

Automatic Generation of Systolic Array Designs For Reconfigurable Computing

Dr. J. Greg Nash, Centar (centar@att.net)

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.

1. Introduction

Meeting latency and throughput requirements is a critical concern in many embedded signal and image processing applications. Consequently, a vast 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, discrete 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 provide an adequate level of speed, density and programmability in the form of reconfigurable computers, boards, and chips. Such hardware could allow dynamic implementation of systolic algorithms leading to inexpensive "programmable" systolic array hardware. Furthermore, the architectural characteristics of much FPGA hardware matches that required by systolic processing, because this technology is constructed from tiling identical memory and logic blocks along with supporting mesh interconnection networks.

The primary reason for the limited role of systolic processing is that to date not a single 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, and few people possess these
- 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
- 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 [2],[12],[13].

The intention of this paper is to provide a broad overview of the SPADE tool and environment, with a specific emphasis on how a circuit designer could use it to explore a variety of parallel array implementations.

2. Related Work

Over the past 20 years much research has been directed at finding systematic methodologies for finding optimal parallel implementations of recursive or iterative algorithms. The practical motivation has always been that such systematic techniques could be reduced to a software tool, which would enable a much larger, non-experienced user community to rapidly and easily develop special purpose, fine-grained parallel computing hardware. Three basic approaches to designing relevant architectures have been proposed, those based on data dependencies [5],[12], index transformations [10],[14],[15], and the parameter method [8].

Index transformations

This design 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.

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

$$\begin{aligned} & \text{for } 1 \leq i, j, k \leq N \\ & a[i, j+1, k] = a[i, j, k] \\ & b[i+1, j, k] = b[i, j, k] \\ & c[i, j, k+1] = c[i, j, k] + a[i, j, k] * b[i, j, k] \end{aligned} \tag{1}$$

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.

Many mapping tools have been designed for such sets of uniform recurrence equations and the related problem of serially coded loops nests with uniform dependencies [11]. A recent effort has resulted in a tool (PICO-N), which generates synthesizable VHDL code describing embedded systolic arrays from functions expressed as loop nests in C [16].

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.

Our tool approach generalizes the acceptable input algorithm forms to systems of affine recurrence equations that closely resemble the form 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

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

rather (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 .

Data Dependencies

Rather than focus on the indices used by variables, the data dependency approach is based on creation of a "dependence graph" that shows the flow of dependencies between variables. An early tool, VACs [6], creates a visual "Space-Time" dependence graph and gives the user enough flexibility to choose how nodes in this graph are allocated to virtual processors and how to schedule the nodes. The major limitation is that the graph, and hence the original algorithm, must possess a shift invariant regularity that restricts its utility in the same way that described for uniform algorithms.

A more elaborate design methodology, the Multi-mesh Graph (MGG) method [12] does not impose uniform dependency regularity restrictions on the input algorithm. Two partial tool efforts based on this methodology, MAMACG [6] and ALIAS [8], have provided some demonstration of the MMG methodology. These tools generate a visually complete algorithm dependence graph to guide algorithm mapping, but it is still left for the user to decide what transformations to apply and their order. Furthermore, all variables must be heuristically embedded in this dependence graph space and there are no design guides to achieving optimality.

SPADE

The difficulty in working with non-uniform equations like (2) is that dependencies between variables are no longer local and thus the path to a systolic implementation is more difficult. There are methodologies that can generate mappings with localized dependencies [15], but scheduling these remains an unsolved problem.

In order to solve the general problem of non-uniform affine recurrence equations when desirable architectural constraints are imposed, it is necessary to use non-linear integer programming methods [4]. 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 Cathedral IV [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 [1] uses algorithm latency as an objective function metric. One important difference between the two tools is that the DESCARTES search is structured to find optimal designs, whereas the Cathedral search

process is more limited and optimal designs aren't guaranteed. Optimal parallel solutions are critically needed for embedded applications where system size, weight, power, and volume constraints drive costs extremely high for non-optimal solutions. 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. It 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.

