situation, if we want a really modular system, one that is purely systolic, is highly undesirable. Furthermore, a possible clock skew (in the case of a large number of PEs) has to be eliminated.

The architectures in use today, for example Warp, has been built with devices that are rather old and conventional. If these units are replaced with better ones, the resulting SAP may improve its performance dramatically. (For instance, the Warp cell uses the Am2910A microsequencer, which is slow and with many limitations). But then, retiming of much of the system would be necessary so that a timing imbalance is avoided. The portability of the systolic devices has to be examined, too.

While ad hoc designs are usually efficient, systematic methodologies (mapping) for large classes of algorithms are necessary. The mapping methods should deal with a broader class of algorithms than what they do now. These methods should also include more parameters in their design procedures that are related for instance, to the host, to the memories used, etc. The design is thus optimized and "fitted" best into the specific environment. SAPs should be easily attached to a number of host computers and execute a variety of compute - intensive algorithms (partly or in whole) thus speeding up the execution of the host. In our opinion, methods that are based on the dependencies of the variables are the ones most likely to increase in the near future. They resemble other procedures that have been used in VLSI technology already, and are based on a background - theory that is solid and expandable.

REFERENCES

Annaratone, M., et al, "Warp Architecture and Implementation", Proc. 13th Int. Symp. Comp. Architecture, pp. 346-356, June 1986.

Apostolico, A., Negro, A., "Systolic Algorithms for String Manipulations", IEEE Trans. Comp., C-33, 4(1984), pp. 361-364.

Barke, D. F., ed., "VLSI: Fundamentals and Applications", 1980.

Brent, R. P., Kung, H. T., "Systolic VLSI Arrays for Polynomial GCD Computation", IEEE Trans. Comp., C-33, 9(1984), pp. 731-736.

Bromley, K., et al, "Systolic Array Processor Developments", in "VLSI Systems and Computations", ed. Kung, H. T., et al, pp. 273-284, 1981.

Cappello, P., Steiglitz, K., "Digital Signal Processing Applications of Systolic Algorithms", in "VLSI Systems and Computations", ed. Kung, H. T., et al, pp. 245-254, 1981.

Chazelle, B., "Computational Geometry on a Systolic Chip", IEEE Trans. Comp., C-33, 9(1984), pp. 774-785.

Drake, B. L., et al, "SLAPP: A Systolic Linear Algebra Parallel Processor", Computer, 20, 7(1987), pp. 45-49.

Fisher, A. L., "Systolic Algorithms for Running Order Statistics in Signal and Image Processing", in "VLSI Systems and Computations", ed. Kung, H. T., et al, pp. 265-272, 1981.

Fortes, J. A. B., Wah, B. W., "Systolic Arrays: From Concept to Implementation", Computer, 20, 7(1987), pp. 12-17.

Fortes, J. A. B., Wah, B. W., "Systolic Arrays: A Survey of Seven Projects", Computer, 20, 7(1987), pp. 91-103.

Foster, M. J., Kung, H. T., "The Design of Special Purpose

VLSI Chips", Computer, 13, 1(1980), pp. 26-40.

Foulser, D. E., Schreiber, R., "The Saxpy Matrix-1: A General - Purpose Systolic Computer", Computer, 20, 7(1987), pp. 35-43.

Gaudet, G., Stevenson, D., "Optimal Sorting Algorithms for Parallel Computers", IEEE Trans. Comp., C-27, 1(1978), pp. 84-87.

Guerra, C., et al, "Synthesizing Non-Uniform Systolic Designs", Intenational Conference on Parallel Processing, 1986, pp. 765-772.

Hockney, R. W., Jesshope, C. R., "Parallel Computers: Architecture, Programming and Algorithms", Bristol, England, 1981.

Hwang, K., Briggs, F. A., "Computer Architecture and Parallel Processing", New York, 1984.

Kung, H. T., "Let's design Algorithms for VLSI Systems", Proc. of the Caltech Conf. on VLSI, ed. Seitz, C. L., Pasadena, California, pp. 65-90, Jan. 1979.

