1984

Systolic algorithms for the CMU warp processor

H. T. Kung
Carnegie Mellon University

CMU-CS-84-158

Systolic Algorithms for the CMU Warp Processor

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

January 1984
(Last revised September 1984)

A preliminary version of this paper appeared in *Proceedings of the 7th International Conference on Pattern Recognition*, Montreal, Canada, July 1984, pp. 570-577, as the text of an invited talk.

The research was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659, in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and in part by a Guggenheim Fellowship.

Abstract

CMU is building a 32-bit floating-point systolic array that can efficiently perform many essential computations in signal processing like the FFT and convolution. This is a one-dimensional systolic array that in general takes inputs from one end cell and produces outputs at the other end, with data and control all flowing in one direction. We call this particular systolic array the *Warp* processor, suggesting that it can perform various transformations at a very high speed.

We expect to have wide applications for the Warp processor, especially for the CMU prototype which has high degrees of flexibility at the expense of a relatively high chip count for each cell. The prototype has 10 cells, each of which is capable of performing 10 million floating-point operations per second (10 MFLOPS) and is build on a single board using only off-the-shelf components. This 10-cell processor for example can process 1024-point complex FFTs at a rate of one FFT every 600 μs. Under program control, the same processor can perform many other primitive computations in signal, image and vision processing, including two-dimensional convolution and complex matrix multiplication, at a rate of 100 MFLOPS. Together with another processor capable of performing divisions and square roots, the processor can also efficiently carry out a number of difficult matrix operations such as solving covariant linear systems, a crucial computation in real-time adaptive signal processing.

This paper outlines the architecture of the Warp processor and describes how the signal processing tasks are implemented on the processor.

1. INTRODUCTION

Very high performance computer systems must rely heavily on parallelism, since there are severe physical and technological limits on the ultimate speed of any single processor. The systolic array concept allows effective use of a very large number of processors in parallel. In recent years many systolic array algorithms have been designed and several prototypes of systolic array processors have been built [2, 11, 21, 23]. Major efforts have now started in attempting to use systolic array processors in large applications. Practical issues on the implementation of systolic array processors have begun to receive substantial attention.

To implement systolic array designs, appropriate architectures of the underlying processors must be developed. Architectural optimization for the implementation of a narrow set of algorithms is usually not very difficult, but designing an architecture which can efficiently implement a wide class of algorithms is a non-trivial task. The challenge is to achieve a balance between many conflicting goals, such as the generality of the system vs. ease of programming, flexibility vs. efficiency, and performance of the system vs. its design and implementation costs. Therefore, a key research issue regarding the implementation of systolic arrays is the identification of processor architectures that have the right tradeoffs between these conflicting goals. Along this line the CMU Programmable Systolic Chip (PSC), for example, represents a research effort in identifying architectures of general purpose microprocessors that can efficiently implement systolic arrays for a variety of application areas [4, 6, 5]. The Warp processor considered in this paper, on the other hand, resulted from research in identifying architectures for implementing very high performance systolic arrays only in the special area of signal processing.

The design and construction of a prototype Warp processor are currently being carried out at CMU, using off-the-shelf components. This prototype will be used as an attached processor for a general-purpose host computer.

Section 2 briefly describes the architecture of the Warp processor, and features of the CMU prototype. Main results of this paper are in Sections 3 to 5, which justify the architecture of the Warp processor by showing how it can efficiently implement convolution, interpolation, matrix multiplication and the FFT. Section 6 contains some concluding remarks and brief discussions on the use of the Warp processor in solving linear systems.

This paper does not address implementation details of the CMU prototype, which are subjects of other papers [16].

2. THE WARP PROCESSOR ARCHITECTURE

The Warp processor is a one-dimensional or linear systolic array that takes inputs at one end of the array and produces outputs at the other end, with data and control all flowing in one direction, as depicted in Figure 1.

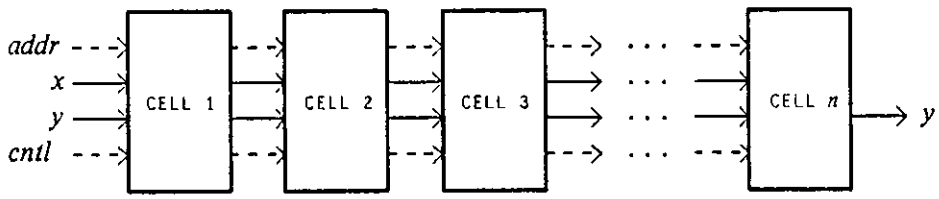

Figure 1. The Warp processor

There are several advantages of having this simple interconnection scheme, besides the obvious ease of its design and implementation. Linear arrays require the minimum-possible I/O, in the sense that only the two end cells communicate with the outside world. Thus an n-cell Warp processor can perform $O(n)$ computations for each I/O operation. This property is desirable in practice, because usually it is the I/O bandwidth with the outside world a major limiting factor for achieving high performance. Linear arrays have the additional advantage that they can always be safely synchronized by a simple, global clock [3]. Finally, by having data and control all flow in one direction, we can use efficient fault-tolerant techniques to deal with faulty cells in a systolic array [14].

As to be shown later in the paper, cells of the Warp processor do not have to be complex either, in order to implement the target signal processing tasks. The following are the major functional features of each cell:

- MPY *and* ALU. These are arithmetic units for multiplication (MPY) and ALU operations. For the CMU prototype, the MPY and ALU are implemented with the Weitek 32-bit floating-point multiplier and ALU chips, respectively [22]. To get the best-possible throughput for the Warp cell, these chips are used in their pipeline mode. That is, a chip starts a new 32-bit arithmetic operation every cycle, although the result of an operation will not emerge from the chip's output ports until five cycles after the operation starts.

  The Warp processor array built from these pipelined arithmetic units therefore supports pipelining at both the array and the cell levels. These *two levels of pipelining* greatly enhance the system throughput. [14, 17].

- MPY *register file*, and ALU *register file*. These are general register files, each implemented with a copy of the Weitek 32×32 six port register file

chip. The MPY register file can also compute approximate inverse and inverse square root functions, using the look-up unit on the Weitek register file chip.

- *Data Memory.* Having a memory at each cell for buffering data, implementing look-up tables, or storing intermediate results is essential for reducing the I/O bandwidth requirement of the cells. Also by using its local memory to store temporary data, a cell can be multiplexed to implement the functions of multiple cells in a systolic array design. As a result, for example, the Warp can implement algorithms designed for two-dimensional systolic arrays, despite that it is a linearly-connected processor array. For the CMU prototype, the local data memory in each cell has 4K words, and can be expanded to have 16K words in the future. The memory can perform both a read and a write simultaneously every cycle, using address selected from the adr-file, crossbar, or the data memory itself (i.e., indirect addressing).

