Toward a Systematic Approach to the Design of Systolic Arrays

Weichang Du

Department of Computer Science
University of Victoria
B.C. Canada V8W 2Y2
wdu@uvunix.bitnet

Abstract

Intensional programming has the potential for the systematic design of systolic arrays. It can be used to study the three aspects of the design of systolic arrays: systolic algorithms, systolic architectures and systolic computing. A systolic algorithm can easily be expressed in a well-designed intensional language; a program in the intensional language with spatial and temporal operators can specify a systolic architecture; and the eduction model for computing intensional programs can also be a basis for the simulation of systolic computation.

An intensional language, Plane Lucid can be extended to specify both systolic algorithms and systolic architectures formally. When a systolic algorithm is expressed by a program in Plane Lucid, the spatial relations defined by spatial operators among variables in the program specify the size and topology of the desired systolic array; the functions describe the operations performed by processors; and data dependencies constrained by the spatial and timing relations specify the order and timing of data communication.

An intensional 3-D spreadsheet based on Plane Lucid can be used as a design environment and a test bed for systolic array designs. We can treat a set of adjacent cells in the spreadsheet as a processor in the systolic array, the formulas in the set of cells define the operations performed by the processor, and the spatial and timing relations between two sets specify the communication between the two processors.
1. Introduction

Systolic arrays are the result of advances in semiconductor technology and of applications that require extensive throughput. Their realization requires a combination of various techniques and tools for algorithm development, architecture design, and parallel programming. These areas are often identified as systolic algorithms, systolic architectures, and systolic computing.

The key concept in systolic algorithms is the decomposition of the problem into subcomputations that are assigned to dedicated processors with the data flowing through the processors, visiting all or an appropriate subset of processors to complete the computation for the input. The systolic architectural concept was originally developed by Kung and associates at Carnegie-Mellon University[17], and many versions of systolic arrays have been designed by universities and industrial organizations. A systolic system consists of a set of processors, each of which is capable of performing some simple operations. All the processors are interconnected to form a systolic array. Information in a systolic system flows between processors in a pipelined fashion; data passes from one processor to neighboring ones in a regular, rhythmic pattern. The communication with the outside world in a systolic array occurs only at the boundary processors.

The merit of systolic approach lies in these properties: locality of communication, a regular communication structure, and having only a few types of simple processors. Systolic algorithms use a limited number of input/output interface processors, while operating at roughly the same speed as many other algorithms that require many more input interfaces; if the processors are arranged in a grid-like way, systolic algorithms will minimize propagation delay among processors. Furthermore, from the point of view of VLSI design and implementation, simple, regular communication and control structure have substantial advantages over complicated ones.

To build a systolic array from the description of an algorithm, the designer needs a thorough understanding of, and familiarity with, the principles behind four things: systolic computing, the application, the algorithm, and the technology. Such skilled designers can provide excellent heuristic designs for important algorithms. However, the process is slow and error prone, and the resulting designs are not guaranteed to be optimal or correct. While numerous systolic designs are known today, the question of their automatic design is still open, though progress has been made in the development of systematic design techniques to automate this process[18][19]. The systematic design techniques are expected to provide tools and formal concepts to assist designers in searching for various desirable designs for a given application. For implementation and proof purposes, most of the techniques are concerned with the derivation of a relatively high-level language with formal and rigorous notation to specify, verify, and optimize the systolic array architecture from a description of the systolic algorithm. Typically, such a language can specify the size and topology of the array, the operations performed by each processor, the order and timing of data
communication, and inputs and outputs. Thus, a program written in the language is both an expression of the systolic algorithm and the description of the corresponding systolic architecture abstracted at some level. The parallel evaluation of the program, which depends on the order and timing of data dependency among various variables in the program, is a simulation of systolic computation in the architecture.

In this paper, we intend to show that intensional programming has potential for systematic design of systolic arrays. Intensional logic is concerned with assertions and other expressions whose meaning depends on an implicit context. Intensional programming means that programming in a language which is at the same time a formal system based on intensional semantics and provides with context switching operators that allow values from different contexts to be combined without explicit context manipulation[PaWa85]. Intensional programming techniques can be used to study the three aspects of the design of systolic arrays. A systolic algorithm can easily be expressed in a well-designed intensional language; a program in the intensional language with spatial and temporal operators can specify a systolic architecture; and the eductive model for computing intensional programs can also be a basis for simulation of systolic computation.