Kung, H. T., Lehman, P. L., "Systolic (VLSI) Arrays for Relational Database Operations", Dept. Com. Sc., Carnegie Mellon Univ., 1980.

Kung, H. T., "The Structure of Parallel Algorithms", in "Advances in Computers", ed. Yovits, 19, pp. 65-112, 1980.

Kung, H. T., "A Two-Level Pipelined Systolic Array for Convolutions", in "VLSI Systems and Computations", ed. Kung, H. T., et al. pp. 255- 264, 1981.

Kung, H. T., "Why Systolic Architecture", Computer, 15, 1(1982), pp. 37-46.

Kung, H. T., "Notes on VLSI Computation", in "Parallel Processing Systems", ed. Evans, D. J., pp. 339-356, 1982.

Kung, H. T., Yu, S. Q., "Integrating High - Performance Special - Purpose Devices into a System", in "VLSI Architecture", ed. Randell. B., et al, pp. 205-211, 1983.

Kung, H. T., Webb, J. A., "Mapping Image Processing Operations onto a Linear Systolic Machine", Tech. Rep., Carnegie Mellon Univ., March 1986.

Kung,S. Y., et al, "Wavefront Array Processor: Language, Architecture and Applications", IEEE Trans. Comp., C-31, 11(1982), pp.1054-1066.

Kung, S. Y., "On Supercomputing with Systolic/ Wavefront Array Processors", Proc. IEEE, 72, 7(1984), pp. 867-884.

Kung, S. Y., et al, "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems", IEEE Trans. Comp. C-36, 5(1987), pp. 603-614.

Kung, S. Y., et al, "Wavefront Array Processors - Concept to Implementation", Computer, 20, 7(1987), pp. 18-33.

Lehman, P. L., "A Systolic (VLSI) Array for Processing Simple Relational Queries", in "VLSI Systems and Computations", ed. Kung, H. T., et al, pp. 285-295, 1981.

Leiserson, C.E., "Systolic Priority Queues", Proc. of the Caltech Conf. on VLSI, ed. Seitz, C. L., pp. 199-214, Jan. 1979.

Leiserson, C. E., "Area - Efficient VLSI Computation", MIT Press, 1983.

Leiserson. C. E., "Systolic and Semisystolic Design", IEEE International Conference on Computer Design: VLSI in Computers, pp. 627-632, 1983.

Li, G. J., Wah, B. W., "The Design of Optimal Systolic Arrays", IEEE Trans. Comp., C-34, 1(1985), pp. 66-77.

McCanny, J. V., McWhirter, J. G., "Some Systolic Array Developments in the United Kingdom", Computer, 20, 7(1987), pp. 51-63.

Mead, C., Conway, L., "Introduction to VLSI Systems", Addison - Wesley, 1980.

Miranker, W. L., "Space - Time Representations of Computational Structures", Computing, 32, 2(1984), pp. 93-114.

Moldovan, D. I., "On the Analysis and Synthesis of VLSI Algorithms", IEEE Trans. Comp., C-31, 11(1982), pp.1121-1126.

Moldovan, D. I., "On the Design of Algorithms for VLSI Systolic Array", Proc. IEEE, pp. 113-120, Jan. 1983.

Moldovan, D. I., Fortes, J. A. B., "Partitioning and Mapping Algorithms Into Fixed Size Systolic Arrays", IEEE Trans. Comp., C-35, 1(1986), pp. 1-12.

Navarro, J. J., et al, "Solving Matrix Problems with no Size Restriction on a Systolic Array Processor", International Conference on Parallel Processing, 1986, pp. 676-683.

Navarro, J. J., "Computing Size-Independent Matrix Problems on Systolic Array Processors", 13th International Symposium on Computer Architecture, 1986, pp. 271-279.

Navarro, J. J., et al, "Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors", Computer, 20, 7(1987), pp. 77-89.

O'Keefe, M. T., et al, "A Comparative Study of Two Systematic Design Methodologies For Systolic Arrays", International Conference on Parallel Processing, 1986, pp. 672-675.