- *Two data I/O ports (x, y) and one address I/O port (addr).* The Warp cell can input as well as output two words, and a pair of (read/write) addresses for the data memory, every cycle.

- *x-file, y-file,* and *addr-file.* These register files (of 128 words each) are provided mainly to implement programmable delays to ensure that *x*, *y*, and *addr* streams are properly synchronized, as required by systolic algorithms. To implement programmable delays, these files are equipped with counters that can automatically increment read and write addresses every cycle. When not implementing programmable delays, these files can be used as scratchpad register files, by setting or holding the counters using the microcode.

- *Crossbar.* The functional units, data memory, register files, and I/O ports of the Warp cell are linked by a crossbar. The crossbar has 8 read ports, including one that accepts literals from microcode, and 6 write ports. The crossbar can be reconfigured every cycle under control of microcode to allow a read port to get data from *any* of the 6 write ports.

- *Input muxes.* These are used to implement computations using the wraparound mode or bi-directional dataflows. In the wrap around mode the outputs of the cell is fed back to its inputs, hence wrapping around the cell. The wraparound mode multiplexes the use of one cell to implement the function of several. This mode is useful for increased utilization of the Warp in case of host I/O bottleneck. It also increases the virtual size of the array for problems requiring larger array size.

Figure 2 summarizes the cell datapath for the CMU prototype. Note that by setting the MUXes outputs of a cell can be used immediately as inputs to the same cell in the next cycle. This "wraparound" facility allows a single physical cell to implement a number of consecutive logic cells. Also, the y-input of each cell can take values from the y-output of the cell to the right. As to be discussed in Section 6, this feature allows the Warp processor to implement linear systolic arrays with bi-directional data flows.

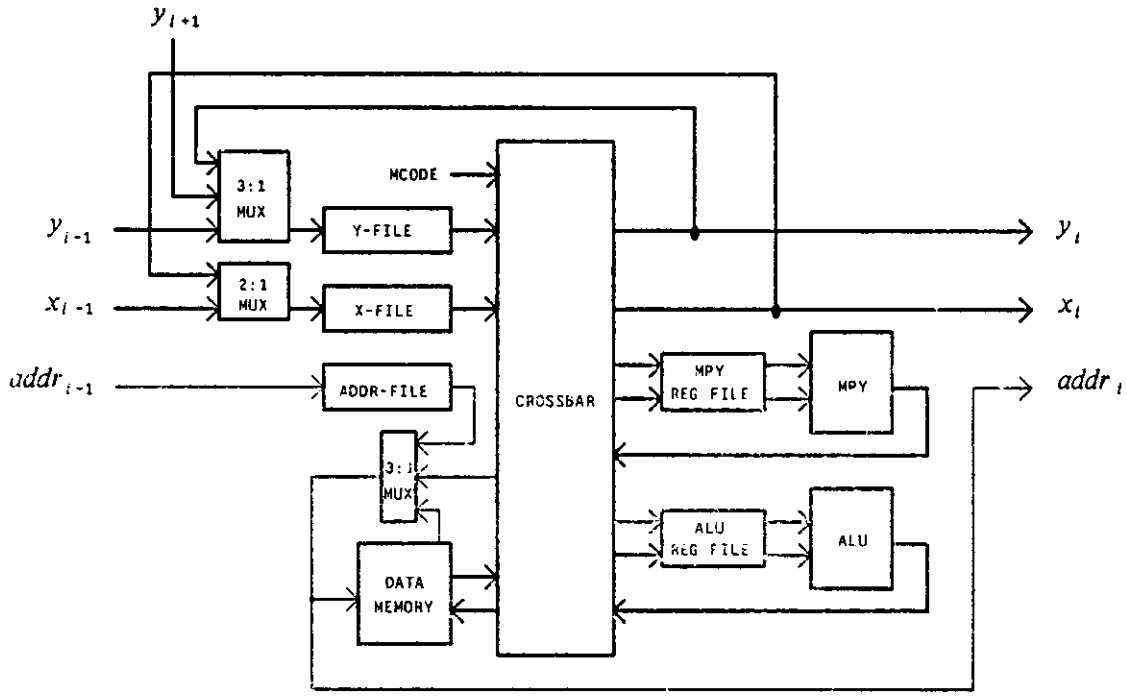

Figure 2. Datapath for the Warp cell

3. CONVOLUTION

We first present systolic array designs for one-dimensional (1-D) convolutions that the Warp processor can implement efficiently. Then we extend these designs to handle two-dimensional (2-D) and higher-dimensional convolutions.

3.1. One-Dimensional Convolution

The 1-D convolution problem is defined as follows:

> Given a kernel as a sequence of weights $(w_1, w_2, \ldots, w_k)$
> and an input sequence $(x_1, x_2, \ldots, x_n)$,
> compute the output sequence $(y_1, y_2, \ldots, y_{n-k+1})$, defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \cdots + w_k x_{i+k-1}.$$

Depicted in Figure 3 is one of the many possible systolic array designs for 1-D convolutions, given at an abstract level [10]. Weights are preloaded into the array, one for each cell. During the computation, both inputs x_i and partial results for y_i flow from left to right, but the x_i move twice as fast as the y_i. The speed difference ensures that each y_i can meet all the k consecutive x_i that it depends upon. More precisely, each y_i stays inside every cell it passes for one cycle, thus it takes twice as long to march through the array as does an x_i. It is an easy exercise to see that each y_i, initialized to zero before entering the leftmost cell, is indeed able to accumulate all its terms while moving to the right. For example, y_1 accumulates w_1x_1, w_2x_2, and w_3x_3 in three consecutive cycles at the first, second, and third cell from the left, respectively.

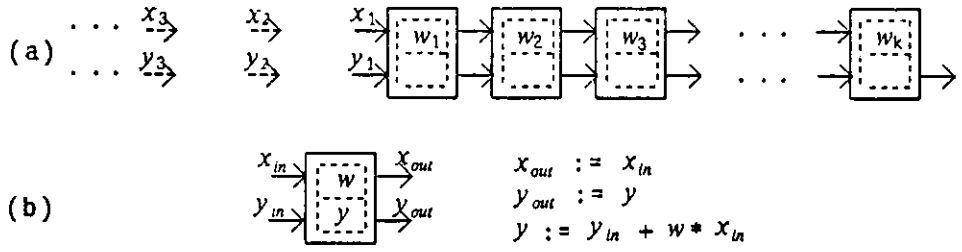

Figure 3. (a) Systolic array for 1-D convolution using a kernel of size k, and (b) the cell specification

Figure 4 depicts the internal structure of two consecutive cells, assuming that multiplication together with addition can be done in one cycle. Note that the y-data stream has two latches for each cell, as opposed to one latch for the case of the x-data stream. Thus data on the two data streams travel at two speeds, as required by the systolic array algorithm of Figure 3.

