Constraint Directed CAD Tool For Automatic Latency-Optimal Implementation of 1-D and 2-D Fourier Transforms

J. Greg Nash, Centar

ABSTRACT

A specialized CAD tool is described that will take a user's high level code description of a non-uniform affinely indexed algorithm and automatically generate abstract latency-optimal systolic arrays. Emphasis has been placed on ease of use and the ability to either force conformation to specific design criteria or perform unconstrained explorations. How such design goals are achieved is illustrated in the context of LU decomposition and the matrix Lyapunov equation. The tool is then used to generate new 1-D and 2-D hardware efficient systolic arrays for the discreet Fourier transform that take advantage of the use of the radix-4 matrix decomposition.

Keywords: signal and image processing, FPGA CAD tool, systolic, DFT, FFT, parallel algorithm, algorithm mapping

1 INTRODUCTION

Systolic arrays have been shown to be suitable for many structured applications because they provide a solution to meeting latency and throughput requirements in many embedded applications[1]. The original systolic concept was that an array design could be rapidly implemented via ASIC technology and then directly inserted into a system; however, it has been recognized that system/architectural issues play a major role in constraining circuit design decisions. Consequently, it is important that a systolic design tool be able to adapt to such considerations. For example, a system design might require that systolic inputs and outputs occur at array boundaries, that all intermediate computations are performed internally in the array, and that any coefficient matrices used remain in a fixed position internally so that external memory accesses are not necessary.

Alternatively, new FPGA based hardware technologies provide enormous flexibility in making design choices. In this case a possible design scenario might have the system architects request all interesting systolic designs. With such feedback it might be possible to make early decisions that would more efficiently focus their FPGA hardware options.

Our symbolic parallel algorithm development environment (SPADE) is constructed to allow a designer to easily and rapidly explore the design space of systolic array implementations so that system tradeoffs can be cost-effectively analyzed at an early stage in system design from both the design points of view described above[2,3]. The intention is to allow a user to specify his algorithm using traditional high-level code, set some architectural constraints and design objectives and then view the results in a meaningful graphical format.

Here we describe how SPADE's constraints and optimization criteria can be used to direct selection of different array designs. LU factorization and the Lyapunov matrix equation were chosen as examples because they do a good job illustrating such features. In particular LU factorization shows how code changes can be made to improve array designs and the Lyapunov matrix equation shows that being able to examine sub-optimal solutions is important. Most of the array designs generated here by SPADE are new.

Finally, SPADE is applied to searching the solution space for new systolic array implementations of the 1-D and 2-D discreet Fourier transform (DFT). We show that the array designs SPADE generates are hardware efficient compared to previous systolic versions.

2 SPADE DESCRIPTION

SPADE takes as input a coded form of a system of non-uniform affine recurrence equation

$$\begin{aligned} &w_1(A_1(I)) = g(\ldots w_i(B_{i1}(I)),\ldots) \qquad for\,all\,I\,in\,\mathrm{I}_1 \\ &\quad\ldots \\ &w_n(A_n(I)) = f(\ldots w_i(B_{in}(I)),\ldots) \qquad for\,all\,I\,in\,\mathrm{I}_n \end{aligned} \tag{1}$$

where *f,g* represent the functional variable dependencies, I_j is the index range for equation *j*, w_i is one of the algorithm variables and the affine indexing functions are $A(I) = AI + a$, $B(I) = BI + b$, where *A/a,* and *B/b* are integer matrices/vectors. For each algorithm variable x SPADE finds an affine transformation, T, such that $T(x) = T_x(A_x I + a_x) + t_x$, where T_x is a matrix and t_x is a vector. Thus, T_x and t_x represent affine "mappings" from one index set to another index set, where the latter index set includes one index representing time with the remaining indices representing spatial coordinates. Every variable and its associated calculation receive a unique mapping to "space-time". The projection of the space-time structure along the time axis produces a systolic array after an elemental processing element (PE) is associated with each grid coordinate. In addition to the outputs *T,t* for each of the algorithm variables, SPADE computes a set of vectors indicating the direction of data flow between PEs for each dependency in the algorithm. SPADE uses an exhaustive search process that examines all combinations of possible transformations to find designs that result in the minimum computation time, defined as the maximum difference between temporal extrema in space-time (latency optimal).

Input to SPADE is in the form of high-level code based on an Algol-like language that is a subset of the Maple™ programming language. It is possible to go directly from a scientific expression to equivalent code because the Maple language provides special syntax options for the commutative and associative operators (multiply, add, minimum, maximum) this tool supports. For example, the matrix-matrix algorithm can be written in this language directly as

```
      for i to N do for j to N do C[i,j] := add(A[i,k]*B[k,j],k=1..N) od od;          (2)
```

where the Maple "add" construct performs a mathematical summation. Maple treats the loop structure in traditional way, but SPADE does not use any lexicographic interpretation of the loops; rather it uses the loop limits only to determine the index space of the inner statement body. When conditionals are used in code input to SPADE, they override the loop limits in determining the index space.

2.1 Secondary objective function criteria

To provide the designer perspective in choosing his systolic array options, SPADE includes three secondary objective functions that are used to select designs: minimum PE area, maximum “regularity”, and minimum bandwidth. That is, after SPADE finds all minimum latency solutions, one of these secondary optimization criteria can be used to choose sub-solutions from among these. Although the different secondary objective criteria hope to identify specific architecture types, a cross purpose is just to identify very different, but “interesting” designs.

Maximum regularity: This criterion is intended to select designs that are spatially uniform, and thus easiest to build. Because specification of what constitutes a regular array design is subjective, SPADE uses four different criteria, the relative weightings of which are user programmable.

- *Dependency relations between variables:* Because a systolic design is inherently pipelined, it is possible to find many different alignments between variables. Therefore, the regularity criterion penalizes designs where variable surfaces are not optimally placed.
- *Boundary I/O*: Designs that don't put variable inputs and outputs on the edge of an array are penalized.
- *Number and orientation of variables that are time-aligned:* This criterion penalizes an array design for each variable that is time aligned (Section 2.2), because this imposes a non-uniformity in the overall design. That is, a problem size amount of memory, O(N) per PE, is required for each PE and special data buffering is required. A design with a time-aligned variable that is not orthogonal to the x/y axes is further penalized.
- *Interconnection network topology:* The topology desired is one that minimizes the number of different interconnects and prefers orthogonal data movement. An array design is penalized for each different data flow and doubly penalized for each different diagonal flow of data.