3. Spade Description

3.1 Introduction

Past difficulty in obtaining optimal solutions for parallel algorithm implementations has been due to the many complex, interrelated steps that are involved and the lack of any systematic approaches to dealing with them:

1. **Algorithm representation:** How a user describes his problem to the software tool involves various tradeoffs. For example, finding an algorithm form like (1) is difficult, but, once achieved, it simplifies the mapping process.
2. **Scheduling:** Finding an execution sequence for the pieces of an overall computation (e.g., making sure that all variables necessary to do the computation are available at the right PE at the correct time).
3. **Reindexing:** Altering the indexing relationships between different variables to best expose regularity.
4. **Localization or regularization:** The process of decomposing global dependencies, like "broadcast" and "fan-in" operations, into sequences of local data movement, so that all data movement between PEs can be performed by a nearest neighbor connection network.
5. **Allocation:** The process of allotment of computations to virtual processors in spatial coordinates

The steps above are not independent because a choice made at any step can affect those at other steps. For example, allocation and scheduling steps would be in conflict if two computations with the same scheduled execution time were allocated to the same processor. The "reindexing step" has been particularly difficult because it involves making choices in the indexing relationships between variables. Historically users have had to make such choices based on previous experience and knowledge of an algorithm's peculiarities.

There is no prescribed order to the steps above and different mapping methodologies sequence the steps differently. For example use of uniform recurrence equations like (1) require that the localization process occur first because it is inherent in the algorithm description.

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.

3.2 Space-Time Mapping

In this section a specification is provided for the form of non-uniform recurrence equations that SPADE accepts. Also, the space-time mapping formalism used is summarized

A formal description of non-uniform recurrence equations has been provided by Baltus [1], where they are termed conditional affine recurrence equations. In this context "conditional" is intended to imply that each equation can have its own unique index space. (In an alternative but similar form [15], they are termed "linear indexed weak single assignment codes"). A system of non-uniform affine recurrence equations is

$$\begin{aligned} w_1(A_1(I)) &= g(\dots w_i(B_{i1}(I)), \dots) && \text{for all } I \text{ in } I_1 \\ &\dots && \\ w_n(A_n(I)) &= f(\dots w_i(B_{in}(I)), \dots) && \text{for all } I \text{ in } I_n \end{aligned} \quad (3)$$

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 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

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

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. Consequently, every point in space-time associated with $T(a)$ represents a source of a data element moving in direction v_{ca} to a corresponding point in $T(c)$. The transformations T are such that this data element arrives just in time to be used (along with other data) in a calculation that produces an element of the result c .

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 $\Lambda_x = [\lambda_{x1} \ \lambda_{x2}]$, SPADE considers λ_{x1} and λ_{x2} as "search" variables (as distinct from previously mentioned "algorithm variables"). Following Baltus [1], the allocation matrices like S_x are treated as a single variable, unimodular matrices are used to ensure that the space-time mapping is "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. Baltus [1] solves this problem by introducing computational and architectural constraints that limit the space of solutions that has to be searched. 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

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

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

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 PE array (hence it's projection onto the PE array is a line). This constraint requires that the normal n_a to the plane of a satisfy $u_t \bullet n_a = 0$. Also, to be at the edge of the PE array it must not be within the convex hull of the polygon defining the PE array. Such constraints as these considerably limit choices for Λ_a and S_a .

The search strategy used in SPADE is similar to that of DESCARTES but with several modifications. For example, DESCARTES's search is depth first with a suitable ordering of search variables that maximizes its efficiency. This works best when desired architectures were highly constrained and the solution space was small. However, when fewer constraints are involved, it was found to be more computationally efficient to introduce a degree breadth first processing, e.g., often the overhead from invoking a complex matrix computation is large enough to justify processing all matrix search variables at that search depth rather than the one search variable in a depth first approach.

