Automatic Generation of Systolic Array Designs For Reconfigurable Computing

J. Greg Nash
Centar
Los Angeles, CA, U.S.A.

Abstract: *The problem of rapidly generating optimal parallel circuit implementations from high level, formal descriptions of affinely indexed algorithms is addressed here in the context of reconfigurable FPGA-based computing. A specialized software tool, SPADE, is described that will take a user's high level code description of his algorithms and automatically generate an abstract latency-optimal, locally-connected parallel array of elemental processing elements. A design example, the Faddeev algorithm, is used to illustrate the tool's capabilities and to show a potential algorithm basis for a reconfigurable array processor.*

Keywords: systolic, CAD, reconfigurable, FPGA, parallel, algorithm

1. Introduction

Meeting latency and throughput requirements is a critical concern in many embedded signal and image processing applications. Consequently, a substantial literature has appeared over the last two decades describing fine-grained parallel algorithms, well suited to meeting these requirements when instantiated in regular, array-based architectures that involve pipelined movement of data. Such algorithms, typically referred to generically as "systolic", have been shown to be suitable for a very large range of structured problems (i.e., linear algebra, graph theory, computational geometry, number-theoretic algorithms, string matching, sorting/searching, dynamic programming, discreet mathematics).

Usage of this systolic architecture class has not been widespread in the past, in part because programmable hardware that supported this computing paradigm was not cost-effective to build. However, suitable hardware has now begun to appear in the form of complex FPGAs, which are constructed from tiling identical memory and logic blocks along with supporting mesh interconnection networks, in a way that matches systolic array architectures. Such hardware could allow dynamic implementation of systolic algorithms leading to inexpensive "programmable" systolic array hardware.

The primary reason for the limited role of systolic processing is that to date no commercial software tool has appeared that allows rapid generation and exploration of systolic array designs. Without such a tool, systolic arrays are very difficult to design because parallel algorithm/architecture design is an intricately complex process requiring extensive knowledge of algorithms, architectures, and hardware, past work on finding parallel algorithm implementations are largely "point" designs that offer little insight into the numerous tradeoffs required as part of any embedded system design and there are no systematic design methodologies with adequate generality and ease of use

The symbolic parallel algorithm development environment (SPADE) described here is being developed 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. The intention is to allow a user to specify his algorithm using traditional high-level code, set some architectural constraints and then view the results in a meaningful graphical format. SPADE includes a simulator that has an embedded computational model to facilitate the transition to systolic FPGA hardware. The results described here show new tool generated optimal systolic algorithm mappings for the Faddeev algorithm, which was chosen because it is a reasonably complex demonstration example, it is useful for performing a wide variety of matrix based computations, and useful detailed systolic analyses already exist [8],[9]. Other tool-generated mappings are presented elsewhere [10].

2. Related Work

The most common approach to designing relevant architectures have been those based on index transformations [7],[11],[2] although data dependency based efforts have been pursued as well [5],[8]. The former category makes use of mathematical transformations which, when applied to systems of "uniform" or "regular" recurrence equations or their equivalent, result in parallel algorithms that represent "mappings" to an architectural model consisting of large arrays of simple, locally connected abstract processing elements (PEs). More specifically, these techniques calculate matrices that transform the index set describing the original algorithm to an index set containing at least one time dimension with the remaining indices used for spatial coordinates, i.e., a "space-time" mapping.

Proceedings Engineering of Reconfigurable Systems and Algorithms (ERSA '02)
International Multiconference in Computer Science
Las Vegas, Nevada, June 24, 2002.

