MAPPING FIR FILTERING ON SYSTOLIC RINGS

Angelos Varvitsiotis
Dept. of Electrical Engineering
National Technical University of Athens
Athens, GR-15773, Greece

Sergios Theodoridis
Dept. of Computer Engineering
University of Patras
Patras, GR-16500, Greece

Rami Melhem
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

During the past decade, systolic arrays have been designed for a wide variety of scientific applications, which are based on highly parallel linear system manipulations. Partitioning and mapping of systolic algorithms has been a key issue for real implementations, in terms of both cost and manageability. In this paper, we demonstrate the mapping of triangular systolic array algorithms onto a one-dimensional ring of processors, so that the resulting architecture features an asymptotically optimal utilization factor in pipelined operation. The problems of least squares system identification and FIR filtering using QR Decomposition via Givens rotations are used as a vehicle for the demonstration of uni- and bi-directional dataflow algorithms on systolic rings.

Introduction

Systolic arrays have been introduced as a parallel architecture particularly well suited for VLSI implementation. Their main characteristics are regularity in layout and communication interconnections, simplicity of processors and synchronous operation [1,2]. A growing number of scientific computing problems that require massive parallelism in applying a restricted set of operations over large sets of data have been shown to fit in that model of computation [5]. For many such problems the resulting systolic designs, although powerful, result in arrays whose number of processors is quadratic in the size of the problem. This can be very expensive if the problem size is large. A number of techniques have therefore been developed to partition and map these algorithms onto fixed size and/or linear systolic arrays [6–8]. Traditional mapping techniques, however, result in sub-optimal processor utilization in the target arrays, especially for problems whose dependence graphs (DGs) are non-rectangular or irregular [3,12]. Recent research efforts focus on efficient dimensionality reduction and mapping techniques, such as processor clustering, refined mapping and geometric techniques [15–18]. For example, Bu et al. [16] unify several mapping techniques under a generic framework that allows designs to be expressed as a series of formal steps. Clauss et al. [15] present a geometric approach that minimizes the total execution time, resulting however in irregular target designs that suffer in terms of scalability. A case study for the matrix multiplication algorithm presented by Benaini and Tchuente [18] provides some geometrical insight regarding the internals of that specific algorithm. Varadarajan and Ravichandran [17] mention the notion of piecewise linear transformations as a means of fine-grained efficient mapping. In this paper we concern ourselves with the mapping of a class of triangular systolic arrays used for Signal Processing applications onto a ring of processors, such that the target array features an optimal utilization factor in pipelined operation. We use a technique similar

TH0382-2/91/0000/0087$01.00 © 1991 IEEE

to these of the abovementioned references, employing piecewise linear transformation functions and taking advantage of the insight provided by geometric interpretations. We use these latter as visualization tools in several steps of the mapping. We are using Least Squares System identification and FIR filtering algorithms as a vehicle for demonstrating our method.

The rest of this paper is organized as follows. A short introduction to the context of the above applications and the respective existing systolic designs is given in the second section. The third section describes our method for single dataflow arrays. The method is described in terms of projecting the existing systolic arrays onto a ring structure and determining a valid schedule so that data dependences of the original array are respected. The derivation of the projection and of a valid schedule are explained in detail, and the target data dependences are proved correct. The mapping that we are presenting features asymptotically optimal utilization in pipelined operation. In the fourth section we provide the arguments for the mapping of a bidirectional data flow array onto a double ring of processors. Hardware considerations are taken into account and discussed so that the resulting array is effectively implementable. Finally, in the fifth section we provide some concluding remarks.

Background on Signal Processing systolic applications

In this section, we consider the problems of least squares system identification and FIR filtering, both of major importance in a number of Signal Processing applications such as Communications, Control, Spectral analysis etc. [14]. The main task is to compute the LS estimates of an unknown FIR system's impulse response based on the minimization of the total squared error between the actual and the desired response signal over a given time interval. In many cases, the statistics of the pertinent process is slowly varying, thus adaptive schemes that estimate the unknown impulse vector on a sample by sample basis are of particular interest.

A very powerful and numerically sound technique for the computation of the LS estimates is the one based on QR triangularization of the input data matrix [3]. QR decomposition is achieved via a sequence of Givens rotations and the unknown coefficients are derived from the resulting triangular matrix via back substitution. An alternative technique has been suggested in [9], which is appropriate for adaptive operation and based on Givens rotations as well. In many applications, only the error between the actual and the desired response signals is of interest. It has been shown in [13] that in this case the backsubstitution step is not required and the error can be obtained directly in terms of the Givens rotation variables.