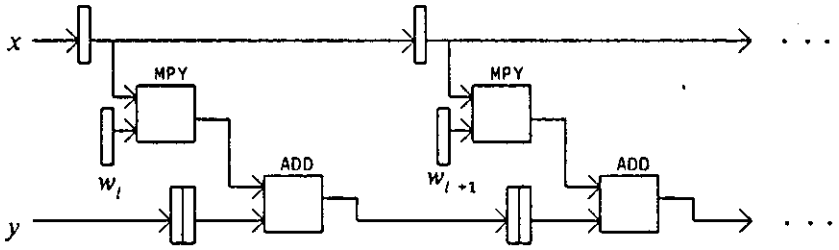

Figure 4. Cell structure for the 1-D convolution array of Figure 3

To use the Warp processor, we consider now the situation that the pipelined multiplier and adder each have five pipeline stages. From Figure 5 we see that in this case it will take five steps, instead of two steps as in the case of Figure 4, for each y_i to pass a cell. To compensate the additional three delays on the y-data stream, we use 4 ($=3+1$) latches on the x-data stream for each cell [14].

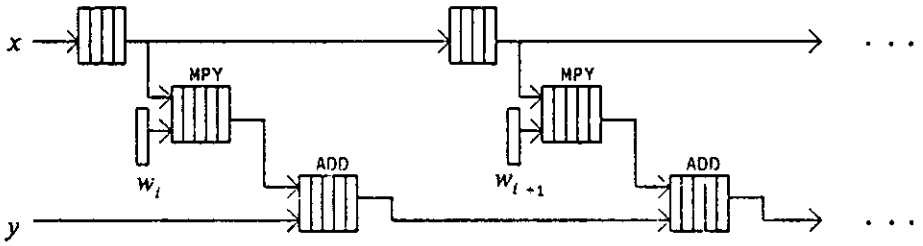

Figure 5. Two-level pipelined systolic array for 1-D convolution

Figure 6 shows another design that the Warp processor can implement. In this design weight w_i used by each cell at every cycle is selected on-the-fly from the data RAM by a systolic address flowing from cell to cell on the *addr* stream. By associating each y_i with an address on the *addr* stream, this design [17] allows different sets of weights to be used to compute different y_i, as required, for example, in interpolation and resampling of signals. Note that by going through the same delay at each cell, that is, a delay of four steps for the design of Figure 6, both y_i and its associated address are synchronized in the sense that they arrive at each cell at the same time.

3.2. Two-Dimensional Convolution

The 2-D convolution problem is defined as follows:

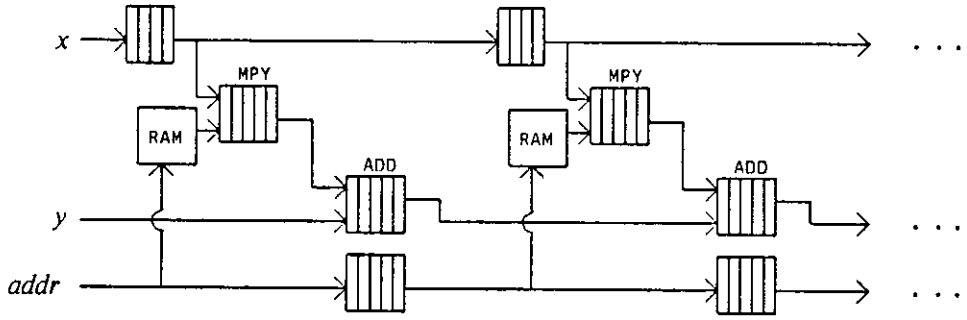

Figure 6. Systolic array using adaptive weights for interpolation and resampling

Given the weights w_{ij} for $i=1,2,\ldots,k$, $j=1,2,\ldots,p$ that form a $k\times p$ kernel, and an input image x_{ij} for $i=1,2,\ldots,m$, $j=1,2,\ldots,n$, compute the output image y_{ij} for $i=1,2,\ldots,m-k+1$, $j=1,2,\ldots,n-p+1$ defined by

$$y_{ij}=\sum_{h=1}^{k}\sum_{l=1}^{p} w_{hl}x_{i+h-1,j+l-1}.$$

It has been shown that any 2-D convolution problem can be converted into a 1-D convolution problem with the 1-D input sequence and kernel defined as follows [12]. The input sequence is

$$x_{1*}, x_{2*}, \ldots, x_{m*},$$

where $x_{i*}=x_{i1}, x_{i2}, \ldots, x_{in}$. That is, the input sequence is the concatenation of the rows of the given 2-D image and has a total length of mn. The kernel is

$$w_{1*}, (n-p)!0, w_{2*}, \ldots, (n-p)!0, w_{k*},$$

where $w_{i*}=w_{i1}, \ldots, w_{ip}$ and $(n-p)!0$ stands for a vector of $n-p$ zeros. Thus the kernel is the concatenation of the rows of the given 2-D kernel, with a vector of $(n-p)$ zero elements inserted between each consecutive pair of rows. The length of the kernel is therefore $n(k-1)+p$.

The method described here, that converts a 2-D problem into a 1-D one, will generate $p-1$ invalid results for every set of n outputs [12]. Fortunately, the fraction of invalid results, which will be ignored, is very small because $p \ll n$.

If the systolic array design of Section 3.1 is applied directly to perform the 1-D convolution derived from a 2-D convolution, then a large number of cells, that is, $n(k-1)+p$ cells would be needed in the array, and cells with zero weights would perform no useful work. From Figure 5, we see that the only effects of a cell with a zero weight are to delay data on the y-data stream by five cycles and those on the

x-data stream by four cycles. Therefore this cell can be replaced with a cell that just introduces zero cycle delay for the x-data stream and a single cycle delay for the y-data stream. This degenerate cell may in turn be absorbed into the cell to the right by introducing one shift register stage to that cell. In general, if there are q non-zero elements in the kernel, then the systolic array needs only q cells.

Applying the above cell-saving technique to 1-D convolutions derived from 2-D convolutions, we conclude that the convolution of an $m \times n$ image with a $k \times p$ kernel can be performed on a systolic array of kp cells, where cells $ip+1$, $i = 1, 2, \ldots, k-1$, each have a shift register of $n-p$ stages. Figure 7 depicts such a two-level pipelined systolic array for 2-D convolutions. The data RAM of each cell is used to implement the required shift register that may be needed. The read and write addresses required at every cycle for the implementation of the shifter are supplied by the *addr* stream. It is straightforward to see that the Warp processor can efficiently implement this systolic array design.