A classic example of such uniform algorithms is the multiplication of two matrices *A* and *B* to produce the result *C*:

$$\begin{aligned} & for \ \ 1 \le i,j,k \le N \\ & a[i,j+1,k] = a[i,j,k] \\ & b[i+1,j,k] = b[i,j,k] \\ & c[i,j,k+1] = c[i,j,k] + a[i,j,k] * b[i,j,k] \end{aligned} \qquad (1)$$

where each variable element (e.g., *a[i,j,k]*) takes on a unique value (single assignment property) for each affinely referenced index vector *I*, ($I=(i,j,k)^T$). All algorithm variables are matrices or vectors. The most important characteristic of these algorithms is that all dependencies are uniform for all values of the index space *I*. For example *c[I]* above depends upon *c[I-D]*, where *D=[0,0,-1]* , for all values of *I*. Because dependence vectors like *D* will always contain only small integer values, data is inherently "localized" or pipelined in the systolic array implementation. That is, calculations associated with indices in space-time only make use of variable values obtained from the same or adjacent points in that index space. Significant extensions to this model have been made with the goal of either generalizing the target class of algorithms [2],[3],[13],[14] or converting non-uniform equations to a uniform equivalent form [6].

There is not sufficient space to describe the many previous tool efforts intended to automate the systolic array design process, but useful summaries exist [1],[8],[16]. Most past tools have targeted uniform recurrence equations and the related problem of serially coded loop nests with uniform dependencies. The disadvantage of tool methodologies based on uniform dependencies is that most algorithms are not naturally expressed in this form and the process of putting them in this form can involve substantial effort because there is no systematic way of doing so. Also, potential parallelism is not fully exploited [1] and inefficiencies are introduced by unnecessarily requiring all variables to exist at all points in the algorithm index space[12].

Active systolic tool efforts such as PICO [15] and DG2VHDL [16] take as inputs a uniform algorithm (PICO) or a user supplied dependence graph (DG2VHDL), and focus on the production of efficient hardware implementations. The MMalpha environment provides a high level language for expressing affine recurrence equations that lets the user create architectures and hardware by invoking built in code transformations[4].

3. Spade Description

3.1 Introduction

The primary goal of SPADE is to enable a designer to easily, automatically and rapidly generate abstract array designs that represent the best of system tradeoffs. For example, a system design might require that systolic inputs and outputs occur at array boundaries, that all intermediate computations remain internal to the array, and that any coefficient matrices used also remain internal so that no "edge based" memory structures need be created. 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 tool approach generalizes acceptable input algorithm forms to one more closely resembling that in which an algorithm is naturally expressed. For example in the case of matrix-matrix multiplication, it would be much more desirable to accept inputs directly in the familiar form

$$c[i,j] = \sum_{k=1}^{N} a[i,k] * b[k,j] \qquad (2)$$

rather than (1). As can be seen in (2), it is not required that there exist a constant vector D that relates the dependencies between *c* and *a* and *b*. Note also that (2) is very general and unlike (1) doesn't impose specific relationships between variables that inherently restrict subsequent choice of architectures. Systolic design solutions for the algorithm form (2) are often termed "non-uniform" because local dependencies between variables in a systolic array design do not have to be the same for all points in the index space *I*.

The difficulty in working with equations like (2) is that dependencies between variables are no longer local and thus the path to a systolic implementation is more difficult. In order to solve the general problem of finding optimal mappings of non-uniform affine recurrence equations that simultaneously consider scheduling, reindexing, localization and allocation in the presence of architectural constraints, it is necessary to use integer programming methods [1]. Two tool efforts that have been proposed for this class of algorithms both solve the integer-programming problem by structuring them as customized searches [1],[17]. The focus in [17] is to maximize processor utilization for problems with a given throughput and input-output scheme using as a metric a sum of dependence lengths. Alternatively, DESCARTES uses algorithm latency as an objective function metric [1]. One important difference between the two tools is that the DESCARTES search is structured to find optimal designs, whereas [17] is more limited in this respect and therefore optimal designs aren't guaranteed. DESCARTES places strong reliance on the use of architectural constraints to reduce the space of possible systolic solutions, making the search strategy feasible.