A short introduction to the derivation of the QRD method will be given below. This is not to be considered a complete description of the QRD method, so the interested reader is referred to [3,4,6,7] for a more detailed discussion. It is well known that the LS problem is equivalent to finding a solution to the overdetermined system of linear equations

$$Ax = b,$$

where A is an $m \times n$ matrix with $m > n$. This solution should satisfy the least squares criterion, that is, it should minimize

$$(Ax - b)(Ax - b)^T.$$

In general, we compute this vector as the solution to the problem

$$Rx = d,$$

where

$$\begin{bmatrix} R & d \\ 0 & d' \end{bmatrix} = Q^T \cdot [A \quad b]$$

The matrix Q represents a linear transformation, such that R is an upper triangular, $n \times n$ matrix. Q must satisfy the two following properties:

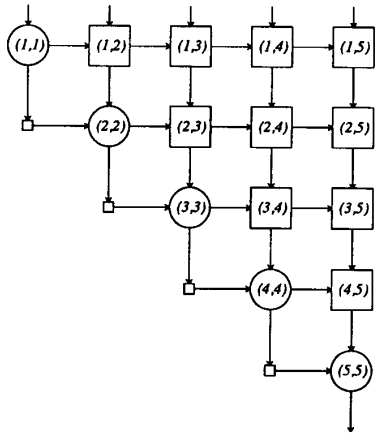
$$\|Q\|^2_{\text{eucl}} = 1,$$

and

$$Q \cdot Q^T = I,$$

where $\| \cdot \|_{\text{eucl}}$ is the Euclidian norm of a matrix, and I is the identity matrix. There are more than one methods to find such matrices, but the one mostly employed in systolic algorithms [3,4,6,13] is to compute Q as a product of 2×2 Givens rotations.

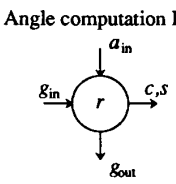
Figure 1a: The triangular systolic array for QR decomposition



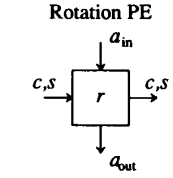
A 2×2 Givens rotation is defined as follows: consider two vectors x and y, where

$$x = (x_1, x_2, \ldots, x_k)$$
$$y = (y_1, y_2, \ldots, y_k)$$

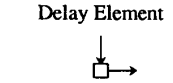
Figure 1b: The operations performed at each PE of the QR array



1. $t = \sqrt{a_{in}^2 + r^2}$
2. if $t = 0$
   $c = s = 0$
   else
   $c = r/t, \quad s = a_{in}/t, \quad r = t$
   $g_{out} = c \cdot g_{in}$



1. $a_{out} = -sr + ca_{in}, \quad r = cr + sa_{in}$



Then, we can compute two vectors x' and y', such that

$$x' = (x'_1, x'_2, \ldots, x'_k)$$
$$y' = (0, y'_2, \ldots, y'_k),$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix},$$

and

$$c = \frac{x_1}{\sqrt{x_1^2 + y_1^2}}, \quad s = \frac{y_1}{\sqrt{x_1^2 + y_1^2}},$$

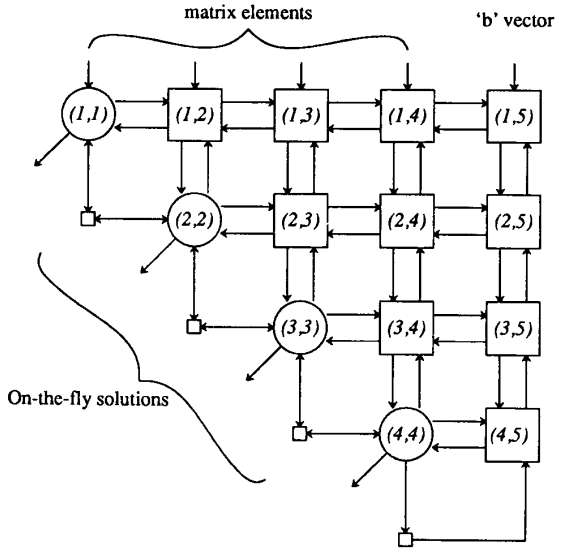
thus eliminating y_1. Assuming that vectors x and y are rows of A, we can eliminate the first element of all but the first rows of A with this method; then, we can proceed by eliminating the second element of all but the first two rows and so on, ending with a triangular matrix R.

The way a systolic array like the one shown in Figure 1 operates, is that it holds a current version of the matrix R (initially, the null matrix). Then, new rows of the data matrix A enter the top row PEs and are eliminated, by being successively rotated with all rows

of the matrix R stored in the array, thus leaving an updated version of R inside the array. According to McWhirter [13], if x is not required, and only the error is of interest, the same array can be used for the computation of this error. This is accomplished by operations performed in the diagonal elements, as shown in Figure 1b. If, on the other hand, x is required, the algorithms described in [11] can be used. As a better alternative, we have shown in [9] that the same array layout can be used to compute the solution to *all* all intermediate triangular systems $R^{(i)}x = d^{(i)}$ for all data frames i. The data flow of this array is shown in Figure 2, where the operations performed by each PE are explained in [9]. The main difference between this and other existing QR systolic arrays is that it uses bidirectional data flow, and thus each PE has to queue results for later computations.