Minimum Bandwidth: Given that it is not unusual for a systolic design to experience more that 10 unique flows of data associated with variable dependencies, it would not be surprising to see such designs wiring limited. When this happens it might be preferable to select a design that minimizes the need for nearest neighbor interconnection bandwidth. The minimum bandwidth secondary objective function achieves this by doing an analysis of the spatial extent of the variable dependencies. For 3-D space-time each dependency corresponds to a convex polygon in the spatial plane after a projection along the time axis. To characterize the bandwidth load of a nearest neighbor wiring network, an adjacency matrix is created with rows and columns representing the different variable dependencies. An overlap between polygons associated with two dependencies results in an entry in the adjacency matrix. These entries are normalized by rate of data movement of the different dependencies. After the adjacency matrix has been filled, the set of dependencies where maximum overlap occurs is used as an estimate of the bandwidth required for that array design. Of course this selection criteria tends to find arrays that are larger in area due to the necessity to spatially separate dependency flows.

2.2 Constraints

To further facilitate identification of suitable array designs, SPADE allows the designer to eliminate from the search space designs that fail certain constraint tests that define architectural features. For example, if it is desired that a variable appear only at the PE array edge, then the position points in space-time of that variable's elements, e.g., *A[i,k]* in (2), must be such that the unit time vector u_t is in the plane of these position points (hence *A*'s projection onto the PE array is a line). This "time align" constraint requires that the normal n_a to the plane of *A* satisfies $u_t \bullet n_a = 0$. (If it is desired that a variable not project to a line, then a constraint can be set that requires $u_t \bullet n_a \neq 0$.) Also, for the projection of *A* to be at the edge of the PE array it must constrained to not be within the convex hull of the polygon defining the PE array. Constraints like these are useful because they provide the designer with only the solutions in which he is interested, while considerably limiting the space of possible systolic designs that SPADE has to explore

Constraint settings also exist to selectively prevent overlap between variables in space-time. These are best used interactively after viewing space-time outputs to force more desirable configurations. For example, sometimes it is desired that certain types of variables be allowed to overlap in order to prevent good designs from being eliminated; in other cases more regular designs can be obtained disallowing overlaps.

2.3 LU decomposition

LU decomposition is addressed here because it serves as a useful benchmark in comparing previous tool and methodology efforts since many of these use it as an example[1]. In particular it shows how mathematical solutions can be taken directly as an input and how code changes and constraints lead to new designs. Starting with a decomposition of the form

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

it can be determined from inspection that

$$l_{11} = a_{11};\ \ l_{21} = a_{21};\ \ l_{31} = a_{31};\ \ u_{12} = a_{12} / l_{11};\ \ u_{13} = a_{13} / l_{11}$$
$$and\ for\ i \geq j, j > 1, i \leq 3: \quad l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}; \quad for\ j > i, i > 1, j \leq 3: \quad u_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}) / l_{i,i} \tag{3}$$

from which it is possible to go directly to the code form

```
for i to N do
  for j to N do
   if j=1 and i>=1 and i<=N then l[i,j]:=a[i,j];
   elif i=1 and j>1 and j<=N then u[i,j]:=a[i,j]/l[i,i] fi;                    (4)
   if i>=j and j>1 and i<=N then l[i,j]:=a[i,j]-add(l[i,k]*u[k,j],k=1..j-1) fi;
   if j>i and i>1 and j<=N then
       u[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..i-1))/l[i,i]
   fi;
  od
od;
```

where the only semantic change has been to use the Maple "add" construct in the place of the mathematical summation sign and replacing "3" with the "problem size" parameter N This code is saved as a text file and read into SPADE as a text file at the beginning of processing by the parser.

In this first mapping example the architectural constraints were set to require input (variable *a*) to take place from the systolic array boundary and look for the minimum time latency solution that has the least PE array area. For this case SPADE found six unique mappings with the minimum time latency of 3N-3. Each array design was triangular in shape.

Four of these had less desirable "diagonal" interconnection patterns between PEs; SPADE output for two designs that have the more desired "rectangular" interconnect structure are shown in Figs. 1a and 1b.

Here, as with later mappings, the x and y axes represent the spatial coordinates of a 2-D mesh grid with the array of PEs embedded in this grid at intersection points. The grid above is NxN in size and the embedded systolic array is triangular with an area defined by the N(N+1)/2 PEs. The different shadings correspond to regions associated with the different algorithm variables *a*, *l* and *u* and are labeled as such.

SPADE also provides a space-time view of each algorithm array mapping solution, one of which is shown in Fig. 1c and corresponds to the spatial-only view in Fig. 1b. This diagram indicates what computational activity is taking place for each PE at array location *x,y* for each time *t*. The space-time view is shown from two different perspectives in order to help in its interpretation (in the SPADE environment this view can be easily manipulated along all axes in real-time). Fig. 1c is more complex because it shows additional "intermediate" variables, *IM1* and *IM2* (*IM1[i,j,k]=l[i,k]*u[k,j]*; *IM2* has the same dependence on *l* and *u*, but a different index domain and both IM1 and IM2 are polytopes in the space-time view) which are created automatically in the parser and are there to keep running sums associated with the summations in (3). Projections of intermediate variables onto the spatial plane are not shaded.