SPADE uses a search methodology that is based on that used in DESCARTES, but involves a different formalism and is organized to provide better coverage of the architectural solution space for cases where solutions are less architecturally constrained. In particular SPADE separates the scheduling and reindexing steps so that all schedule solutions can be enumerated prior to reindexing, allocation and localization. This is useful when solutions exist that are latency optimal, but are inefficient (sub-optimal) for other reasons. SPADE also incorporates a parser to handle standard nested loop inputs, does additional analysis of potential solutions to give the designer more control over design tradeoffs, includes a simulator that uses an embedded model of computation and is instrumented to provide useful architectural statistics.

SPADE performs space-time mapping first. That is, scheduling, reindexing and allocation are done first. After all minimum latency solutions are found, an attempt is made to localize these solutions. If this isn't possible, then the next best minimum latency space-time mapping solutions are chosen and the process repeated until at least one solution is found. (SPADE can also examine designs associated with non-optimal latencies.)

3.2 Space-Time Mapping

A formal description of non-uniform recurrence equations, termed "conditional affine recurrence equations," has been provided in [1]. In this context "conditional" is intended to imply that each equation can have its own unique index space. A system is

$$\begin{aligned} w_1[A_1(I)] &= g(...w_i[B_{i1}(I)],...) \qquad \mathit{for\,all\,I\,in}\,\mathrm{I}_1 \\ &\;\;... \qquad\qquad\qquad\qquad\qquad\qquad\qquad (3) \\ w_n[A_n(I)] &= f(...w_i[B_{in}(I)],...) \qquad \mathit{for\,all\,I\,in}\,\mathrm{I}_\mathrm{n} \end{aligned}$$

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

$$\begin{aligned} A(I) &= AI + a \\ B(I) &= BI + b \end{aligned}$$

Here, *A/a,* and *B/b* are integer matrices/vectors. All assignments of values to variable elements in the system of the equations must not involve a reuse of a variable. For each algorithm variable SPADE finds an affine transformation, *T* that maps this algorithm variable's indices to space-time, e.g., for a variable *x*:

$$T(x) = T_x(A_x I + a_x) + t_x$$

where T_x is a matrix and t_x is a vector. Thus, every variable element $x[I^T]$ gets mapped to a unique point in the space-time domain. The transformation *T* can be thought of as consisting of two parts, one that determines the scheduling index and one that determines the spatial index. That is, writing $T(x)$ using

$$T_x = \begin{bmatrix} \Lambda_x \\ S_x \end{bmatrix}, \qquad t_x = \begin{bmatrix} \gamma_x \\ s_x \end{bmatrix} \qquad (4)$$

means that variable $x[I^T]$ would be mapped to a time index $\Lambda_x(A_x I + a_x) + \gamma_x$ (Λ_x / γ_x is a vector/scalar), and to a spatial index $S_x(A_x I + a_x) + s_x$ (S_x / s_x is a matrix/vector with a number of rows/elements equal to the dimension of the spatial array).

Each time index corresponds to potential activity (data transfer or calculation) in all PEs with that same index value. The basic execution cycle for a time index value consists of two steps: first there is a local movement of data between adjacent PEs, followed by a computation step in those PEs. The algorithm latency is the total number of these time steps needed to execute the algorithm computation.

SPADE's primary outputs are the values *T,t* for each of the algorithm variables. In addition a set of vectors are computed indicating the direction of data flow for each dependency in the algorithm. For the example (2) it can be seen that *c* needs input from *a*, so corresponding to this dependence is a uniform flow of data from *T(a)* to *T(c)* in the space-time domain. Since this flow is one dimensional, a vector v_{ca} is calculated to indicate its direction.

3.3 Solution search

From (4) it is clear that to specify the space-time mapping for *x* it is necessary to find the elements of Λ_x, S_x, s_x and the scalar γ_x. For example, given $\Lambda_x = \begin{bmatrix} \lambda_{x1} & \lambda_{x2} \end{bmatrix}$, SPADE considers λ_{x1} and λ_{x2} as "search" variables (as distinct from previously mentioned "algorithm variables"). Following [1], the allocation matrices like S_x are treated as a single variable and are derived from unimodular matrices to ensure that space-time mapping solutions are more "dense". The small dimensionality of *S* limits the number of unique matrices that have to be considered. The fact that each algorithm variable can have a different spatial mapping *S* is equivalent to it being "reindexed" with respect to other algorithm variables.

