The Use of aFP to Design Regular Array Algorithms*

Yen-Chun Lin

Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan 10764, R.O.C.

Ferng-Ching Lin

Dept. of Computer Science and
Information Engineering
National Taiwan University
Taipei, Taiwan 10764, R.O.C.

ABSTRACT

We design a language called aFP (array FP), which is a dialect of FP containing many desirable parallel constructs. It extends FP with additional primitive functions and higher-order functions for the purpose of designing regular arrays. In aFP, one can think and describe algorithms in a parallel way to a large degree. We give the mappings of aFP functions together with input patterns to regular array structures. The mapped basic array algorithms can then be used as building blocks to construct more complex ones. In addition, aFP can be adapted to be a programming language in programmable systolic systems.

1. Introduction

Many have explored VLSI hardware design based on functional programming language FP, which was advocated by Backus [2, 3]. Schlag [25] was able to extract information from an FP program to obtain a visual sketch of a combinational circuit, showing its behavioral and structural organization. Meshkinpour and Ercegovic [22] illustrated the use of the FP concept to support the specification of both combinational and sequential systems, the mapping of specification into a gate level implementation, and the simulation of its functional behavior. Patel et al. [23] described a method based on FP for the specification, evaluation, and synthesis of gate level hardware algorithms. Sheeran [26, 27] used a variant of FP as a VLSI design language to describe the semantics (behavior) and layout (floor plan) of a circuit and showed how FP could be adopted to design regular array architectures. But the final intent was only to use FP as an aid to existing design techniques. Jones and Luk [11] suggested to apply mathematical methods to circuit design and outlined a theory of orthogonally connected circuits based on FP.

The dramatic development of very large scale integration (VLSI) technology has made it possible to implement algorithms directly in hardware and hence promoted great interest in designing algorithmically specialized processing components. Following Kung's systolic concept [13, 14], many computing arrays have been proposed to handle various compute-bound problems. Systolic array processors generally consist of a regular array of simple and nearly identical processing elements (PEs) in which data are communicated locally and operated on rhythmically. The simplicity, regularity, and locality of the systolic arrays render them suitable for VLSI implementation. High performance is achieved by the concurrent use of a large number of PEs in the array. Many methods have also been proposed for synthesizing systolic algorithms; a survey of these contributions can be found in [9]. Usually, they start in the conventional, sequential way of thinking, and come to the result of systolic array design through tedious transformations.

The purpose of this paper is to introduce a new method of designing regular arrays (systolic or semisystolic [17], in particular) through FP programs. First, we design a language called aFP, which is a dialect of FP containing many desirable parallel constructs. In aFP, one can think and describe algorithms in a parallel way to a large degree. Then, the aFP programs are transformed into array algorithms rather systematically by considering the programs and their (input) data structures. So, aFP can be used as a programming language for built programmable systolic systems [1, 20, 28], in addition to being a good tool for synthesizing specialized regular arrays.

In this paper, we focus on mapping aFP programs to 1-D (linearly-connected) regular arrays. Although simple, 1-D arrays are very powerful and more implementable than 2-D arrays on VLSI chips. We shall show how 1-D array algorithms for problems like matrix-vector multiplication, key enumeration, and finite impulse response filtering can easily be derived.

2. Brief introduction to aFP

The language aFP (for array FP) extends FP proposed by Backus with additional primitive functions and higher-order functions (HOFs) for the purpose of designing regular arrays. We will adopt the postfix notation for FP, which provides good programming style and helps make large programs readable [8, 24].

2.1. Object level programming

A program in aFP is a function mapping objects to objects. Objects are atoms (symbols, numbers, or character strings) or lists of objects. Primitive functions, the basic functions of aFP, consist of:

```
arithmetic functions
  <2,3>: + = 5      <2,3>: * = 6
  <8,4>: - = 4      <8,4>: + = 2
predicates
  lt (less than)    <2,3>: lt = T
  gt (greater than) <2,3>: gt = F
  eq (equal)        <2,3>: eq = F
  ne (not equal)    <2,3>: ne = T
Boolean functions
  <T, F>: and = F    <T, F>: or = T
identity function
  <2,<3,4>>: id = <2,<3,4>>
```

and structural functions:

```
length
  <1, 2, 3>: length = 3
selector functions
  <2,<4>>: 2 = <4>    <2,3>: 1 = 2
tail
  <1, 2, 3, 4>: tail = <2, 3, 4>
distribute from left; distribute from right
  <2,<3,4>>: dist_l = <<2,3>,<2,4>>
  <<x,y>,a>: dist_r = <<x,a>,<y,a>>
```

*This research was partially supported by National Science Council of R.O.C. under contract NSC-76-0408-E-002-05.

```

transpose
  <<1,2,3>,<x,y,z>>: trans
    = <<1,x>,<2,y>,<3,z>>
take from left
  <<a,b,c,d>,>3>: take_1 = <a,b,c>