An existing intensional language Plane Lucid[Du86], which is an extension of Lucid[WaAs85], is one of many possible intensional languages whose specialized version may be used as a specification language to describe both systolic algorithms and abstract systolic architectures. Plane Lucid is a "three-dimensional" intensional language in which every variable varies in two spatial dimensions and a temporal dimension. In other words, the value of a variable in a Plane Lucid program depends on the spatial position and the moment in time, i.e. where and when the variable is evaluated. When a systolic algorithm is expressed by a program in Plane Lucid, the spatial relations defined by spatial operators among variables in the program specify the size and topology of the desired systolic array; the functions describe the operations performed by processors; and data dependencies constrained by the spatial and timing relations specify the order and timing of data communication.

Also, an existing intensional 3-D spreadsheet[Du86], in which Plane Lucid is the definition language, may be extended to form a design environment and a simulation bed for systolic array designs. In the intensional spreadsheet, different from conventional spreadsheets, the whole spreadsheet is considered as an entity that varies in space and time. The relations among cells in the spreadsheet are defined by the intensional (or temporal and spatial) operators in formulas of cells. We can treat a set of adjacent cells as a processor in the systolic array, the formulas in the set of cells define the operations performed by the processor, and the spatial and timing relations between two sets specify the communication between the two processors.

In the next section we briefly introduce Plane Lucid and the intensional spreadsheet. In Section 3, we discuss the possibility of realizing systematical design of systolic arrays by using intensional programming.
Section 4 is some concluding remarks.

2. Plane Lucid and the Intensional Spreadsheet

2.1. Plane Lucid

A natural extension of Lucid is formed by allowing intensions to vary in space as well as in time. The idea is to change the semantics of Lucid by allowing the values of expressions to be time sequences of elements that themselves vary in space. If Lucid is considered as a one-dimensional (temporal) language, the extended language may be considered as a multi-dimensional (a temporal dimension and one or more spatial dimensions) language. In particular, we are interested in a three dimensional (a temporal dimension and two spatial dimensions) version called Plane Lucid.

Plane Lucid provides a number of operators that permit the values of expressions to vary in space and time. These operators are called \textit{intensional operators}. For the all three dimensions, there are five kinds of intensional operators to switch context. The intensional operators for the horizontal dimension in space are: right, left, hsby(for "horizontally succeeded by"). hpby(for "horizontally preceded by"). and side. Similarly, the corresponding intensional operators for the vertical dimension in space are: up, down, vsby(for "vertically succeeded by"). vphy(for "vertically preceded by") and edge. The corresponding intensional operators for the temporal dimension are: next, before, fby(for "followed by"). pby(for "preceded by") and first. Other operators in Plane Lucid are pointwise. In the following, we informally describe the meanings of the spatial operators in the horizontal dimension. The meanings of others are not difficult to understand from these descriptions.

The unary operator \texttt{right} and \texttt{left} shifts the context one point right and left in space, respectively. The value of \texttt{right} x and \texttt{left} x at a given spatial point and a given time is the value of x at the spatial point immediately to the right and the left of the given point in space, respectively, at the same time.

The binary operator \texttt{hsby} combines two different spatial operations, the \texttt{left} operation and the pointwise operation. If the horizontal coordinate of the given spatial point is positive, the value of x \texttt{hsby} y at a given spatial point and a given time is the value of y at the spatial point immediately to the left of the given one at the same time (which is the same as a \texttt{left} operation); otherwise, it is the value of x at the same spatial point at the same time. For example, the definition

\begin{verbatim}
  x where x = 0 hsby (x + 1) ; end ;
\end{verbatim}

defines x as a variable whose value at any spatial point is equal to the horizontal coordinate of that point, no matter what the time is, if the horizontal coordinate is positive; otherwise the value of x is 0.

The other binary operator \texttt{hpby} combines the \texttt{right} operation and the pointwise operation in a similar way. For example, the definition

\begin{verbatim}
\[ x \text{ where } x = (x-1) \text{ hpby } 0 \; ; \text{ end} \]
defines \( x \) as a variable whose value at any spatial point is equal to the horizontal coordinate of that point, no matter what the time is, if the horizontal coordinate is negative; otherwise the value of \( x \) is 0. Thus the combination of the above two definitions of \( x \)

\[ x \text{ where } x = (x-1) \text{ hpby } 0 \text{ hsby } (x+1) \; ; \text{ end} \]
defines \( x \) as the index of the whole horizontal dimension.

The unary operator side evaluates the value of its operand at the horizontal coordinate 0. The value of \( x \) at a given spatial point at a given time is the value of \( x \) at the spatial point which has the same vertical coordinate as the given point but has the horizontal coordinate 0, and at the same time.