Finally, given a set of search variables, SPADE examines all possible combinations and chooses those that produce the minimum algorithm latency. The difficulty in doing this is that there are potentially a large number of these search variables and even though each need take only a small range of values, the search can be computationally infeasible. By introducing computational and architectural constraints that limit the space of choices that has to be searched the solution space can be limited [1]. For example, causality requires that a computation not occur if its arguments are not available. From the example (2) it can be seen that computation of *c* depends upon input *a*. Thus, it must follow that temporally

$$\Lambda_c(A_c I + a_c) + \gamma_c - \Lambda_a(B_a I + b_a) - \gamma_a \geq 0 \qquad (5)$$

Typically, there are many dependencies and thus a large number of such constraints are generated from these criteria.

Architectural constraints are just as important. For example, if it is desired that input of data occur from the edge of the PE array, then the position in space-time of an input matrix, such as *a[i,j]* in (2) , must be such that the unit time vector u_t is in the plane of the space-time array of variable *a* (hence it's projection onto the PE array is a line). This constraint requires that the normal n_a to the plane of *a* satisfies $u_t \bullet n_a = 0$, and thus *a* is "time aligned". Also, for the projection of *a* to be at the edge of the PE array it must not be within the convex hull of the polygon defining the PE array. Constraints like these considerably limit choices for Λ_a and S_a

3.4 Input

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. Very often 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 (2) 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)
end do end do;
```

where the Maple "add" construct directly replaces the mathematical summation sign. Maple treats the loop structure in traditional way, but SPADE does not make any lexicographic interpretation of the loops; rather it uses the loop limits only to determine the index space of the inner statement body. Computational ordering is determined directly from the loop body statements. When conditionals are used in code input to SPADE, the loop limits are ignored in determining the index space.

Other inputs pertain to architectural constraints desired and objective function criteria used to select solutions from the search space. Architectural constraints specify (1) which variables should be constrained to align with the time axis ("time-aligned") and/or the PE array boundary as outlined in Section 3.3 and (2) whether overlap of variables in space-time is allowed. After SPADE finds all minimum latency solutions, secondary optimization criteria can be used to choose sub-solutions from among these. Presently these criteria find sub-solutions with (1) minimum area, (2) maximum regularity and (3) minimum array bandwidth.

4. Faddeev Design Example

4.1 Algorithm Derivation

The Faddeev algorithm [9] computes the expression *CX+D* subject to the constraint *AX=B*, where *C,D,A*, and *B* are matrices and *A* is a full rank *NxN* matrix. It is convenient to describe the algorithm as an extended matrix *a* using the representation

$$a = \frac{A \mid B}{-C \mid D}. \tag{6}$$

The algorithm begins by adding a linear combination of the rows A|B to the rows C|D or

$$\frac{A \mid B}{WA - C \mid WB + D}.$$

If W is chosen so that WA-C=0, then W=CA^{-1} and this substitution in the lower right hand corner provides the desired result there or

$$\frac{A \mid B}{0 \mid CA^{-1}B + D}. \tag{7}$$

It can be seen that matrix operations are "programmable" based on the entries in (6), e.g.,

$$\frac{A \mid B}{-I \ \mid 0} \rightarrow X,\ \frac{A \mid I}{-I \mid 0} \rightarrow A^{-1},\ \frac{I \mid B}{-C \mid 0} \rightarrow CB.$$

It is not necessary to actually compute the value of W; it is only necessary to annul the elements of C, a process that can be done using an LU decomposition algorithm in which *a* is decomposed into the product of a lower triangular matrix (*l*) and an upper triangular (*u*) matrix or

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & - & - \\ l_{31} & l_{32} & - & - \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & u_{43} & u_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

where the "-" above indicates these elements l_{ij} are not computed and each matrix is taken as 4x4. Here the matrix *u* is the desired final result and corresponds to the transformation from *a* to the form in (7). This has been achieved by stopping the factorization process at the point where the elements of C have been annulled. Consequently, the desired result from the lower right hand corner of (7) is

$$d = \begin{bmatrix} u_{33} & u_{34} \\ u_{43} & u_{44} \end{bmatrix} = CA^{-1}B + D.$$

An explicit mathematical formula for the *l's* and *u's* above can be obtained by induction. That is,

$$u_{11} = a_{11}; u_{12} = a_{12}; u_{13} = a_{31}; u_{14} = a_{14}; l_{11} = 1;$$
$$l_{21} = a_{21}/u_{11}; l_{31} = a_{31}/u_{11}$$

$$\begin{aligned} &for\, i \geq j, j > 1, i \leq 4, j \leq 2 \ \rightarrow\ l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}) / u_{ii} \\ &for\, j \geq i, i > 1, i \leq 2, j \leq 4 \ \rightarrow\ u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \\ &for\, i > 2, j > 2, i \leq 4, j \leq 4 \ \rightarrow\ u_{ij} = a_{ij} - \sum_{k=1}^{N} l_{ik} u_{kj} \end{aligned} \tag{8}$$

From the mathematical expressions in (8) above it is possible to go directly to the coded form by replacing the summation by "add()":