```

An HOF in aFP maps functions or objects into functions. In this section we use $p, g, f, f_1, \dots, f_n$ to denote functions;

specifically, p is a predicate or Boolean function.

```

reduce(/)
  <1,...,n>: / f
    = <<1,...,n-1>: / f,n>: f
  <1>: / f = 1
left-insert (/L)
  <x,<1,...,n>>: /L f = <x,1,...,n>: / f
composition (*)
  x: (f * g) = (x: f): g
construction ([ ])
  x: [f1,...,fn] = <x:f1,...,x:fn>
apply-to-all (α)
  <1,...,n>: α f = <1:f,...,n:f>
constant (!)
  x: !y = y
condition
  x: (p --> f; g) = x: f if (x:p) = T
  x: g otherwise
power_scan (/P)
  <x,n>: \P f = <x:f0, x:f1,...,x:fn-1>
  f0 = id, fm = f * fm-1 for m > 0

```

Of all functions the precedence of "composition" is the lowest; however, one can always use parentheses to change the execution order or make it more explicit. Examples are α (trans $\cdot \alpha$ f) = α trans $\cdot \alpha$ (α f) and α (α f) = α α f.

Programmers can use the above functions to define new ones, called user-defined functions (UDFs). For example, we define here a new function Add1 and show how it applies to a proper argument:

```

Def Add1 == [id,!1] * +
5: Add1 = 5: ([id,!1] * +) = <5,1>: + = 6

```

Here is a more complex function Matrix_Vector:

```

Def Matrix_Vector == dist_r * α trans * α (α *) * α (/+)
  <<<1,2,3>,<2,4,6>,<3,6,9>>,<1,1,2>>: Matrix_Vector
  = <<<1,2,3>,<1,1,2>>,<<2,4,6>,<1,1,2>>,<<3,6,9>,<1,1,2>>>:
    α trans * α (α *) * α (/+)
  = <<<1,1>,<2,1>,<3,2>>,<<2,1>,<4,1>,<6,2>>,<<3,1>,<6,1>,<9,2>>>: α (α *) * α (/+)
  = <<1,2,6>,<2,4,12>,<3,6,18>>: α (/+)
  = <9,18,27>

```

In another form of definition (extended definition, xdef), programmers can name parameters in UDFs and refer to them in the function body [3]. As an example, we define function Greater as follows:

```

xdef [k,m] Greater == [k,m] * (gt --> !1; !0)

```

Here the function Greater takes two arguments k and m ; if k is greater than m then the result is 1, else 0. To improve the readability, we may use abbreviation forms for function definitions involving "condition", such as:

```

xdef [k,m] Greater == k gt m --> !1; !0

```

The predicate "gt" in the above definition acts as if it were an infix operator.

Based on the above-mentioned built-in primitive functions and HOFs, we can define many UDFs. Many thus definable functions either make aFP programs readable or have close equivalents in array structures. The close equivalents make the mappings from aFP programs to regular arrays very straightforward. They are presented here and regarded as built-in functions.

```

skew
  xdef [x,k] skew ==
    [[x,[x * length,k] * - * Add1] * \P tail,
     k] * dist_r * α take_1
  <<x1,x2,...,xk,...,xn>,>k>: skew
  = <<x1,...,xk>,<x2,...,xk+1>,...,<xn-k+1,...,xn>>
windows
  xdef [w,x] windows ==
    [w,[x,w * length] * skew] * dist_l * α trans
  <<w1,...,wk>,<x1,...,xk,...,xn>>: windows
  = <<<w1,x1>,...,<wk,xk>,<<w1,x2>,...,<wk,xk+1>>,<<w1,xn-k+1>,...,<wk,xn>>>
pairing
  Def pairing == dist_r * α dist_l

```

2.2. Structure level programming

For proper inputs, an aFP function changes not only the data objects, but also the data structures in most cases. (Atoms and lists of atoms are structures, so are lists of lists.) However, an aFP function maps only objects of a certain structure to objects of a certain, perhaps different, structure. Therefore, it is helpful to program aFP at an abstract level, the structure level. First, some notations must be introduced.

Assume that m, n , and k are integers. The letter M (resp. N, K) is used to represent a list of m (resp. n, k) atoms; i.e., M can represent the structure (abstraction) of the (real) object $\langle a_1, a_2, \dots, a_m \rangle$. $M \times N$ denotes a list consisting of m

components, each of them being an N ; i.e., $M \times N$ is the structure of $\langle \langle a_{11}, \dots, a_{1n} \rangle, \langle a_{21}, \dots, a_{2n} \rangle, \dots, \langle a_{m1}, \dots, a_{mn} \rangle \rangle$.

Similarly, $M \times K \times N$ denotes a list consisting of m components, each of them being a $K \times N$. Lists like $M \times N$ and $M \times K \times N$ can be respectively abbreviated as MN and MKN , if they cause no confusion. A list of structure M , $M \times N$, or $M \times K \times N$ is called regularly structured list (RSL). Infix arithmetic operators, such as $+$, $-$, and $*$, as well as numbers, can also be used for representing RSLs; e.g., $(M+N-1)$ denotes a list of $m+n-1$ atoms.

Assume X, Y, Z are atoms or RSLs. We use (X, Y) to denote a list consisting of two components, X and Y ; in particular, (a, N) represents a list whose first and second components are an atom and a list of n atoms, respectively.

