Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 217-233

OPTIMISATION OF BIDIRECTIONAL SYSTOLIC ARRAYS WITH SPARSE INPUT BY "FOLDING"

L. Ruff (Cluj-Napoca, Romania)

Abstract. We present a study of an interesting transformation procedure (called "folding") which can be applied on bidirectional systolic arrays with sparse input in order to improve their efficiency (the number of processing elements is reduced to the half while accomplishing the same throughput). The transformation is exemplified on a systolic array for sequences comparison.
The presented work is used as an additional optimization step of our automatic systolic array design method implemented as a set of rewrite rules on top of the computer algebra system Mathematica.

1. Introduction

The automatic systolic array design methods available in the literature (see [13] for a short survey) often lead to arrays which work inefficiently from a certain point of view. The idea of optimising such arrays is usually tackled separately and/or it is performed rather intuitively than in an automatic manner.

We demonstrate in this paper starting from a particular (inefficient) array type how the optimisation step for this case can be performed automatically by using the power of the rewriting technique. The rewrite rules are detected after a former study of the starting and resulting array types which have to be computationally equivalent.

A typical example for inefficiently working systolic arrays are the arrays with bidirectional sparse data flow. The sparse input insures that the elements of a data stream will meet each of the elements of the other data stream advancing

in the opposite direction, however because of the sparse input the processing elements (PEs) will perform a useful computation only at each second time step.

The transformation presented in this paper improves the efficiency of such arrays. It does not affect the timing of the computations (as other circuit-optimising transformations like retiming [8, 3]), it rather changes (only) the placing of the computations such that the number of processors needed for the computation is reduced to the half. A set of rewrite rules is determined, with the help of which the transformation can be done automatically and it is also easy to implement.

The presented transformation is used as additional optimisation step of our *functional–based* automatic systolic array design method [5, 11] implemented [essentially] also as a set of rewrite rules.

The paper is organised as follows: we first present the notations (referred to as *list notation* [5, 11]) used in this paper which allow us to reason about the data streams of the array in a concise way. Using these notations we present the functioning of bidirectional arrays with sparse input, then that of the "folded" array. By induction on the time t we show that starting from a given input scheme, the output pattern of the two arrays is the same, thus they are computationally equivalent. We detect the set of rewrite rules which can be applied onto the transition function of the array with sparse input in order to get the transition function of the folded array which needs only the half of the number of PEs to perform the same computation.

The application of the rewrite rules is exemplified on a case study presenting a non trivial problem, namely that of sequences comparison: a bidirectional systolic array with sparse input (which was generated with an automatic design method in [14]) is transformed into a more efficient "folded" array.

2. List notation

We represent the input streams of a systolic array by lists of fixed size objects (that is lists of scalars), using the following notations and conventions:

- We denote by X_i the *infinite list* $\langle x_i, x_{i+1}, x_{i+2} \ldots \rangle$ (note that i can also be negative). X stands for X_0. $X_{n,n+m}$ (where $n \in \mathbb{Z}$ and $m \in \mathbb{N}$) denotes the finite list having $m+1$ elements $\langle x_n, x_{n+1}, \ldots x_{n+m} \rangle$.
- We denote by a^n the list of n elements all equal to a and by a^∞ the infinite constant list with all elements equal to a.
- For any list $X = \langle x_0, x_1, \ldots, x_n, \ldots \rangle$, we denote by $H[X] = x_0$ the *head* of it and by $T[X] = \langle x_1, \ldots, x_n, \ldots \rangle$ the *tail* of it.

- The k^{th} *tail* respectively *head* of X:
  $T_k[X] = \langle x_k, x_{k+1}, \ldots, x_n, \ldots \rangle$, for $k \geq 0$ ($T_0[X] = X$, $T_1 = T$)
  T_k, for $k < 0$ is obtained by iterating T_{-1} $|k|$ times, where $T_{-1}[X_i] = X_{i-1}$ (if x_{i-1} is not defined, then a blank value is inserted in the front of the list).
  Intuitively, the application of the tail function of order k on a list X_i, when k is negative should be understood as if we were operating on the list which is infinite in both directions: $\langle \ldots, x_{-1}, x_0, x_1, \ldots \rangle$. X_i can be seen as its sublist, then the application of T_k means $|k|$ steps "backward" in this "extended" list.
  $H_k[X] = H[T_k[X]]$ gives the $(k+1)^{\text{th}}$ element of X (thus $H_0 = H$).

- The *prefix* of order n of a list is $P_n[X] = \langle x_0, \ldots, x_{n-1} \rangle = X_{0,n-1}$.

- The *concatenation* of two lists is denoted by "$\frown$":
  $\langle a_0, a_1, \ldots, a_k \rangle \frown X = \langle a_0, a_1, \ldots, a_k, x_0, x_1, \ldots \rangle$.
  The first operand must be finite, but the second may also be infinite. We also use "$\frown$" for *prepending* a scalar to a (finite or infinite) list: $a \frown X = \langle a \rangle \frown X$.

- We use (as in the theory of cellular automata) a special *quiescent symbol* "\$" (which belongs to all scalar types) in order to encode the "blank" values.

- A *sparse list* is defined with the help of the list function $Sparse_k$:
  $Sparse_k[X_i] = H[X_i] \frown (\$^k \frown Sparse_k[T[X_i]])$.
  For brevity we denote $Sparse_k[X_i]$ by $(X_i)_{\$k}$.
  $(X_i)_{\$1} = (X_i)_{\$} = \langle x_i, \$, x_{i+1}, \$, \ldots \rangle$. By convention $(X_i)_{\$0} = X_i$.