In what follows, we will show how to project a triangular array on a ring of processors. We will also examine the case of the forementioned adaptive filtering array, whose multiple dataflow renders partitioning a non-trivial task using existing partitioning methods.

Figure 2: The dual data flow array for solution of intermediate triangular systems



Mapping the QR systolic array onto a ring

Let us first consider the QRD triangular array which features regular, unidirectional data flow. By "regular", we mean that the time/data dependency vectors at each processor are constant throughout the array and do not change with time. This assumption is realistic for other systolic matrix algorithmsas well, e.g., Gaussian elimination [15]. Another characteristic of such arrays is the non-uniformity of computations at various PEs. For example, PEs on the diagonal perform angle computation and interior PEs perform plane rotations. After projection, different computations might be mapped onto the same processor. Hence, it would be useful to think of the original triangular array as consisting of identical "omnipotent" processors, able to perform all kinds of computations needed. The actual function is selected by means of a small opcode field in the incoming data. If the number of different kinds of computations in the array is P, then $\lceil \log_2(P) \rceil$ bits are needed to select the necessary operation. In the original array, we can think of these bits as being hardwired to the desired function at each processor, whereas in the target array, the opcode will be part of the incoming data. In what follows, the actual mapping is accomplished in two steps. The first step is a *projection* from the two-dimensional original array onto the one-dimensional ring. The second step involves the derivation of a valid *schedule* for the target ring.

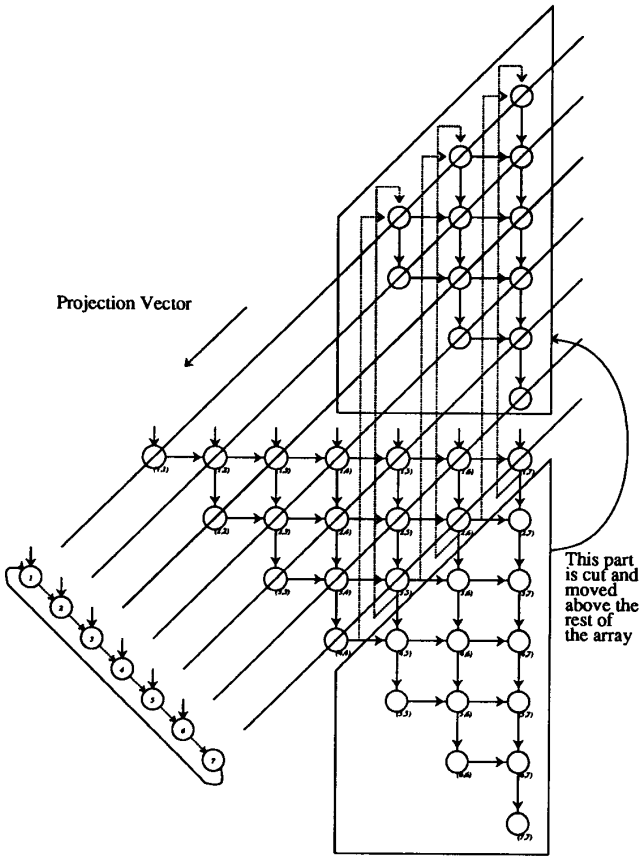
Projection: Projection is performed on the 2D systolic array and not on the initial 3D DG. The philosophy behind this choice is that the 2D array is a direct implementation of the Signal Flow Graph (SFG) derived by the most efficient [16] projection and schedule. Taking also into account the fact that further dimensionality reduction of existing arrays is not an uncommon practice [16], improving on this array seems the most promising method to follow. We project along a vector perpendicular to the *schedule vector* [10], that is, if the computation performed by PE (i,j) in the original array is mapped to the processor $\pi(i,j)$ of the ring, then $\pi(i,j) = i+j-1$ (see Figure 3). The reason for choosing this vector is that all computations performed at the same time instant on the original array are mapped onto the same processor of the ring, thus simplifying data dependences: each processor still needs to communicate only to its closest neighbors. However, if we project along all the $2n-1$ cross-diagonals that are parallel to the projection vector onto $2n-1$ processors, the load at each processor will not be uniform. For that reason, a more elaborate projection method is needed.

If the dimension of the original array is n, it will contain $\frac{n(n+1)}{2}$ PEs. The maximum number of PEs mapped onto the same processor occurs in the cross-diagonal with the elements $(1,n),(2,n-1),\ldots$. This number is $l = \lceil \frac{n}{2} \rceil$. Then, due to symmetry, for each projection line that maps k PEs onto one processor, there exists another projection line that maps m PEs onto the same processor and either $k+m = l$ or $k+m = l+1$. Moreover, the cross-diagonal excluded, there are exactly $n-1$ such pairs, with each projection line of each pair on either sides of the cross-diagonal. So the objective will be to map all PEs on each *pair of projection lines* onto the same processor, thus achieving uniform computational load per processor. Specifically, we define

$$\pi(i,j) = \begin{cases} \pi_1(i,j) = i+j-1, & \text{if } i+j-1 \leq n; \\ \pi_2(i,j) = i+j-1-n, & \text{if } i+j-1 > n. \end{cases}$$