Let $f, f_1, f_2, \dots, f_n$ represent functions. The following are basic structure level programs, that are most useful:

```

(MX, X): dist_r = M2X
MNX: trans = NMX
MX: α f = MY if X: f = Y
(MX, MY): trans = M(X, Y)
X: f * g = Y if X: f = Z and Z: g = Y
(X, Y): 1 = X
(X, Y): 2 = Y
X: id = X
2: * = a
MX: / f = X if (X, X): f = X
(X, NX): /L f = X if (X, X): f = X

```

(M, N): pairing = MN2
 (K, N): windows = (N-K+1)K2
 $X: [f_1, f_2, \dots, f_n] = (X: f_1, X: f_2, \dots, X: f_n)$

Note that a basic structure level program listed above may imply more than one such programs. For example, $MX: / f = X$ implies $M: / f = a$ and $MN: / f = N$. The application of Matrix_Vector to a proper input presented in Section 2.1 can be abstracted, at the structure level, as follows:

(3×3, 3): Matrix_Vector
 = 3×2×3: $\alpha \text{ trans} * \alpha (\alpha *) * \alpha (/ +)$
 = 3×3×2: $\alpha (\alpha *) * \alpha (/ +)$
 = 3×3: $\alpha (/ +)$
 = 3

3. Mapping aFP programs to array algorithms

3.1. Sequenced data and structures

Regular arrays operate on regularly synchronized data streams, which usually belong to one, or combinations, of the three patterns: serial, parallel, and skewed. We can have data structures subscripted with S, P, or K, called sequenced structures, to denote serial, parallel, and skewed data streams, respectively. A sequencing level aFP program is an aFP function mapping a sequenced structure to a sequenced structure. Not all structure level programs have a corresponding sequencing level program; however, some structure level programs have more than one corresponding sequencing level programs. They are elaborated in the following.

As previously mentioned, $\langle a_1, a_2, \dots, a_m \rangle$ can be abstracted as M; then the sequenced structure M_S denotes that these m atoms move serially, and can be depicted as

$a_m \quad \dots \quad a_2 \quad a_1 \rightarrow$

M_P denotes that $a_1, a_2, \dots$, and a_m move in parallel, and can be depicted as

$a_1 \quad a_2 \quad \dots \quad a_m$
 $\downarrow \quad \downarrow \quad \dots \quad \downarrow$

M_K denotes that the m atoms move in a time- and space-skewed fashion; i.e., they move in their respective lines (like parallel) and a_i is one time step ahead of a_{i+1} for $1 \leq i < m$ (like serial). It can be depicted as

$a_1 \quad a_2 \quad \dots \quad a_m$
 $\downarrow \quad \downarrow \quad \dots \quad \downarrow$

Let Q1, Q2, Q3 stand for any of P, S, and K. Then we extend the sequenced structures to complex structures. $M_{Q1}N_{Q2}$ denotes

that the m lists move in the Q1 fashion, while the n atoms in each of the m lists move in the Q2 fashion. For example, $M_S N_P$ can be depicted as

$a_{m1} \quad a_{m2} \quad \dots \quad a_{mn}$
 $\cdot \quad \cdot \quad \dots \quad \cdot$
 $\cdot \quad \cdot \quad \dots \quad \cdot$
 $a_{21} \quad a_{22} \quad \dots \quad a_{2n}$
 $a_{11} \quad a_{12} \quad \dots \quad a_{1n}$
 $\downarrow \quad \downarrow \quad \dots \quad \downarrow$

$M_S N_K$ can be depicted as

$a_{m1} \quad a_{m2} \quad \dots \quad a_{mn}$
 $\cdot \quad \cdot \quad \dots \quad \cdot$
 $\cdot \quad \cdot \quad \dots \quad \cdot$
 $a_{21} \quad a_{22} \quad \dots \quad a_{2n}$
 $a_{11} \quad a_{12} \quad \dots \quad a_{1n}$
 $\downarrow \quad \downarrow \quad \dots \quad \downarrow$

$M_P N_P$ can be depicted as

$a_{11} \quad a_{12} \quad \dots \quad a_{1n} \quad a_{21} \quad \dots \quad a_{mn}$
 $\downarrow \quad \downarrow \quad \dots \quad \downarrow \quad \downarrow \quad \dots \quad \downarrow$

Similarly, for $M_S N_K^2 P$, its sequenced structure can be depicted like

$a_{m1} \quad b_{m1} \quad a_{m2} \quad b_{m2} \quad \dots \quad a_{mn} \quad b_{mn}$
 $\cdot \quad \cdot \quad \cdot \quad \cdot \quad \dots \quad \cdot \quad \cdot$
 $\cdot \quad \cdot \quad \cdot \quad \cdot \quad \dots \quad \cdot \quad \cdot$
 $a_{21} \quad b_{21} \quad a_{22} \quad b_{22} \quad \dots \quad a_{2n} \quad b_{2n}$
 $a_{11} \quad b_{11} \quad a_{12} \quad b_{12} \quad \dots \quad a_{1n} \quad b_{1n}$
 $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \dots \quad \downarrow \quad \downarrow$