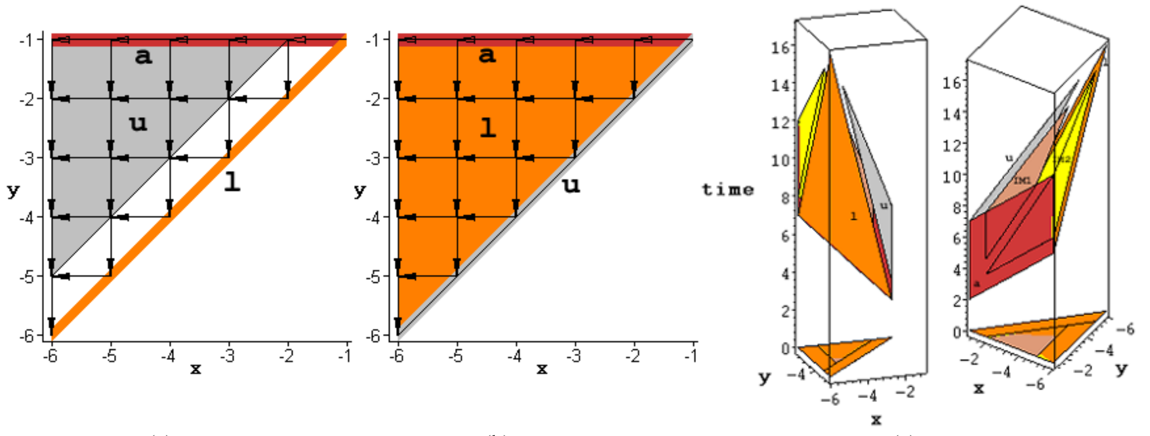


Figure 1. Here (a) and (b) are two minimum area array designs (N=6) and (c) is the position of algorithm variables *u*, *l*, and *a* in space-time shown for two different orientations of the design (b).

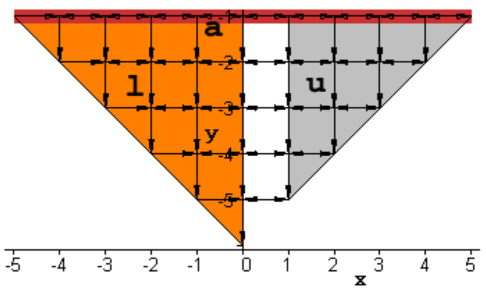
The space-time view is an important output because imparts a good deal more information than the spatial-only view. For example it shows where and when array activity associated with the different algorithm variables takes place, it provides a visible view of 3-D data flow between algorithm variables, it provides an estimate of potential throughput, it imparts a rough estimate of how efficiently PEs are used by what percent of the total space-time volume is occupied by polytopes/polygons, and it provides insight as to where to look for design modifications.

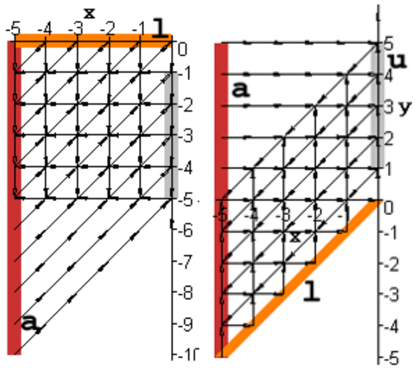
Use of the maximum regularity secondary objective with *a* still constrained to an array edge results in the single optimal design with latency 3N-3 and area N^2 shown in Fig. 2a. Here neither *l* or *u* is time aligned as in Fig. 1a and 1b because the regularity criterion tries to avoid creating memory structures in the array. If *l*, *u* and *a* were all constrained to appear along the array boundary, two minimum area solutions are found (Fig. 2b). However, the latency of the algorithm with this new constraint increased to 4N-4 and area to N(3N-1)/2.

Although the array designs in Fig. 1a and 1b look equivalent, there is a significant difference between the two in terms of hardware usage. From `(4)` it can be seen that the line

```
if j>i and i>1 and i<=N and j<=N then
     u[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..i-1))/l[i,i]             (5)
fi;
```

implies that at each point in the *i,j* index range a division is necessary to compute `u[i,j]`. Consequently, in Fig. 1a a triangular array of dividers corresponds to the internal shaded region labeled *u*, whereas in Fig. 1b only a linear array associated with the linear projection of *u* is required.



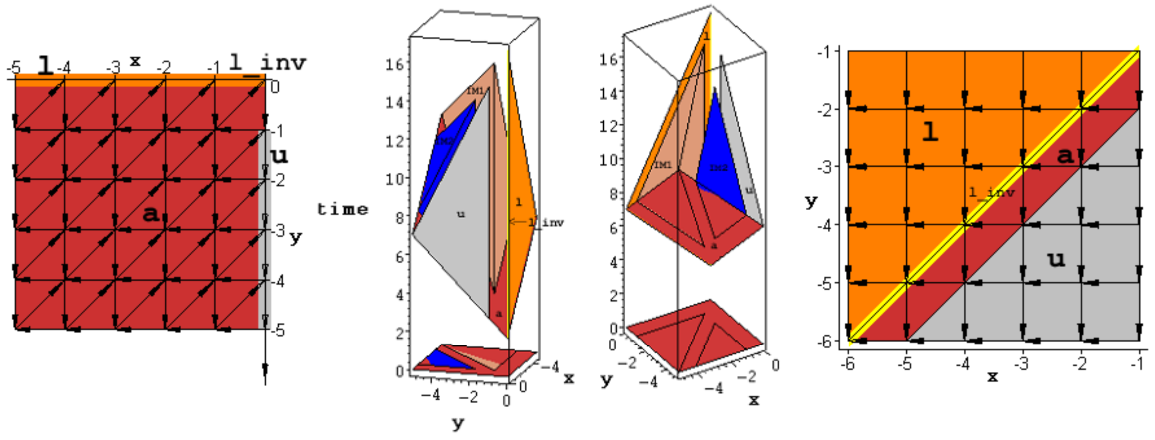


(a) (b)

Figure 2.(a) Single optimal design for *a* restricted to array boundary and maximum regularity criterion and (b) two optimal minimum area designs for *a*,*l*, and *u* constrained to array boundaries (N=6).