  We have extended the T_m function to negative m values, too. Here we define how T_{-1} applies to sparse lists, then T_m is obtained by iterating the T_{-1} function $|m|$ times:

$$\begin{array}{lcl} T_{-1}[Sparse_k[X_i]] & = & \$ \frown Sparse_k[X_i] \\ \quad \forall 0 < n \leq k-1, & & \\ T_{-1}[\$^n \frown Sparse_k[X_i]] & = & \$^{n+1} \frown Sparse_k[X_i] \\ T_{-1}[\$^k \frown Sparse_k[X_i]] & = & Sparse_k[X_{i-1}] \end{array}$$

The following properties can be either deduced directly from the definition of the head and tail function, respectively the $Sparse_k$ function (see the first two properties) or can be easily proven with mathematical induction on m, both for positive and negative m values.

Properties:

$$
\begin{array}{ll}
T\ Sparse_k & = \$^k \frown Sparse_k\ T \\
H\ Sparse_k & = H \\
T_m\ Sparse_k & = Sparse_k\ T_{\frac{m}{k+1}}\ ,\ \text{if } (k+1)|m \\
\text{in particular } (k=1): & \\
T_m\ Sparse_1 & = \begin{cases} Sparse_1\ T_{\frac{m}{2}} & \text{if } m \text{ is even,} \\ \$ \frown Sparse_1\ T_{\frac{m+1}{2}} & \text{if } m \text{ is odd.} \end{cases}
\end{array} \tag{1}
$$

3. Bidirectional arrays with sparse input

Bidirectional systolic arrays have a two directional data flow. We consider in this paper arrays where the data advances with the same velocity in both, left-to-right and right-to-left directions. The sparse input insures that the elements of one data stream will meet each element of the data stream advancing in the opposite direction.

Each PE has a left-to-right and a right-to-left input channel, denoted by x, respectively y and an internal state register (also called local memory variable), r.

The global input to the array consists of two lists: X is the left-to-right input list, respectively $\$ \frown Y$ in case of an even number of PEs (or Y, if the number of PEs is odd) is the right-to-left input list. Note that there is no restriction to only one input stream in each direction: the elements of both lists may also be tuples of scalars. The same holds for r which also may be a fixed-size tuple of internal state registers, then $r = \langle r|1|, r|2|, \ldots, r|k|\rangle$.

One (or both) of the output streams which leave the array at the boundary PEs is (are) considered as the result computed by the array. In some special cases the elements of the result are computed in the internal registers. In this case the left-to-right or right-to-left communication channels can be used at the end of the computation in order to transmit the results towards the boundary PEs or an additional direct communication channel may be used specially for the collection of the results from each PE.

Figure 1 depicts the initial state of such an array for the case when the number of PEs (n) is even

The initial values of the input channels of PE_i, for $0 \leq n-1$ are the elements of $P_n[\$ \frown X_{-(\frac{n}{2}-1)_\$}]$ and $P_n[Y_{-\frac{n}{2}\$}]$, as indicated on Fig 1. Besides the elements of the global input $X_\$$ and $\$ \frown Y_\$$ these initial values are also considered to be given as parameters of the problem.

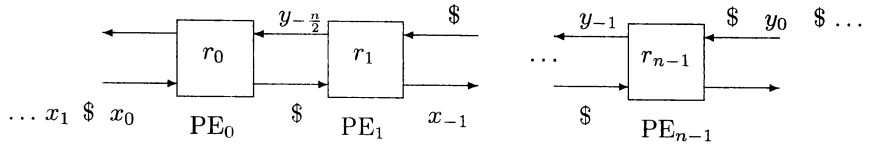


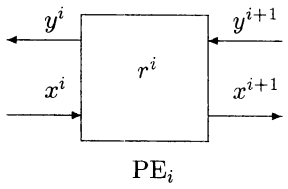
Figure 1. Bidirectional array with sparse input (n is even). Initial state

We denote by x_t^i the value of the communication channel x^i at time step t. Note the difference between the above mentioned notation and the way, how the list elements are denoted: let us suppose for example, that the elements of the sparse list $X_\$$ are fed into the communication channel x^i, then at a certain time step t the value of the input channel x^i could be x_j (or \$), that is $x_t^i = x_j$ ($x_t^i = \$$) holds.

x_0^i and y_0^{i+1} denote the initial values of the x^i respectively y^{i+1} input channels of PE_i, $\forall i, 0 \leq i \leq n-1$. Then

$$
(2) \qquad
\begin{aligned}
x_0^i &= H_{-i}[X_\$] \overset{(2)}{=} \begin{cases} x_{-\frac{i}{2}} & \text{if } i \text{ is even,} \\ \$ & \text{if } i \text{ is odd,} \end{cases} \\
y_0^{i+1} &= H_{-(n-1-i)}[\$ \smile Y_\$] \overset{(2)}{=} \begin{cases} y_{-\frac{n-i}{2}} & \text{if } i \text{ is even,} \\ \$ & \text{if } i \text{ is odd.} \end{cases}
\end{aligned}
$$

The transition function of a PE (that is the computations performed by a PE) is described on Fig. 2. We assume that the functions f_x, f_y and f_r are producing blanks when applied to blank arguments and if at least one of the arguments is blank, then the corresponding value is transmitted unchanged, that is: $x_{t+1}^{i+1} = x_t^i$, $y_{t+1}^i = y_t^{i+1}$, respectively $r_{t+1}^i = r_t^i$.