The meaning of sequenced structures can be made one step further to include non-RSLs. $(X_{Q1}, Y_{Q2})_{Q3}$ denotes that the components of X and Y move in the Q1 and Q2 fashions, respectively. Furthermore, the coordination of X and Y is in fashion Q3. For example, $(M_S, N_S)_P$ represents two parallel data streams, as depicted by

$a_m \quad \dots \quad a_2 \quad a_1 \rightarrow$
 $b_n \quad \dots \quad b_2 \quad b_1 \rightarrow$

$(a, N_K)_P$ denotes that an atom, say c, keeps pace with the first atom of a set of skewed atoms, and can be depicted as

$c \quad b_1 \quad b_2 \quad \dots \quad b_n$
 $\downarrow \quad \downarrow \quad \downarrow \quad \dots \quad \downarrow$

The data streams of $M_S(a, N_K)_P$ can be depicted as

3.2. Basic mappings

An array algorithm consists of not only an array of PEs and the operations performed on the PEs, but also the timing of input data streams. To present the mapping to array algorithms, we use $\langle c_1, c_2, \dots, c_m \rangle$ and $\langle\langle c_{11}, \dots, c_{1n} \rangle, \langle c_{21}, \dots, c_{2n} \rangle, \dots, \langle c_{m1}, \dots, c_{mn} \rangle\rangle$ respectively to represent the objects of M and MN for most cases in the sequencing level programs, emphasizing that c_i and c_{ij} can be other than atoms.

The basic sequencing level programs are to be used as a base for match searching. Whenever a basic program is matched, its corresponding array algorithm is suggested. Fig. 1(a), (b), and (c) respectively depict the array algorithms for the following three sequencing level programs:

$$M_P: \alpha f = M_P$$

$$M_S: \alpha f = M_S$$

$$M_K: \alpha f = M_K$$

(a) Parallel apply-to-all.

(b) Serial apply-to-all.

(c) Skewed apply-to-all.

Fig. 1. Array algorithms for " αf ".

Note that, as we have mentioned, M can be a list of m lists. The application of " αf " to any of M_P , $M_P N_S$, and $M_P N_S K_S$, for example, matches the first program and hence can make use of the array algorithm depicted in Fig. 1(a).

Fig. 2 depicts the array algorithm for the following program

$$M_S N_K: \alpha (\alpha f) = M_S N_K$$

Fig. 2. Array algorithm for $M_S N_K: \alpha (\alpha f)$.

There are eight other sequenced structures for the structure MN , and therefore eight corresponding array algorithms. The reader should be able to construct the rest array algorithms.

Let " $\#$ " represent a PE that outputs its input untouched and a " D " marked on the link connecting two PEs denote a delay (clocked latch). Fig. 3 depicts the array algorithm for $M_K: / f = X_P$, while Fig. 4 depicts that for $M_S: / f = X_P$.

Fig. 3. Array algorithm for $M_K: / f$.

Fig. 4. Array algorithm for $M_S: / f$.

Note that each of the results produced by the above two algorithms is output at the m -th time step after the first input is ready, and in Fig. 4, the result is accompanied by a value 2. Next, let us consider the following four functions:

$$M_S N_K: \alpha (/ f) = M_S$$

$$M_S(a, N_K)_P: \alpha (/ L f) = M_S$$

$$(M_S, N_S)_P: \text{pairing} = M_S N_K 2_P$$

$$(K_S, N_S)_P: \text{windows} = (N-K+1)_S K_K 2_P$$

Their mapped array algorithms are depicted in Fig. 5, 6, 7, 8, respectively.

Fig. 5. Array algorithm for $M_S N_K: \alpha (/ f)$.

Fig. 6. Array algorithm for $M_S(a, N_K)_P: \alpha / L f$.

Fig. 7. Array algorithm for $(M_S, N_S)_P$: pairing.

Fig. 8. Array algorithm for $(K_S, N_S)_P$: windows.

Note that the arrays in Fig. 7 and Fig. 8 are semisystolic since the upper links between PEs are not marked by "D" and are actually for broadcast use. In Fig. 7, $m-1$ 0-valued flags are required to guarantee correct results. In Fig. 8, $n-k$ 0-valued flags are required to ensure that the array can generate results correctly. A semisystolic array can always be converted into a systolic one by retiming [18].

If the output sequenced structure of an aFP function conforms to the input sequenced structure of another aFP function, their corresponding arrays can be connected, with or without a delay on each of the connected links, to accomplish more complicated computation. When such delays are not shown, it means that the connected PEs are to be merged into a bigger one.

The following gives a basic sequencing level program that does not map to any array algorithm:

$$(M_S X_K, M_S Y_K)_P: \text{trans} = M_S(X_K, Y_K)_P$$

Its role is to transform structures, but without changing the meanings (physical sequencing) of sequenced data objects.

4. Examples of regular array design

In this section, we show, by examples, how the mappings presented in the previous section can be employed to construct useful regular array algorithms. Since our approach to array design is quite different from other approaches, it is not unusual that we have different results in appearance, if not in array structure or operations of PEs.