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 Maple¹ 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;
```

¹ Maple is a commercial "scientific" programming environment (Waterloo Maple Inc).

where the Maple "add" construct directly replaces the mathematical summation sign. This code as written runs correctly in both SPADE and Maple. 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. It does not always happen that code written for SPADE will run correctly in Maple and vice-versa; so the user must be wary of the distinctions between recurrence equations and traditional serial code. When conditionals are used in code input to SPADE, they override the loop limits 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 and (2) whether overlap of variables in space-time is allowed. After SPADE finds all minimum latency solutions, a 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 Algorithm Design Example

4.1 Introduction

In this section the Faddeev algorithm is used as an example to illustrate the functionality of SPADE. The two main goals of this section are to show that:

1. mathematical derivations can be used almost directly as an input to SPADE, without the necessity for heuristically obtaining a uniform algorithm form
2. a circuit designer can easily explore a variety of array implementation tradeoffs
3. it is not necessary for the user to be intimately familiar with parallel algorithms or architectures

Details related to array designs, such as PE construction, partitioning, where variables appear and how they flow in the array are not covered here since much information on such subjects is already available [2],[12],[13]. Also, comparing the different designs presented here to previously published designs or even isolating all the interesting parallel Faddeev architectures is well beyond the scope of this paper.

4.2 Algorithm Derivation

The Faddeev algorithm [3] 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 $N \times N$ matrix. It is convenient to describe the algorithm as an extended matrix using the representation

$$\frac{A \mid B}{-C \mid D} \quad (6)$$

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

$$\frac{A \mid B}{0 \mid CA^{-1}B + D} \quad (7)$$

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

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

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. If we let

$$a = \frac{A \mid B}{-C \mid D},$$

then the desired decomposition is

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

where the "-" above indicates these elements l_{ij} are not computed and each matrix is taken as 2×2 . 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,

$$\begin{aligned} u_{11} &= a_{11}; u_{12} = a_{12}; u_{13} = a_{13}; u_{14} = a_{14}; \\ l_{11} &= 1; l_{21} = a_{21}/u_{11}; l_{31} = a_{31}/u_{11} \\ &\text{for } i \geq j, j > 1, i \leq 4, j \leq 2 \\ l_{ij} &= (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{ii} \\ &\text{for } j \geq i, i > 1, i \leq 2, j \leq 4 \\ u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \\ &\text{for } i > 2, j > 2, i \leq 4, j \leq 4 \\ u_{ij} &= a_{ij} - \sum_{k=1}^N l_{ik}u_{kj} \end{aligned} \quad (8)$$

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(1[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
      u[i,j] := a[i,j] - add(1[i,k]*u[k,j],k=1..N)
    fi;
    if j=1 and i>=1 and i<=2*N then
      1[i,j] := a[i,j]/u[j,j];
    fi;
    if i>=j and j>1 and i<=2*N and j<=N then
      1[i,j] := (a[i,j] - \
        add(1[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 $N \times N$ in size. In this example it is possible to run this code directly in Maple to verify it's functionality.

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 SPADE 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)
    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 saved as a text file and read into SPADE at the beginning of processing by the parser.

This completes process associated with algorithm design. It can be seen that a suitable algorithm input to SPADE can be derived from first principles and the corresponding mathematical expressions can be used directly as an input. It is noteworthy that no steps in this process involved issues requiring significant understanding of parallel algorithms or architectures.