The negative space coordinates permit intensional programming on a full plane. The negative time coordinates are also expected to be useful for some applications. Table 1 lists the informal meanings of all intensional operators in Plane Lucid.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Evaluation Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>right ( x )</td>
<td>( x ) at the spatial point to the right of the evaluation point</td>
</tr>
<tr>
<td>up ( x )</td>
<td>( x ) at the spatial point above the evaluation point</td>
</tr>
<tr>
<td>next ( x )</td>
<td>( x ) at the temporal point next to the evaluation point</td>
</tr>
<tr>
<td>left ( x )</td>
<td>( x ) at the spatial point to the left of the evaluation point</td>
</tr>
<tr>
<td>down ( x )</td>
<td>( x ) at the spatial point below the evaluation point</td>
</tr>
<tr>
<td>before ( x )</td>
<td>( x ) at the temporal point before the evaluation point</td>
</tr>
<tr>
<td>( x ) hsby ( y )</td>
<td>( \text{(left } y\text{) if the current horizontal coordinate } \geq 0 \text{ else } x )</td>
</tr>
<tr>
<td>( x ) vsby ( y )</td>
<td>( \text{(down } y\text{) if the current vertical coordinate } \geq 0 \text{ else } x )</td>
</tr>
<tr>
<td>( x ) fbby ( y )</td>
<td>( \text{(before } y\text{) if the current time } \geq 0 \text{ else } x )</td>
</tr>
<tr>
<td>( x ) hpby ( y )</td>
<td>( \text{(right } x\text{) if the current horizontal coordinate } \leq 0 \text{ else } y )</td>
</tr>
<tr>
<td>( x ) vpby ( y )</td>
<td>( \text{(up } x\text{) if the current vertical coordinate } \leq 0 \text{ else } y )</td>
</tr>
<tr>
<td>( x ) pbby ( y )</td>
<td>( \text{(next } x\text{) if the current time } &lt; 0 \text{ else } y )</td>
</tr>
<tr>
<td>side ( x )</td>
<td>( x ) at the spatial point with horizontal coordinate 0 and the current vertical coordinate</td>
</tr>
<tr>
<td>edge ( x )</td>
<td>( x ) at the spatial point with vertical coordinate 0 and the current horizontal coordinate</td>
</tr>
<tr>
<td>first ( x )</td>
<td>( x ) at time 0</td>
</tr>
</tbody>
</table>

Table 1 The meanings of intensional operators

2.2. An Intensional 3-D Spreadsheet

As an application of Plane Lucid, an intensional 3-D spreadsheet has been designed in which Plane Lucid is the definition language of the spreadsheet[Du86]. In the spreadsheet, all definitions of cells (or formulas in cells) as well as of variables and user-defined functions, can be thought of as constituting a Plane Lucid-like program.
The spreadsheet, consisting of infinitely many rows and columns of cells, is considered as an entity which is called the spreadsheet variable. The spreadsheet variable varies in a two-dimensional space, called the spreadsheet plane, as well as in time. The value of a cell is the value of the spreadsheet variable at a particular spatial point at a given time. The distinguished identifier $S$ denotes the spreadsheet variable.

Apart from ordinary operators, intensional operators are used to refer to other cells intentionally in the definitions of cells. Basically these operators are $\text{left}$, $\text{right}$, $\text{up}$ and $\text{down}$. If a cell in the spreadsheet is defined as $\text{left E}$ or $\text{right E}$, where $E$ is any expression, the result of evaluating the cell is the value $E$ would have in the cell immediately to the left or the right, respectively. Similarly, if a cell is defined as $\text{up E}$ or $\text{down E}$, the value of the cell is the value of $E$ in the cell immediately above or below.

In the spreadsheet, a global variable is similar to the spreadsheet variable varying in the spreadsheet plane as well as in time. The main difference between a global variable and the spreadsheet variable is that a global variable is defined globally with a single expression, while the spreadsheet variable is defined pointwise by defining its cells. For instance, if a global variable, say $g$, is defined as $g = 5$, then $g$ is a variable with the value 5 at every point of the plane at any time.

Furthermore, during programming the spreadsheet, users can also define various functions as in high level language programs. All operators used in the definitions of variables can also be used in the definition of user-defined functions. User-defined functions can be called in the definitions of cells of the spreadsheet variable and in the definitions of variables.