4.1. Matrix-vector multiplication

Suppose A is an $m \times n$ matrix and B is an $n \times 1$ column vector. They can be respectively represented as lists

$$A = \langle \langle a_{11}, a_{12}, \dots, a_{1n} \rangle, \langle a_{21}, \dots, a_{2n} \rangle, \dots, \langle a_{m1}, \dots, a_{mn} \rangle \rangle,$$

$$B = \langle b_1, \dots, b_n \rangle.$$

To compute their product AB , when input is of the form $\langle A, B \rangle$, we can use the function `Matrix_Vector` defined in Section 2:

$$\text{Def Matrix_Vector} == \text{dist_r} \cdot \alpha \text{ trans} \cdot \alpha (\alpha *) \cdot \alpha (/ +)$$

Since $\langle A, B \rangle$ is of the structure (MN, N) , we first show the application of `Matrix_Vector` to (MN, N) at the structure level in the following:

$$\begin{aligned} (MN, N): \text{dist_r} \cdot \alpha \text{ trans} \cdot \alpha (\alpha *) \cdot \alpha (/ +) \\ &= M2N: \alpha \text{ trans} \cdot \alpha (\alpha *) \cdot \alpha (/ +) \\ &= MN2: \alpha (\alpha *) \cdot \alpha (/ +) \\ &= MN: \alpha (/ +) \\ &= M \end{aligned}$$

To determine the mapping of `Matrix_Vector` with input (MN, N) to a proper array algorithm, we examine the application of the constituent functions to their respective input structures and match each of them with a basic sequencing level program. The matching may or may not follow the order the constituent functions applied; i.e., it can be either forward or backward. The only factor guiding the matching order is to avoid backtracking caused by the mismatch of data sequencing in finding suitable basic sequencing level programs.

The only basic sequencing level program that the last application of

$$MN: \alpha (/ +) = M$$

matches is

$$M_S N_K: \alpha (/ f) = M_S \quad (1)$$

This means that when " $\alpha (/ +)$ " accepts structure MN and produces structure M , the I/O sequenced structures (i.e., data sequencing) must be $M_S N_K$ and M_S , respectively. To be more specific, the

array algorithm is the same as that shown in Fig. 5, except that the operation in each PE is "+" instead of "f".

Then, for the application of " $\alpha (\alpha *)$ " to obtain $M_S N_K$, we have the following basic sequencing level program:

$$M_S N_K: \alpha \alpha f = M_S N_K \quad (2)$$

We obtain the algorithm depicted in Fig. 9 from program (2) by utilizing the mapping in Fig. 2, as well as considering the input data object of " $\alpha (\alpha *)$ ", which is the result of applying " $\text{dist_r} \cdot \alpha \text{ trans}$ " to $\langle A, B \rangle$, i.e.,

$$\begin{aligned} &\langle \langle \langle a_{11}, b_1 \rangle, \langle a_{12}, b_2 \rangle, \dots, \langle a_{1n}, b_n \rangle \rangle, \\ &\langle \langle a_{21}, b_1 \rangle, \langle a_{22}, b_2 \rangle, \dots, \langle a_{2n}, b_n \rangle \rangle, \dots, \\ &\langle \langle a_{m1}, b_1 \rangle, \langle a_{m2}, b_2 \rangle, \dots, \langle a_{mn}, b_n \rangle \rangle \rangle \end{aligned}$$

The functions " dist_r " and " $\alpha \text{ trans}$ " do not map to any array algorithms, but affect the input data objects and structures as mentioned above. Therefore, we achieve the array algorithm depicted in Fig. 10 by composing the above two array algorithms. That the two arrays can link well is guaranteed by the fact that the output sequencing of one array conforms to the input sequencing of the other.

Fig. 9. Array algorithm for $M_S N_K^2: \alpha(\alpha *)$.

Fig. 10. Systolic algorithm for matrix-vector multiplication.

A similar result can be obtained for computing $AB + C$, where C is an $m \times 1$ column vector. When input is of the form $\langle C, \langle A, B \rangle \rangle$, the program Matrix_Vector can be revised as follows:

Def MVC == {1, 2 • dist_r • α trans • $\alpha(\alpha *)$ • trans • $\alpha(L +)$

The application of MVC to $(M, (MN, N))$, the structure of $\langle C, \langle A, B \rangle \rangle$, is as follows:

$(M, (MN, N)): [1, 2 \bullet \text{dist}_r \bullet \alpha \text{trans} \bullet \alpha(\alpha *)]$
 $\bullet \text{trans} \bullet \alpha(L +)$
 $= (M, (MN, N): \text{dist}_r \bullet \alpha \text{trans} \bullet \alpha \alpha *): \text{trans} \bullet \alpha(L +)$
 $= (M, M2N: \alpha \text{trans} \bullet \alpha \alpha *): \text{trans} \bullet \alpha(L +)$
 $= (M, MN2: \alpha \alpha *): \text{trans} \bullet \alpha(L +)$
 $= (M, MN): \text{trans} \bullet \alpha(L +)$
 $= M(a, N): \alpha(L +)$
 $= M$

The basic sequencing level aFP program that

$M(a, N): \alpha(L +) = M$

matches is

$$M_S(a, N_K)_P: \alpha(L f) = M_S \quad (3)$$

Then, the following basic sequencing level programs help determine the array algorithm.