4.3 Faddeev Algorithm Design Results

The critical function of SPADE is to provide the circuit designer with a complete range of architectural options so that tradeoffs can be adequately analyzed. This is necessary because available FPGA chips, boards and virtual computers are built with different architectural features and any system implementation based on such technologies will have its own unique set of constraints. The purpose of this section is to provide a few examples of how such considerations can result in very different array designs.

4.3.1 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's projected 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 requirement can be done by setting a constraint that eliminates exploration of designs that don't map 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 $\frac{1}{2}N(3N+1)$ PEs, that are shown in Figures 1 and 2. The PE arrays in Figure 1 and 2 are 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 Figure 1 and 2. Thus, some PEs can experience many different data streams passing through them. There are five additional dependencies in which data 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.

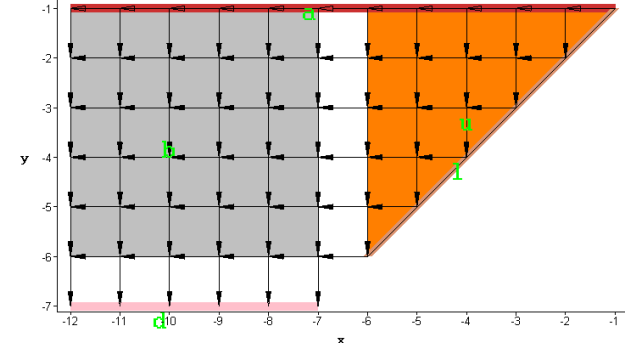


Figure 1. The first of two minimum area designs for Faddeev algorithm with variables a and l constrained to appear on the array boundary ($N=6$).

The picture in Figures 1 and 2 show 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. Typically data movement begins at one edge of the array and proceeds outwards in a "wavefront-like" fashion.

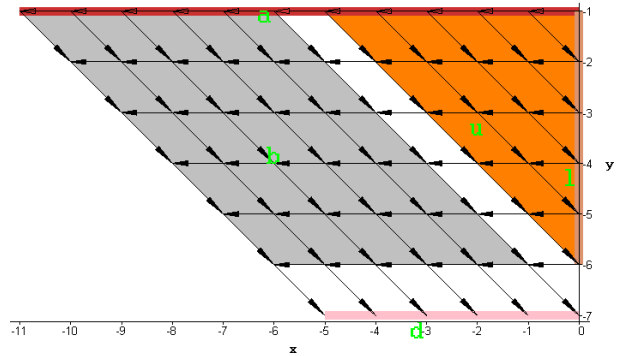


Figure 2. The second of two minimum area Faddeev algorithm mappings with input variables a and l constrained to appear on the array boundary ($N=6$).

While the design in Figure 2 is only slightly different topologically, it could be of interest because of the orthogonal placements of a , l and d . The design in Figure 1 is the same architecture on which most past analyses of the Faddeev algorithm have been based [2],[12],[13].

4.3.2 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 here is one that minimizes the number of different interconnects and prefers orthogonal data movement. An array design is penalized for each different data flow arrow 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, as that shown in Figure 3. Therefore, the regularity criteria penalize designs of this type.

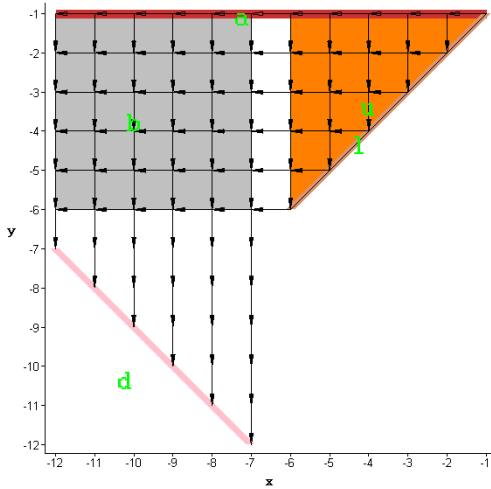


Figure 3. A valid array design ($5N-2$ latency) with undesirable placement of output, d .

Using the criteria above as the secondary objective function, SPADE produces the single optimal design ($5N-2$ latency) shown in Figure 4. 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 Figure 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.

It is easily possible to find a design similar to the square array in Figure 4 that places only the l variable on a PE edge boundary (linear array of dividers). This is done by looking amongst the set of designs comprising those whose regularity criteria are at least in the top three. However, this design has additional interconnects and has higher data bandwidth requirements.

4.3.3 Other Faddeev Array Implementations

As mentioned in Section 3, SPADE array designs are based on the variables, their indexing relationships, and index domains specified by the input code. Here, the input code (10) is modified further by introducing other variables to illustrate this point, in particular those that would arise when the a variable, which contains all inputs, is

broken into its component matrices. This gives SPADE additional flexibility to place variables in different locations that yield new array designs. This modified code (11) includes separate input variables A and C . While introduction of new variables increases the design space, it also increases the computational complexity of the search process. Consequently, the inputs B and D are handled differently by initializing b and d with these values. This provides much of the flexibility but without additional computational requirements. The resulting code is