Since it is known that systolic arrays that do LU decomposition are possible which use only one divider[1], it is useful to ask how SPADE would achieve such a result. Here this is done by introduction of a new variable. This is necessary because it is clear that if a variable positioned in space-time is to project onto the spatial plane at a single point (corresponding to the divider), this variable can't be represented by a polygon, but must be represented by a line. Hence it must be specified by a single index. Also, it is clear in `(5)` that although there are O(N^2) divides associated with this statement, only N values of *l[i,i]* are used and this could be specified by a single index. Therefore it follows that the number of divisions can be reduced by changing `(5)` to read

```
if j>i and i>1 and i<=N and j<=N then
  u[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..i-1))*l_inv[i]                (6)
fi;
if i>=1 and i<=N then l_inv[i]:=1/l[i,i] fi;
```



(a) (b) (c)

Fig. 3. (a) Array implementation for single divider, (b) space-time view, and (c) maximum regularity solution (N=6).

and replacing instances of division by *l* to multiplication by *l_inv* elsewhere in (4). If this change is made along with a time aligned boundary constraint on *l_inv* and no other time align constraints, we get the single optimal solution shown in Fig. 3a with an area of N^2 (minimum area secondary objective function) and time latency of 3N-3. The corresponding space-time view is shown in Fig. 3b.

In this design variables *l*, *u* and *a* are placed internally and there is additional data flow movement (nine flows vs. the six the array in Fig. 1a). While the creation of a new variable often increases the overhead in terms of data movement and computational requirements, this doesn't happen here because the variable elements *l_inv[i]* remain in the same PE (no data movement). If all constraints are removed and the maximum regularity secondary objective function is used, the single optimal design shown in Fig. 3c is found. Here, all variables are mapped to the array interior to avoid special memory structures (*a* is mapped to all PEs, although this can't be shown with the shading scheme due to variable overlap).

2.4 Lyapunov matrix equation

Here solutions of the matrix Lyapunov equation (find the solution to *AX+XB=C,* where *A* is an NxN non-singular lower triangular matrix, *B* is an NxN non-singular upper triangular matrix, *C* is an NxN matrix and *X* is an NxN unknown[4], are explored to illustrate a variety of design solutions, in particular how non-latency-optimal designs can be found that can be superior to those that are latency-optimal.

The matrix equation can be represented in the equivalent form $c[i,j]=\sum_{k=1}^{i} a[i,k]*x[k,j]+\sum_{m=1}^{j} b[m,j]*x[i,m]$, which leads to

$$for\quad 1\le i,j\le N: \qquad x(i,j)=\frac{c(i,j)-\sum_{k=1}^{i-1} a(i,k)*x(k,j)-\sum_{m=1}^{j-1} b(m,j)*x(i,m)}{a(i,i)+b(j,j)}. \tag{7}$$

This is a non-uniform recurrence equation of the form (1) and the code equivalent can be written directly as

```
for i to N do for j to N do                                                    (8)
  x[i,j]:=(c[i,j]-add(a[i,k]*x[k,j],k=1..i-1)-
     add(b[m,j]*x[i,m],m=1..j-1))/(a[i,i]+b[j,j])
od od;
```

where the SPADE parser ignores summation "add" over *k* when *i*=1 and over *m* when *j*=1.

Minimum area designs: With no time alignment constraints set, two optimal mappings were found, each with a latency of 4N-3 and area 2N^2. The two SPADE generated array designs are shown in Fig. 4a, along with a space-time mapping of one of one these solutions in Fig. 4b. As in the LU decomposition algorithm two intermediate variables, *IM1[i,j,k]=a[i,k]*x[k,j]* and *IM2[i,j,m]=b[m,j]*x[i,m]*, are created automatically. Since *c* has the same affine dependency as *x*, it is placed by SPADE (optionally) in the same locations as the variable *x*.

(a) (b)

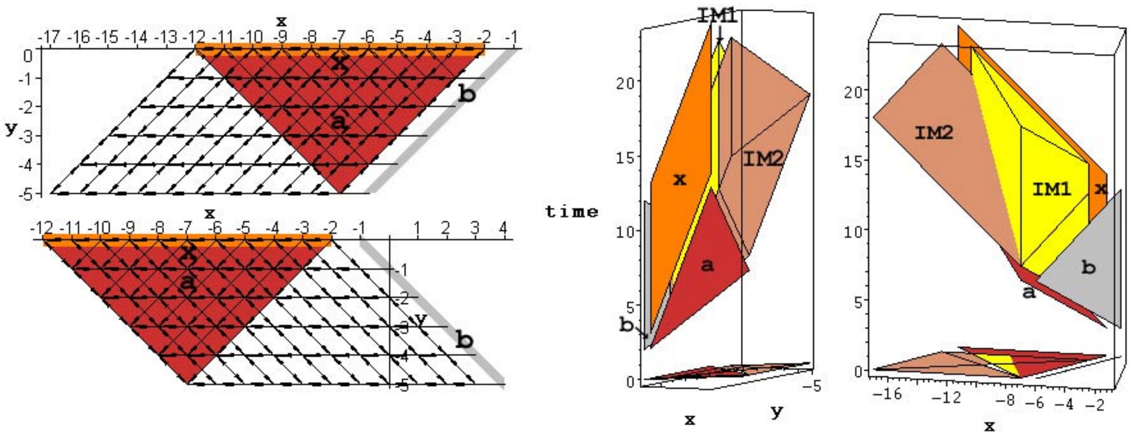


Fig. 4. (a) Area optimal array designs for 4N-3 latency and (b) space-time view of lower design in (a) (N=6).

Sub-optimal latency solutions can be easily explored because all of them have been identified in the scheduling stage of the search. This is very important when useful designs have different but closely spaced latencies. For example, if SPADE explores minimum area designs for latencies of duration 4N-2, four optimal solutions are found (Fig. 5) with a much smaller area, N(N+1), than for the 4N-3 latency solutions.

*Maximum regularity desi*gns: Looking at all designs having a latency of 4N-2 with no time alignment constraints, four unique maximum regularity designs are found, each having an area (N+1)(N+1), In this case algorithm variables *a* and *b* were mapped to the array edge, although *x* and *c* were distributed across the array. Each of these four designs has the same systolic architecture and PE connectivity shown in Fig. 6a; however, each of the four designs has different data flows among these connections (this can only be seen by viewing the corresponding space-time views).