O'Leary, D. P., "Systolic Arrays for Matrix Transpose and Reorderings", IEEE Trans. Comp., C-36, 1(1987), pp. 117-122.

Ramakrishnan, I. V., et al, "On Mapping Homogeneous graphs on a Linear Array Processor Model", International Conf. Parallel Proc., pp. 440-447, 1983.

Rogers, M. H., "Specification of Algorithms for Systolic Array Elements", in "VLSI Architecture", ed. Randell, B., et al, pp. 212- 224, 1983.

Shih, Z. C., et al, "Systolic Algorithms to examine All Pairs of Elements", CACM, 30, 2(1987), pp. 161-167.

Savage, C., "A Systolic Data Structure Chip for Connectivity Problems", in "VLSI Systems and Computations", ed. Kung, H. T., et al, pp. 296-300, 1981.

Savage, C., "A Systolic Design for Connectivity Problems", IEEE Trans. Comp., C-33, 1(1984), pp. 99-104.

Varman, P. J., Fussell, D. S., "Design of Robust Systolic Algorithms", International Conf. Parallel Proc., pp.458-460, 1983.

APPENDICES

APPENDIX A

PROOF OF THE SYSTOLIC THEOREM

The proof for the theorem of systolic processing will be given here. The proof is quite clear and uses direct manipulation (composition) of vectors to derive the relations (1) - (6). The assumptions made for the proof are (without loss of generality):

(a) The SAP consists of PEs that are orthogonally connected and with diagonal connections. Obviously the situation for a linear array would be much simpler, while a hexagonal SAP is similar to the model assumed here (as far as communication goes).

(b) The periods are assumed to be positive and equal, i.e. $t_k = t_{kx} = t_{ky} > 0$.

Assume A, B, C and D are four PEs, that <u>do</u> <u>not</u> <u>have</u> <u>to</u> be directly connected; why this can be true will be obvious as the proof progresses.

<u>Proof</u> <u>of</u> (<u>1</u>), (<u>2</u>): While PE C ic computing $z^{k}_{i,j} = f(z^{k-1}_{i,j}$, x(i,k), y(k,j)), the next element of X, $x_{x(i,k+1)}$ is in PE B and the next element of Y, $y_{y(k+1,j)}$ is in PE D. (Notice how the elements of X, Y are referenced using the

subscript - access functions) (Fig. 21(a)). Since the periods are positive i.e. $t_{kx} = t_{ky} > 0$,
$\overrightarrow{CB}$ represents $\vec{x}_{ks}$, (i) and $\overrightarrow{CD}$ represents $\vec{y}_{ks}$, (ii).
According to the characteristics of systolic processing, the operands needed for the next iteration (recursion) must arrive at the <u>same PE simultaneously</u> after t_k units of time. (Remember, this period is the time difference between computation of recurrences k and k+1, for z).
Hence, $z_{i,j}^{k}$, $x_{x(i,k+1)}$, $y_{y(k+1,j)}$, arrive at PE A simultaneously, which computes recurrence $z_{i,j}^{k+1} = f(z_{i,j}^{k}, x(i,k+1), y(k+1,j))$, (see Fig. 21(b)). We have:
$\overrightarrow{CA} = t_k \vec{z}_d$, (iii); $\overrightarrow{BA} = t_k \vec{x}_d$, (iv); $\overrightarrow{DA} = t_k \vec{y}_d$, (v).
Furthermore, from the principle of vector composition, we have
$\overrightarrow{CB} + \overrightarrow{BA} = \overrightarrow{CA}$, and using (i), (iv) and (iii) we obtain (1).
Also, $\overrightarrow{CD} + \overrightarrow{DA} = \overrightarrow{CA}$, and using (ii), (v) and (iii) we obtain (2). (Do not forget that the periods are assumed equal).
The cases in which t_{kx} and t_{ky} have different signs or are both negative can be proved similarly.