```

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 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]:=b[i,j]-add(l[i,k]*b[k,j],k=1..i-1)
    fi;
    if j>=1 and i>=1 and j<=N and i<=N then
      d[i,j]:=d[i,j]-add(l[i+N,k]*b[k,j],k=1..N)
    fi;
    if j=1 and i>=1 and i<=N then
      l[i,j]:=A[i,j]/u[j,j];
    fi;
    if i>=j and j>1 and i<=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;
    if j=1 and i>N and i<=2*N then
      l[i,j]:=C[i-N,j]/u[j,j];
    fi;
    if i>N and j>1 and i<=2*N and j<=N then
      l[i,j]:= (C[i-N,j]-\
        add(l[i,k]*u[k,j],k=1..j-1))/u[j,j]
    fi;
  od
od;

```

As expected, this code produces the same designs as previously seen, but others as well. For example, with the secondary objective function set to find minimum area solutions, although no new designs with less than a $5N-2$ time latency were found, two additional minimum area designs were found similar to those in Figures 1 and 2, one of which is shown in Figure 5. Here the main difference is that the variable C is now located coincident with the variable l on the PE array diagonal. Other potentially useful designs were found as well.

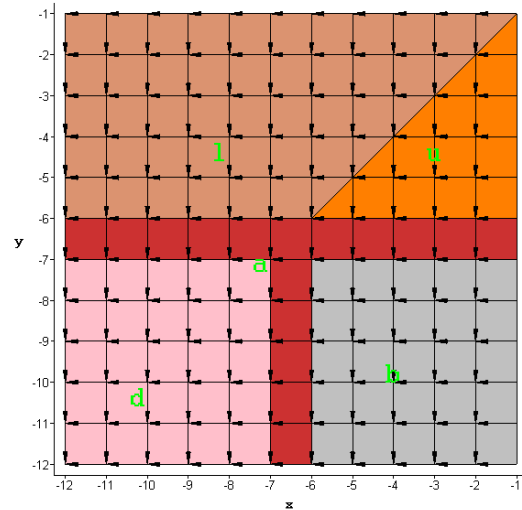


Figure 4. Faddeev algorithm mapping obtained using regularity as secondary objective function ($N=6$).

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 tools 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 the previous figures 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. A space-time view example is shown from two different perspectives in Figure 6 in 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 Figure 6 corresponds to the spatial-only view in Figure 4.

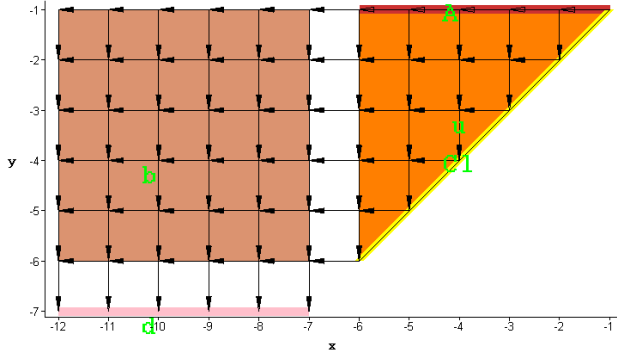


Figure 5. Minimum area Faddeev algorithm mappings obtained using code (11) and same constraints as designs in Figures 1 and 2 (N=6).

Figure 6 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).

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.

Finally, there is a third output for algorithms for which the space-time dimension is two. In such cases there is only one 2-D graphical output. This shows the data flow directions with arrows and is overlaid by polygons and lines indicating the positioning of variables in space-time.

6. Simulator

There is also a simulator that can be run to mimic operation of the systolic array. It takes as inputs the various mathematical quantities that specify the algorithm mapping and then simulates activity in each PE during the systolic computation. Each simulator time step consists of two phases. First there is data transfer between PEs; then there is (potentially) a computation within each PE.

This is a very abstract model of a real systolic array; that is, it presumes there is a data path available for all data flow directions, that there is a physical PE for each virtual PE, that there is no PE control "overhead", and that all computation is equivalent and can be done in one-half time step.

However, the simulator does represent a useful model of computation. In particular it is initialized in somewhat the same manner a real systolic array would be and the simulation mimics the actual flow of data and abstract computation activity in each