To visualize the above steps, consult Figure 3. We first *cut* the part of the array which is below the cross-diagonal, then move them above the first row of PEs and finally we project all elements along cross-diagonals onto the same processor. Often, each element of the original triangular array has a small amount of memory it operates on. For example, in the QR factorization array, each PE (i,j) stores the element $r_{i,j}$ of the triangular

Figure 3: Projecting the triangular array onto a ring



matrix R produced so far. When projecting many PEs to one processor, we have to preserve these stored data, typically in a circular queue. In this queue, together with the data items that constitute the original contents of each PE, it is convenient to store as well the *opcode* that defines the operation of the PE on these data items, as defined

above. So, the discipline of each resulting processor is now to get a set of data items and the respective opcode from the circular queue, communicate with its neighbors get any necessary input (or, depending on the opcode, get external inputs) compute the new data set, send any results to its neighbors and append the computed data items at the end of the queue. At this point we note that the above projection results in an optimal schedule only for n odd; for n even, the projection method that guarantees uniform load is somewhat different. The schedule derivation methodology for n even is similar to what follows, however, due to lack of space, in the following we will only describe the case of odd n.

Schedule: Two factors are of major importance in deriving a schedule. The first is to preserve data dependences of the original array. The second is to derive an optimal schedule for *pipelined* operation, i.e., a schedule that will allow processors of the resulting ring to operate on successive instances of the same problem optimizing their utilization. This is particularly needed in Least Squares minimization, where the matrices that have to be triangularized are typically long ($m \gg n$). In the QRD context, the rows of the matrix A represent the successive instances of the same problem, that of annihilation of a vector through a triangular matrix.

A different problem encountered in scheduling is the use of buffering. Buffers should be inserted between each two processors to hold results produced by one processor until they are used by the next. With respect to buffering we are concerned with the amount of memory needed for buffers and the discipline (FIFO, LIFO, or other) according to which data are stored in the buffer. These problems will be examined in turn after the mapping methodology has been discussed.

The schedule of the original systolic array can be defined as a function f from the set of *space indices* $\{(i,j)\}$ of a computation to a set of *time indices* $\{t_{i,j}\}$, such that $t_{i,j} = f(i,j)$ is the time at which the computation (i,j) takes place. A *pipelined schedule* is a function from the set of triples $\{(i,j,k)\}$, where i and j are the space indices of computations and k is the *data set index* of the computation, onto the set of time indices $t_{i,j,k}$. The idea here is that the PE (i,j) operates on successive rows of the matrix A, called *data sets*, which are indexed by k. A *pipelined* schedule maps computations of data sets onto available time slots. In the original array, a schedule derived using standard data dependence analysis methods [5,10] is used. This schedule is $t_{i,j,k} = f(i,j,k) = i+j+k-2, 1 \leq i,j \leq n, 1 \leq k \leq m$. This schedule is described in the bibliography in terms of *computational hyperplanes*. The intersection of each hyperplane with the plane of the systolic array defines the PEs that are simultaneously performing operations on the same data set. Hyperplanes are parallel to each other and at each time step move one unit 'east' and 'south'. The direction of this displacement is constant throughout the array and can be described by a *schedule vector*. The magnitude of the schedule vector represents the amount of displacement of hyperplanes for each time unit.

In the above context, mapping a computation performed by a triangular systolic array onto a ring of processors is just an alternative pipelined schedule, such that data dependences of the original computation are preserved. This is denoted by a function g from the set of space/dataset indices $\{(i,j,k)\}$ to the set of time indices $\{\tau_{i,j,k}\}$ such that $\tau_{i,j,k} = g(i,j,k)$. For the function g to represent a valid schedule, it must satisfy the following conditions for all i,j,k,i',j',k':

$$g(i,j,k) > g(i',j',k'), \quad \text{if} \quad f(i,j,k) > f(i',j',k'); \tag{1.a}$$

$$g(i,j,k) \neq g(i',j',k), \quad \text{for} \quad \pi(i,j) = \pi(i',j'),\ (i,j) \neq (i',j'); \tag{1.b}$$

$$g(i,j,k) \neq g(i',j',k'), \quad \text{for} \quad \pi(i,j) = \pi(i',j'),\ (i,j) \neq (i',j') \text{ and } k \neq k'. \tag{1.c}$$

Condition (1.a) implies that data dependences will be satisfied in the ring if they are satisfied in the original array. Condition (1.b) implies that, for the same processor of the ring and the same data set, no overlap arises according to the schedule. Condition (1.c) implies that, for the same processor of the ring and different data sets, there will be no overlap as a result of the schedule.