```
for j to 2*N do
    for i to 2*N do
      if i=1 and j>=1 and j<=2*N then u[i,j]:=a[i,j] fi;
      if j>=i and i>1 and j<=2*N and i<=N then
         u[i,j]:=a[i,j]-add(l[i,k]*u[k,j],k=1..i-1)
      fi;
      if j>N and i>N and j<=2*N and i<=2*N then              (9)
         u[i,j]:=a[i,j]-add(l[i,k]*u[k,j],k=1..N)
      fi;
      if j=1 and i>=1 and i<=2*N then l[i,j]:=a[i,j]/u[j,j] fi;
      if i>=j and j>1 and i<=2*N and j<=N then
         l[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..j-1))/u[j,j]
      fi;
   od
od;
```

Here, the loop limits have been replaced by the problem size parameter, N, where it has been assumed that each matrix is NxN in size. Solutions obtained are independent of the size parameter N, which is handled symbolically during the solution search.

While the code in (9) is suitable for input to SPADE, there are many different ways to code the Faddeev algorithm. Each of the different codings can result in substantially different array implementations. In general it is best to assign unique variables to the quantities that you want to calculate. In (9) the result *d* is not called out separately, which limits the number of array implementations that SPADE can explore. The reason for this is that for (9) SPADE must find a single polygon in space-time that contains all the values of *u*, only some of which contain *d*. If instead the input code called out *d* separately, then SPADE would have an easier task because it would have more freedom to separately place the polygons that represent *d* and *u*. Therefore, (9) has been modified as follows:

```
for j to 2*N do
   for i to 2*N do
      if i=1 and j>=1 and j<=N then u[i,j] := a[i,j] fi;
      if i=1 and j>=1 and j<=N then b[i,j] := a[i,j+N] 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)
      fi;
      if j>=1 and i>=1 and j<=N and i<=N then
         b[i,j]:=a[i,j+N]-add(l[i,k]*b[k,j],k=1..i-1)          (10)
      fi;
      if j>=1 and i>=1 and j<=N and i<=N then
         d[i,j] := a[i+N,j+N]-add(l[i+N,k]*b[k,j],k=1..N)
      fi;
      if j=1 and i>=1 and i<=2*N then l[i,j]:=a[i,j]/u[j,j] fi;
      if i>=j and j>1 and i<=2*N and j<=N then
         l[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..j-1))/u[j,j]
      fi;
   od
od;
```

where we have also made the substitution

$$b = \begin{bmatrix} u_{13} & u_{14} \\ u_{23} & u_{24} \end{bmatrix}.$$

as well for similar reasons. The code (10) is read as a text file into SPADE at the beginning of processing by the parser.

From the steps above, it can be seen that a suitable algorithm input to SPADE can be derived directly from the corresponding mathematical expressions. It is noteworthy that no steps in this process involved issues requiring significant understanding of parallel algorithms or architectures.

4.2 Faddeev Algorithm Design Results

The purpose of this section is to provide a few examples of how high level architectural constraints can result in very different array designs that could serve as reconfigurable computer options.

4.2.1 Minimum Area Array Designs

In this first mapping example the architectural constraints were set to find array designs with the least PE array area among the minimum time latency solutions. In addition, since computation of values of *l* in (10) require hardware-demanding division operations, it would be desirable to minimize them. One way of doing this is to require that the variable *l* be mapped such that it is projected (time aligned) onto the PE-array in a line. In this case division operations would only be necessary in a linear array of PEs as opposed to a 2-D array of PEs. Finally, it is a common characteristic of systolic arrays, which are physically tied to sensors, to have sensor data stream into the array from one or more array-edge boundaries. This is also consistent with FPGA based virtual computers that contain a lot of buffer memory at the board inputs. This input data configuration can be forced by setting a constraint that eliminates exploration of designs that don't map (time align) *a* to a linear array of PEs on the edge of the complete array.

With these constraints set, a SPADE search discovered 2 unique solutions with time latency 5N-2, each with ½ N(3N+1) PEs, one of which is shown in Fig. 1. The PE array in Fig. 1 is uniform in terms of the interconnection pattern and there are twelve different flows of data associated with the various variable dependencies, each of which moves along an orthogonal path defined by the arrows in Fig. 1. There are five additional dependencies in which a variable does not move spatially, but rather is updated and reused in the same PE. This corresponds to data "movement" along the time axis in space-time.

The picture in Fig. 1 shows a superposition of data flow at all times. The actual time variation of data flow is more complex, with the size of uniform sub-regions of data movement growing and shrinking with time. The design in Fig. 1 is the same architecture on which most past analyses of the Faddeev algorithm have been based [8],[9].

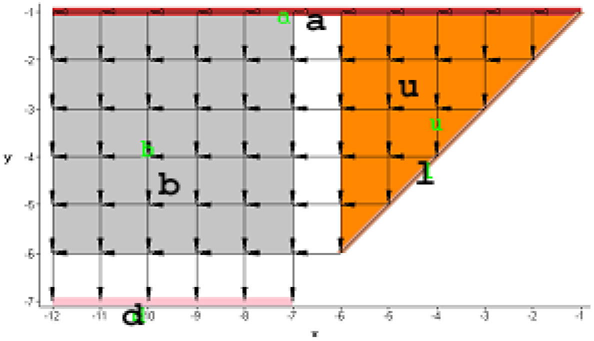

Figure 1. One of two minimum area designs for Faddeev algorithm with variables *a* and *l* constrained to appear on the array boundary (N=6).

4.2.2 Maximum Regularity Design

While a minimum PE array area criterion is useful as a secondary objective function, it is possible to devise a variety of other criteria to help select designs. The main function of different secondary objective criteria should be to identify very different architectures, rather than to identify a single "best" one. This is important because the SPADE is capable of generating many different architectures, but at the abstract level at which these are generated, practical FPGA hardware considerations can have a big impact on choices. In this section a heuristically determined maximum regularity criteria is described. The goal of this criterion is to select designs that are maximally spatially uniform, and thus easiest to build. Because specification of what constitutes a regular array design is subjective, SPADE uses three different criteria.

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 in proportion to the number of different data flows and doubly penalized for each different diagonal flow of data.

Number and orientation of variables that are time-aligned: This criterion penalizes an array design for each variable that is time aligned, 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.

Dependency relations between variables: Because a systolic design is inherently pipelined, it is possible to find many different alignments between variables. In other words a variable plane can simply be shifted one unit or rotated in space-time and still represent a valid design. Therefore, the regularity criteria penalize designs of this type.

Using the criteria above as the secondary objective function, SPADE produces the single optimal design (5N-2 latency) shown in Fig. 2. Clearly this is a much larger design, but it is completely uniform with respect to PEs and interconnections, it contains no variables that are time aligned, and the overall number of different data-flow paths is less by 25% than the design in Fig. 1. On the other hand it requires an array of dividers (in the area labeled *l*), and the load/unload cycle required by the input/output data could be undesirable.

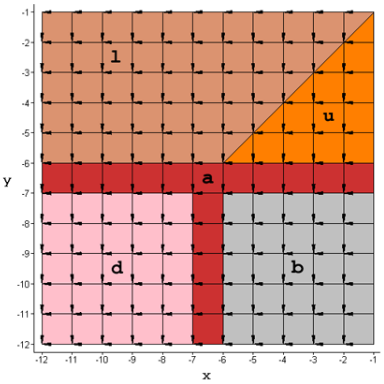

Figure 2. Faddeev algorithm single optimal design obtained using regularity as secondary objective function (N=6).

4.2.3 Single Divider Implementation

It is known that the Faddeev algorithm can be mapped to a design that uses only one divider [9], so a useful question would be what code modifications would be necessary to achieve this. The code (10) shows only divisions by u[j,j]; consequently, a 1-D variable can replace a 2-D variable limiting the number of divisions to N with "u_inv[j]:=1/u[j,j]". Adding this to (10) and changing the divisions by u[j,j] to multiplications by u_inv[j] results in the code (11), where "...." refers to the other statements in (10) that don't involve division, yields