The temporal operations of the spreadsheet mark a significant change from conventional 2-D spreadsheets. In this case the spreadsheet becomes three-dimensional. The addition of the temporal dimension is especially useful when we consider an object varying not only in space but also in time. In principle, the spreadsheet represented by the spreadsheet variable consists of an infinite number of planes which themselves are infinite as well. In other words, a plane is a matrix including all values of the spreadsheet variable at all spatial points and a particular temporal point. At each time only one of many planes of the spreadsheet can be the "current" one, a part of which can be shown on the screen, depending on the "current" time decided by the user. Therefore, on one hand, the points on a plane of the spreadsheet may have different values depending on their spatial positions; on the other hand, the spreadsheet may have different planes depending on the times they are associated with. One analogy is the world’s weather, which varies from place to place and changes every day.

Now, by an example, we show the features of the spreadsheet. We use the intensional spreadsheet to play the game of life. The game of life may briefly be described as follows. Each gridpoint may contain an organism. Every gridpoint is adjacent to eight other gridpoints. We use $\text{occ}(k)$ to represent the number of the adjacent gridpoints which are occupied by the organisms. By using two simple rules, we can obtain
a new organism configuration from the previous one:

1. If \(2 \leq \text{occ}(k) \leq 3\), then the organism in the point \(k\) can survive, otherwise it will die.
2. If \(\text{occ}(k) = 3\), then a new organism will be born at point \(k\) which is currently empty.

Programming the intensional spreadsheet to solve this problem is very simple, because many features of the spreadsheet are well suited to the problem. Table 2 shows a spreadsheet program for the game of life played in a 5x5 array and initially (at time 0) has seven organisms.

\[
\begin{align*}
t(x) &= \text{if } x \text{ eq } "*" \text{ then } 1 \text{ else } 0 \text{ fi;} \\
occ &= t(\text{left } S) + t(\text{right } S) + t(\text{up } S) + t(\text{down } S) + \\
& \quad t(\text{up right } S) + t(\text{up left } S) + t(\text{down right } S) + t(\text{down left } S); \\
survive &= \text{if } \text{occ eq } 2 \text{ or } \text{occ eq } 3 \text{ then } "*" \text{ else } "0" \text{ fi;} \\
birth &= \text{if } \text{occ eq } 3 \text{ then } "*" \text{ else } "1" \text{ fi;} \\
\text{life} &= \text{if } S \text{ eq } "*" \text{ then } \text{survive else } \text{birth fi};
\end{align*}
\]

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
<tr>
<td>D2</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
<tr>
<td>D3</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
<tr>
<td>D4</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
<tr>
<td>D5</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
<tr>
<td>D6</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
<td>fby life</td>
<td>01</td>
</tr>
</tbody>
</table>

**Table 2 A spreadsheet program for the game of life**

In the above spreadsheet program, "*" in a cell means that the cell contains an alive organism. The spatial relations among the defined cells are denoted by the global variable \(\text{occ}\). Every time the \(\text{occ}\) is evaluated at a spatial point (cell) and a moment in time, it has the value which is the number of the alive organisms among the eight adjacent spatial points of the given one at given time. The temporal relations among the defined cells are denoted by the definition, initial-value fby life, for each cell. When a cell is evaluated, the variable \(\text{life}\), hence \(\text{birth or survive}\) and \(\text{occ}\), be evaluated one moment before the current time, so that the action of "survive" or "birth" can be done according to the previous situations of the neighbors of the cell. It is easy to see that, as the time increases, the organism configurations represented by the spreadsheet exactly follows from the introduced rules.

3. A Systematic Approach to Systolic Array Designs

3.1. Expressing Systolic Algorithms in Lucid

Systolic algorithms are a special kind of pipelined algorithms. A pipelined algorithm has the property that, in the algorithm, an original problem is decomposed into subcomputations. Each subcomputation is assigned to a dedicated processor, and all the processors are connected in some way to form a network. Usually input of the algorithm is a sequence of data items. A processor accepts input either from the input of the algorithm or from other connected processors. Similarly, output from a processor is either as input to other connected processors or as the output of the algorithm. With data flowing through the network, the
input of the algorithm and the intermediate results by processors visit all appropriate processors to complete the computation.

A systolic algorithm has other special properties besides being pipelined.

(1) The communication of processors is local: each processor only exchange data with its neighbor processors which are connected to it; only the processors in the border of the network deal with input or output.

(2) The structure of the network is regular; each processor is connected to others in a very similar manner.

(3) There are only a few types of processors in the network, each type has a relatively simple computing function.

(4) A computation in a processor can be performed in a time unit, and the communication between two processors also spends one or more time units. In other words, the delay of the output of a processor as the input for its neighbor(s) is at least one time unit.

(5) The computation and communication in the network are synchronous. All processors start and finish computation at the same time and all processors send out their results at the same time.