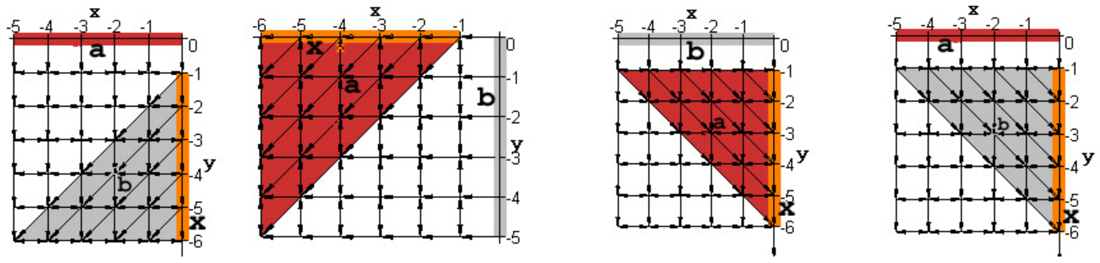
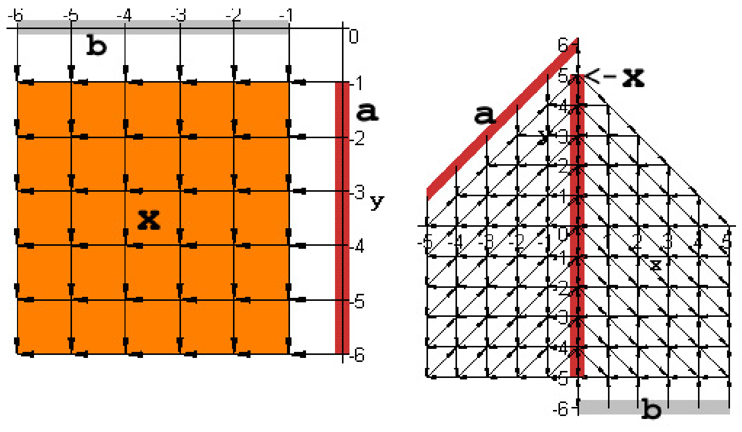


Fig. 5. Optimal array designs for 4N-2 latency (N=6).

The Lyapunov designs typically need to support eight separate flows of data. Consequently, looking for those designs that minimize the movement on the mesh network could be of value. Setting the search to isolate minimum bandwidth solutions with the variables *a*,*b*, and *x* time aligned, generated four designs with latency 6N-4. That design having the minimum PE area is shown in Fig. 6b. Each of these four designs selected by SPADE had the dependencies $x \leftrightarrow IM1$ and $x \leftrightarrow IM2$ spatially separated to reduce the flow of data in/out of a given PE. However, for this to work the linear array of PEs allocated to *x* was placed at the array interior, creating the need for O(N) memory storage at these locations. While such an array design is unusual in the traditional systolic sense, it would not be difficult to create such a design from FPGA hardware, as it would require only a memory structure in between two sets of FPGA logic chips.



(a) (b)

Figure 6. (a) Maximum regularity and (b) minimum bandwidth optimal arrays.

3 1-D AND 2-D DISCREET FOURIER TRANSFORMS[†]

Computational schemes for computing the discreet Fourier transform (DFT) have been intensively studied because of their central importance to various digital signal processing applications. Systolic arrays have been proposed for computing the DFT because of their inherent design regularity and scalability[1] compared to the more conventional "pipelined" implementations[5]. However, past systolic implementations have been hardware intensive because they require N complex multipliers and adders for an N point transform, independent of the dimensionality of the transform[6]. Here we show that the amount of hardware required for a systolic DFT calculation can be reduced considerably by employing the well known radix-4 arithmetic efficiency long effectively used in pipelined FFT implementations. Our approached is derived from the use of a "base-4" frequency/time transformation of the Fourier coefficient matrix, that was applied to make a pipelined FFT calculation more regular[5], and thus more amenable to VLSI implementation. We

[†] Patent Pending

show that this same base-4 regularity can be effectively applied to a systolic implementation of the FFT as well. Although our base-4 systolic design exploits the computational efficiency of a radix-4 butterfly, transform lengths must only be divisible by 16 in the implementation described here, compared to traditional radix-4 designs which must satisfy $N = 4^m$, where N is the transform length and m is an integer.

3.1 Base-4 DFT Representation

The DFT is defined as

$$Z[k] = \sum_{n=1}^{N} X[n]\, e^{-j(2\pi/N)(k-1)(n-1)} \qquad k = 1,2...N \tag{9}$$

where *X[n]* are the time domain input values and *Z[k]* are the frequency domain outputs. It is convenient to represent (9) in the matrix terms

$$Z = CX \tag{10}$$

where C is a coefficient matrix containing elements $W_N^{kn} = e^{-2*j*\pi*(n-1)*(k-1)/N}$. With an appropriate set of permutations applied to Z and X, the matrix form of the DFT can be written in the base-4 form as[5]

$$\begin{aligned} Y &= W_M^t \cdot C_{M1} X \\ Z &= C_{M2} Y^t \end{aligned} \tag{11}$$

where W_M is the $(N/4)\times(N/4)$ array,

$$W_M = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots \\ 1 & W^1 & W^2 & W^3 & \cdots \\ 1 & W^2 & W^4 & W^6 & \cdots \\ 1 & W^3 & W^6 & W^9 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{12}$$

C_{M1} and C_{M2} contain N/16 radix-4 decimation-in-time butterfly submatrics C_B of the form $C_{M1} = \left[C_B^t \mid C_B^t \mid ...\right]^t$ and $C_{M2} = \left[C_B \mid C_B \mid ...\right]$, where

$$C_B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}$$

and *Z,X* have been redefined as the matrices,

$$Z = \begin{bmatrix} Z_1 & & Z_{N/4} \\ Z_{1+N/4} & \cdots & Z_{N/2} \\ Z_{1+N/2} & & Z_{3N/4} \\ Z_{1+3N/4} & & Z_N \end{bmatrix}, \quad X = \begin{bmatrix} X_1 & & X_{N/4} \\ X_{1+N/4} & \cdots & X_{N/2} \\ X_{1+N/2} & & X_{3N/4} \\ X_{1+3N/4} & & X_N \end{bmatrix}. \tag{13}$$

By comparing (11) with (10), the computational advantages of the manipulation leading to (11) are readily evident. In (11) the matrix products $C_{M1}X$ and $C_{M2}Y^t$ involve only additions because the elements of C_{M1} and C_{M2} contain only ± 1 or $\pm j$, whereas the product CX in (10) requires complex multiplications. Also, the size of the coefficient matrix W_M in (11) is $(N/4)\times(N/4)$ vs. the $N \times N$ size of C in (10); consequently the number of overall direct multiplications in (11) is reduced compared to (10); additionally, W_M is applied element-by-element, further reducing the number of multiplications. Note that for systolic implementations a distribution of the elements, $C_{M1}[i,j]$ and $C_{M2}[i,j]$, in (11) does not impose significant bandwidth requirements because full complex numbers are not used.