```
for j to 2*N do
  for i to 2*N do
      .....
      if j<=N and j>=1 then u_inv[j]:=1/u[j,j] fi;
      if j=1 and i>=1 and i<=2*N then l[i,j]:=a[i,j]*u_inv[j] fi;
      if i>=j and j>1 and i<=2*N and j<=N then
          l[i,j]:=(a[i,j]-add(l[i,k]*u[k,j],k=1..j-1))*u_inv[j]
      fi;
    od
od;
```

The only way all divisions can occur in the same PE happens when u_inv[j] (a line in space-time) aligns with the time axis. Thus, an architectural constraint is set that only looks for these solutions. With this constraint set, SPADE finds one area-optimal design, that shown in Fig. 3, which has the same 5N-2 time latency as that in Fig. 1, but $(2N-1)^2$ area. (Note that in Fig. 3 the single PE with the divider is in the upper right hand corner of the figure and isn't labeled.)

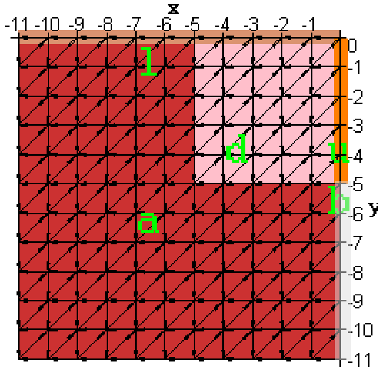

Figure 3. Single optimal design with one divider (N=6).

5. Space-time graphical output

Unfortunately, the mathematical outputs *T,t* provide little insight into the nature of the solution, especially from the designer's point of view. For this purpose graphical outputs have been included as part of SPADE. The two primary mapping views when the dimension of space-time is three are the 2-D spatial-only views seen in Figs.1-3 and a 3-D view that shows the mapped algorithm variables in space-time along with a projection of all these variables onto just the spatial plane. Such a space-time view example is shown from two different perspectives in Fig. 4 order to help in its interpretation (in the SPADE environment this view can be easily manipulated along all axes in real-time). The result in Fig. 4 corresponds to the spatial-only view in Fig. 2. Fig. 4 is more complex because it shows additional variables, *IM1* through *IM4* (e.g., *IM1[i,j,k]=l[i,k]*u[k,j]*). These new variables are created automatically [1] in the parser and are there to keep running sums associated with the summations in (10).

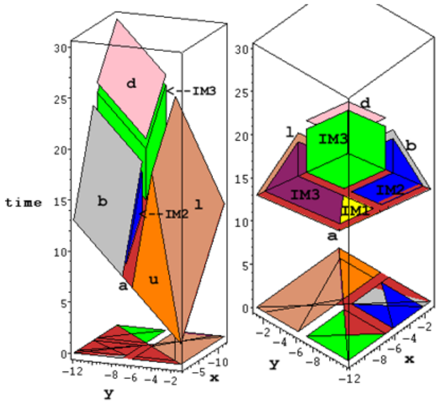

Figure 4. Space-time mappings corresponding to design in Fig. 2

The space-time view 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, and it imparts a rough estimate of how efficiently PEs are used by what percent of the total space-time volume is occupied by polytopes and polygons.

6. Acknowledgements

This work was supported in part by DARPA Contracts DAAH01-96-C-R135 and DAAH01-97-C-R107.

7. References