Computations:

$$
\begin{aligned}
x_{t+1}^{i+1} &= f_x[x_t^i, r_t^i, y_t^{i+1}] \\
y_{t+1}^i &= f_y[x_t^i, r_t^i, y_t^{i+1}] \\
r_{t+1}^i &= f_r[x_t^i, r_t^i, y_t^{i+1}]
\end{aligned}
$$

Figure 2. Computations of a PE in a bidirectional array with sparse input

As already mentioned, the sparse input insures that each element of the two input lists will meet each other, but on the other hand this is also the reason for which the array is inefficient: the PEs are namely idle at each second time step.

A commonly used idea is to merge two PEs which are working alternately (that is to map the computations of two different PEs which are working alternately onto one single PE) in order to transform the array into a more efficient one: if the array with sparse input had n PEs, then the same problem can be solved using only $n/2$ PEs.

One possibility is to merge two neighbouring PEs (PE_i and PE_{i+1}) into one single PE. This is possible because two neighbouring PEs are active in alternating time steps, thus the merged PE will do the computations of PE_i, respectively PE_{i+1} in the even respectively odd time steps.

The compression of the array by merging more PEs into a single processor is also the subject of the so called partitioning problem, which means the mapping of the systolic algorithm onto an array with a fixed number of processors. The partitioning problem has two different forms: LPGS (locally parallel, globally sequential) [9] and LSGP (locally sequential, globally parallel) [2, 4]. The latter approach permits not only the synthesis of systolic arrays with a fixed number of PEs, but as a particular case it can improve the efficiency of the PEs by compressing the array. Merging neighbouring PEs is considered to be generally more advantageous if we want to pipeline many problems on the same array.

In the sequel we present in detail a different solution of array compression, based on the same idea of merging two PEs which are working alternately, but not the neighbouring PEs are considered. In our case, that is the optimisation of bidirectional arrays, the delay between two problems to be solved on the same compressed array is not affected by the fact that we merge a PE active at the beginning with a PE active much later. Moreover, the input and output lists will be introduced into, respectively collected from the array at the same (leftmost) PE, which can be useful from the point of view of the implementation.

Thus, the main difference between the two array types resulted after the two mentioned compression types consists in their architecture, the complexity of a PE and the time required for the computation is the same.

We assume that the number of PEs, n, is even (otherwise an additional "dummy" PE should be introduced before merging two PEs).

4. "Folded" array

An interesting solution which enhances the efficiency of the bidirectional array with sparse input consists in mapping the computations of PE_{n-1-i} onto PE_i, for $i = 0, \ldots, \frac{n}{2} - 1$. We call the resulted array "folded" array, because the mapping process can be intuitively understood as folding the inefficient array in the middle and merging the functionality of the overlapping PEs, as shown on Fig 3.

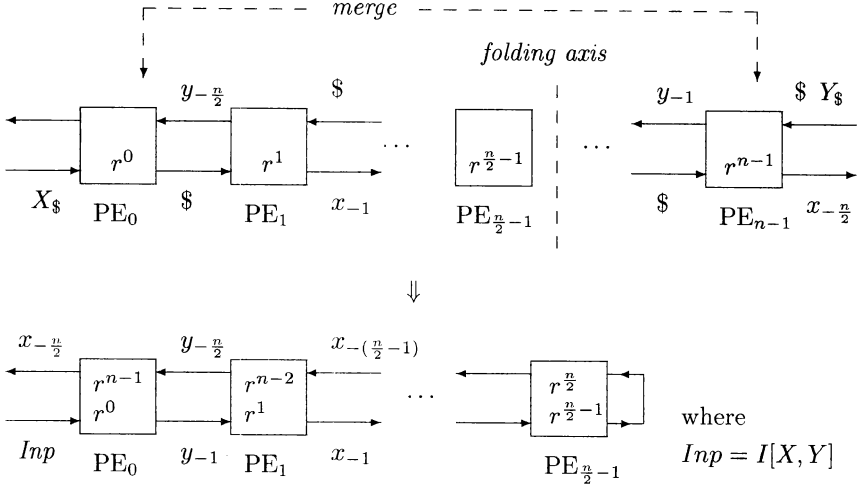


Figure 3. Bidirectional "folded" array. Initial state

Now each PE has to perform different computations in the successive time steps. The PEs are not aware of the time, but this problem can be solved with either the introduction of a control signal with alternating 0, 1 values, or with the help of an additional internal state register s. The s^i state register of PE_i, $0 \leq i \leq \frac{n}{2} - 1$ is initialised with the value s_0^i such that:

$$s_0^i = \begin{cases} 0 & \text{if } i \text{ is even,} \\ 1 & \text{othervise.} \end{cases}$$

The computations performed by a PE are shown on Fig. 4. The last PE, that is $PE_{\frac{n}{2}-1}$ is slightly different from the other PEs of the array: its left-to-right output channel is connected to its right-to-left input.