For example, sets of registers are defined that are associated with specific data dependencies, such that when a data item is transferred, its direction of propagation is included along with the data. This information is used to control its movement during subsequent time steps. Also, each PE is initialized with (or without) information about what computations it performs. In a real implementation PE control

could possibly be localized with each PE having its own unique finite state machines.

The simulator is instrumented to provide various outputs useful in designing systolic arrays (e.g., computational efficiency) and in understanding how the design works via visual pictures of data positions as a function of time.

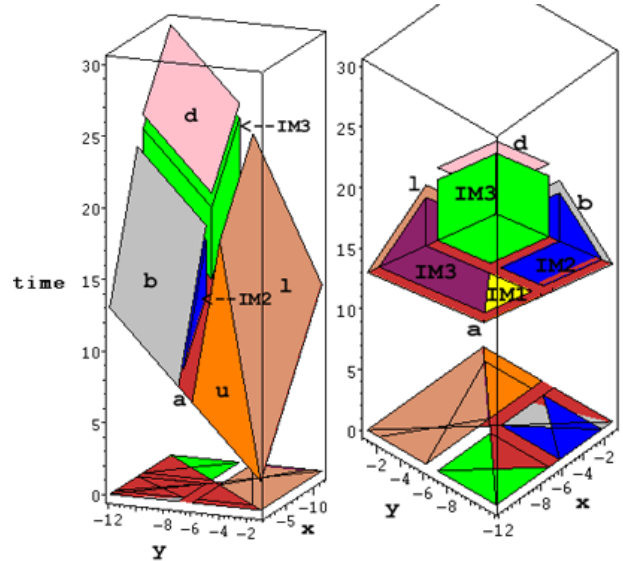


Figure 6. Space-time view of algorithm mapping shown in Figure 4 (N=6).

7. Matrix Transformations

Once an array design is selected, the actual transformation matrices are saved and available as inputs to other components of the environment, in particular the simulator (Section 6). For illustration the dependency information (data flow vectors) is provided in Table 1 for the particular design shown in Figure 1 and the transformation matrices are given in Table 2. Although there are a large number of dependencies (data flow directions), each of these apply over only a fraction of the spatial plane, so their overall number does not have a direct bearing on the I/O bandwidth of a specific PE.

8. Summary

Although FPGAs conceptually provide an excellent implementation strategy for systolic arrays, a designer faces a large number of design options, given the variety of reconfigurable computers, boards and chips that are becoming available. For a given algorithm different systolic array mappings will present different tradeoffs from which he must make optimal choices. Further, design complexity is introduced when building programmable systolic arrays that support several different algorithms. In this case a designer must consider architectural tradeoffs among systolic array mapping possibilities for each of the different algorithms he wants to support. Having available only a few systolic "point" designs will lead to sub-optimal implementations. Consequently, the utility of this tool is that it can facilitate identification of such tradeoffs rapidly and easily.

It should be noted that the exhaustive search procedures used in SPADE will not always guarantee mapping solutions. The reason for this is that dependencies expressed in the algorithm code might preclude a mapping; in such cases alternative codings must be explored. The primary goal of SPADE is to move the level of abstraction at which the designer must work away from the realm of detailed architectural and timing issues to the more familiar world of high-level code. An example of how this might be accomplished was demonstrated by showing how code changes in (10) and (11) could result in different array designs.

Table 1. List of direction vectors ($time,x,y$) of data flow for the various dependencies corresponding to the transformation matrices in Table 2..

dependencies	v
$u \rightarrow a$	$[1 \ 0 \ -1]$
$u \rightarrow IM1$	$[1 \ 0 \ -1]$
$b \rightarrow a$	$[1 \ 0 \ -1]$
$b \rightarrow IM2$	$[1 \ 0 \ -1]$
$IM1 \rightarrow l$	$[1 \ -1 \ 0]$
$IM1 \rightarrow u$	$[1 \ 0 \ 0]$
$l \rightarrow a$	$[1 \ 0 \ -1]$
$l \rightarrow u$	$[1 \ 0 \ 0]$
$l \rightarrow IM4$	$[1 \ 0 \ -1]$
$IM2 \rightarrow l$	$[1 \ -1 \ 0]$