<u>Proof of (3), (4)</u>: Suppose that while PE D is computing $z_{i,j}^{k} = f(z_{i,j}^{k-1}, x(i,k), y(k,j))$, PE C is computing $z_{i+1,j}^{p}$, p<k, and PE B has $x_{x(i+1,k)}$; (see Fig. 21(c)).
Therefore $\overrightarrow{DC} = \vec{z}_{is}$, (vi), and $\overrightarrow{DB} = \vec{x}_{is}$, (vii).
According to the characteristics of systolic processing, k-p steps of the iterative computation are performed <u>after</u> t_i units of time and $z_{i+1,j}^{k-1}$, $x_{x(i+1,k)}$, and $y_{y(k,j)}$ arrive at PE A

for the computation of $z^{k}_{i+1,j}$. Let us briefly clarify why this happens: t_i is defined as $t_i = T_c(z^{k}_{i+1,j}) - T_c(z^{k}_{i,j})$. It is the time elapsed between the computation of two successive z's along the i - dimension. Obviously, since element $z^{p}_{i+1,j}$ and element $z^{k}_{i,j}$ are computed at the instant described, after t_i units of time, (and k-p steps of computation), $z^{p}_{i+1,j}$ becomes $z^{k-1}_{i+1,j}$ and now $z^{k}_{i+1,j}$ can be computed at PE A in terms of $z^{k-1}_{i+1,j}$, $x_{x(i+1,k)}$ and $y_{y(k,j)}$. So: $\overrightarrow{BA} = t_i \vec{x}_d$, (viii); $\overrightarrow{CA} = t_i \vec{z}_d$, (ix); $\overrightarrow{DA} = t_i \vec{y}_d$, (x). From the principle of vector composition we have: $\overrightarrow{DB} + \overrightarrow{BA} = \overrightarrow{DA}$, and using (vii), (viii) and (x) we obtain (3). Also, $\overrightarrow{DC} + \overrightarrow{CA} = \overrightarrow{DA}$, and using (vi), (ix) and (x) we get relation (4). (With the i, j periods of z being equal). Finally, the proofs for (5), (6) are absolutely similar to (3) and (4), respectively.

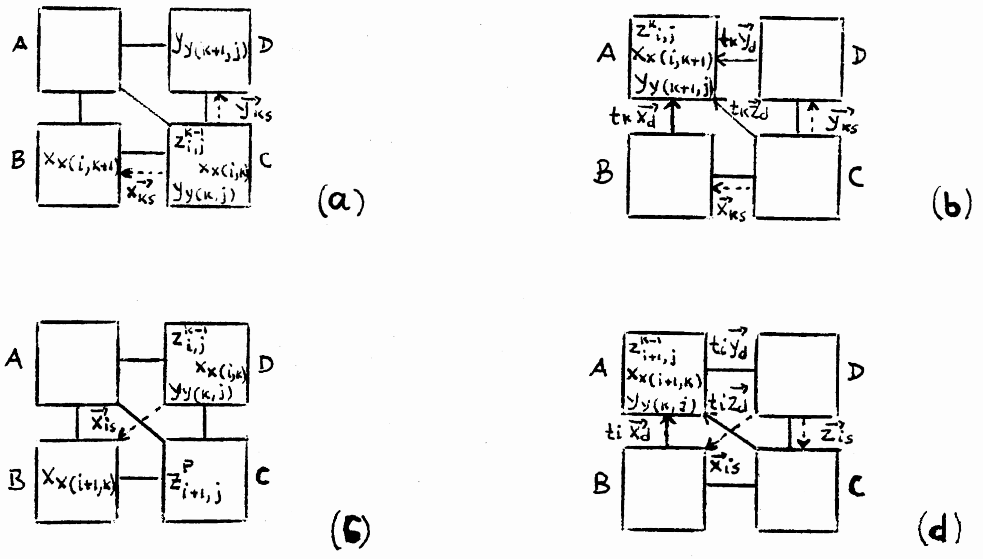

Figure 21. Snapshots of the Systolic Theorem.