Computations: (different computations for $i = \frac{n}{2} - 1$)

$$
\begin{array}{rcl}
s^i_{t+1} = 1 - s^i_t & & \\
\text{case } s^i_t & = & 0 \\
b^i_{t+1} & = & f_y[a^i_t, \underline{w}^i_t, b^{i+1}_t]\ b^{\frac{n}{2}-1}_{t+1} = f_y[a^{\frac{n}{2}-1}_t, \underline{w}^{\frac{n}{2}-1}_t, a^{\frac{n}{2}}_t] \\
a^{i+1}_{t+1} & = & f_x[a^i_t, \underline{w}^i_t, b^{i+1}_t] \\
\underline{w}^i_{t+1} & = & f_r[a^i_t, \underline{w}^i_t, b^{i+1}_t] \\
\text{case } s^i_t & = & 1 \\
a^{i+1}_{t+1} & = & f_y[b^{i+1}_t,\ \bar{w}^i_t, a^i_t] \\
b^i_{t+1} & = & f_x[b^{i+1}_t,\ \bar{w}^i_t, a^i_t] b^{\frac{n}{2}-1}_{t+1} = f_x[a^{\frac{n}{2}}_t,\ \bar{w}^{\frac{n}{2}-1}_t, a^{\frac{n}{2}-1}_t] \\
\bar{w}^i_{t+1} & = & f_r[b^{i+1}_t,\ \bar{w}^i_t, a^i_t]
\end{array}
$$

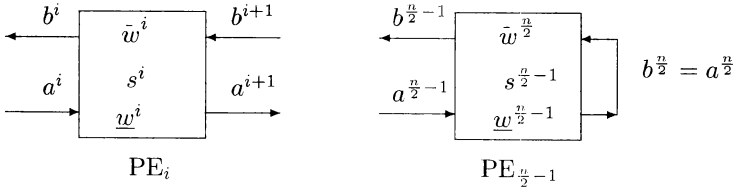


Figure 4. Computation of a PE in a bidirectional "folded" array

Let us consider the function I which interleaves two lists:

$$I[a \smile A, b \smile B] = \langle a, b \rangle \smile I[A, B] \ .$$

The input list Inp can be defined as $Inp = I[X, Y]$. The initial values of the input channels are

$$
\begin{array}{rcll}
a^k_0 & = & H_{-k}[Inp] & 0 \le k \le \frac{n}{2}, \\
b^k_0 & = & H_{-(n-k)}[Inp] & 0 \le k \le \frac{n}{2}.
\end{array} \tag{3}
$$

Note that

$$
H_i[I[A,B]] \overset{(2)}{=} \begin{cases}
H_i[A_\$] = HT_iA_\$ = HA_{\frac{i}{2}} & \text{if } i \text{ is even,} \\
H_{i-1}[B_\$] = HT_{i-1}B_\$ = HB_{\frac{i-1}{2}} & \text{if } i \text{ is odd.}
\end{cases} \tag{4}
$$

The computations presented on Fig. 4 describe the transition function of a PE of the "folded" array in the classical way, which is rather concise and is recommended to be used in case of the implementation of the systolic array.

We introduce here, however, a new notation (see Fig. 5) for the same transition function which is more intuitive, thus the functioning of a PE of this particular array can be expressed more clearly. This notation reveals namely one of the main characteristics of the PE of this particular array: the input channels receive/output the values of the lists X, respectively Y in the alternating time steps. Thus the new notation makes the understanding and the reasoning about the functioning of the array much easier.

By convention we denoted an x value by $\bar{x}$, when it is input from right to left (see the upper input channel on Fig. 5 a)) and by $\underline{x}$, when it is input from left to right (Fig. 5 b)). In the same way, we use the notation $\bar{y}$ and $\underline{y}$ for the y values.

The connection between the two notations can be expressed in the following way:

$$(5)\qquad \begin{aligned} a_t^i &= \begin{cases} \underline{y}_t^i & \text{if } i+t \text{ is even } (s_{t-1}^i = 1), \\ \underline{x}_t^i & \text{if } i+t \text{ is odd } (s_{t-1}^i = 0), \end{cases} \\ b_t^i &= \begin{cases} \bar{x}_t^i & \text{if } i+t \text{ is even } (s_{t-1}^i = 1), \\ \bar{y}_t^i & \text{if } i+t \text{ is odd } (s_{t-1}^i = 0). \end{cases} \end{aligned}$$

4.1. Computational equivalence of the two array types

Hereafter we show that the "folded" array described in this section and the bidirectional array with sparse input, presented in Sect. 3 perform the same computation. That is, we demonstrate by induction on t that for the input scheme given in Fig. 3 the following holds (we denote the PEs of the "folded" array by PE'): at a certain time step t, the transition function of PE'$_i$ outputs the same result as the transition function of PE$_i$, if $i+t$ is odd (which is equivalent to $s_{t-1}^i = 0$), respectively PE$_{n-1-i}$, if $i+t$ is even ($s_{t-1}^i = 1$).

Formally:

$$\forall i, \quad 0 \le i \le \frac{n}{2} - 1, \quad \forall t \ge 0$$

$$(6)\quad \text{if } s_t^i = 0, \begin{cases} \bar{y}_{t+1}^i &= y_{t+1}^i, \\ \underline{x}_{t+1}^{i+1} &= x_{t+1}^{i+1}, \\ \underline{w}_{t+1}^i &= r_{t+1}^i, \\ (\bar{w}_{t+1}^i &= \bar{w}_t^i), \end{cases} \quad \text{if } s_t^i = 1, \begin{cases} \underline{y}_{t+1}^{i+1} &= y_{t+1}^{n-1-i}, \\ \bar{x}_{t+1}^i &= x_{t+1}^{n-i}, \\ \bar{w}_{t+1}^i &= r_{t+1}^{n-1-i}, \\ (\underline{w}_{t+1}^i &= \underline{w}_t^i). \end{cases}$$