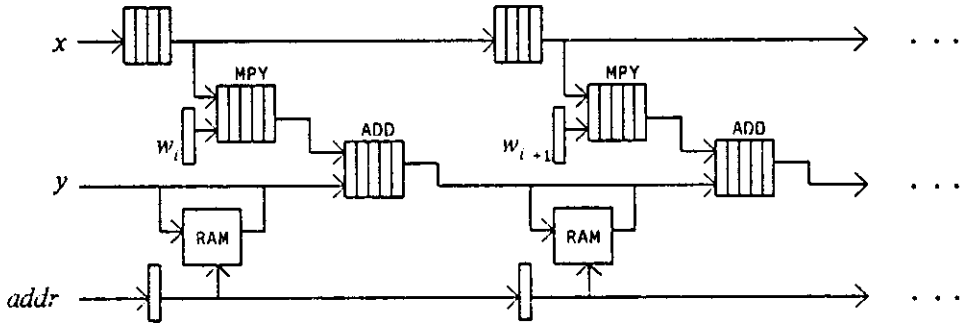

Figure 7. Two-level pipelined systolic array for two or higher dimensional convolutions

Three or higher dimensional convolutions can also be converted into 1-D convolutions in a similar way [12]. By using the conversion and cell-saving techniques, the architecture of the Warp processor can handle convolutions of any dimensionality. But the size of the data RAM at each cell must increases as does dimensionality. For example, convolving an $n \times n$ image with a $k \times k$ kernel the data RAM must be large enough to hold $n-k$ words, as shown above, whereas convolving an $n \times n \times n$ 3-D image with a $k \times k \times k$ kernel the data RAM must be large enough to hold $(n-k)(n+1)$ words.

The design of Figure 7 has a dual design for which data on the y-data stream travel at a higher speed than those on the x-data stream. More precisely, each cell of the dual design has six instead of four delays in the x-data stream, as depicted in Figure 8 [12]. The dual design has the property that the data RAM keeps data from the x-data stream rather than from the y-data stream. This allows a reduction on the size of the data RAM for each cell, when the word size for data on the (input) x-data stream is smaller than that for data on the (output) y-data stream.

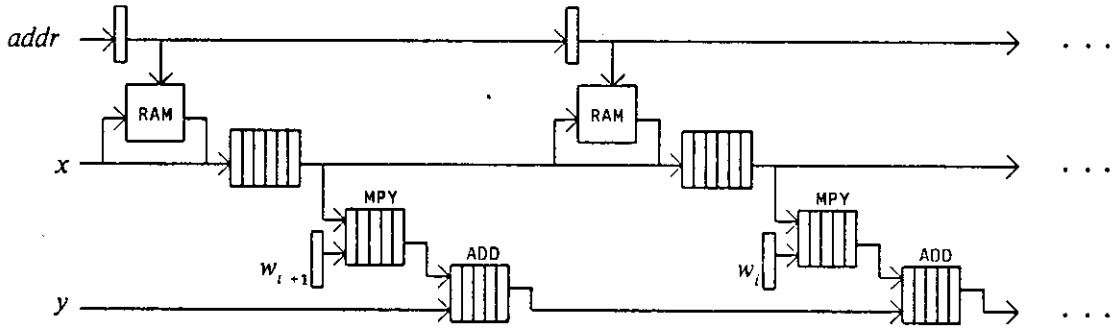

Figure 8. Another two-level pipelined systolic array for two or higher dimensional convolutions

For convolutions with very large kernels, well known transform methods based on the FFT should be used instead. In Section 5 we will describe how the Warp processor can be programmed to perform the FFT.

4. MATRIX MULTIPLICATION

Given $n \times n$ matrices $X=(x_{ij})$ and $W=(w_{ij})$, we want to compute their product $Y=(y_{ij})$. We present two systolic array designs for matrix multiplication that the Warp processor can efficiently implement—the design of Figure 11 for real matrix multiplication and the design of Figure 12 for complex matrix multiplication.

Figure 9 depicts a simple linear systolic array at an abstract level for the matrix multiplication problem [18], with $n=8$. Each cell of the array performs a multiply-accumulate operation every cycle. The j-th cell from the left computes the inner product y_j of vectors $(x_{i1},x_{i2},\ldots,x_{in})$ and $(w_{1j},w_{2j},\ldots,w_{nj})$, for each i. By pumping the entries of X into the array serially in the row-major ordering and by recirculating $(w_{1j},w_{2j},\ldots,w_{nj})$ at cell j for each j, entries in the product matrix $Y=XW$ will be computed, and output also in the row-major ordering.

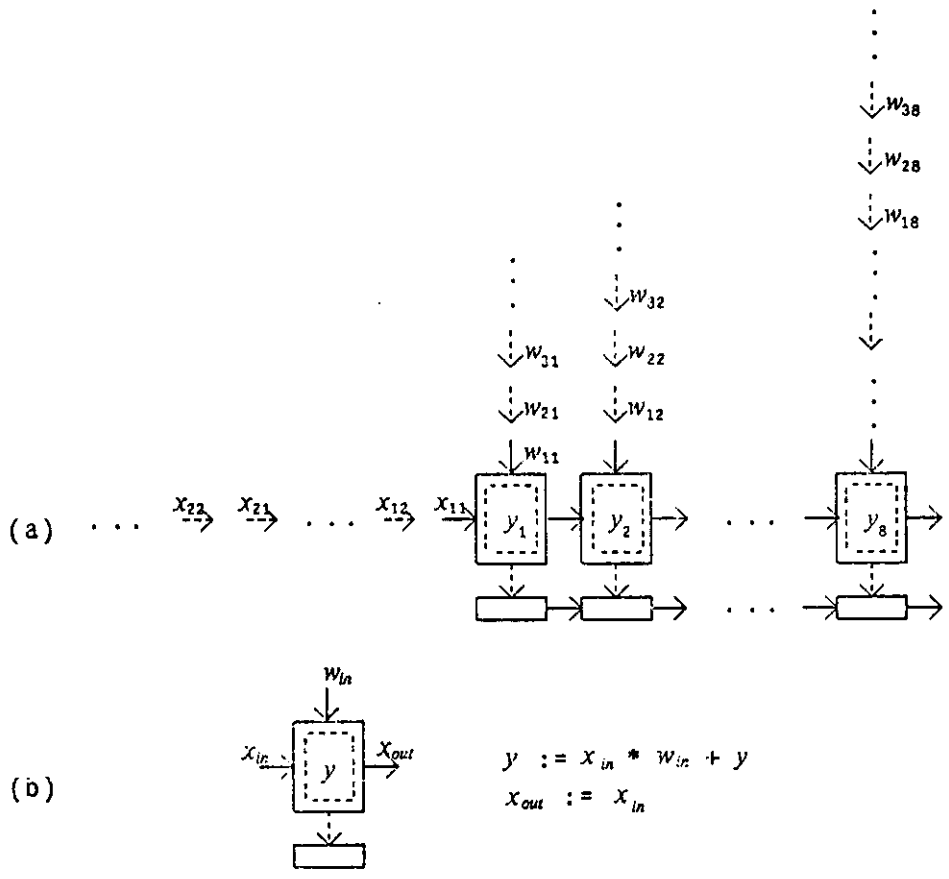

Figure 9. (a) Systolic array for matrix multiplication, and (b) cell specification

Figure 10 shows the internal structure of two consecutive cells for computing y_i and y_{i+1}, assuming that multiplication together with addition can be done in one

cycle. Each cell performs a multiply-accumulate operation every cycle and the result is kept in the y register. After n multiply-accumulate operations, the contents of the y register is transferred to the corresponding latch on the y-data stream. These computed results on the y-data stream shift to the right systolically at the rate of one cell every cycle. When they reach the right end cell of the systolic array, they are output. Note that on the x-data stream two rather than one latch is provided for each cell. This ensures that computation for y_i be completed two cycles earlier than that for y_{i+1}, and therefore the computed y_i and y_{i+1} will not collide with each other on the y-data stream.