[1] Baltus, Donald and Allen, Jonathon, "Efficient Exploration of Nonuniform Space-Time Transformations for Optimal systolic Array Synthesis," Proc. Application specific Array Processors, 1993, pp.428-441.

[2] Darte, A., Robert, Y., Vivien, F., "Scheduling and Automatic Parallelization", Birkhauser, 2000.

[3] Feautrier P., "Some Efficient Solutions to the Affine Scheduling Problem, Part III",Int.J. of Parallel Programming,Vol. 21(6):389-420, Dec. 1992.

[4] Guillou, A., Quinton, P, Risset, T., Massicotte, D., "Automatic Design of VLSI Pipelined LMS Architectures", Proc. Int. Conf. Parallel Computing in Electrical Engineering (PARELEC'00).

[5] Kung,S.Y., "VLSI Array Processing", Prentice Hall, 1988.

[6] Manjunathaiah, M., Megson, G.M., Rajopadhye, S., and Risset, T., "Uniformization of Affine Dependence Programs for Parallel Embedded System Design, Proc. 2001 Int. Conf. Parallel Proc. (ICPP 2001),pp.205-213.

[7] Moldevan, D. I., "Parallel Processing", Kaufmann, 1993.

[8] Moreno, J. and Lang, T., "Matrix Computations on Systolic-Type Arrays," Kluwer Publishers, 1992.

[9] Nash, J.G.and Hansen, S., "Modified Faddeeva Algorithm for Concurrent Execution of Linear Algebraic Operations," IEEE Trans. Computers, Feb.1988, pp.129.

[10] Nash, J. G., "Constraint Directed CAD Tool For Automatic Latency-Optimal Implementation of 1D and 2D Fourier Transforms", Proc. SPIE, Boston, July 29, 2002, and www.centar.net

[11] Quinton, P. and Robert, Y., "Systolic Algorithms and Architectures", Prentice Hall 1991.

[12] Roychowdhury, V.P, "Derivation, Extensions and Parallel Implementations of Regular Iterative Algorithms," Ph.D Thesis, Stanford, 1989.

[13] Roychowdhury, V.P., Rao, S.K., Thiele, L., Kailath, T.,"On the Localization of Algorithms for VLSI Processor Arrays", VLSI Signal Processing III, Ed. by R.W. Brodersen and H.Moscovitz,IEEE Press 1988,pp.459-470.

[14] Rajopadhye, S.V. and Fujimoto, R.M., "Automating the Design of Systolic Arrays," Integration, the VLSI Journal 9, (1990), Elsevier Science Pub., pp.225-242.

[15] Schreiber, R., et. al., "High-Level Synthesis of Nonprogrammable Hardware Accelerators", Proc. IEEE Int. Conf. Application Specific Systems, Architectures, and Process, (ASAP 2000).

[16] Stone, A. and Manolakos, E., "DG2VHDL: A Toll to Facilitate the High Level Synthesis of Parallel Processing Array Architectures", J. VLSI Signal Processing Systems, Vol.24, 99-120 (2000).

[17] Van Swaaij, M. , Catthoor, F., and DeMan, H., "Nonlinear Transforms for High Level Regular Array Synthesis: A Case Study," J. VLSI Signal Processing, vol. 4., pp. 259-268, 1992.