Computations:

$$\underline{y}^{i+1}_{t+1} = f_y[\bar{x}^{i+1}_t,\ \bar{w}^i_t, \underline{y}^i_t]$$

$$\bar{x}^i_{t+1} = \begin{cases} f_x[\bar{x}^{i+1}_t,\ \bar{w}^i_t, \underline{y}^i_t], & \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_x[\underline{x}^{i+1}_t,\ \bar{w}^i_t, \underline{y}^i_t], & \text{if } i = \frac{n}{2} - 1 \end{cases}$$

$$\bar{w}^i_{t+1} = f_r[\underline{x}^{i+1}_t,\ \bar{w}^i_t, \underline{y}^i_t]$$

$$\underline{w}^i_{t+1} = \underline{w}^i_t$$

$$\bar{y}^i_{t+1} = \begin{cases} f_y[\underline{x}^i_t,\ \underline{w}^i_t, \bar{y}^{i+1}_t], & \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[\underline{x}^i_t,\ \underline{w}^i_t, \underline{y}^{i+1}_t], & \text{if } i = \frac{n}{2} - 1 \end{cases}$$

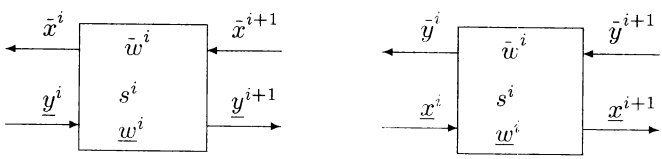
$$\underline{x}^{i+1}_{t+1} = f_x[\underline{x}^i_t, \underline{w}^i_t, \bar{y}^{i+1}_t]$$

$$\underline{w}^i_{t+1} = f_r[\underline{x}^i_t, \underline{w}^i_t, \bar{y}^{i+1}_t]$$

$$\bar{w}^i_{t+1} = \bar{w}^i_t$$

common computation:

$$s^i_{t+1} = 1 - s^i_t$$



a) $s^i_t = 1$ **b)** $s^i_t = 0$

Figure 5. Computations of a PE in a bidirectional "folded" array. A different view

Base step: we verify that (6) holds for $t = 0$.

Indeed, for the case when $s^i_t = s^i_0 = 0$ we have (i is even):

$$\bar{y}^i_1 = \begin{cases} f_y[\underline{x}^i_0, \underline{w}^i_0, \bar{y}^{i+1}_0], \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[\underline{x}^i_0, \underline{w}^i_0, \underline{y}^{i+1}_0], \text{if } i = \frac{n}{2} - 1 \end{cases} \overset{(5)}{=} \begin{cases} f_y[a^i_0, r^i_0, b^{i+1}_0], \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[a^i_0, r^i_0, a^{i+1}_0], \text{if } i = \frac{n}{2} - 1 \end{cases} =$$

$$\overset{(3)}{=} \begin{cases} f_y[H_{-i}[Inp], r^i_0, H_{-(n-i-1)}[Inp]], \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[H_{-i}[Inp], r^i_0, H_{-(i+1)}[Inp]], \text{if } i = \frac{n}{2} - 1 \end{cases} =$$

$$\overset{(4)}{=} \begin{cases} f_y[x_{-\frac{i}{2}}, r_0^i, y_{-\frac{n-i}{2}}], \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[x_{-\frac{i}{2}}, r_0^i, y_{-\frac{i}{2}-1}], \text{if } i = \frac{n}{2} - 1 \end{cases} =$$

$$= f_y[x_{-\frac{i}{2}}, r_0^i, y_{-\frac{n-i}{2}}], \text{if } 0 \le i \le \frac{n}{2} - 1 =$$

$$\overset{(2)}{=} f_y[x_0^i, r_0^i, y_0^{i+1}] = y_1^i$$

We can similarly check that (6) holds for $\underline{x}$ and $\underline{w}$, then we can verify the case $s_t^i = s_0^i = 1$ (i is odd) similarly.

Induction step: **if** (6) holds for $t = k$ (*induction hypothesis*), **then** (6) also holds for $t = k + 1$.

We consider here only the computation of $\bar{y}^i$ respectively $\underline{y}^{i+1}$. One can similarly check the induction step for the other computations from (6).

For the case $s_t^i = s_{k+1}^i = 0$ (that is, $i + k$ is odd) we have:

$$\bar{y}_{k+2}^i \overset{\text{def.}}{=} \begin{cases} f_y[\underline{x}_{k+1}^i, \underline{w}_{k+1}^i, \bar{y}_{k+1}^{i+1}] & \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[\underline{x}_{k+1}^i, \underline{w}_{k+1}^i, \underline{y}_{k+1}^{i+1}] & \text{if } i = \frac{n}{2} - 1 \end{cases} =$$

$$\overset{\text{(ind. hyp.)}}{=} \begin{cases} f_y[x_{k+1}^i, r_{k+1}^i, y_{k+1}^{i+1}] & \text{if } 0 \le i \le \frac{n}{2} - 2 \\ f_y[x_{k+1}^i, r_{k+1}^i, y_{k+1}^{n-1-i}] & \text{if } i = \frac{n}{2} - 1 \end{cases} =$$

$$= f_y[x_{k+1}^i, r_{k+1}^i, y_{k+1}^{i+1}] \text{ if } 0 \le i \le \frac{n}{2} - 1 =$$

$$\overset{\text{def.}}{=} y_{k+2}^i \,.$$

In the same way, for $s_t^i = s_{k+1}^i = 1$ (that is, $i + k$ is even):

$$\underline{y}_{k+2}^{i+1} \overset{\text{def.}}{=} f_y[\bar{x}_{k+1}^{i+1}, \bar{w}_{k+1}^i, \underline{y}_{k+1}^i] =$$

$$\overset{\text{(ind. hyp.)}}{=} f_y[x_{k+1}^{n-(i+1)}, r_{k+1}^{n-1-i}, y_{k+1}^{n-1-(i-1)}] =$$

$$= f_y[x_{k+1}^{n-1-i}, r_{k+1}^{n-1-i}, y_{k+1}^{n-i}] \overset{\text{def.}}{=} y_{k+2}^{n-1-i} \,.$$