$IM2 \rightarrow b$	$[1 \ 0 \ 0]$
$d \rightarrow a$	$[1 \ 0 \ -1]$
$d \rightarrow IM3$	$[1 \ 0 \ -1]$
$IM3 \rightarrow l$	$[1 \ 1 \ 0]$
$IM3 \rightarrow b$	$[1 \ 0 \ 0]$
$IM4 \rightarrow l$	$[1 \ -1 \ 0]$
$IM4 \rightarrow u$	$[0 \ 0 \ 0]$

9. Acknowledgements

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

10. 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] H.Y.H. Chuang and He, G., "A Versatile Systolic Array for Matrix Computations," pp. 315-322, Proc. 12th Sym. on Computer Architecture (June 1985).
- [3] Faddeev, D.K., and Faddeeva, V.N., "Computational Methods of Lindar Algebra, pp. 150-158, W. H. Freeman and Co. (1963)
- [4] Feautrier, Paul, "Fine Grain Scheduling under Resource Constraints," 7th Workshop on Language and Compilers for Parallel Computers, Aug. 1994.
- [5] Kung, S.Y., "VLSI Array Processing", Prentice Hall, 1988.
- [6] Kung, S.Y., and Jean, S.N., "A VLSI Array Compiler System (VACS) for Array Design", Proc. IEEE Workshop on VLSI Signal Processing, pp.495-508,1988.
- [7] Le, Dinh, et.al, "MAMACG: A Tool for Automatic Mapping Matrix Algorithms Onto Mesh Array Computational Graphs",

Proc. 1992 Application Specific Array Processors, IEEE Computer Society Press, p. 511-525.

- [8] Li, G.I., and Wah, B.W., "The Design of Optimal Systolic Arrays," IEEE Trans. on Computers, Vol. C-34, pp. 66-77, Jan. 1985.
- [9] Liu, James and Ercegovic, M.D., "ALIAS Environment: A Design Tool for Application Specific Arrays", In Fifth IEEE Symposium on Parallel and Distributed Processing, Dec. 1993.
- [10] Moldevan, D. I., "Parallel Processing", Kaufmann, 1993.
- [11] Moldevan, D.I., "ADVIS: A Software Package for the Design of Systolic Arrays," IEEE Trans Computer Design, pp.33-40, 1987.
- [12] Moreno, Jaime and Lang, T., Matrix Computations on Systolic-Type Arrays, Kluwer Publishers, 1992.
- [13] Nash, J.G. and Hansen, Siegfried, "Modified Faddeeva Algorithm for Concurrent Execution of Linear Algebraic Operations," IEEE Trans. Computers, Feb.1988, pp.129.
- [14] Quinton, P. and Robert, Y., "Systolic Algorithms and Architectures", Prentice Hall 1991.
- [15] Roychowdhury, V.P., S.K. Rao, L. Thiele, and T. Kailath, "On the Localization of Algorithms for VLSI Processor Arrays," VLSI Signal Processing III, IEEE Press, 1988, pp. 459-470.
- [16] Schreiber, R., et. al., "High-Level Synthesis of Nonprogrammable Hardware Accelerators", Proc. IEEE Int. Conf. Application Specific Systems, Architectures, and Processors, IEEE Computer Society, July, 2000, p. 113-124. (Includes many references to other related tools.)
- [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.

Table 2. Transformation for array mapping shown in Figure 1.

name	T	t
a	$\begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$
u	$\begin{bmatrix} 2 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
l	$\begin{bmatrix} 1 & 2 \\ 0 & -1 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
$IM1,4$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
$IM2$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} N-1 \\ -N \\ 0 \end{bmatrix}$
b	$\begin{bmatrix} 2 & 1 \\ 0 & -1 \\ -1 & 0 \end{bmatrix}$	$\begin{bmatrix} N-1 \\ -N \\ 0 \end{bmatrix}$
d	$\begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 3N \\ -N \\ -N-1 \end{bmatrix}$
$IM3$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 2N-1 \\ -N \\ 0 \end{bmatrix}$