$$(M_S, M_S N_K)_P: \text{trans} = M_S(a, N_K)_P \quad (4)$$

$$M_S N_K: \alpha(\alpha f) = M_S N_K \quad (5)$$

Program (5) is the same as program (2), their input data objects are exactly the same, so are their mappings. As mentioned before, program (4), whose function is "trans", does not map to any array algorithm. For program (3), the corresponding data object of $M_S(a, N_K)_P$ is

$$\langle \langle c_1, \langle p_{11}, p_{12}, \dots, p_{1n} \rangle \rangle, \langle c_2, \langle p_{21}, p_{22}, \dots, p_{2n} \rangle \rangle, \dots, \langle c_m, \langle p_{m1}, p_{m2}, \dots, p_{mn} \rangle \rangle \rangle$$

where $p_{ij} = a_{ij} * b_j$, for $1 \leq i \leq m, 1 \leq j \leq n$. Therefore, the

array algorithm is like that in Fig. 6, except that "+" should replace "f" operation in each PE. The sequencing of other data streams are guaranteed by programs (5), (4), and (3), in that sequence. The output of (5), $M_S N_K$, is part of the input of (4),

and the output of (4), $M_S(a, N_K)_P$, is the input of (3).

Furthermore, the input and output of (4) are the same at the sequencing level, in spite of their difference at the structure level. Therefore, we can obtain the algorithm depicted in Fig. 11 by connecting the corresponding arrays of programs (5) and (3).

Fig. 11. MVC systolic algorithm.

4.2. Key enumeration

The enumeration sort (sorting by counting) [12] is composed of ranking (key enumeration) and rearranging. The ranking process inputs a sequence of keys $k_1, k_2, \dots, k_n$ and outputs a sequence of ranks $r_1, r_2, \dots, r_n$ to represent that k_i is the (r_i+1) th smallest key in the input sequence. Then in the rearranging process the records are rearranged according to the ranks of their keys.

Assume that the input is $\langle k_1, \dots, k_n \rangle$, we write a program as follows:

Def KE == [id, id] • pairing • $\alpha(\alpha \text{ Greater}) \bullet \alpha(/ +)$

The following shows the application of KE to a proper input:

$\langle 3, 5, 1 \rangle: \text{KE}$
 $= \langle \langle 3, 5, 1 \rangle, \langle 3, 5, 1 \rangle \rangle: \text{pairing} \bullet \alpha(\alpha \text{ Greater}) \bullet \alpha(/ +)$
 $= \langle \langle \langle 3, 3 \rangle, \langle 3, 5 \rangle, \langle 3, 1 \rangle \rangle, \langle \langle 5, 3 \rangle, \langle 5, 5 \rangle, \langle 5, 1 \rangle \rangle, \langle \langle 1, 3 \rangle, \langle 1, 5 \rangle, \langle 1, 1 \rangle \rangle \rangle: \alpha(\alpha \text{ Greater}) \bullet \alpha(/ +)$
 $= \langle \langle 0, 0, 1 \rangle, \langle 1, 0, 1 \rangle, \langle 0, 0, 0 \rangle \rangle: \alpha(/ +)$
 $= \langle 1, 2, 0 \rangle$

The application of KE to the structure N is as follows.

N: KE
 $= (N, N): \text{pairing} \bullet \alpha(\alpha \text{ Greater}) \bullet \alpha(/ +)$
 $= NN2: \alpha(\alpha \text{ Greater}) \bullet \alpha(/ +)$
 $= NN: \alpha / +$
 $= N$

The basic sequencing level aFP program that satisfies

$NN: \alpha / + = N$

is

$$M_S N_K: \alpha / f = M_S \quad (6)$$

Then the following series of basic sequencing level programs are matched to help construct the array algorithm for key enumeration.

$$M_S N_K: \alpha \alpha f = M_S N_K \quad (7)$$

$$(M_S, N_S)_p: \text{pairing} = M_S N_K \quad (8)$$

The corresponding array algorithms of programs (6), (7), and (8) have the same array structures as those in Fig. 5, 2, and 7, respectively. Fig. 12 depicts the result of connecting the arrays. Since the main operation of the first row of the array is to generate data pairs for the second row, these two rows can actually be merged into a single row.

Fig. 12. Semisystolic array for key enumeration.

4.3. All-nearest-neighbors and closest pair

Two computational geometry problems which are related to the key enumeration problem are discussed in the following. The all-nearest-neighbors problem: given n points in the space, for each point find the point nearest to it. The closest-pair problem: given n points in the space, find the two that are closest together among them. We modify the program KE and derive systolic arrays for these problems.

When input is of the form $\langle p_1, p_2, \dots, p_n \rangle$, where $p_1, \dots, p_n$ are coordinates of n points, here is an aFP program to solve the all-nearest-neighbors problem:

```
Def All_Nearest_Neighbors ==
  [id, id] * pairing *  $\alpha \alpha$  [Distance, id] *  $\alpha$  (/ Closer)

xdef [[di, [p,r]], [dj, [q,s]]] Closer == di eq !0 --> 2;
  dj eq !0 --> 1;
  di lt dj --> 1; 2
```

Distance is a UDF to compute the distance between two points, and is not explicitly defined here. Closer identifies a pair from $\langle p,r \rangle$ and $\langle q,s \rangle$ that is closer than the other pair by judging from their respective distances d_i and d_j . However, in the context of All_Nearest_Neighbors, p and q are always the same; the real function of Closer is to identify the closer point (r or s) to the point p . Thus, α Closer finds the point closest to a certain point among the set of given points. Since the program structure of All_Nearest_Neighbors is identical with that of KE above, their array structures are exactly the same.