3.2 1-D Base-4 Systolic Designs

In this section the constraint based functionality of SPADE is used to explore possible systolic designs for calculating the DFT from (11). The primary design criteria are to find those architectures that provide the highest throughput, are easiest to design and are hardware (area) efficient. Even though SPADE uses a minimum latency objective function, the designs it finds are often the most regular and therefore most suitable for providing high throughput. SPADE's flexibility in examining non-optimal solutions is used as well.

The input to SPADE is a coded form of (11),

```
for j to N/4 do
  for k to N/4 do Y[j,k] := WM[j,k]*add(CM1[j,i]*X[i,k],i=1..b)  od;              (14)
  for k to b do Z[k,j] := add(CM2[k,i]*Y[j,i],i=1..N/4) od;
od
```

where b is the base (b=4) and N is the DFT size. This is obtained directly since the matrix multiplications can be equivalently described by summations over rows and columns.

The analysis process starts by setting all the desired architectural constraints, and then relaxing these successively in order of decreasing importance, if no satisfactory designs are found. Desired architectural constraints are found by examining (14) and (11) in conjunction with desired characteristics of systolic arrays. Typically, some experimentation is required to find the best tradeoff between throughput and area.

The most important constraint to impose is that the number of complex multipliers be minimized since these contribute most heavily to area usage. As noted earlier, the products $C_{M1}X$ and $C_{M2}Y^t$ only require complex additions. However, from (14) it can be seen that there has to be a complex multiplication associated with every element *Y[j,k]* of *Y*. (The multiplications take place at the space-time locations of *Y*). Therefore it is important that *Y* be time aligned so that it projects onto the PE array as a linear array of multipliers rather than a 2-D array of multipliers. Next, it would also be desirable to avoid a network flow of complex data from W_M to Y, because of the bandwidth required to support this. To avoid this possibility SPADE can be set to automatically use the same transformation matrices for both W_M and Y, so that both variables are mapped to the same points in space-time. Finally, SPADE constraints can be set so that array I/O (X and *Z*), occur at array boundaries, while coefficient matrices, C_{M1}, C_{M2}, can be constrained to reside in a non-time aligned fashion internal to the PE array, in which case edge-based memory structures do not have to be created to distribute them to the systolic array.

With all the constraints mentioned above set, using the minimum area secondary objective function, SPADE found several systolic array designs; however, all of these had relatively high latencies, low throughputs and consumed substantial areas. Experimentation revealed that the designs with the lowest latencies, smallest areas and good throughputs required one of either X or Z to be resident internally in the array at the end of a single computation. SPADE found six such latency optimal designs (L=latency=*N*/2 + 8) with a "bounding box" area of 4(*N*/4+5). As a measure of throughput the number of cycles between successive transform computations is used and denoted by CPD (cycles per datum). Each of these designs had the same throughput, CPD=*N*/4 + 6. For two of the designs *Z*/ C_{M1} were time aligned and for the other four *X*/ C_{M2} were time aligned.

These designs were computational efficient in those regions of space-time where the variable calculations were performed; that is, there was a computation at each internal space-time point each cycle for each variable at which a computation occurs. High computational efficiencies are necessary to achieve the best latencies. However, to achieve the maximum throughput, it equally important to "fill" the entire space-time volume with computations. The six minimum area designs contained irregularities in space-time, that prevented all of the space-time volume from being filled and thus the throughputs aren't optimal. In order to achieve better throughputs SPADE was used to generate more regular solutions with sub-optimal areas by setting the secondary objective function to look for maximum regularity arrays. By focusing on systolic arrays with larger areas, two latency optimal (L=*N*/2 + 8) and throughput optimal (CPD=*N*/4 + 1) solutions were found (Fig. 7). Again each of these requires one of either X or Z to be time aligned. However, the design in Fig. 7 requires an unusual placement of a linear array of PEs with O(*N*) memory per PE at the array center.

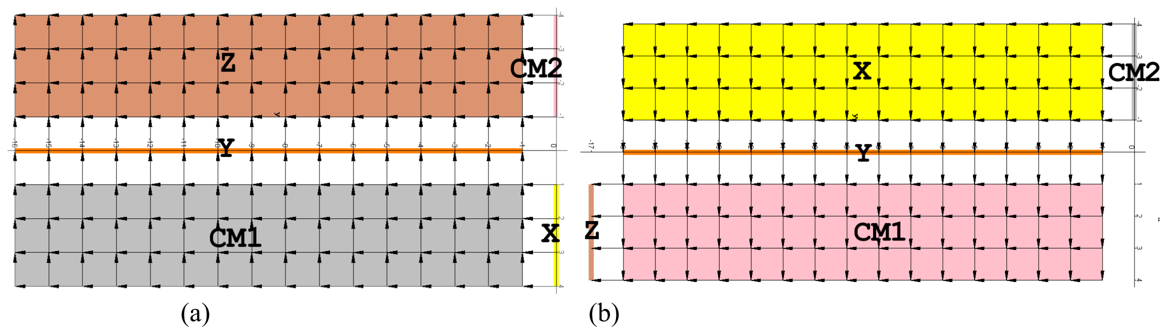


Fig. 7. Two latency optimal, throughput optimal 1-D DFT systolic designs (N=64). WM is mapped to the same locations as Y.