We now want to derive a linear function g, that is, $g(i,j,k) = c_1 i + c_2 j + c_3 k + c_4$, such that g represents a valid schedule. Condition (1.a) is satisfied for c_1, c_2 and c_3 positive and c_4 such that $c_1 + c_2 + c_3 + c_4$ is positive. For the schedule to reduce as much as possible the overall execution time, $c_1 \ldots c_4$ should take minimum values. Moreover, condition (1.b) requires $c_1 \neq c_2$. Thus, two alternatives are possible: either $c_1 = 1$ and $c_2 = 2$, or $c_1 = 2$ and $c_2 = 1$. The first alternative corresponds to scheduling first the computations performed in the first row of the original array, then those of the second row, and so on, thus we call it "First Row First" (FRF). The second one corresponds to the same scheme columnwise, so we call it "First Column First" (FCF). Both schemes can be shown valid if $\pi(i,j)$ is a continuous linear function. However, our definition of π is by cases, so π is only continuous in its two sub-domains. Thus, before deriving values for c_3 and c_4, we will have to check the properties of our schedule separately for each sub-domain of $\pi(i,j)$.

Let us apply the FRF scheme to the first half of the original array, that is, to PEs (i,j) with $1 \leq i \leq (n+1)/2, 1 \leq j \leq n+1-i$. Then, condition (1.c) is satisfied by $c_3 \geq (n+1)/2$. We choose $c_3 = (n+1)/2$, in order to satisfy the abovementioned minimality criterion. Finally, adjusting c_4 so that g is always positive, we get

$$g_1(i,j,k) = 2i + j + \frac{n+1}{2}(k-1) - 2, \quad \text{for} \quad 1 \leq i \leq \frac{n+1}{2},\ 1 \leq j \leq n-i+1.$$

	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
14			$22^{(3)}$	$14^{(3)}$			$35^{(2)}$	$27^{(2)}$	$19^{(2)}$
13			$13^{(3)}$			$34^{(2)}$	$26^{(2)}$	$18^{(2)}$	$\mathbf{55^{(1)}}$
12		$12^{(3)}$			$33^{(2)}$	$25^{(2)}$	$17^{(2)}$		$\mathbf{46^{(1)}}$
11	$11^{(3)}$				$24^{(2)}$	$16^{(2)}$		$\mathbf{45^{(1)}}$	$\mathbf{37^{(1)}}$
10				$23^{(2)}$	$15^{(2)}$		$\mathbf{44^{(1)}}$	$\mathbf{36^{(1)}}$	$\mathbf{28^{(1)}}$
9			$22^{(2)}$	$14^{(2)}$			$\mathbf{35^{(1)}}$	$\mathbf{27^{(1)}}$	$\mathbf{19^{(1)}}$
8			$13^{(2)}$			$\mathbf{34^{(1)}}$	$\mathbf{26^{(1)}}$	$\mathbf{18^{(1)}}$	
7		$12^{(2)}$			$\mathbf{33^{(1)}}$	$\mathbf{25^{(1)}}$	$\mathbf{17^{(1)}}$		
6	$11^{(2)}$				$\mathbf{24^{(1)}}$	$\mathbf{16^{(1)}}$			
5				$\mathbf{23^{(1)}}$	$\mathbf{15^{(1)}}$				
4			$\mathbf{22^{(1)}}$	$\mathbf{14^{(1)}}$					
3			$\mathbf{13^{(1)}}$						
2		$\mathbf{12^{(1)}}$							
$t=1$	$\mathbf{11^{(1)}}$								

Figure 4: Example for the FRF mapping

In order to help visualizing this schedule, we denote by $ij^{(k)}$ the computation performed by the PE (i,j) of the original array on the data set k. Figure 4 shows an example of

FRF for $n = 9$. As it can be seen in this figure, the rest of the computations should be scheduled in the triangular 'free time-slot' area between two successive data set frames. Unfortunately, the FRF scheme does not work for this area. The reason is that the remaining computations cannot be scheduled to fit inside this area without violating condition (1.c). In order for the remaining computations to fit in the free slot, they have to be skewed timewise, resulting in a "skewed" FRF. This latter schedule, however, violates condition (1.a), that is, it does not preserve the data dependences of the original array.

This problem can be rectified if an FCF schedule is used for PEs (i, j) with $(n+1)/2 < j \leq n, n+1-j < i \leq n$. The values produced by such a schedule are (with appropriate values for c_3 and c_4) complementary to those of FRF, that is, FCF computations fit exactly in the free time slot area mentioned in the previous paragraph. To see this, using the same method as before, we derive the minimum c_3, which is equal to $(n+1)/2$, as for FRF. The value of c_4 determines the starting time for the entire set of operations that correspond to the second half of the original array, and should be set so that conditions (1.a) and (1.c) are satisfied. Let us, for the sake of derivation of this value, fix (i, j) so that $\pi(i, j) = 1$. From the definition of π, this means that either $i = j = 1$, and $g(i, j, k) = g_1(1, 1, k) = 1 + (n+1)(k-1)/2$, or $i + j = n + 2$, and $g(i, j, k) = g_2(i, j, k)$, where

$$g_2(i, j, k) = i + 2j + \frac{n+1}{2}k + c_4.$$

In order to satisfy condition (1.c), we have to define c_4 so that

$$\forall k > 0 \quad \exists k' > 0: \quad \forall i, j \text{ with } i + j = n + 2: \\ g_1(1, 1, k') < g_2(i, j, k) < g_1(1, 1, k' + 1).$$