The aFP program for the closest-pair problem is straightforward by making use of All_Nearest_Neighbors:

```
Def Closest_Pair == All_Nearest_Neighbors * / Closer
```

As the output of All_Nearest_Neighbors is of the structure M_S , the basic sequencing level program

$$M_S X_p: / f = X_p \quad (9)$$

is matched, and its corresponding array, like that in Fig. 4, is to be connected to the array for All_Nearest_Neighbors.

4.4. Finite impulse response (FIR) filtering

We want to compute $y_i = w_1 x_i + w_2 x_{i+1} + \dots + w_k x_{i+k-1}$, $1 \leq i \leq n-k+1$. Assume that the input is $\langle \langle w_1, w_2, \dots, w_k \rangle, \langle x_1, \dots, x_n \rangle \rangle$, we write an aFP program for the purpose:

```
Def FIRf == windows *  $\alpha$  ( $\alpha$  *) *  $\alpha$  (/ +)
```

Without loss of generality, we use an example input to show the application of FIRf to a proper input as follows:

```
<<1, 2, 2, 1>, <1, 2, 3, 4, 5, 6>>: FIRf
= <<<1, 1>, <2, 2>, <2, 3>, <1, 4>>,
  <<1, 2>, <2, 3>, <2, 4>, <1, 5>>,
  <<1, 3>, <2, 4>, <2, 5>, <1, 6>>>:  $\alpha$  ( $\alpha$  *) *  $\alpha$  (/ +)
= <<1, 4, 6, 4>, <2, 6, 8, 5>, <3, 8, 10, 6>>:  $\alpha$  (/ +)
= <15, 21, 27>
```

The application of FIRf to the structure (K, N) is as follows:

```
(K, N): FIRf
= (N-K+1)K2:  $\alpha$  ( $\alpha$  *) *  $\alpha$  (/ +)
= (N-K+1)K:  $\alpha$  (/ +)
= (N-K+1)
```

The first application of "windows" to (K, N) is matched with the basic sequencing level program

$$(K_S, N_S): \text{windows} = (N-K+1)_S K_K 2_p \quad (10)$$

which has a corresponding array algorithm as shown in Fig. 8.

Then, for the application of α (α *) to $(N-K+1)_S K_K 2_p$, we use the following basic sequencing level program:

$$M_S N_K: \alpha \alpha f = M_S N_K \quad (11)$$

to produce $(N-K+1)_S K_K$.

Finally, the basic sequencing level program

$$M_S N_K: \alpha / f = M_S \quad (12)$$

is used to generate $(N-K+1)_S$ for the application of α (/ +) to $(N-K+1)_S K_K$.

Fig. 13 depicts the array algorithm for FIR filtering, which is composed of the corresponding array algorithms of programs (9), (10), and (11). The first and second rows of the array can further be merged.

Fig. 13. Semisystolic algorithm for FIR filtering.

5. Concluding remarks

Cremers and Hibbard [6, 7], based on the state transition model, proposed a LISP-like data space notation for the specification of parallel algorithms. Kung et al. [16], aiming at describing matrix and other related parallel algorithms, presented a special purpose language MDPL for programming otherwise derived algorithms on a VLSI wavefront array processor. A programmer can address an entire front of PEs; therefore, the complexity of programming can be reduced. Gross and Lam [10] used an Algol-like language to describe the operations executed on each PE of a systolic system. Receive and send primitives are provided to specify inter-PE communication.

All these languages are not for design purpose, but rather for specifying operations performed on the PEs. While our notation aFP is good for expressing array algorithms without the need to explicitly specify the synchronization and communications between the PEs, it is able to address an entire row of PEs because of the employment of higher-order functions, thereby reducing the size and complexity of programs drastically. Furthermore, as we have shown, aFP can be used to synthesize new regular array algorithms.

Another notation that can describe parallel algorithms, as well as synthesize systolic algorithms, without explicit synchronization and communications, is Chen's Crystal [4, 5]. A Crystal program is a set of first order recursion equations based on the state transition model. While Crystal programs are more general in that they can be mapped to structures other than regular arrays, the programs and mapping are generally more complicated than those of the aFP method.

Given algorithms in aFP, we have demonstrated in this paper how to systematically derive regular array algorithms. We have presented not only mapping rules from aFP functions to array structures, but also several examples of array design. The key point of our mapping method is to use function-array equivalents that are easily provable as building blocks, and construct the array algorithms with building blocks following the composition sequence of functions in a program. Therefore, the correctness of the array algorithms can easily be verified.

In our examples of regular array design, we have used an array model that requires no data preloading. In fact, there is another equivalent model with data preloading and one can be transformed into the other [21].

Since our design method is quite different from others', many of the array algorithms thus derived are also different from theirs. Some of our designs perform better than the previous ones; ours for key enumeration is better than that described in [18]. While some of our designs are essentially the same as others', they may be more intuitive and easier to understand because of our step-by-step construction; our design for FIR filtering is such an example as compared with that by Kung [15].