An intensional language such as Lucid is very suitable to express systolic algorithms. Generally speaking, a Lucid program has a number of sequences of data items as its inputs and a sequence of data items as its output. A data item of the output sequence is computed through the operation network defined by the program, from the element(s) of the input sequence(s). If we consider one or more operations in the program are performed by a processor, in this sense, the algorithm expressed by the program is pipelined.

To express a systolic algorithm by a Lucid program, we can think of a variable in the program as a processor, the expression defining the variable as the computation performed by the processor. In order to preserve the locality and regularity of the algorithm, the definition of a variable only involves the variables representing the connected processors, and every variable refers to other variables in a regular way. The definitions of variables in the program are simple functions which may be used to define many other variables. The most important feature of Lucid for expressing systolic algorithms is the control of the timing. Using temporal operators of Lucid, the timing relations of processors can be described exactly and the synchronization of computation and communication can be simulated correctly. We may specify that the evaluation of a variable (or the operation of a processor) is performed at a moment in time, but the resultant value of the variable (or the output of the processor) is available for others demanding it at one moment later.

In the following, we show how a systolic algorithm can be expressed by a Lucid program. Consider the problem of multiplying a matrix $A = (a_{ij})$ with a vector $X = (x_1, ..., x_n)^T$. The elements in the product $Y = (y_1, ..., y_n)^T$ can be computed by the following recurrence relations:

\begin{align*}
y_1^{(1)} &= 0, \\
y_1^{(k+1)} &= y_1^{(k)} + a_{1k}x_k, \\
y_i^{(k+1)} &= y_i^{(k)} + a_{ik}x_k, \\
y_i &= y_i^{(n+1)}.
\end{align*}

Suppose $A$ is an $n \times n$ band matrix with band width $w$, where the matrix of bandwidth $w$ may have $w$
diagonals that are not all zeros and the entries outside the diagonal band are all zeros. Then the above recurrences can be evaluated by pipelining $x_i$'s and $y_i$'s through $w$ linearly connected processors. The general scheme of the systolic algorithm to solve the problem can be viewed as follows: the $y_i$'s which are initially zero, the $x_i$ and $a_{ij}$ move into the network in different directions synchronously, such that each $y_i$ is able to accumulate all its non-zero terms, namely, $a_{11} x_1, a_{12} x_2, \ldots, a_{in} x_n$, before it leaves the network. Consider the case that $w = 4$. We need four linear connected processors to implement the algorithm. Figure 1 shows the structure of the systolic array[MeCo80].

$$
\begin{bmatrix}
    a_{11} & a_{12} & & 0 \\
    a_{21} & a_{22} & a_{23} & \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{42} & a_{43} & a_{44} & a_{45} \\
    0 & & & & \\
\end{bmatrix}
\begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    x_4 \\
\end{bmatrix}
= 
\begin{bmatrix}
    y_1 \\
    y_2 \\
    y_3 \\
    y_4 \\
\end{bmatrix}
$$

Figure 1(a) Multiplication of a vector by a band matrix with $w = 4$

Figure 1(b) The linearly connected systolic array for the problem shown in Figure 1(a)

Below is a Lucid program expressing the algorithm.

Example 1:

```
hd(processor1) whenever P
where
  processor1 = [0, 0, first X %] fby [% mul_add(next X, A1, hd(processor2)), next x %]);
  processor2 = [0, 0, 0] fby [% mul_add(hd2(processor1), A2, hd(processor3)), hd2(processor1) %]);
  processor3 = [first Y, 0 %] fby [% mul_add(hd2(processor2), A3, hd(processor4)), hd2(processor2) %]);
```
processor4 = [% 0, 0 %] fby [% mul_add(hd2(processor2), A3, next Y), hd2(processor3) %];

mul_add(a, b, c) = c + a * b;
Y = 0;
P = false fby false fby false fby false fby true fby (next next next P);
hd2(x) = hd(tl(x));
end

The program needs five input sequences which are values of diagonals of matrix A and vector X:

A1 = (a11, 0, a21, 0, a31, 0, ...
A2 = (a12, 0, a22, 0, a32, 0, ...
A3 = (a13, 0, a23, 0, a33, 0, ...
A4 = (0, a31, 0, a41, 0, a51, 0, ...
X = (x1, 0, x2, 0, x3, 0, ...

The input sequence X moves from processor1 through processor4 by one time unit a step; The input sequence Ai's move to processori; and each element y_i of Y = A \times X is accumulated in each processor from processor4 backwards to processor1. The inserted zero's in the input sequences guarantee that, with the movements of Ai's, X and Y, a_{i2}, x_2 and y_1 can arrive at a processor at the same time and the accumulated y_i can be computed correctly. After w time units, y_1 is completely computed and output in processor1 and then at every other moment in time y_i's (i=2,3,...) are also computed and outputed.