Each of the designs shown in Fig.7 require that either X or Z be pre-loaded into or unloaded from the systolic array. In principal this is a disadvantage because extra bandwidth must be provided so that this operation, can be "buried" in the computation cycle. However, there is an inherent advantage for the internal positioning of X and Z in this case because the load/unload would be such that data I/O is in sequential rather than radix-4 order. This is not a small consideration because input/output formatters can consume considerable amount of space.

In order to fully characterize the systolic array designs in Fig. 2 it is necessary to specify the transformation matrices for variables in (14). For example those for the array of Fig.7b are provided in Tables 1 and 2. Table 1 provides the *T/t* matrices for the variables and Table 2 provides the corresponding vectors ([*time,x,y*]) showing the data flow direction for each dependency in (14). Note that the data flow for two dependencies indicates no spatial movement. This means that the variable remains in the PE to which it was originally assigned and gets reused in that same PE each cycle.

An actual hardware implementation would need to consider among other things keeping copies of the four elements of C_B in each PE to eliminate the need to actually circulate them as elements of CM1 and CM2. Also, it might a good tradeoff to use one high-speed horizontal (Fig. 7) bus and collapse the array in the vertical direction using an array of registers properly interfaced to an array of adders at each PE along the horizontal bus.

var	Y	IM var1	CM1	X	Z	IM var2	CM2
T	$\begin{bmatrix} 1 & 1 \\ 0 & 0 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 1 \\ 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & -1 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$
t	[5, 0, 0]	[0, 5, 0]	[0, 5, 0]	[0, 5, 0]	[27, -5, 0	[10, -5, 0]	[10, -5, 0

Table 1. Transformation matrices for variables corresponding to Figure 7(b)

Y->IM_var1	*IM_var1->CM1*	*IM_var1->X*	*Z->IM_var2*	*IM_var2->CM2*	*IM_var2->Y*
[1, -1, 0]	[1, 0, 0]	[1, 0, -1]	[1, 0, 0]	[1, 0, -1]	[1, -1, 0]

Table 2. Dependencies (top) and corresponding propagation vectors (below) for design in Fig. 7(b).

3.3 2-D Base-4 Systolic Designs

Here we only consider 2-D NxN DFT designs based on the same algorithm formulation (14) as that for the 1-D transform. An attractive strategy from an algorithmic point of view would be the use of an NxN arrays of PEs that can perform the column transforms followed by row transforms without the need for a matrix transposition in between[6,7,]. However, this 2-D array of multiplier/adders imposes significant hardware penalty. Alternatively, we propose here to build on the less hardware intensive linear designs in Fig.7 by noting that the output Z of the design in Fig. 7a could be used directly as the input X to the design of Fig. 7b. That is the 1-D column transforms can be done with the array of

Fig.7a, followed by the row transforms using the same array configuration of Fig. 7b. However, for this approach to work directly, it would be necessary to transpose the data in between the row and column DFTs, a very time intensive extra operation. Alternatively, it is possible to do the transposition "on-the-fly" during the course of the column DFTs performed in Fig. 7a. This can be done by systolic rotation of the elements of the matrices CM1, CM2 and WM as the column DFTs are performed. To illustrate this concept the code in `(14)` can be altered to do N successive column DFTs including these rotations by writing it as

```
for n to N
  for  j to N/4 do
    for k to N/4 do Y[j,k] := WM[j,k]*add(CM1[j,i]*X[i,k],i=1..b) od;              (15)
    for k to b do Z[k,j] := add(CM2[k,i]*Y[j,i],i=1..N/4) od;
   WM := matrix_rotate(WM,"down");
   CM1 := matrix_rotate(CM1,"down");
   if n mod(b)=0 then CM2 := matrix_rotate(CM2,"down") fi;
  od;
od;
```

where the procedure "matrix_rotate()" does a circular rotation of the argument matrices in the direction indicated. Here the rotations of WM and CM1 cause a rotation of the *Z* output columns one position to the right, and the rotation of CM2 causes *Z* to be rotated downward by one row. The *Z* outputs from the code `(15)` for the first two column DFTs of a 64x64 2-D DFT obtained from this code would then be

$$\begin{bmatrix} z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & z_9 & z_{10} & z_{11} & z_{12} & z_{13} & z_{14} & z_{15} & z_{16} \\ z_{17} & z_{18} & z_{19} & z_{20} & z_{21} & z_{22} & z_{23} & z_{24} & z_{25} & z_{26} & z_{27} & z_{28} & z_{29} & z_{30} & z_{31} & z_{32} \\ z_{33} & z_{34} & z_{35} & z_{36} & z_{37} & z_{38} & z_{39} & z_{40} & z_{41} & z_{42} & z_{43} & z_{44} & z_{45} & z_{46} & z_{47} & z_{48} \\ z_{49} & z_{50} & z_{51} & z_{52} & z_{53} & z_{54} & z_{55} & z_{56} & z_{57} & z_{58} & z_{59} & z_{60} & z_{61} & z_{62} & z_{63} & z_{64} \end{bmatrix}, \begin{bmatrix} z_{16} & z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 & z_9 & z_{10} & z_{11} & z_{12} & z_{13} & z_{14} & z_{15} \\ z_{32} & z_{17} & z_{18} & z_{19} & z_{20} & z_{21} & z_{22} & z_{23} & z_{24} & z_{25} & z_{26} & z_{27} & z_{28} & z_{29} & z_{30} & z_{31} \\ z_{48} & z_{33} & z_{34} & z_{35} & z_{36} & z_{37} & z_{38} & z_{39} & z_{40} & z_{41} & z_{42} & z_{43} & z_{44} & z_{45} & z_{46} & z_{47} \\ z_{64} & z_{49} & z_{50} & z_{51} & z_{52} & z_{53} & z_{54} & z_{55} & z_{56} & z_{57} & z_{58} & z_{59} & z_{60} & z_{61} & z_{62} & z_{63} \end{bmatrix}$$

Thus after all 64 column DFTs are computed by the array in Fig. 7a, the rows would be distributed over the input PEs *X* in Fig.7b. The row DFTs could then be performed with another similar set of rotations so that the final 2-D output is in the same radix-4 order is the original input. The rotations occur such that at the end of a row or column DFT, the matrices have returned their original positions. Also, the twiddle factors in the matrix WM are always in the correct positions for both row and column DFT processing.