A fundamental issue in designing VLSI array architectures is how to express parallel algorithms in a notation that is easy to understand by humans and possible to compile into efficient VLSI array processors. We feel that aFP is a promising candidate.

References

- [1] M. Annaratone, et al., Warp architecture and implementation, Proc. 13th Int. Symp. Computer Architecture, pp. 346-356, June 1986.
- [2] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, CACM 21 (8), pp. 613-641, August 1978.
- [3] J. Backus, The algebra of functional programs: function level reasoning, linear equations, and extended definitions, Lecture Notes in Computer Science #107, pp. 1-43, Springer-Verlag, 1981.
- [4] M.C. Chen, Synthesizing systolic designs, Proc. Int'l Symp. on VLSI Technology, System, and Applications, Taipei, Taiwan, May 1985, pp. 209-215.
- [5] M.C. Chen, A parallel language and its compilation to multiprocessor machines or VLSI, Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 131-139.
- [6] A.B. Cremers, and S.Y. Kung, On programming VLSI concurrent array processors, Integration, Vol. 2, 1984, pp. 15-26.
- [7] A.B. Cremers, and T.N. Hibbard, Executable specification of concurrent algorithms in terms of applicative data space notation, in VLSI and Modern Signal Processing (S.Y. Kung et al., eds.), Englewood Cliffs, NJ: Prentice-Hall, 1985.
- [8] A.C. Fleck, Structuring FP-style functional programs, Comput. Lang. Vol. 11, No.2, pp.55-63, 1986.
- [9] J.A.B. Fortes, K.S. Fu, and B.W. Wah, Systematic approaches to the design of algorithmically specified systolic arrays, Proc. IEEE ICASSP, Tampa, FL, pp. 300-303, 1985.
- [10] T. Gross and M.S. Lam, Compilation for a high-performance systolic array, Proc. SIGPLAN 86 Symp. on Compiler Construction, ACM SIGPLAN, June 1986, pp. 27-38.
- [11] G. Jones and W. Luk, Exploring designs by circuit transformation, in Systolic Arrays (W. Moore, A. McCabe, and R. Urquhart eds.), pp. 91-98, Adam Hilger, 1987.
- [12] K.E. Knuth, The art of computer programming, Vol. 3: Sorting and Searching, Addison-Wesley, 1973.
- [13] H.T. Kung and C.E. Leiserson, Systolic arrays (for VLSI), Proc. SIAM Sparse Matrix Symp., pp. 256-282, 1978.
- [14] H.T. Kung, Why systolic architectures? Computer, Vol. 15, pp. 37-46, Jan. 1982.
- [15] H.T. Kung, Notes on VLSI computation, in Parallel Processing Systems (D.J. Evans ed.), Cambridge, England: Cambridge University Press, pp. 339-356, 1982.
- [16] S.Y. Kung, D.S. Arun, R.J. Gal-Ezer, and D.V. Bhaskar Rao, Wavefront array processor: Language, architecture, and applications, IEEE Trans. Comput. C-31 (11), 1982, pp. 1054-1066.
- [17] C.E. Leiserson, Systolic and semisystolic design, 1983 IEEE Int. Conf. on Computer Design/VLSI in Computers, 1983.
- [18] F.C. Lin and K. Chen, On the design of a unidirectional systolic array for key enumeration, to appear in IEEE Trans. on Computers, 1988.
- [19] F.C. Lin and I.C. Wu, Broadcast normalization in systolic design, to appear in IEEE Trans. on Computers, 1988.
- [20] W.T. Lin, C.Y. Chin, and C.Y. Ho, Integrating systolic arrays into a supersystem, IEEE Computer, Vol. 20, pp. 100-101, July 1987.
- [21] Y.C. Lin and F.C. Lin, A functional programming approach to systolic design, to appear in Journal of the Chinese Institute of Engineers, 1988.
- [22] F. Meshkinpour and M. D. Ercegovac, A functional language for description and design of digital systems: sequential constructs, Proc. 22nd Design Automation Conference, pp. 238-244, June 1985.
- [23] D. Patel, M. Schlag and M. Ercegovac, nuFP: an environment for the multi-level specification, analysis, and synthesis of hardware algorithms, Lecture Notes in Computer Science #201, Springer-Verlag, pp. 238-255, 1985.
- [24] A.D. Robison, Illinois Functional Programming: a tutorial, Byte, pp. 115-125, Feb. 1987.
- [25] M. Schlag, Extracting geometry from FP for VLSI layout, Tech. Rep. CSD-840043, UCLA Computer Science Dept, Los Angeles, California, Oct. 1984.
- [26] M. Sheeran, muFP, a language for VLSI design, Proc. ACM Symp. on Lisp and Functional Programming, pp. 104-112, 1984.
- [27] M. Sheeran, Designing regular array architectures using higher order functions, Lecture Notes in Computer Science #201, pp. 220-237, Springer-Verlag, 1985.
- [28] L. Snyder, Introduction to the configurable, highly parallel computer, IEEE Computer, pp. 47-56, Jan. 1982.