We have shown that for $t = k + 1$

$$\begin{aligned} \bar{y}_{t+1}^i &= y_{t+1}^i && \text{if } s_t^i = 0, \\ \underline{y}_{t+1}^{i+1} &= y_{t+1}^{n-1-i} && \text{if } s_t^i = 1. \end{aligned}$$

The induction step can be performed in a similar way for the computation of $\underline{x}^{i+1}$, $\bar{x}^i$, $\underline{w}^i$ and $\underline{w}^i$.

It turns out that the "folded" array with constant internal state registers defined in this section computes the same list of results as the corresponding bidirectional array with sparse input.

The above considerations allow us to formulate the transformation which should be applied on the transition function of a bidirectional array with sparse input, in order to get the transition function of the "folded" array as a set of rewrite rules. Let us denote the expression of the transition function of the array with sparse input corresponding to the computation of x, y and r with E_x, E_y and E_r, respectively.

The set of rules to be applied for example onto E_x are the following (see the set of rules on the right hand side of the operator "/.")

$$
\begin{array}{lll}
\text{case } s_t^i & = 0 & \\
 & E_x/.\{x \to \underline{x}, r \to \underline{w}, y \to \bar{y}\}, & \\
\text{case } s_t^i & = 1 & \\
 & E_x/.\{x_{t+1}^{i+1} \to \bar{x}_{t+1}^i, x_t^i \to \bar{x}_t^{i+1}, r \to \bar{w}, y_t^{i+1} \to \underline{y}_t^i\} & \text{if } 0 \le i \le \frac{n}{2} - 2, \\
 & E_x/.\{x_{t+1}^{i+1} \to \bar{x}_{t+1}^i, x_t^i \to \underline{y}_t^{i+1}, r \to \bar{w}, y_t^{i+1} \to \underline{y}_t^i\} & \text{if } i = \frac{n}{2} - 1.
\end{array}
$$

There is no speed-up in the computation of the results, but the number of the PEs used for the computation of the same result was reduced to the half which increases the efficiency of the array. The PEs of the folded array are active at each time step.

5. Systolic array for sequences comparison

The problem of similarity determination has applications specially in bioinformatics and natural language processing. It means the comparison of two sequences while allowing certain mismatches between them. In order to get the similarity between two sequences, we have to align them at first.

Given two sequences (e.g. DNA or RNA), the problem consists in finding the "best" alignment between them. The problem is largely presented in the literature [1]. Most algorithms used are based on dynamic programming.

The problem consists in making the sequences to be of the same size, by inserting gaps. The best alignment is one that maximises some scoring function (for instance, we score $+1$ for each match, -1 for each mismatch and -2 for each gap).

Let $p = p_1 p_2 \dots p_m$ and $t = t_1 t_2 \dots t_k$ be the two sequences. The matrix $A(m \times k)$ will contain the items of p along the rows and the items of t along the columns, and each entry $A(i, j)$ corresponds to the optimal alignment of the i^{th} prefix of p with the j^{th} prefix of t.

$$(7) \qquad A(i,j) = \max \begin{cases} A(i-1,j) - 2 & \text{align } p(i) \text{ with a gap,} \\ A(i,j-1) - 2 & \text{align } t(j) \text{ with a gap,} \\ A(i-1,j-1) \pm 1 & \text{align } p(i) \text{ with } t(j), \\ & +1 \text{ for a match,} \\ & -1 \text{ for a mismatch.} \end{cases}$$

Note. The value $A(i,0)$ stands for aligning the i^{th} prefix of p with the 0^{th} prefix of t. The optimal score is -2. Analogously, $A(0,j)$ stands for aligning the 0^{th} prefix of p with the j^{th} prefix of t. The optimal score is also -2.

5.1. Bidirectional systolic array with sparse input for the problem of sequences alignment

Figure 6 shows the input scheme of a bidirectional systolic array with sparse input for the sequences alignment problem, presented in [14]. Note that the array was generated in an automatic manner using the ideas from the space-time transformation methodology [12, 13] and from the method of [6, 7].

The size of the array is $n = m + k - 1$ (where m is the length of the sequence p and k is the length of t). We have two input streams in both, left-to-right and right-to-left directions: the elements of B and p, respectively A and t. The C values are computed in the internal state registers. Note that the initial values for B and C are copies of the corresponding initial values of the A matrix.

Let us denote the sparse list of the elements of p by $P_\$$ and that of t by $L_\$$ (instead of $T_\$$, in order to avoid the confusion with the tail function). $A_\$$ and $B_\$$ are the two input lists which collect the partial results, such that:

$$\begin{aligned} H_i[A] &= A_{0,i+1}, \\ H_i[B] &= B_{i+1,0}. \end{aligned}$$

The list of initial values for the internal state registers are

$$R_{1,m+k-1} = \langle C_{m-1,0}, C_{m-2,0}, \ldots, C_{0,0}, C_{0,1}, \ldots, C_{0,k-1} \rangle.$$

The elements of $P_\$$ and $L_\$$, $A_\$$ and $B_\$$ are preloaded in the array, which means that the $X_\$$ sparse input list corresponding to the description of the array with sparse input from Sect. 3 is $X_\$ = T_{m-1}[\langle B_\$, P_\$ \rangle]$, and $Y_\$ = T_k[\langle A_\$, L_\$ \rangle]$.