3.2. Specifying Systolic Architectures in Plane Lucid

Expressing a systolic algorithm in Lucid such as the above matrix-vector multiplication algorithm is relatively simple, but in the program the systolic structure corresponding to the algorithm is not clearly described. The issues about specifying the systolic architecture that implements the systolic algorithm should also be concerned, e.g. the global structure of the systolic array, the layout of the processors inside the array, the physical connections of the processors in architectural sense, the connections between I/O and the border processors, and so on. For example, by looking at Example 1, we cannot figure out what the architecture of the systolic array would be. Using spatial operators in Plane Lucid, we may rewrite the above algorithm, which will not only express the systolic algorithm for the matrix-vector multiplication, but also describe the corresponding systolic structure at an abstract level.

Example 2:

hd(SysArr wherever P)
where
SysArr =
case [% x, y %] of
[0,1]: A1
[0,2]: A2
[0,3]: A3
[0,4]: A4
[1,0]: [% hd(right SysArr), X%]
[1,1]: [1,2]: [1,3]: [1,4]:
[0,0] fby [% mul_add(hd2(left SysArr), up SysArr, hd(right SysArr)), hd2(left SysArr) %]
[1,5]: [% Y, hd2(left SysArr) %];
end;
mul_add(a, b, c) = c + a*b;
Y = 0;
P = if [%x, y%] eq [1, 0] then true else false fi;
x = 0 hsby x+1;
y = 0 vsby y+1;
hdz(x) = hd(t(x));
end

In the above Plane Lucid program, we may think the systolic structure is built on a plane consisting of infinitely many gridpoints or cells located by their coordinates. All defined cells represent processors which constitute the desired systolic array. The cells defined by input variables and/or producing the program output represent I/O processors, in the program the processors at the locations (0,1),(0,2),(0,3),(0,4) are input processors and the processor at (1,0) is an input and output processor. The cells whose definitions involving values of other cells represent computing processors, such as the processors at the locations (1,1),(1,2),(1,3) and (1,4). All I/O processors are on the border of the array.

The operations performed by a computing processor is the function called in the definition of the processor. The communication between two processors is denoted through the use of spatial operators left, right, up and down in the arguments of the calls of their defining functions. For example, the processor at location (1,1) obtains its input values from the output values of the processors at its left, right and above it. The use of "local" spatial operators like left and right guarantees the locality and regularity of the array. The program also specifies the directions of movements of input sequences and intermediate results: the input sequence for the vector X enters the array from the left and moves to the right; the input sequence Ai for the band matrix A enters the array from the top and moves down; the output vector Y enters the array from the right with initial value 0 and moves to the left. The value of each element of Y is outputted sequentially from the leftmost I/O processor(1,0) at the rate of every other time step after the delay of w time units.

3.3. The Intensional Spreadsheet As a Design Environment

Plane Lucid may be extended so that it can specify systolic arrays formally, but for the systolic array designers these forms of the specifications are obviously not intuitive. The designer must have the geometric layout and connections of processors of the array in mind. Since most of systolic arrays are designed in a grid-like way, a specialized and extended version of the intensional spreadsheet may create an environment for the designers.

Now we show this potential of the intensional spreadsheet by the same example as above. In the following systolic array design for the matrix-vector multiplication algorithm, we consider a more detailed model of the array. Each processor has three registers, Ra, Rx and Ry, which will hold entries in A, X and Y, respectively. Initially, all registers contain zeros. Each step of the algorithm consists of the following
operations:

1. Shift.
   Ra gets a new element in the band of matrix A.
   Rx gets the contents of register Rx from the left neighbor.
   (The Rx in processor 1 gets a new component of x.)
   Ry gets the contents of register Ry from the right neighbor.
   (Processor 1 outputs its Ry contents and the Ry in processor w gets 0.)

2. Multiply and Add.
   \( Ry \rightarrow Ry + Ra \times Rx. \)

During the computation, for odd-numbered time steps, only odd-numbered processors are activated, and for even-numbered time steps, only even-numbered processors are activated.