To facilitate the recirculation of $(w_{1j}, w_{2j}, \ldots, w_{nj})$, we store at cell j these values in a shifter implemented by a RAM. The particular w to be used in any given cycle is picked up by an address (*addr*), which is input to the cell every cycle. Since the address patterns for all the cells are the same, they are passed systolically from cell to cell. To synchronize the *addr* stream with the x-data stream, two latches are provided for the *addr* stream at each cell.

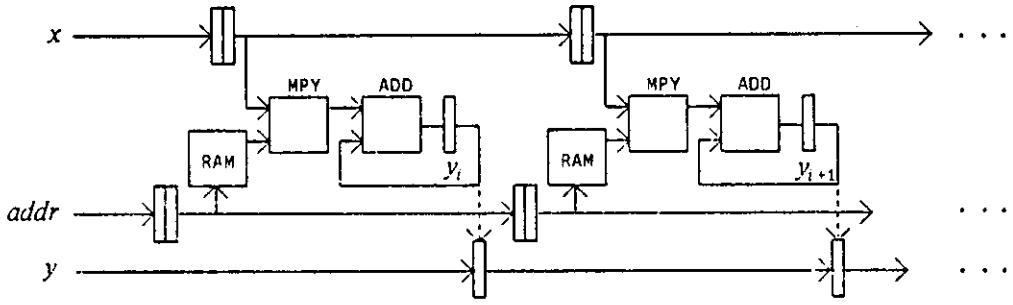

Figure 10. Cell structure for the systolic matrix multiplication array of Figure 9

4.1. Interleaving Multiple Matrix Multiplications

We now consider the case that the multiplier and adder each have five pipeline stages. Since the adder has five stages, accumulations can take place only once every five cycles. In order to make full use of the adder and multiplier, we interleave computations for five independent matrix multiplications on the systolic array. (These matrix multiplications may actually be subtasks of a single, large matrix multiplication.) This implies that a new task can enter the adder every cycle, and that five independent accumulations can be updated simultaneously at various stages of the adder at any given cycle. However, with this interleaving scheme, cells will output results in bursts. That is, at every n-th cycle a cell will start outputting results for five consecutive cycles. To avoid collisions on the y-data stream, among outputs from different cells, we use 6 ($= 5 + 1$) latches on the x-data stream for each cell, as depicted by Figure 11.

4.2. Complex Matrix Multiplication

Complex matrix multiplications are common in many signal processing applica-

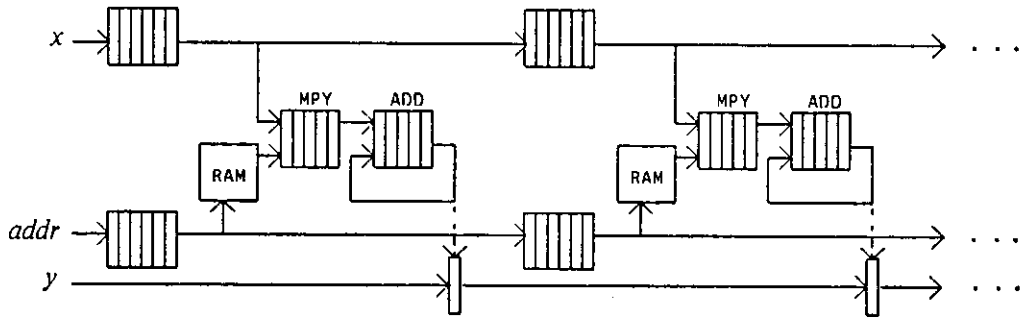

Figure 11. Two-level pipelined systolic array for matrix multiplication

tions like beamforming. A complex multiply-accumulate operation,

$$(x_r + jx_i)\cdot(w_r + jw_i) + (y_r + jy_i)$$
$$= [(x_r \cdot w_r - x_i \cdot w_i) + y_r] + j[(x_r \cdot w_i + x_i \cdot w_r) + y_i],$$

involves four real multiplications and four real additions. They will be done in four cycles using the multiplier and adder of each cell. The real and imaginary parts of a complex number are processed in separate cycles; in particular, the real part of a result is computed two cycles earlier than its imaginary part. In the scheme of Figure 12, the four multiplications, $x_r \cdot w_r$, $x_i \cdot w_i$, $x_r \cdot w_i$, and $x_i \cdot w_r$, occupy four consecutive stages of the multiplier. The resulting products enter directly into the adder to form $x_r \cdot w_r - x_i \cdot w_i$ and $x_r \cdot w_i + x_i \cdot w_r$, and these two additions occupy two stages of the adder. Interleaved with these two stages are stages for performing the other additions involving y_r and y_i. We append a three-stage shift register to the adder, so that two independent (complex) accumulations, that involve updating a total of four real numbers, can be simultaneously maintained inside the eight-stage pipeline.

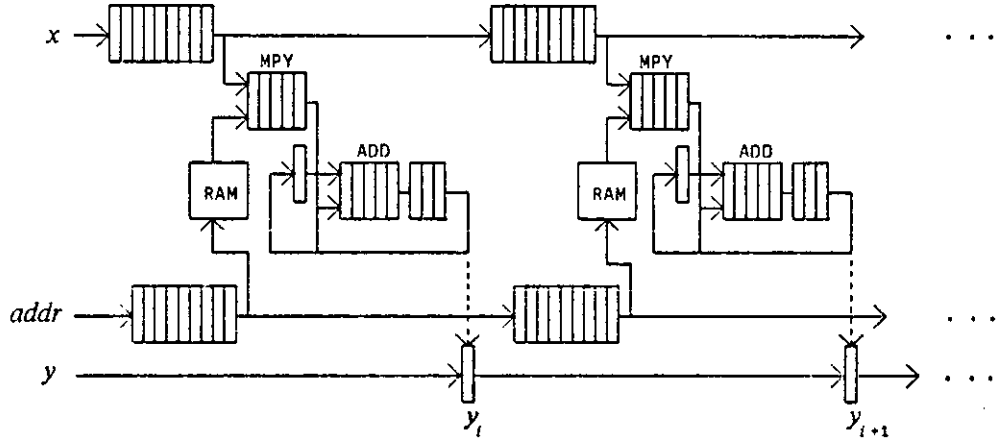

Figure 12. Two-level pipelined systolic array for complex matrix multiplication

5. FAST FOURIER TRANSFORM

The problem of computing an n-point discrete Fourier transform (DFT) is as follows:

Given $x_0, x_1, \ldots, x_{n-1}$,
compute $y_0, y_1, \ldots, y_{n-1}$ defined by

$$y_i = x_0\omega^{i(n-1)} + x_1\omega^{i(n-2)} + \cdots + x_{n-1},$$

where ω is a primitive n-th root of unity.