The computations of a PE are shown on Fig. 7, where

$$f_1[a,b,c] = \text{Max}[a,b,c], \text{ and } f_2[x,y] = \begin{cases} 1 & \text{if } x = y, \\ -1 & \text{if } x \neq y. \end{cases}$$

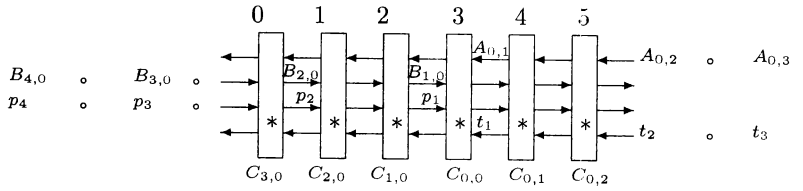


Figure 6. Linear systolic array for sequences alignment ($m = 4$, $k = 3$) [14]

Computations:

$$
\begin{aligned}
a^i_{t+1} &= f_1[a^{i+1}_t - 2, b^i_t - 2, f_2[p^i_t, l^{i+1}_t] + r^i_t] \\
b^{i+1}_{i+1} &= f_1[a^{i+1}_t - 2, b^i_t - 2, f_2[p^i_t, l^{i+1}_t] + r^i_t] \\
r^i_{t+1} &= f_1[a^{i+1}_t - 2, b^i_t - 2, f_2[p^i_t, l^{i+1}_t] + r^i_t] \\
p^{i+1}_{t+1} &= p^i_t \\
l^i_{t+1} &= l^{i+1}_t
\end{aligned}
$$

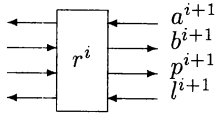


Figure 7. Computations of a PE in the linear array for sequences alignment

5.2. The optimised array

Applying the rewrite rules described in Sect. 4.1 we automatically get the following transition function computed by the PE of the "folded" array:

$$
\begin{aligned}
s^i_{t+1} &= 1 - s^i_t, \\
\text{if } s^i_t &= 0: \\
\langle \bar{a}^i_{t+1}, \bar{l}^i_{t+1} \rangle &= \begin{cases} \langle f_1[\bar{a}^{i+1}_t - 2, \underline{b}^i_t - 2, f_2[\underline{p}^i_t, \bar{l}^{i+1}_t] + \underline{w}^i_t], \bar{l}^{i+1}_t \rangle \\ \qquad \text{if } 0 \le i \le \frac{n}{2} - 2, \\ \langle f_1[\underline{a}^{i+1}_t - 2, \underline{b}^i_t - 2, f_2[\underline{p}^i_t, \underline{l}^{i+1}_t] + \underline{w}^i_t], \underline{l}^{i+1}_t \rangle \\ \qquad \text{if } i = \frac{n}{2} - 1, \end{cases} \\
\langle \underline{b}^{i+1}_{t+1}, \underline{p}^{i+1}_{t+1} \rangle &= \langle f_1[\bar{a}^{i+1}_t - 2, \underline{b}^i_t - 2, f_2[\underline{p}^i_t, \bar{l}^{i+1}_t] + \underline{w}^i_t], \underline{p}^i_t \rangle, \\
\underline{w}^i_{t+1} &= f_1[\bar{a}^{i+1}_t - 2, \underline{b}^i_t - 2, f_2[\underline{p}^i_t, \bar{l}^{i+1}_t] + \underline{w}^i_t], \\
\bar{w}^i_{t+1} &= \bar{w}^i_t, \\
\text{if } s^i_t &= 1: \\
\langle \underline{a}^{i+1}_{t+1}, \underline{l}^{i+1}_{t+1} \rangle &= \langle f_1[\underline{a}^i_t - 2, \bar{b}^{i+1}_t - 2, f_2[\bar{p}^{i+1}_t, \underline{l}^i_t] + \bar{w}^{i+1}_t], \underline{l}^i_t \rangle, \\
\langle \bar{b}^i_{t+1}, \bar{p}^i_{t+1} \rangle &= \begin{cases} \langle f_1[\underline{a}^i_t - 2, \bar{b}^{i+1}_t - 2, f_2[\bar{p}^{i+1}_t, \underline{l}^i_t] + \bar{w}^i_t], \bar{p}^{i+1}_t \rangle \\ \qquad \text{if } 0 \le i \le \frac{n}{2} - 2, \\ \langle f_1[\underline{a}^i_t - 2, \underline{b}^{i+1}_t - 2, f_2[\underline{p}^{i+1}_t, \underline{l}^i_t] + \bar{w}^i_t], \underline{p}^{i+1}_t \rangle \\ \qquad \text{if } i = \frac{n}{2} - 1, \end{cases}
\end{aligned}
$$

Concerning the efficiency of the algorithms for sequences comparison, the best known sequential algorithms solve the problem in $O(mn)$ time and $O(m+n)$ space, where m and n are the lengths of the two sequences [1]. The best space and time optimal parallel algorithm for the same problem requires only $O((m+n)/p)$ space and $O(mn/p)$ time, where p is the number of processors used [10].