To design the systolic array according to the above description, we can easily write a program in the environment of the intensional spreadsheet, specifying the structure of the array. In the spreadsheet, a computing processor is represented by three cells, which denote Ra, Rx and Ry respectively; an input processor is represented by a cell containing an input sequence. an input and output processor is represented by two cells containing an input sequence and an output sequence, respectively. Figure 2 shows the spreadsheet program when \( w = 4 \), where a computing processor is represented by three vertically adjacent cells.

\[
egin{array}{c}
Ra = 0 \text{ fby action(up S)};
Ry = 0 \text{ fby action(mul_add(left S, up S, right S))};
Rx = 0 \text{ fby action(left S)};
Output = right S;
Y = 0;
\text{mul_add}(a,b,c) = c + a \times b;
\text{action}(x) = \text{if OddProc then if (next OddTime) then x else S fi}
\text{else if (next OddTime) then S else x fi}
\end{array}
\]

where
- OddProc = false if by not OddProc;
- OddTime = false if by not OddTime;

end;

<table>
<thead>
<tr>
<th>S</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ra</td>
<td>Ra</td>
<td>Ra</td>
<td>Ra</td>
<td>Ra</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Output</td>
<td>Ry</td>
<td>Ry</td>
<td>Ry</td>
<td>Ry</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td>Rx</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 A matrix-vector multiplication systolic array

In the above program, the actual arguments in the calls of function action in the definitions of Ra, Ry and Rx for a processor describe the operations of the processor when it is activated. The function action(x) describes the behavior of a processor. The boolean variable OddProc is alternatively true or false along the horizontal dimension; and the boolean variable OddTime is true at every odd and false at every even
time step. Thus the function defines that processors with odd (even) numbers are activated at odd (even) time steps and are not activated at other time steps. In the function, we use nextOddTime predicate in the second if-then-else condition because, when the function is called, the operator fy causes it to be evaluated at one time step before the current time.

Using the spatial operators, Plane Lucid and the intensional spreadsheet can describe various structures of systolic arrays. The following are some examples. Let Fx(y) be the function defining the operation that register Rx performs in a processor. The processor network of a two-dimensional square array can be specified in the spreadsheet shown in Figure 3, in which each processor consists of two registers, Ru and Rl, and data flows from top down and left to right.

![Figure 3(a) The structure of a two-dimensional square systolic array](image)

Ru = 0 fby Fu(up up S);
Rl = 0 fby Fl(left S);

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ru</td>
<td>Ru</td>
</tr>
<tr>
<td>2</td>
<td>Rl</td>
<td>Rl</td>
</tr>
<tr>
<td>3</td>
<td>Ru</td>
<td>Ru</td>
</tr>
<tr>
<td>4</td>
<td>Rl</td>
<td>Rl</td>
</tr>
</tbody>
</table>

![Figure 3(b) The specification of (a) on the intensional spreadsheet](image)

The processor network of a two-dimensional hexagonal array can be specified in the spreadsheet as in Figure 4, in which each processor consists of three registers, Ru, Rl and Rul, and data flows from top down, left to right and along the main diagonal.
Ru = 0 fby Fu (up up S);
RI = 0 fby Fl (left left S);
Rul = 0 fby Fu (left left up up S);

<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rul</td>
<td>Ru</td>
<td>Rul</td>
<td>Ru</td>
</tr>
<tr>
<td>2</td>
<td>RI</td>
<td>RI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Rul</td>
<td>Ru</td>
<td>Rul</td>
<td>Ru</td>
</tr>
<tr>
<td>4</td>
<td>RI</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4(b) The specification of (a) on the intensional spreadsheet

Other systolic array structures can also be specified, such as a binary tree. The following is a specification in the spreadsheet for a complete binary tree with seven nodes. In the tree, data flows from top down, and each processor (node) has one register called RI, Rr, Ru or Rd.

Figure 5(a) The structure of a systolic tree with seven nodes
RI = 0 fby F(right S);
Rr = 0 fby F(left S);
Ru = 0 fby F(down S);
Rd = 0 fby F(up S);

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ru</td>
<td></td>
<td>Ru</td>
</tr>
<tr>
<td>2</td>
<td>RI</td>
<td>input</td>
<td>Rr</td>
</tr>
<tr>
<td>3</td>
<td>Rd</td>
<td></td>
<td>Rd</td>
</tr>
</tbody>
</table>

Figure 5(b) The specification of (a) on the intensional spreadsheet

From the binary tree example, one can find the method we have used is not general. For some non-grid-like systolic structures, we need appropriate algorithms to embed the structures into the grid. An approach to apply existing embedding algorithms to the Plane Lucid specifications for systolic arrays is being investigated by the author.