Assume that n is a power of two. The well known fast Fourier transform (FFT) method solves an n-point DFT problem in $O(n \log n)$ operations, while the straightforward method requires $O(n^2)$ operations. The FFT involves $\log_2 n$ stages of $n/2$ butterfly operations, and data shufflings between any two consecutive stages. The so-called constant geometry version of the FFT algorithm allows the same pattern of data shuffling to be used for all stages [20], as depicted in Figure 13 with $n=16$. In the figure the butterfly operations are represented by circles, and number h by an edge indicates that the result associated with the edge must be multiplied by ω^h.

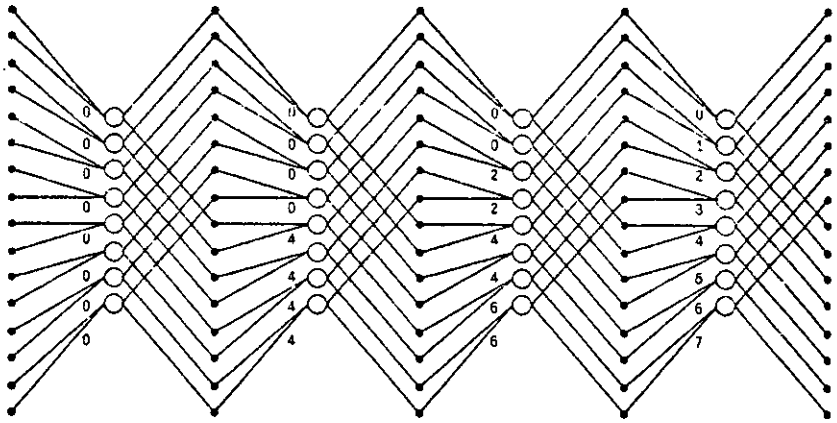

Figure 13. Constant geometry version of the FFT *algorithm*

We show that the constant geometry version of the FFT can be implemented efficiently with the Warp processor. In the systolic array, all the butterfly operations in the i-th stage are carried out by cell i, and results are stored to the data RAM of cell $i+1$. While the data RAM of cell $i+1$ is being filled by the outputs of cell i, cell $i+1$ can work on the butterfly operations in the $(i+1)$-st stage of another FFT problem. In practical applications, there are often a large number of FFT's to be processed, or there are FFT problems being continuously generated. Thus it is possible that a new FFT problem can enter the first cell, as soon as the cell becomes free. In this way all the cells of the systolic array can be kept busy all the time.

We now describe how butterfly operations are executed by each cell. A butterfly operation,

$$(a_r + ja_i) \pm (b_r + jb_i)\cdot(w_r + jw_i)$$
$$= [a_r \pm (b_r \cdot w_r - b_i \cdot w_i)] + j[a_i \pm (b_r \cdot w_i + b_i \cdot w_r)],$$

involves four real multiplications and six real additions. Thus, just to do the necessary additions, it will occupy six cycles of the adder of the cell. By techniques similar to those used in the design of Figure 12, we can derive a two-level pipelined systolic design of Figure 14, for which each cell can in fact process a new butterfly operation every six cycles. At any time, executions of up to four independent butterfly operations are interleaved at various stages of a cell. Note that inputs of the butterfly operation each are used twice and so are the intermediate results. As illustrated in Figure 14, this can be efficiently supported by the pipeline registers linked to the input registers of the arithmetic units.

Inputs a_r, a_i, b_r and b_i of the butterfly operations are obtained from the data RAM of each cell. And the outputs of the butterfly operations are stored in the data RAM of the next cell on the right. The constant geometry version of the FFT allows the use of the same address patterns for all the cells. Therefore addresses for the RAM of each cell can be passed from cell to cell systolically along the *addr* stream.

The other inputs $w_r + jw_i$ needed for the butterfly operations, are obtained from the x-data stream. For illustration, consider the 16-point problem depicted in Figure 13. The $w_r + jw_i$ needed by the eight butterfly operations at each stage are various powers of a primitive 16-th root of unity ω. In particular, we use

$\omega^0,\omega^0,\omega^0,\omega^0,\omega^0,\omega^0,\omega^0,\omega^0$ for the first stage,

$\omega^0,\omega^0,\omega^0,\omega^0,\omega^4,\omega^4,\omega^4,\omega^4$ for the second stage,

$\omega^0,\omega^0,\omega^2,\omega^2,\omega^4,\omega^4,\omega^6,\omega^6$ for the third stage, and

$\omega^0,\omega^1,\omega^2,\omega^3,\omega^4,\omega^5,\omega^6,\omega^7$ for the fourth stage.

For each cell to obtain the proper ω^h at each cycle, we pump elements in $(\omega^0,\omega^1,\omega^2,\omega^3,\omega^4,\omega^5,\omega^6,\omega^7)$ sequentially into the leftmost cell of the systolic array, and move them from left to right systolically along the x-data stream. Cell 1 buffers the first entry ω^0 and uses it for eight butterfly operations. Cell 2 buffers the first entry ω^0 and uses it for four butterfly operations, and after that it buffers the fifth entry ω^4 and uses it for four butterfly operations. Similarly cell 3 buffers its inputs ω^0, ω^2, ω^4, and ω^6, and use each of them for two butterfly operations. Cell 4 just uses whatever entry on x-data stream at every cycle. All these operations can be easily controlled, for example, by a counter at each cell, which is denoted by CNT in Figure 14.