The efficiency of our systolic solution is comparable to that of the best space and time optimal parallel algorithms as it achieves a linear speedup (the execution time is $O(m+n)$), wile using $max(m,n)$ processors and requiring constant space per processor. Besides, it was designed and optimised in an automatic way.

6. Conclusions

We have described a transformation process for the optimisation of bidirectional arrays with sparse input, which can be formulated as a set of rewrite rules. The rules were detected after we have shown by induction, using list notation that the bidirectional array with sparse input described in Sect. 3 and the "folded" array presented in Sect. 4 are computationally equivalent. Thus the implementation of this optimisation process is easy and can be performed in a completely automatic manner.

The fact that the automatisation of the optimisation step requires a former analysis of the arrays can be seen as a disadvantage, however, on the other hand, once we detected the rules, they can be easily applied to that class of arrays, which makes it worth to study the optimisation problem in this way for other array types, too.

In Sect. 3 we have mentioned another solution of optimisation (not detailed in this paper), which consists in merging the functionality of the neighbouring PEs (also subject to the more general array partitioning problem [2, 4]). It is a commonly used optimisation procedure, for which the rewrite rules could be detected in the same manner. The main difference between the two implementations consist in the architecture of the resulted arrays, specially the introduction of the input elements into the array, which in our case is performed at the same (leftmost) PE rather than at the two different PEs from the edge. This fact would facilitate the implementation of the problem for the case, when we would like to solve the same problem repeatedly, for input lists having variable lengths. While in the case of the other array type one of the input PEs would always change according to the required array length, in our case the input is pipelined into the array always at the same place, respectively the results can be collected from the same PE as well. We would need additional control registers only to determine the leftmost PE, which has a slightly different behaviour than the others.

The presented work is used as additional optimisation step of our automatic systolic array design method, also implemented as a set of rewrite rules [5, 11] on top of the computer algebra system Mathematica [15]. This points out that it is possible to handle both, the design and optimisation problem for systolic arrays in a completely automatic manner, using the same framework. The presented solution serves as a good example which demonstrates the applicability and efficiency of rewriting technique used in the process of automatic systolic array design and optimisation. It also motivates the further investigation concerning other optimising-transformations which can be applied to systolic arrays.

References

[1] **Aluru S., Futamura N. and Mehrotra K.,** Parallel biological sequence comparison using prefix computations, *J. Parallel Distrib. Comput.,* **63** (3) (2003), 264-272.

[2] **Bu J., Deprettere E. F. and Dewilde P.,** A design methodology for fixed-size systolic arrays, *Application specific array processors*, eds. S.-Y. Kung and E.E. Swartylander, IEEE Computer Society, 1990, 591-602.

[3] **Even G.,** The retiming lemma: A simple proof and applications. INTEG: Integration, *The VLSI Journal,* **20** (1996), 123-137.

[4] **Darte A.,** Regular partitioning for synthesizing fixed-size systolic arrays. Integration, *The VLSI Journal,* **12** (3) (1991), 293-304.

[5] **Jebelean T. and Szakács L.,** Functional-based synthesis of systolic online multipliers, *SYNASC-05 Int. Symp. on Symbolic and Numeric Scientific Computing,* eds. D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Cicortas, A. Abraham and M. Paprzycki, IEEE Computer Society, 2005, 267-275.

[6] **Kazerouni L., Rajan B. and Shyamasundar R.K.,** Mapping linear recurrences onto systolic arrays, *IPPS 10th Int. Parallel Processing Symp.,* IEEE Computer Society Press, 1995, 891-897.

[7] **Kazerouni L., Rajan B. and Shyamasundar R.K.,** Mapping linear recurrence equations onto systolic architectures, *Int. J. of High Speed Computing (IJHSC),* **8** (3) (1996), 229-270.

[8] **Leiserson C.E. and Saxe J.B.,** Optimizing synchronous systems, *Journal of VLSI and Computer Systems,* **1** (1) (1983), 41-67.

[9] **Moldovan D.I. and Fortes J.A.B.,** Partitioning and mapping algorithms into fixed size systolic arrays, *IEEE Trans. Computers,* **35** (1) (1986), 1-12.

[10] **Rajko S. and Aluru S.,** Space and time optimal parallel sequence alignments, *Transactions on Parallel and Distributed Systems,* **15** (12) (2004), 1070-1081.

[11] **Ruff L. and Jebelean T.,** Functional-based synthesis of a systolic array for GCD computation, *Proc. of the* 18^{th} *Int. Symp. on Implementation and Application of Functional Languages IFL'06, Budapest, August 2006,* Technical Report No:2006-S01, ISBN 9634638767, 44-61.

[12] **Song S.W.,** *Systolic algorithms: concepts, synthesis, and evolution,* Temuco, CIMPA School of Parallel Computing, Chile, 1994.

[13] **Szakács L.,** *Automatic design of systolic arrays: A short survey,* Technical Report 02-27, RISC Report Series, University of Linz, Austria, 2002.

[14] **Szakács L. and Chiorean I.,** Automatic derivation of a systolic algorithm for sequences comparison, *Analele Universitatii din Timisoara, Seria Matematica - Informatica, Special Issue on Computer Science - Proc. of SYNASC'03,* **XLI** (2003), 213-227.

[15] **Wolfram S.,** *The Mathematica book,* 5th edition, Wolfram Media, 2003.

L. Ruff
Babeş-Bolyai University
Cluj-Napoca, Romania
laura@cs.ubbcluj.ro