APPENDIX B

PARAMETER METHOD - AN EXAMPLE

This appendix provides an example of how to apply the parameter method to obtain a systolic array for convolutions. The problem can be expressed as a <u>one-dimensional</u> linear recurrence and is thus simpler than the two-dimensional examples discussed so far. A recurrence for this problem is

$$y_i^0 = 0, \qquad 1 <= i <= n$$
$$y_i^k = y_i^{k-1} + w_{m-k+1} \; x_{m-k+i} \, ,$$
$$1 <= i <= n, \quad 1 <= k <= m.$$

This recurrence evaluates the terms in reverse order as we will see, so that y_n is the first computed term. The inputs, i.e. the set of weights w and x, are accessed in the same order - decreasing, and hence $t_{kw} = t_{kx}$. (The reader is urged to refer to the definitions presented previously). The function to be minimized is #PE x T^2.

<u>Completion time</u>. It takes mt_k units of time to compute y_1 (m is the number of weight coefficients and t_k is the time difference between computing successive recurrences of y); in addition to that, $(n-1)|t_i|$ units of time are needed

to compute the remaining y_i's. (Remember that $t_i = T_c(y^K_{i+1}) - T_c(y^K_i)$). Hence the total computation time is $T = mt_K + (n-1)|t_i|$, excluding possible load and drain times. Let us assume that m=4 and n=6. In this case $T_{serial} = 6 \times 4 = 24$.

<u>Bounds on the periods</u>. We can now find the bounds of the periods. It is required that $T <= T_{serial}$, i.e.
$4t_K + 5\ |t_i| <= 24$.
By using the minimum value (=1) of one of the periods in the above inequality, we obtain an upper bound for the other.
So, $t_{Kmax} = \lceil 19/4 \rceil = 5$ and $t_{imax} = \lceil 20/5 \rceil = 4$.
The problem is formulated as:

$$\text{Minimize } \#PE \times (\ 4|t_K| + 5|t_i| + \text{load time} + \text{drain time}\ )^2.$$

subject to the equations of the systolic Theorem

$$t_{Kx}\ \vec{x}_d + \vec{x}_s = t_{Kx}\ \vec{y}_d \qquad (1)$$
$$t_{Kw}\ \vec{w}_d + \vec{w}_s = t_{Kw}\ \vec{y}_d \qquad (2)$$
$$t_i\ \vec{x}_d + \vec{x}_s = t_i\ \vec{w}_d \qquad (3)$$
$$t_i\ \vec{y}_d + \vec{y}_s = t_i\ \vec{w}_d \qquad (4)$$

and a set of constraints (see corresponding part in the method)

$1/4 <= |\vec{w}_d| <= 1$ or $|\vec{w}_d| = 0$

$1/5 <= |\vec{y}_d| <= 1$ or $|\vec{y}_d| = 0$

$1 <= t_K <= 5$ $\qquad 1 <= |t_i| <= 4$

$|\vec{w}_s| \neq 0$ $\qquad |\vec{x}_s| \neq 0$ $\qquad |\vec{y}_s| \neq 0$

$|t_i||\vec{w}_d| = k_1 <= 4$ $\qquad t_\kappa|\vec{y}_d| = k_2 <= 5$

$t_\kappa = |t_{\kappa x}| = |t_{\kappa\omega}|$ $\qquad t_{\kappa x} = t_{\kappa\omega}$

It is not necessary to bound $|\vec{x}_d|$ because $\vec{x}_d$ is uniquely determined when t_κ, t_i, $\vec{y}_d$ and $\vec{w}_d$ are set. At this point we feel it is proper to discuss two points.

1) The procedure refers to ".. enumerating over the limited search space..". This means: examine all possible combinations of values of periods and vectors, and choose the ones that are minimal and lead to a feasible solution. (This way we obtain the complexity of the search space, as discussed previously in the thesis). So, it is valid not to bound $\vec{x}_d$, since it is determined by the other values.