The above is clearly satisfied by $c_4 = l(n+1)/2$. Our last step is to fix a minimal value for l, so that condition (1.a) is satisfied. This we do by examining the data dependences of the computations that belong to the first frame ($k = 1$) and are scheduled using FCF. These depend immediately on computations (i, j) with $i + j = n + 1$. The last such computation happens for $i = j = (n+1)/2$, at time $g_1((n+1)/2, (n+1)/2, 1) = 3(n+1)/2 - 2$. The first computation that uses its result must be, according to FCF, for $i = (n+1)/2, j = 1 + (n+1)/2$. We can fix this to happen at time $2 + 3(n+1)/2$, by setting $l = -1$. This corresponds to scheduling the FCF computations for the first frame exactly after the fourth FRF frame, since $g_1(1, 1, 4) = 1 + 3(n+1)/2$. That is, the first three FCF frames are left empty; we will use this result when we discuss buffering. The function g_2 is therefore defined as

$$g_2(i, j, k) = i + 2j + \frac{n+1}{2}(k-1), \quad \text{for} \quad \frac{n+1}{2} < j \leq n, \; n+1-j < i \leq n.$$

and g is defined as

$$g(i, j, k) = \begin{cases} g_1(i, j, k), & \text{for } 1 \leq i \leq \frac{n+1}{2}, \; 1 \leq j \leq n - i + 1; \\ g_2(i, j, k), & \text{for } \frac{n+1}{2} < j \leq n, \; n+1-j < i \leq n. \end{cases}$$

In order for the above proof to be complete, we must show that g satisfies the three conditions (1.a–1.c) for all i, j, k within the appropriate ranges. This last proof (by contradiction) is straightforward but lengthy, and thus it is omitted due to space limitations. Using this combination of FRF and FCF, pipelined operation can be achieved and data

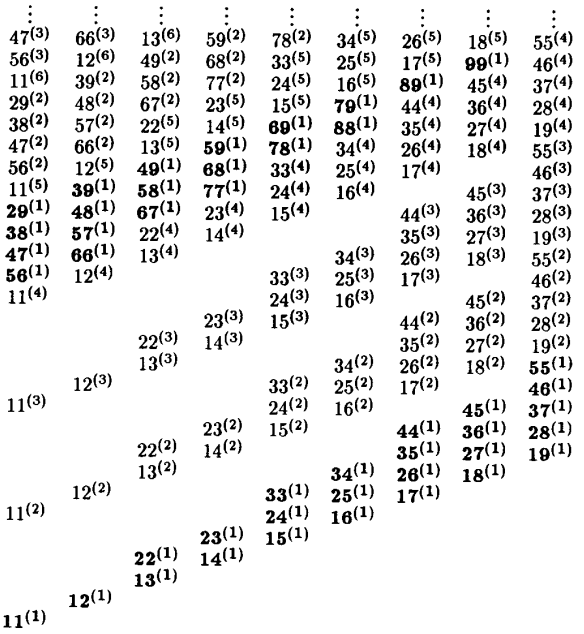


Figure 5: The combination of FRF and FCF yields a valid mapping (the computations of frame 1 are shown in boldface)

dependences are preserved. Moreover, it can be shown that g asymptotically fills all the available time slots. That is, the utilization of the processors is asymptotically (i.e., excluding the first and last few frames) equal to one, which is an optimal result. An example of the resulting scheduling for $n = 9$ is shown on Figure 5. The computations of the first frame are shown in bold typeface. These are split by the projection and schedule functions into two parts. The first part, on the bottom of Figure 5, contains the computations scheduled using the FRF function g_1. The second part, which looks like being "wrapped around" the ring, involves the computations scheduled by the FCF function g_2.

With respect to scheduling, a factor that deserves some attention is buffering. We can see that, excluding the wrap-around communication step, all results that need to be communicated from a computation to its dependent computations will be consumed before the same computation is repeated on the next frame of data. This is in general a property of the FRF and FCF mapping schemes. With this observation we can show that a single buffer slot for each result that has to be communicated to a neighbor

computation and a FIFO buffering scheme provide an adequate and simple to implement communication method. Unfortunately, this is not true for the wrap-around step. There, since the scheduling scheme changes from FRF to FCF, the computation results are consumed in the opposite order from the order they are produced in, and, since two more frames of results have to be stored between the time the first result is produced and the time it is consumed, three buffer slots for each result communicated between two computations are needed. We can show that three buffer slots are necessary and sufficient. For a proof, see the derivation of the functions g_1 and g_2. In addition to the extraneous buffering space needed, a LIFO buffering scheme has to be adopted for the wrap-around step for results of the same frame, whereas a FIFO scheme must be adopted between results of successive frames. The above buffering discipline can be implemented using a circular queue as follows: the producer puts continuously results in the queue, moving its write pointer, say, clockwise. The consumer, starting at the appropriate position of the queue, consumes $(n+1)/2-1$ items, each time moving its read pointer one position anti-clockwise. After consuming the $(n+1)/2^{\text{th}}$ result, the consumer moves the read pointer $2(n+1)/2-1$ positions clockwise, thus starting with the first result of the next frame the producer put in the queue. This scheme, although seemingly complicated, provides a simple way of LIFO communication between the last and the first processor of the ring for the wrap-around communication step.