The CMU prototype of the Warp processor has ten cells, each having a data RAM of 4K words. Thus by double-buffering the RAMs, the machine can do computations of ten independent 1024-point complex FFTs simultaneously. More

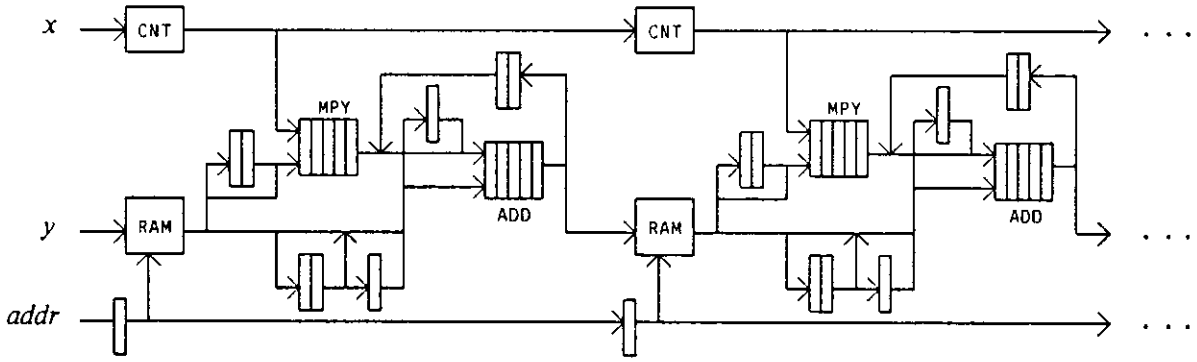

Figure 14. Two consecutive cells in the two-level pipelined systolic array for the FFT

precisely, at any given time all these FFTs are in different stages being carried out by different cells. Since the machine has a cycle time of 200 ns, a cell takes 1.2 μs to process a butterfly operation. This implies that we can process 1024-point complex FFTs at a rate of one FFT every $(1/2)\cdot(1024)\cdot 1.2\ \mu s\ (=614.4\ \mu s)$.

The I/O bandwidth requirement for the Warp processor when performing FFTs is modest, in view of its high performance in throughput. Inputs $w_r + jw_i$ and *addr* to cell 1 are constants and thus can be supplied, for example, by a memory external to the host, as depicted in Figure 15. The host needs only to deliver one set of inputs for a butterfly operation, which amount to four words, every 1.2 μs. This is equivalent to a data rate of 13.4 Mbytes/second. At the same rate the host collects outputs coming out from cell n along the y-data stream. Thus the total data bandwidth requirement between the host and the Warp need not excess 27 Mbytes/second in order to make full utilization of the Warp processor when performing FFTs.

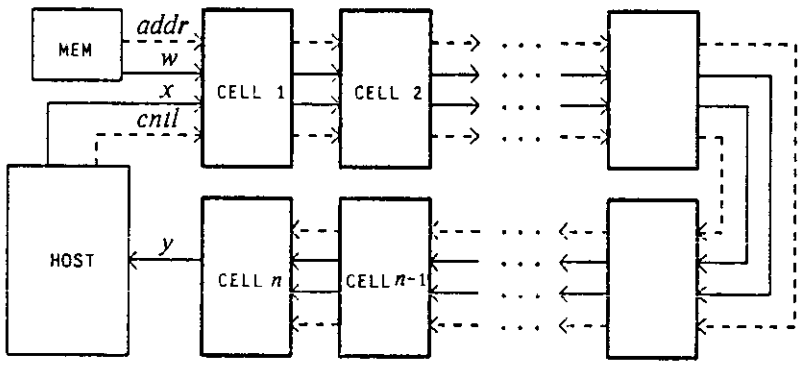

Figure 15. I/O interface of the Warp *processor when performing* FFTs

The linear array of Figure 15 is wrapped around so that both cell 1 and cell n are adjacent to the host. This layout arrangement allows an indefinite expansion of the array while maintaining the I/O interface with the host.

6. CONCLUDING REMARKS

When performing convolution, matrix multiplication and QR-decomposition, the multiplier and adder of each cell are fully utilized every cycle. Therefore, for the 10-cell CMU prototype, each cell can deliver 10 MFLOPS, and results in an aggregate computation rate of 100 MFLOPS.

The Warp processor is highly modular in the sense that the number of cells in the linear array can expand indefinitely to deal with problems of large sizes. For example, to make the CMU prototype to handle 2-D convolutions for 5×5 kernels at the original rate of one output every 200 ns, we can simply extend the processor array to 25 cells.

As discussed at the end of Section 2, by wrapping around data each cell can perform the functions of several consecutive cells, and thus large problems can often be handled by a small array. For example, a 5-cell CMU prototype can perform 2-D convolutions for 5×5 kernels at a rate of one output every microsecond instead of 200 ns. In this case the I/O bandwidth requirement for the prototype to communicate with the outside world is reduced. Thus multiplexing cells is also a useful technique to make full use of the Warp processor when the communication speed with the outside world cannot keep up with the cell speed.

The Warp processor can solve problems beyond those considered in this paper. For example, it can perform polynomial evaluation and discrete Fourier transform at the peak speed [13]. In general, most of the so-called "local" operations in signal and image processing can be efficiently carried out by the Warp processor. Since Batcher's sorting network is realizable with the constant geometry interconnection of Figure 13 [9], the Warp can also be used for sorting.

Together with another processor for performing fast divisions and square roots, the Warp processor can efficiently implement most of the systolic arrays proposed in the literature for solving various kinds of linear systems. For example, it is known that a linear systolic array of Figure 16 can solve triangular linear systems [15], and perform the QR-decomposition of a Hessenberg matrix [7], a key step in real-time adaptive signal processing [1, 19]. The boundary processor, drawn in dotted lines in the figure, is assumed to be capable of performing hard arithmetic operations such as divisions and square roots.

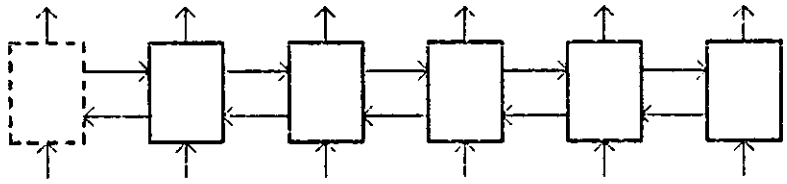

Figure 16. Bi-directional systolic array with a boundary processor

Figure 17 shows that the Warp processor augmented with the boundary processor

can implement the systolic array of Figure 16. Note that the y-input of each cell now takes values from the y-output of the cell to the right, so that the required bi-directional data flows can be implemented.

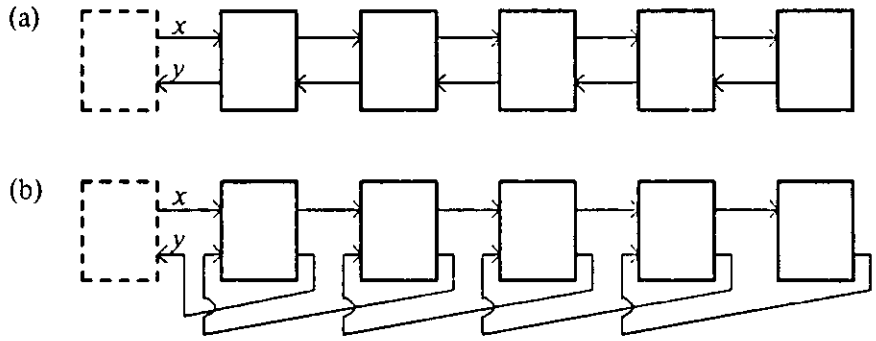

Figure 17. (a) Bi-directional systolic array, implemented by (b) the Warp *processor*

Programming the Warp processor is, however, a non-trivial task, since the architecture is highly pipelined. To ensure correctness of the code, we basically rely on software simulations at this time. One of the current research efforts at CMU is to construct compilers for machines of this type.