2) The reader may have noticed the use of a linear array (which is of size m). This is not arbitrary; the reader is referred to that part of the method that discusses the #PEs. Although the case of one-dimensional problems was not discussed, it is easy to see that a lower bound for #PE, for a problem of size n is either n, or 1. The latter of course does not usually lead to any efficient solutions.

If we assume that $t_{\kappa x} = t_{\kappa\omega} = 1$ and $t_i = -1$ (y is evaluated in reverse order) and $|\vec{w}_d| = 0$, we obtain

$|\vec{y}_d| = |\vec{y}_s| = 1$ (see (4))

$|\vec{x}_d| = |\vec{x}_s| = 1/2$ (see (1))

and $|\vec{w}_s| = 1$ (see (2) and bounds on $|\vec{y}_d|$).

We note that this is a one-dimensional solution and so all the vectors are pointing in the same direction. Furthermore, the assumption $|\vec{w}_d| = 0$ simply means that the weights are statically placed in the cells; this is a natural assumption as the weights are predefined constants. The completion time of the algorithm is m + n - 1 = 9 time units. To see that this design is optimal, the performance measure #PE x T^2 for the specific example is 4 x (9 x 9) = 324. If the number of PEs is decreased to one (the other possibility), then T degrades to T_{serial} = 24 and so the performance measure becomes #PE x T^2 = 1 x (24 x 24) = 576.

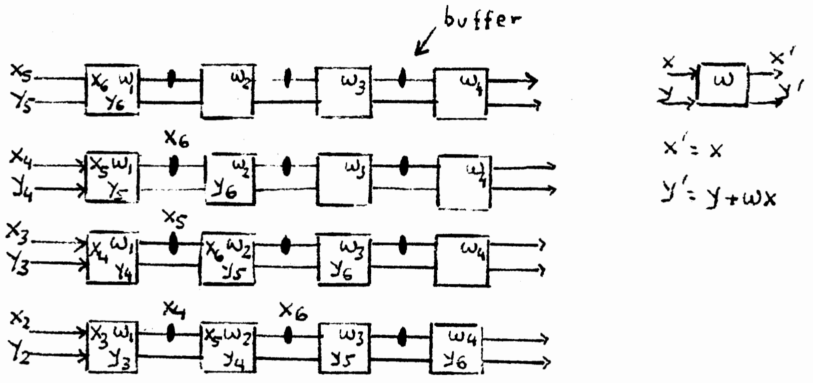

Figure 22. Systolic Array for Convolutions.

APPENDIX C

DEPENDENCY METHOD GENERALIZATION

The mappings Π, S can be defined in general as follows:

$\Pi : J^{n} \dashrightarrow \hat{J}^{k}$ $\qquad n > k$

$S : J^{n} \dashrightarrow \hat{J}^{n-k}$.

Thus the dimensionality of Π, S is marked by an integer k. k is selected such that Π alone establishes the ordering R_T. So the role of Π and S remains unchanged; only now <u>the first k coordinates</u> of elements in $\hat{J}^{n}$ are related to time, while the <u>last n-k coordinates</u> can be related to the geometrical properties of the algorithm.

In an analogous manner we have that the total number of processing cells is $O(N^{n-k})$, (where N is the size of the problem; that is, each of the n indexes in an algorithm of n nested loops, ranges within O(N) values). The running time in this case becomes $O(N^{k})$. Another observation is that keeping k as small as possible should be one goal in designing VLSI algorithms. This will increase the concurrency of operations at the expense of the number of processors.

2

VITA

Leonidas Alexandropoulos

Candidate for the Degree of

Master of Science

Thesis: ON THE EFFICIENT DESIGN AND IMPLEMENTATION OF SYSTOLIC STRUCTURES

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Athens, Greece, June 20, 1962, the son of Vassilis and Chrysoula Alexandropoulos. Was married to Sofia Fafoutaki in December 27, 1985.

Education: Bachelor of Science Degree in Mathematics from University of Patras, Greece, in June, 1985; completed requirements for the Master of Science degree at Oklahoma State University in July, 1988.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August 1987, to May 1988.