Mapping the bidirectional data flow array onto a double ring

In order to map the bidirectional array of Figure 2 onto a ring architecture, we use two rings. The PEs of the original systolic array are projected onto the same processors in both rings. In each PE, we separate the instructions that use the reverse data flow, and assign them to the processors of the second ring. This is done so that results that have to be passed between two computations originally performed at the same PE through the internal queue of each PE, now have to traverse the simplest possible communication path between the two rings, i.e., from a processor to the processor opposite to it.

In the array of Figure 2, each PE has to store results generated by the forward data flow operations in an internal queue, so that they can be used by the reverse data flow computations. These queues are now external to the processors, connecting corresponding processors of the two rings. Therefore, there will be n distinct queues for each processor. However, only one processor from each ring is using one queue at a time, so a single output (for the first ring) or input (for the second one) is needed. We might use some appropriate delivery mechanism for choosing the right queue to place a result in or extract it from.

In order to derive an optimal schedule for the operations mapped to the second ring, we proceed as follows: Formally, the schedule for the reverse dataflow operations is described by a function f' analogous to the function f of the previous section, where:

$$f'(i,j,k) = -i-j+k+4n-1 \quad \text{for } 1 \le i,j \le n, 1 \le k \le m.$$

We will describe the function g' to be used for scheduling these operations in analogy with the function g of the previous section. A valid schedule function g' should satisfy the conditions (1.b) and (1.c), whereas condition (1.a) is substituted by

$$g'(i,j,k) > g'(i',j',k') \quad \text{if} \quad f'(i,j,k) > f'(i',j',k').$$

Using a similar arguments as in the previous section, we result in "reverse" editions of FRF and FCF functions g'_1 and g'_2. Reverse FCF should be used for the second part of

the original array and reverse FRF for the first part. Reverse FCF can be described by a function $g'_2(i,j,k)$, where

$$g'_2(i,j,k) = -i - 2j + \frac{n+1}{2}k + c'_4.$$

The minus signs for i and j result from the respective signs in f'. The constant c'_4 determines the starting time of the entire set of these operations. It should be set so that the first reverse dataflow operation takes places exactly after the last forward dataflow operation, i.e.,

$$g'_2(n,n,1) = g_2(n,n,1) + 1.$$

Therefore, c'_4 takes the value $6n - (n+1)/2 + 1$, and

$$g'_2(i,j,k) = -i - 2j + \frac{n+1}{2}(k-1) + 6n + 1$$

The reverse FRF function g'_1 can be derived with the same method to

$$g'_1(i,j,k) = -2i - j + \frac{n+1}{2}(k+11) - 3$$

The function g' is therefore defined as

$$g'(i,j,k) = \begin{cases} g'_1(i,j,k), & \text{for } 1 \le i \le \frac{n+1}{2},\ 1 \le j \le n-i+1; \\ g'_2(i,j,k), & \text{for } \frac{n+1}{2} < j \le n,\ n+1-j < i \le n. \end{cases}$$

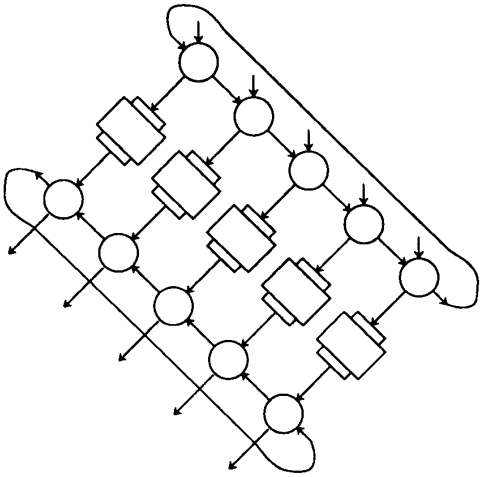
In Figure 5, the second set of operations would start right after $99^{(1)}$, on the respective processor of the second ring. It is easy to verify on that figure that all results needed for a computation in the second ring will be present by the time processor 9 of that ring will start to operate. The reverse dataflow schedule lends itself equally well to pipelined operation, since it can be shown to fill asymptotically all available time slots, thus yielding an optimal processor utilization factor.