Future implementations of the Warp processor are expected to be much more compact than the current CMU prototype described in this paper. For example, when the CMU link and interconnection chip (LINC) [8] (that can efficiently implement data communication in a processor) becomes available in 1985, we will be able to build on a single board three or more Warp cells. Within ten years, we expect that advances in semiconductor technology will allow a Warp cell to be implemented on a single chip.

ACKNOWLEDGMENTS

The Warp project involves extensive collaborations at CMU. Onat Menzilcioglu is responsible for a large portion of the Warp architecture. C. H. Chang has developed and simulated the microcode for most of the systolic array designs described in this paper. T. M. Parng participated in the initial hardware design of the processor, while visiting CMU in the fall of 1983. Takeo Kanade and Jon Webb made many suggestions (and demands) to the architecture of the machine, regarding its use in vision applications. Andreas Nowatzyk provided valuable consultations in hardware design. As of September 1984, the total design team includes C. H. Chang and Peter Lieu (simulator), Monica Lam (assembler and compiler), Onat Menzilcioglu and Ken Sarocky (Warp cell hardware), David Sher and Jon Webb (vision code), Peter Dew (mobile robot demonstration software), and Emmanuel Arnould (interface unit).

REFERENCES

[1] Bowen, B.A. and Brown, W.R.
VLSI Systems Design for Digital Signal Processing. Volume 1: *Signal Processing and Signal Processors.*
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[2] Evans, R.A., Wood, D., Wood, K., McCanny, J.V., McWhirter, J.G. and McCabe, A.P.H.
A CMOS Implementation of a Systolic Multi-Bit Convolver Chip.
In Anceau, F. and Aas, E.J. (editors), *VLSI '83*, pages 227-235. North-Holland, August, 1983.

[3] Fisher, A.L. and Kung, H.T.
Synchronizing Large VLSI Processor Arrays.
In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 54-58. June, 1983.
The final version of the paper is appear in *IEEE Transactions on Computers.*

[4] Fisher, A.L., Kung, H.T., Monier, L.M. and Dohi, Y.
Architecture of the PSC: A Programmable Systolic Chip.
In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 48-53. June, 1983.

[5] Fisher, A.L., Kung, H.T. and Sarocky, K.
Experience with the CMU Programmable Systolic Chip.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII.* Society of Photo-Optical Instrumentation Engineers, August, 1984.

[6] Fisher, A.L., Kung, H.T., Monier, L.M., Walker, H. and Dohi, Y.
Design of the PSC: A Programmable Systolic Chip.
In Bryant, R. (editor), *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 287-302. California Institute of Technology, Computer Science Press, Inc., March, 1983.

[7] Heller, D.E. and Ipsen, I.C.F.
Systolic Networks for Orthogonal Equivalence Transformations and Their Applications.
In *Proceedings of Conference on Advanced Research in VLSI*, pages 113-122. Massachusetts Institute of Technology, Cambridge, Massachusetts, January, 1982.

[8] Hsu, F.H., Kung, H.T., Nishizawa, T. and Sussman, A.
LINC: The Link and Interconnection Chip.
Technical Report, Carnegie-Mellon University, Computer Science Department, May, 1984.

[9] Knuth, D.E.
The Art of Computer Programming. Volume 3: *Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[10] Kung, H.T.
Why Systolic Architectures?
Computer Magazine 15(1):37-46, January, 1982.

[11] Kung, H.T.
On the Implementation and Use of Systolic Array Processors.
In *Proceedings of International Conference on Computer Design: VLSI in Computers*, pages 370-373. IEEE, November, 1983.

[12] Kung, H.T., Ruane, L.M., and Yen, D.W.L.
Two-Level Pipelined Systolic Array for Multidimensional Convolution.
Image and Vision Computing 1(1):30-36, February, 1983.
An improved version appears as a CMU Computer Science Department technical report, November 1982.

[13] Kung, H.T.
Two-Level Pipelined Systolic Arrays for Matrix Multiplication, Polynomial Evaluation and Discrete Fourier Transform.
In *Proceedings of the Workshop on Dynamical Behaviour of Automata: Theory and Applications.* Academic Press, September, 1983.

[14] Kung, H.T. and Lam, M.
Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays .
Journal of Parallel and Distributed Computing 1:32-63, 1984.
A preliminary version appeared in *Proceedings of the Conference on Advanced Research in VLSI*, MIT, January 1984.

[15] Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.
A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Section 8.3, pp. 37-46.

[16] Kung, H.T. and Menzilcioglu, O.
Warp: A Programmable Systolic Array Processor.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII.* Society of Photo-Optical Instrumentation Engineers, August, 1984.

[17] Kung, H.T. and Picard, R.L.
One-Dimensional Systolic Arrays for Multidimensional Convolution and Resampling.
In Fu, King-sun (editor), *VLSI for Pattern Recognition and Image Processing*, pages 9-24. Spring-Verlag, 1984.
A preliminary version, "Hardware Pipelines for Multi-Dimensional Convolution and Resampling," appears in *Proceedings of the 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Hot Springs, Virginia, November 1981, pp. 237-278.

[18] Kung, H.T. and Yu, S.Q.
Integrating High-Performance Special-Purpose Devices into a System.
In Randel, B. and Treleaven, P.C. (editors), *VLSI Architecture*, pages 205-211. Prentice/Hall International, 1983.
An earlier version also appears in *Proceedings of SPIE Symposium, Vol. 341, Real-Time Signal Processing V*, May 1982, pp. 17-22.

[19] Monzingo, R.A. and Miller, T.W.
Introduction to Adaptive Arrays.
John Wiley & Sons, Inc., New York, 1980.

[20] Rabiner, L.R. and Gold, B.
Theory and Application of Digital Signal Processing.
Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[21] Symanski, J.J.
NOSC Systolic Processor Testbed.
Technical Report NOSC TD 588, Naval Ocean Systems Center, June, 1983.

[22] Woo, B.Y., Lin, L., Fandrianto, J. and Sun, E.
A 32 Bit IEEE Floating-Point Arithmetic Chip Set.
In *Proceedings of 1983 International Symposium on VLSI Technology, Systems and Applications*, pages 219-222. 1983.

[23] Yen, D.W.L. and Kulkarni, A.V.
Systolic Processing and an Implementation for Signal and Image Processing.
IEEE Transactions on Computers C-31(10):1000-1009, October, 1982.