Furthermore, we can also specify reconfigurable systolic arrays in Plane Lucid. In a reconfigurable array, the processors are not directly connected to each other, rather connected through one or more switches in a switching network with a regular structure. In the course of computation, the connections among the processors change as the data paths going through the switches vary. The function of a switch can be described simply by using intensional operators. Consider a four-way switch that works in the following way: at every even time step it allows data passing from the above processor to the below one and from the left one to the right one, while at every odd time step it allows data passing from the left to the below and from the above and the right. The connections of the four processors through the switch can be specified as in the following spreadsheet definition. (Figure 6.)
Figure 6(a) The connections of processors in the reconfigurable systolic array at even time steps (circles represent switches and squares represent processors)

Figure 6(b) The connections of processors in the reconfigurable systolic array at odd time steps

Processor = 0 fby F(hd(up S), hd2(left S));
Switch = [% up S, left S%] fby [% left S, up S %] fby Switch
hd2(x) = hd(₁(x));

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Switch</td>
<td>Processor</td>
<td>Switch</td>
</tr>
<tr>
<td>1</td>
<td>Switch</td>
<td>Processor</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Processor</td>
<td>Switch</td>
<td>Processor</td>
</tr>
<tr>
<td>3</td>
<td>Switch</td>
<td>Processor</td>
<td>Switch</td>
</tr>
</tbody>
</table>

Figure 6(c) The specification of the reconfigurable systolic array on the intensional spreadsheet
3.4. Simulating Systolic Computation in Eduction

Up to now, we have assumed that the processors in a systolic array are synchronous as described before. However, it is possible to view the processors as being asynchronous, each computing its output values when all its inputs are available as in a data flow model. In other words, at each time step the input values for a processor, which were produced by other processors at the previous time step, are always available, we can consider systolic computation as data flow computation or synchronous data flow. In fact, in the above sense, systolic computation is a kind of data-driven computation. But when we consider to simulate systolic computation, the data-driven model may not be appropriate because the simulation involving only a part of the systolic array instead of the whole structure often needs better control facilities. Thus demand-driven model may be thought as the best model for simulating systolic computation, since under this model only the processors whose outputs are demanded compute their outputs like in data-driven model. When a part of the array is simulated, the simulation can be controlled by sending demands only to the involved processors.

Currently the intensional spreadsheet is implemented by the eduction technique or tagged demand-driven model[AFH85]. In the implementation[Du86], the definitions of cells or variables are stored in an associated memory called definition warehouse with the labels containing their names and their spatial locations. The cells are evaluated by the demands from other cells or variables, and the initial evaluation is demanded by the user. The values of cells or variables are stored in another associated memory called value warehouse with the label containing their owners, the spatial locations and the times at which they are evaluated. It is expected that eduction is suitable for simulating systolic computation. Using eduction, when a systolic array is specified in a specification language like Plane Lucid or in the intensional spreadsheet, the designer does not need complete the whole array before testing it. Instead the design can be tested partially by simulating only the completed part. In addition, using an interactive design environment such as the intensional spreadsheet, the designer can modify the design at any time and test it immediately.

4. Concluding Remarks

We have showed, through the examples of Plane Lucid and the intensional spreadsheet, the potential of intensional programming as a systematic approach to systolic array designs. This potential also shows that intensional programming is suitable for a rather wide range of applications, especially for those involving objects varying in various dimensions. Here we do not claim that the current Plane Lucid and the intensional spreadsheet is mature enough for the systolic array designs; they certainly need to be developed further. A specialized intensional language for designing systolic arrays systematically may include these important features: formally specifying systolic architectures from systolic algorithms, verifying and
optimizing the specified systolic arrays by analyzing intensional semantics, and simulating the arrays by
eduction. A direct transformation between the formal specification of the systolic array and a design
environment in the sense of intensional programming may make the automatic design of systolic arrays
possible, that is, once the designer specifies the desired systolic array in the design environment, the rest of
the design procedures: verification, optimization, simulation and even prototypes can be implemented
automatically.

References

[AFH85] E.A. Ashcroft, A.A. Faustini and B. Huey, "Eduction A Model of Parallel Computation and
the Programming Language Lucid," Proc. 4th Annual International Conference PCCC, 1985,
pp. 9-15.

Victoria, 1986.

[FaWa85] A.A. Faustini and W.W. Wadge, "Intensional Programming" Dept. of Computer Science,
Univ. of Victoria, DCS-55-IR, June 1986.

[FoWa87] J. Fortes and B.W. Wah, "Systolic Array From Concept to Implementation," IEEE Computer,


37-46.

[MeCo80] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-wesley Publishing Com-

[WaAs85] W.W. Wadge and E.A. Ashcroft, Lucid, the Dataflow Programming Language, Academic