With respect to buffering results between the two arrays, a problem arises, namely that the results from each queue have to be consumed in the opposite order they are produced in. This brings in the same kind of considerations about circular queues as in the second section. Another possibility is that a small memory be used, together with a suitable address generation scheme at both ends (producer–consumer). The reason for doing so would be to avoid using specialized hardware between each two corresponding processors of the two rings. In the original array, the queue at each PE has size $O(n)$ [9], so the amount of specialized hardware would be $O(n^2)$ per processor. Alternatively, attaching a moderate-sized, dual interface memory to each processor is a cheaper and easier solution. To visualize this latter architecture, consult Figure 6, where the two rings are shown, together with their interconnecting memory modules. The two rings can be merged into one single, dual dataflow ring, whose operating details will be exactly the same as those of the double ring discussed in this section.

Concluding remarks

A mapping scheme from triangular systolic arrays with simple unidirectional or bidirectional data flow onto systolic rings has been developed and discussed. In particular, the examples of QR decomposition and its extension to linear filtering have been projected

Figure 6: Mapping the dual dataflow array of Fig. 2 onto a double ring architecture



and mapped onto ring architectures, achieving maximal processor utilization. Although the results in this paper are confined to these two examples, there is a wider class of systolic algorithms for which non-rectangular systolic arrays have been introduced. The technique described herein could be extended to cover these classes of algorithms, provided that there is a geometrical transformation such that after projection onto a linear structure all processors of that structure receive equal computational loads.

References

[1] Kung, H.T. and C.E. Leiserson, "Systolic Arrays for VLSI," *Introduction to VLSI Systems*, Mead and Conway, Eds., Reading, MA: Addison-Wesley, 1980, Sect. 8.3

[2] Kung, H.T., "Why Systolic Architectures," *IEEE Computer*, V.15, Jan 82, pp.37–46

[3] Gentleman, W.M., and H.T. Kung, "Matrix Triangularization by Systolic Arrays," *Proc. SPIE Real Time Processing IV*, 1981, pp.19–26

[4] Bozanczyk, A., R.P. Brent and H.T. Kung, "Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors," *SIAM J. Sci. Stat. Computing*, V.5, 1984, pp.95–104

[5] Moldovan, D.I., and J. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. on Computers*, V.C-35, No.1, Jan 1986

[6] Chuang, H.Y.H., L. Chen and D. Qian, "A Size-Independent Systolic Array for Matrix Triangularization and Eigenvalue Computation," *IEEE Trans. Circuits, Systems*

and Signal Processing, V.7, No.2, 1988, pp.173–189

[7] Torralba, N. and J.J. Navarro, "A One-Dimensional Systolic Array for Solving Arbitrarily Large Least Mean Square Problems," *Proc. IEEE Intl. Conf. on Systolic Arrays*, 1988, pp.103–112

[8] Moreno, J.H., and T. Lang, "Arrays for Partitioning Matrix Algorithms—Tradeoffs Between Cell Storage and Cell Bandwidth," *Proc. SPIE Real Time Signal Processing XI*, V.977, 1988, pp.156–169

[9] Varvitsiotis, A.P., and S. Theodoridis, "A Pipelined Structure for QR Adaptive LS System Identification," *to appear in IEEE Trans. Acoust., Speech and Signal Processing*, Aug 1991

[10] S.Y. Kung et al., "Wavefront Array Processors—Concept to Implementation," *IEEE Computer*, Jul 1987, pp.18–33

[11] Nash, J.G., and S. Hansen, "Modified Fadeeva Algorithm for Concurrent Execution of Linear Algebraic Operations," *IEEE Trans. on Computers*, V.37, No.2, Feb 1988, pp.129–136

[12] Guibas, L.J., H.T. Kung and C.D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," in *Proc. Caltech Conference on VLSI*, L.A., 1979

[13] Mc Whirter, J.G., "Recursive least-squares minimisation using a systolic array," in *Proc. SPIE Real Time Signal Processing VI*, V.431, 1983

[14] Haykin, S. *Adaptive Filter Theory*, Prentice-Hall, 1986

[15] Clauss, Ph., C. Mongenet and G.R. Perrin, "Calculus of Space-Optimal Mappings of Systolic Algorithms on Processor Arrays," in *Proc. Intl. Conf. Appl. Specific Array Processors*, IEEE, 1990, pp. 4–18

[16] Bu, J., E.F. Deprettere and P. Dewilde, "A Design Methodology for Fixed-Size Systolic Arrays," in *Proc. Intl. Conf. Appl. Specific Array Processors*, IEEE, 1990, pp. 591–602

[17] Varadarajan, R., and B. Ravichandran, "Refining Algorithm Mappings for Linear Systolic Arrays," in *Proc. 5th Intl. Parall. Process. Symp.*, IEEE, 1991, pp. 151–154

[18] Benaini, R., and M. Tchuente, "Matrix Product on Modular Linear Systolic Arrays," in *Parallel and Distributed Algorithms*, Elsevier, 1989, pp. 79–88