The entire sequence of column and row DFTs is best understood from viewing the systolic computation in the space-time domain as shown in Figure 8. (Only two of the 64 row and column DFTs are shown.) It can be seen that successive row/column iterations can be "stacked" on top of each other in time so that all PEs are active all of the time and thus that the design is throughput optimal with the exception of the small period of time period at the end of the column DFTs when systolic processing switches from the design of Fig.7a to that of Fig.7b. Control of the PEs can be done in a number of ways, as for example using a set of modulo counters in each PE to keep track of that PE's required functionality or propagating control signals along with the data.

3.4 Design Comparisons

A comparison of the previously mentioned 2-D systolic array design for computing the 2-D DFT and the base-4 design described here is shown in Table 3. The base-4 design is very different in that it is pseudo linear because as the transform size increases, it's width remains the same, while its length increases so that it uses O(N) hardware rather than O(N^2) when comparing non-partitioned designs of each. However, when considering throughput and area together, the base-4 design is more efficient by a factor of 6 for N=64, as indicated in Table 3, where it was assumed that the area of an adder was ¼ that of a multiplier. It should be noted that the base-4 design can be used to efficiently perform a 1-D DFT by factoring it into two sets of size n_1 and n_2 such that $N= n_1\, n_2$ in the same way as the 2-D systolic array[6,7] and performing row and column DFTs on n_1 and n_2.

4 SUMMARY

As detailed above, the CAD goal of SPADE is (1) to provide a designer with a complete picture of all the systolic options available to him for a given algorithm and (2) to help chose only those designs that meet particular hardware/system constraints. Because the systolic designs are generated at a very abstract level, we feel that SPADE is most applicable to the process of making early high-level system design decisions, where optimal choices can have their greatest impact on real-time embedded system performance. We have provided an example of its use in deriving new hardware efficient 1-D and 2-D systolic arrays to compute the DFT.

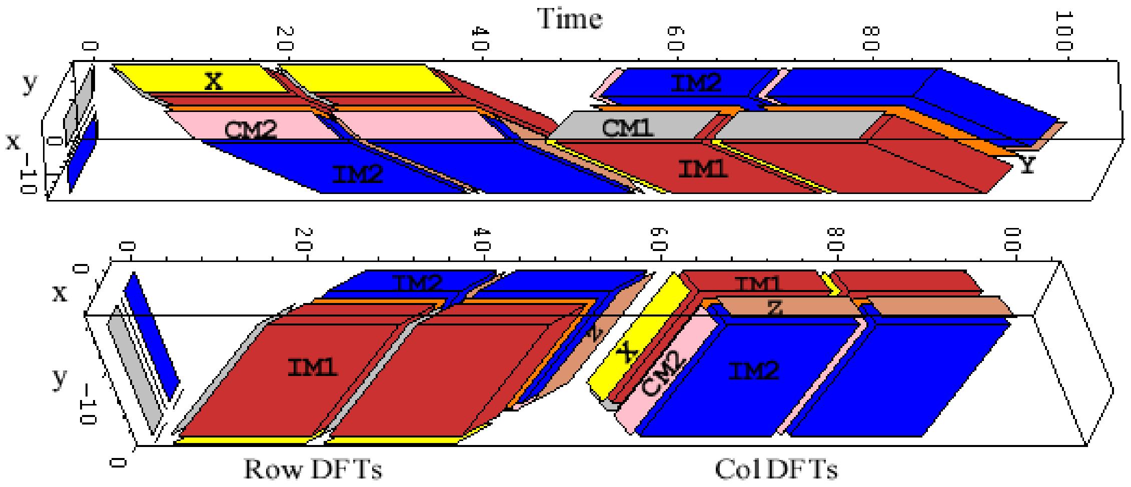


Fig. 8. Two perspectives of space-time for two column and two row DFT iterations. Column and row DFTs are performed by the systolic arrays in Figs. 7a and b, respectively (N=64). The variable *Y* is a plane and can be seen in between the *IM1* and *IM2* variables.

	2-D Array[6,7]	**Base-4 Array**
M	N^2	N/4
A	N^2	2N
CPD	2N+1	$N^2/2 + 6N + 18$
L	4N	$N^2/2+25N/4+19$
~area x throughput = **(M+A/4) CPD**	$5N^3/2+ 5N^2/4$	$3N^3/8+9N^2/2+27N/2$

Table 3. Comparison of area and time characteristics of base-4 and 2-D systolic implementations. Here M and A correspond to the number of complex multipliers and adders.

ACKNOWLEDGMENTS

The development of SPADE was supported in part by DARPA Contracts DAAH01-96-C-R135 and DAAH01-97-C-R107.

REFERENCES

[1] See Kung, S.Y., "VLSI Array Processing", Prentice Hall, 1988; Moldevan, D. I., "Parallel Processing", Kaufmann, 1993; and Quinton, P. and Robert, Y., "Systolic Algorithms and Architectures", Prentice Hall 1991.
[2] J. Greg Nash, "Automatic Generation of Systolic Array Designs For Reconfigurable Computing", 2002 Int. Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA 02), June 24, Las Vegas, NV.
[3] www.centar.net
[4] V.P. Roychowdhury, "Derivations, Extensions and Parallel Implementations of Regular Iterative Algorithms," Ph.D. Thesis, Stanford, 1989, p.192.
[5] Colin C.W. Hui, Tiong Jiu Ding, John McCanny, and Roger Woods, "A New FFT Architecture and Chip Design for Motion Compensation based on Phase Correlation," Proc. Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP 96), pp. 83-92.
[6] H. Lim and E. Swartzlander, "Multidimensional Systolic Arrays for the Implementation of Discrete Fourier Transforms,", IEEE Trans. Computers, Vol.47, No.5, May 1999, pp.1359-1370.
[7] S. Peng, I. Sedukhin, S. Sedukhin, "Design of Array Processors for 2-D Discrete Fourier Transform," IEICE Trans. Inf. & Syst., Vol. E80-D, No.4, April 1997.